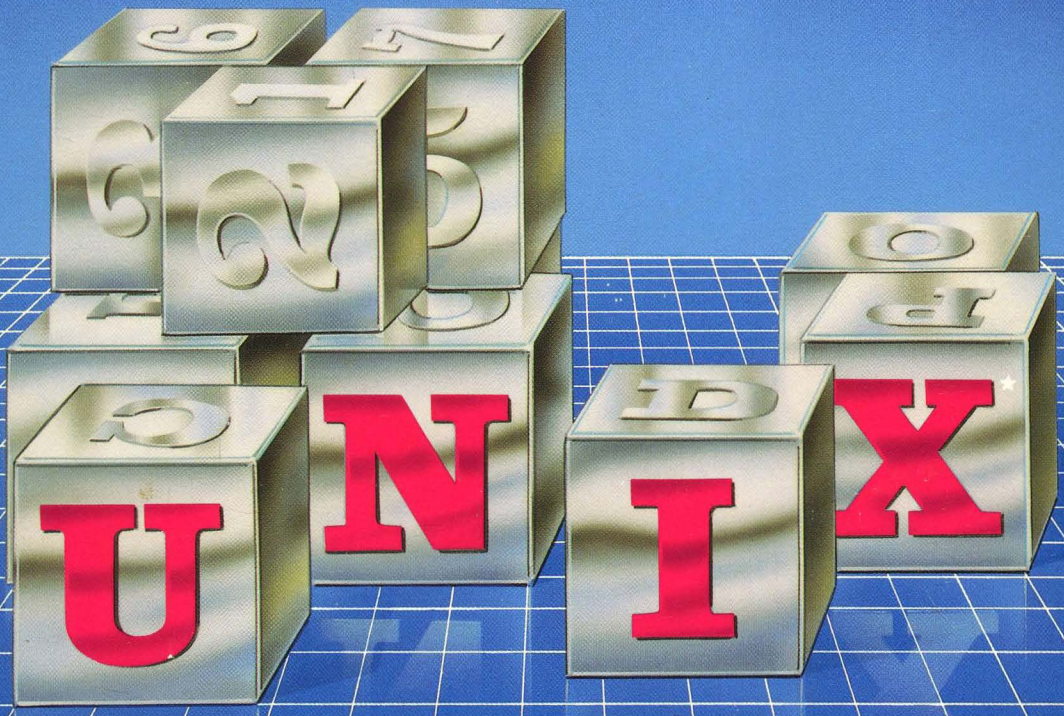
VOLUME 5

LANGUAGES AND SUPPORT TOOLS



programmer's manual

CBS COLLEGE PUBLISHING'S
UNIX* SYSTEM LIBRARY

* TRADEMARK OF AT&T

AT&T

VOLUME 5

LANGUAGES AND SUPPORT TOOLS

UNIX*

programmer's manual

CBS COLLEGE PUBLISHING'S
UNIX SYSTEM LIBRARY

* Trademark of AT&T.

VOLUME 5

LANGUAGES AND SUPPORT TOOLS

UNIX*

programmer's manual

CBS COLLEGE PUBLISHING'S
UNIX SYSTEM LIBRARY

Steven V. Earhart: Editor

HOLT, RINEHART AND WINSTON
New York Chicago San Francisco Philadelphia
Montreal Toronto London Sydney Tokyo
Mexico City Rio de Janeiro Madrid

---

* Trademark of AT&T.

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy of all information in this document, AT&T assumes no liability to any party for any loss or damage caused by errors or omissions or statements of any kind in the *UNIX* Programmer's Manual*, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. AT&T further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. AT&T disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties or merchantability or fitness for a particular purpose.

AT&T reserves the right to make changes without further notice to any products herein to improve reliability, function or design.

This document was set on an AUTOLOGIC, Inc. APS-5 phototypesetter driven by the TROFF formatter operating under the UNIX system on an AT&T 3B20 computer.

---

* Trademark of AT&T.

Address correspondence to:
383 Madison Avenue
New York, NY 10017

Library of Congress Cataloging-in-Publication Data

UNIX programmer's manual.

At head of title: AT&T
Inclues index.
Contents: v. 1. Commands and utilities—v. 2.
System calls and library routines—v. 3. System
administration facilities.—v. 4. Document Preparation. —v. 5. Languages and support tools.
1. UNIX (Computer operating system) I. Earhart,
Steven V. II. American Telephone and Telegraph Company.
QA76.76.063U548 1986 005.4'3 86-311

Select Code 320-035
ISBN 0-03-011204-4

Printed in the United States of America

Published simultaneously in Canada

678 090 98765432

CBS COLLEGE PUBLISHING
Holt, Rinehart and Winston
The Dryden Press
Saunders College Publishing

PREFACE

The *UNIX Programmer's Manual* describes most of the features of the UNIX System V. It does not provide a general overview of the UNIX system nor details of the implementation of the system.

Not all commands, features, or facilities described in this series are available in every UNIX system implementation. For specific questions on a machine implementation of the UNIX system, consult your system administrator.

The *UNIX Programmer's Manual* is available in several volumes.

- Volume 1 contains the Commands and Utilities (sections 1 and 6)
- Volume 2 contains the System Calls and Library Routines (sections 2,3,4, and 5).
- Volume 3 contains the System Administration Facilities (sections 1M, 7, and 8).
- Volume 4 contains the Document Preparation Facilities (**mm**, **tbl**, etc.).
- Volume 5 contains the Languages and Support Tools (**C language**, **lex**, **make**, etc.).

INTRODUCTION

This volume of the *UNIX** *Programmer's Manual* describes the languages and support tools that are available on most UNIX operation system implementations. These facilities are not available on all implemntations, for specific information as to availability consult your system administrator.

Two main programming languages are supported on the UNIX system. The languages include:

- C Language — A medium-level programming language which was used to write most of the UNIX operating system. Discussion includes the following:

  C LANGUAGE— provides a summary of the grammar and rules of the C programming language. The C language as it is implemented on most computers including the AT&T 3B computers, the PDP‡-11 computer, and the VAX‡-11/780 computer. Where differences exist, these chapters try to point out implementation-dependent details. With few exceptions, such dependencies follow directly from the properties of the hardware. The various compilers are generally quite compatible. Computers not listed in the examples, probably follow the same rules and guidelines. Consult your system support group for information.

  LIBRARIES— describe functions and declarations that support the C Language and how to use these functions. Both the C Library and the Object File and Math Libraries are discussed.

  THE "cc" COMMAND— describes the command used to compile C language programs, produce assembly language programs, and produce executable programs.

  A C PROGRAM CHECKER - "lint"— describes a program that attempts to detect compile-time bugs and non-portable features in C programs.

---

* Trademark of AT&T.

‡ Trademarks of Digital Equipment Corporation

A SYMBOLIC DEBUGGER - "sdb"— describes a symbolic debugging program that is used to debug compiled C language programs.

- Fortran — Fortran 77, and rational Fortran preprocessor (Ratfor) are described as follows:

  UNIX SYSTEM COMMANDS FOR FORTRAN— describes the various commands that may be used with Fortran on most UNIX system.

  FORTRAN 77— describes the implementation of Fortran 77 on the UNIX system in terms of the variations from the American National Standard.

  RATFOR— describes the Ratfor preprocessor. This preprocessor provides a means for writing Fortran in a fashion similar to the C language. This preprocessor provides (among other things) simplified control-flow statements.

  EFL— descibes a general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language.

The following paragraphs contain a brief description of the support tools that aid in program development.

- **A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)**— describes a software tool for maintaining, updating, and regenerating groups of computer programs. The many activities of program development and maintenance are made simpler by the **make** program.
- **AUGMENTED VERSION OF "make"**— is now combined with A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make).

- **SOURCE CODE CONTROL SYSTEM (SCCS) USER'S GUIDE**— describes the collection of SCCS programs under the UNIX operating system. The SCCS programs act as a "custodian" over the UNIX system files.
- **M4 MACRO PROCESSOR**— describes a general purpose macro processor that may be used as a front end for rational Fortran, C, and other programming languages.
- **"awk" PROGRAMMING LANGUAGE**— describes a software tool designed to make many common information retrieval and text manipulation tasks easy to state and to perform.
- **LINK EDITOR**— describes a software tool (**ld**) that creates load files by combining object files, performing relocation, and resolving internal references.
- **COMMON OBJECT FILE FORMAT (COFF)**— describes the output file produced on some UNIX systems by the assembler and the link editor.
- **ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC)**— describes a compiler for doing arbitrary precision arithmetic on the UNIX operating system.
- **INTERACTIVE DESK CALCULATOR (DC)**— describes a program implemented on the UNIX operating system to do arbitrary-precision integer arithmetic.
- **LEXICAL ANALYZER GENERATOR (Lex)**— describes a software tool that lexically processes character input streams.
- **YET ANOTHER COMPILER-COMPILER (yacc)**— describes the **yacc** program. The **yacc** program provides a general tool for imposing structure on the input to a computer program.
- **UNIX SYSTEM TO UNIX SYSTEM COPY (UUCP)**— describes a network that provides information exchange (between UNIX systems) over the direct distance dialing network.

The support tools provide an added dimension to the basic UNIX software commands. The "tools" described enable the user to fully utilize the UNIX operating system.

It is assumed that the user of this document has at least two years of specialized training in computer-related fields. The user is also expected to use the UNIX system for software development.

Throughout this document, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *UNIX Programmer's Manual—Volume 3: System Administration Facilities*. Each reference of the form **name**(1) and

name(6) refers to entries in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.* All other references to entries of the form **name**(N), where possibly followed by a letter, refer to entry **name** in section **N** of the *UNIX Programmer's manual—Volume 2: System Calls and Library Routines.*

TABLE OF CONTENTS

LANGUAGES

SUPPORT TOOLS

C LANGUAGE

LEXICAL CONVENTIONS

There are six classes of tokens - identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

Comments

The characters /* introduce a comment which terminates with the characters */. Comments do not nest.

Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names. The external name sizes include:

PDP-11	7 characters, 2 cases
VAX-11	>100 characters, 2 cases
AT&T 3B20	>100 characters, 2 cases

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**external**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Some implementations also reserve the words **fortran** and **asm**.

Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "NAMES." Hardware characteristics that affect sizes are summarized in "Hardware Characteristics" under "LEXICAL CONVENTIONS."

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant. As discussed below, on some machines integer and long values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in **'x'**. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is **int**.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") have type **int**.

Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of **char**" and storage class **static** (see "NAMES") and is initialized with the given characters. The compiler places a null byte (**\0**) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All strings, even when written identically, are distinct.

Hardware Characteristics

The following figures summarize certain hardware properties that vary from machine to machine.

DEC PDP-11 (ASCII)	
char	8 bits
int	16
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$

DEC PDP-11 HARDWARE CHARACTERISTICS

DEC VAX-11 (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$

DEC VAX-11 HARDWARE CHARACTERISTICS

AT&T 3B (ASCII)	
char	8 bits
int	32
short	16
long	32
float	32
double	64
float range	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 308}$

AT&T 3B HARDWARE CHARACTERISTICS

SYNTAX NOTATION

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

$$\{\ expression_{opt}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in "SYNTAX SUMMARY".

NAMES

The C language bases the interpretation of an identifier upon two attributes of the identifier - its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

Storage Class

There are four declarable storage classes:

- Automatic
- Static
- External
- Register.

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "STATEMENTS") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent. In particular, **char** may be signed or unsigned by default.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. **Char**, **int** of all sizes whether **unsigned** or not, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *Arrays* of objects of most types
- *Functions* which return objects of a given type
- *Pointers* to objects of a given type
- *Structures* containing a sequence of objects of various types
- *Unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

OBJECTS AND LVALUES

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if **E** is an expression of pointer type, then ***E** is an lvalue expression referring to the object to which **E** points. The name "lvalue" comes from the assignment expression **E1 = E2** in which the left operand **E1** must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

CONVERSIONS

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, **char** variables range in value from −128 to 127. The more explicit type **unsigned char** forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, '\377' has the value **-1**.

When a longer integer is converted to a shorter integer or to a **char,** it is truncated on the left. Excess bits are simply discarded.

Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a float.

Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** integer is converted to **long**, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

1. First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.

2. Then, if either operand is **double,** the other is converted to **double** and that is the type of the result.

3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.

4. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.

5. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.

6. Otherwise, if either operand is **unsigned,** the other is converted to **unsigned** and that is the type of the result.

7. Otherwise, both operands must be **int**, and that is the type of the result.

Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "STATEMENTS") or as the left operand of a comma expression (see "Comma Operator" under "EXPRESSIONS").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

EXPRESSIONS

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of "SYNTAX SUMMARY".

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

Primary Expressions

Primary expressions involving ., ->, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string
(expression)
primary-expression [expression]
primary-expression (expression-list$_{opt}$ *)*
primary-expression . identifier
primary-expression -> identifier

expression-list:
expression
expression-list , expression

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int** and floating constants have type **double**.

A string is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see "Initialization" under "DECLARATIONS.")

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is **int**, and the type

of the result is "...". The expression **E1[E2]** is identical (by definition) to ***((E1)+(E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, * and +, respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "TYPES REVISITED."

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...," and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "DECLARATIONS."

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from - and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the

structure or union to which the pointer expression points. Thus the expression **E1->MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure, Union, and Enumeration Declarations" under "DECLARATIONS."

Unary Operators

Expressions with unary operators group right to left.

unary-expression:
** expression*
& lvalue
- expression
! expression
~ expression
++ lvalue
--lvalue
lvalue ++
lvalue --
(type-name) expression
sizeof *expression*
sizeof *(type-name)*

The unary * operator means *indirection* ; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...," the type of the result is "...".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator ! is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix **++** is incremented. The value is the new value of the operand but is not an lvalue. The expression **++x** is equivalent to **x=x+1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix **++** operator.

When postfix **++** is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix **++** operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in "Type Names" under "Declarations."

The **sizeof** operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char.**) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(***type***)** is taken to be a unit, so the expression **sizeof(***type***)-2** is the same as **(sizeof(***type***))-2.**

Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
expression * *expression*
expression / *expression*
expression % *expression*

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b + a%b** is equal to **a** (if **b** is not 0).

Additive Operators

The additive operators + and - group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + *expression*
expression - *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as

the original pointer which points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the - operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **int** representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

Shift Operators

The shift operators << and >> group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

shift-expression:
    expression << *expression*
    expression >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

Relational Operators

The relational operators group left to right.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

Equality Operators

equality-expression:
expression == *expression*
expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

Bitwise AND Operator

and-expression:
expression & *expression*

The & operator is associative, and expressions involving & may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ *expression*

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | *expression*

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

Logical AND Operator

logical-and-expression:
expression && *expression*

The && operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Logical OR Operator

logical-or-expression:
expression || *expression*

The | operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, | guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:

lvalue = *expression*
lvalue += *expression*
lvalue -= *expression*
lvalue *= *expression*
lvalue /= *expression*
lvalue % = *expression*
lvalue >>= *expression*
lvalue <<= *expression*
lvalue & = *expression*
lvalue ^= *expression*
lvalue |= *expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* = **E2** may be inferred by taking it as equivalent to **E1** = **E1** *op* **(E2)**; however, **E1** is evaluated only once. In += and -=, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

Comma Operator

comma-expression:

expression , *expression*

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "DECLARATIONS"), the comma operator as described in this subpart can only appear in parentheses. For example,

f(a, (t=3, t+2), c)

has three arguments, the second of which has the value 5.

DECLARATIONS

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
auto
static
extern
register
typedef

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "Typedef" for more information. The meanings of the various storage classes were discussed in "Names."

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are **int** or pointer. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be **auto** inside a function, **extern** outside. Exception: functions are never automatic.

Type Specifiers

The type-specifiers are

type-specifier:
    struct-or-union-specifier
    typedef-name
    enum-specifier
basic-type-specifier:
    basic-type
    basic-type basic-type-specifiers
basic-type:
    char
    short
    int
    long
    unsigned
    float
    double
    void

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The

word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations." Declarations with **typedef** names are discussed in "Typedef."

Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , declarator-list

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in "Initialization". The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

The grouping is the same as in expressions.

Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T** ," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in **'int x"** is just **int**). Then if **D1** has the form

***D**

the type of the contained identifier is "... pointer to **T** ."

If **D1** has the form

D()

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

D[*constant-expression***]**

or

D[]

then the contained identifier has type "... array of T." In the first case, the constant expression is an expression whose value is determinable at compile time , whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

int i, *ip, f(), *fip(), (*pfi)();

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function which returns an integer. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. Using indirection through the (pointer) result to yield an integer. In the declarator **(*pfi)()**, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

float fa[17], *afp[17];

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

static int x3d[3][5][7];

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array" and the last has type **int**.

Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union { *struct-decl-list* }
struct-or-union identifier { *struct-decl-list* }
struct-or-union identifier

struct-or-union:
struct
union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field ;* its length, a non-negative constant expression, is set off from the field name by a colon.

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on the 3B20.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependant boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **int** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with **int** are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as **unsigned**. In all implementations, there are no arrays of fields, and the address-of operator **&**

may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

struct *identifier* { *struct-decl-list* }
union *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

struct *identifier*
union *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the **count** field of the structure to which **sp** points;

```
s.left
```

refers to the left subtree pointer of the structure **s**; and

```
s.right->tword[0]
```

refers to the first character of the **tword** member of the right subtree of **s**.

Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type, and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , initializer-list
{ *initializer-list* }
{ *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in "CONSTANT EXPRESSIONS", or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a one-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y** begins with a left brace but that for **y[0]** does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for **y[1]** and **y[2]**. Also,

```
float y[4][3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **y** (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of **sizeof**), it is desired to supply the name of a data type. This is accomplished using a "type name", which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
    type-specifier abstract-declarator

abstract-declarator:
    empty
    (abstract-declarator)
    ** abstract-declarator*
    abstract-declarator ()
    abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

    (abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration.

The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
int (*[3])()
```

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function returning an integer," and "array of three pointers to functions returning an integer."

Typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
    identifier

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators." For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**, " and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types which could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

STATEMENTS

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage so initialization is not permitted.

Conditional Statement

The two forms of the conditional statement are

if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The "else" ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

While Statement

The **while** statement has the form

while (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

Do Statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

For Statement

The **for** statement has the form:

for (*exp-1*$_{opt}$; *exp-2*$_{opt}$; *exp-3*$_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are simply dropped from the expansion above.

Switch Statement

The **switch** statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be **int**. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in "CONSTANT EXPRESSIONS."

There may also be at most one statement prefix of the form

default :

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a **default**, prefix, control passes to the prefixed statement. If no case matches and if there is no **default**, then none of the statements in the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see "Break Statement."

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

Break Statement

The statement

break ;

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

Continue Statement

The statement

continue ;

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...)        do              for (...)
{                  {               {
    ...                ...             ...
contin: ;          contin: ;       contin: ;
}                  } while (...);  }
```

a **continue** is equivalent to **goto contin.** (Following the **contin:** is a null statement, see "Null Statement".)

Return Statement

A function returns to its caller by means of the **return** statement which has one of the forms

return ;
return *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesized.

Goto Statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (see "Labeled Statement") located in the current function.

Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See "SCOPE RULES."

Null Statement

The null statement has the form

;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

EXTERNAL DEFINITIONS

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "Type Specifiers" in "DECLARATIONS") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

External Function Definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "Scope of Externals" in "SCOPE RULES" for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

The function-body has the form

function-body:
declaration-list$_{opt}$ *compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class which may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition follows

```
int max(a, b, c)
        int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator; **int a, b, c;** is the declaration-list for the formal parameters; { ... } is the block giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...."

External Data Definitions

An external data definition has the form

data-definition:
    declaration

The storage class of such data may be **extern** (which is the default) or **static** but not **auto** or **register**.

SCOPE RULES

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure, Union, and Enumeration Declarations" in "DECLARATIONS") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
    auto int distance;
    ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a mult-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

COMPILER CONTROL LINES

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

Token Replacement

A compiler-control line of the form

#define *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier*(*identifier, ...)token-string*$_{opt}$

where there is no space between the first identifier and the (, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#defined** identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

File Inclusion

A compiler control line of the form

#include "*filename*"

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include <*filename*>

searches only the specified or standard places and not the directory of the **#include**. (How the places are specified is not part of the language.)

#includes may be nested.

Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "CONSTANT EXPRESSIONS"; the following additional restrictions apply here: the constant expression may not contain **sizeof** casts, or an enumeration constant.)

A restricted constant expression may also contain the additional unary expression

defined *identifier*
or
defined(*identifier*

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#if**defined(*identifier*). A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to **#if!defined(***identifier***)**.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

#endif

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a

#else or, lacking a **#else**, the **#endif** are ignored.

These constructions may be nested.

Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

#line *constant "filename"*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by *"filename"*. If *"filename"* is absent, the remembered file name does not change.

IMPLICIT DECLARATIONS

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning ...," it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning **int**."

TYPES REVISITED

This part summarizes the operations which can be performed on objects of certain types.

Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the -> or the . must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int     type;
        } n;
        struct
        {
                int     type;
                int     intnode;
        } ni;
        struct
        {
                int     type;
                float   floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

Functions

There are only two things that can be done with a function m call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine since its appearance in **g(f)** was not followed by **(**.

Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator **[]** is interpreted in such a way that **E1[E2]** is identical to ***((E1)+(E2))**. Because of the conversion rules which apply to **+**, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2 -th** member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If **E** is an *n*-dimensional array of rank i×j×...×k, then **E** appearing in an expression is converted to a pointer to an (n-1)-dimensional array with rank j×...×k. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n-1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

int x[3][5];

Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves multiplying **i** by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "Unary Operators" under"EXPRESSIONS" and "Type Names"under "DECLARATIONS."

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The **char**'s have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that **double** quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B20 computer has 24-bit pointers placed into 32-bit quantities. Most objects are aligned on 4-byte boundaries. **Short**s are aligned in all cases on 2-byte boundaries. Arrays of characters, all structures, **int**s, **long**s, **float**s, and **double**s are aligned on 4-byte boundries; but structure members may be packed tighter.

CONSTANT EXPRESSIONS

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof** expressions, possibly connected by the binary

operators

```
+ - * / % & | ^ << >> == != < > <= >= && |
```

or by the unary operators

```
- ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary **&** operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary **&** can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

PORTABILITY CONSIDERATIONS

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

Expressions

The basic expressions are:

expression:
primary
* *expression*
&lvalue
\- *expression*
! *expression*
~ *expression*
++ *lvalue*

--lvalue
lvalue ++
lvalue --
sizeof *expression*
sizeof (*type-name*)
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string
(expression)
primary (expression-list$_{opt}$ *)*
primary [expression]
primary . identifier
primary -> identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary -> identifier
** expression*
(lvalue)

The primary-expression operators

() [] . ->

have highest priority and group left to right. The unary operators

* & - ! ~ ++ -- **sizeof** *(type-name)*

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:

```
*   /   %
+   -
>>   <<
<   >   <=   >=
==   !=
&
^
|
&&
|
```

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:

```
=  +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=
```

The comma operator has the lowest priority and groups left to right.

Declarations

declaration:

decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:

type-specifier decl-specifiers$_{opt}$

sc-specifier decl-specifiers$_{opt}$

sc-specifier:
auto
static
extern
register
typedef

type-specifier:
struct-or-union-specifier
typedef-name
enum-specifier
basic-type-specifier:
basic-type
basic-type basic-type-specifiers
basic-type:
char
short
int
long
unsigned
float
double
void

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

init-declarator-list:
init-declarator
init-declarator , *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

struct-or-union-specifier:
struct { *struct-decl-list* }
struct *identifier* { *struct-decl-list* }
struct *identifier*
union { *struct-decl-list* }
union *identifier* { *struct-decl-list* }
union *identifier*

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initializer:
= *expression*
= { *initializer-list* }
= { *initializer-list* , }

initializer-list:
expression
initializer-list , initializer-list
{ initializer-list }
{ initializer-list , }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ()
abstract-declarator [constant-expression$_{opt}$ *]*

typedef-name:
identifier

Statements

compound-statement:
{ declaration-list$_{opt}$ *statement-list*$_{opt}$ *}*

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:

compound-statement
expression ;
if *(expression) statement*
if *(expression) statement* **else** *statement*
while *(expression) statement*
do *statement* **while** *(expression) ;*
for *(exp_{opt};exp_{opt};exp_{opt}) statement*
switch *(expression) statement*
case *constant-expression : statement*
default *: statement*
break ;
continue ;
return ;
return *expression ;*
goto *identifier ;*
identifier : statement
;

External definitions

program:

external-definition
external-definition program

external-definition:

function-definition
data-definition

function-definition:

decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:

declarator (parameter-list$_{opt}$ *)*

parameter-list:

identifier
identifier , parameter-list

function-body:
declaration-list$_{opt}$ *compound-statement*

data-definition:
extern *declaration* ;
static *declaration* ;

Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier*(*identifier*,...)*token-string*$_{opt}$
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant* "*filename*"

C LIBRARIES

GENERAL

This chapter and **THE OBJECT AND MATH LIBRARIES** chapter describe the libraries that are supported on the UNIX operating system. A library is a collection of related functions and/or declarations that simplify programming effort by linking only what is needed, allowing use of locally produced functions, etc. All of the functions described are also described in Part 3 of the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines.* Most of the declarations described are in Part 5 of the **2UNIX Programmer's Manual—Volume 2: System Calls and Library Routines.** The three main libraries on the UNIX system are:

C library — This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described later in this chapter.

Object file — This library provides functions for the access and manipulation of object files. This library is described in **THE OBJECT AND MATH LIBRARIES** chapter.

Math library — This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is described in **THE OBJECT AND MATH LIBRARIES** chapter.

Some libraries consist of two portions - functions and declarations. In some cases, the user must request that the functions (and/or declarations) of a specific library be included in a program being compiled. In other cases, the functions (and/or declarations) are included automatically.

Including Functions

When a program is being compiled, the compiler will automatically search the C language library to locate and include functions that are used in the program. This is the case only for the C library and no other library. In order for the compiler to locate and include functions from other libraries, the user must specify these libraries on the command line for the compiler. For example, when using functions of the math library, the user must request that the math library be searched by including the argument **-lm** on the command line, such as:

```
cc file.c -lm
```

The argument **-lm** must come after all files that reference functions in the math library in order for the link editor to know which functions to include in the a.out file.

This method should be used for all functions that are not part of the C language library.

Including Declarations

Some functions require a set of declarations in order to operate properly. A set of declarations is stored in a file under the */usr/include* directory. These files are referred to as *header files*. In order to include a certain header file, the user must specify this request within the C language program. The request is in the form:

```
#include <file.h>
```

where *file.h* is the name of the file. Since the header files define the type of the functions and various preprocessor constants, they must be included before invoking the functions they declare.

The remainder of this chapter describes the functions and header files of the C Library. The description of the library begins with the actions required by the user to include the functions and/or header files in a program being compiled (if any). Following the description of the actions required is information in three-column format of the form:

function reference (N) Brief description.

The functions are grouped by type while the reference refers to section 'N' in the **2UNIX Programmer's Manual—Volume 2: System Calls and Library Routines**. Following this, are descriptions of the header files associated with these functions (if any).

THE C LIBRARY

The C library consists of several types of functions. All the functions of the C library are loaded automatically by the compiler. Various declarations must sometimes be included by the user as required. The functions of the C library are divided into the following types:

- Input/output control
- String manipulation
- Character manipulation
- Time functions
- Miscellaneous functions.

Input/Output Control

These functions of the C library are automatically included as needed during the compiling of a C language program. No command line request is needed.

The header file required by the input/output functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <stdio.h>
```

near the beginning of each file that references an input or output function.

The input/output functions are grouped into the following categories:

- File access
- File status
- Input

- Output
- Miscellaneous.

File Access Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fclose	**fclose** (3S)	Close an open stream.
fdopen	**fopen** (3S)	Associate stream with an **open**(2) ed file.
fileno	**ferror** (3S)	File descriptor associated with an open stream.
fopen	**fopen** (3S)	Open a file with specified permissions. **Fopen** returns a pointer to a stream which is used in subsequent references to the file.
freopen	**fopen** (3S)	Substitute named file in place of open stream.
fseek	**fseek** (3S)	Reposition the file pointer.
pclose	**popen** (3S)	Close a stream opened by **popen.**
popen	**popen** (3S)	Create pipe as a stream between calling process and command.
rewind	**fseek** (3S)	Reposition file pointer at beginning of file.

setbuf	**setbuf** (3S)	Assign buffering to stream.
vsetbuf	**setbuf** (3S)	Similar to **setbuf**, but allowing finer control.

File Status Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
clearerr	**ferror** (3S)	Reset error condition on stream.
feof	**ferror** (3S)	Test for "end of file" on stream.
ferror	**ferror** (3S)	Test for error condition on stream.
ftell	**fseek** (3S)	Return current position in the file.

Input Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fgetc	**getc** (3S)	True function for **getc** (3S).
fgets	**gets** (3S)	Read string from stream.
fread	**fread** (3S)	General buffered read from stream.
fscanf	**scanf** (3S)	Formatted read from stream.

getc	**getc** (3S)	Read character from stream.
getchar	**getc** (3S)	Read character from standard input.
gets	**gets** (3S)	Read string from standard input.
getw	**getc** (3S)	Read word from stream.
scanf	**scanf** (3S)	Read using format from standard input.
sscanf	**scanf** (3S)	Formatted from string.
ungetc	**ungetc** (3S)	Put back one character on stream.

Output Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
fflush	**fclose** (3S)	Write all currently buffered characters from stream.
fprintf	**printf** (3S)	Formatted write to stream.
fputc	**putc** (3S)	True function for **putc** (3S).
fputs	**puts** (3S)	Write string to stream.
fwrite	**fread** (3S)	General buffered write to stream.
printf	**printf** (3S)	Print using format to standard output.

putc	**putc** (3S)	Write character to standard output.
putchar	**putc** (3S)	Write character to standard output.
puts	**puts** (3S)	Write string to standard output.
putw	**putc** (3S)	Write word to stream.
sprintf	**printf** (3S)	Formatted write to string.
vfprintf	**vprint** (3C)	Print using format to stream by **varargs**(5) argument list.
vprintf	**vprint** (3C)	Print using format to standard output by **varargs**(5) argument list.
vsprintf	**vprintf** (3C)	Print using format to stream string by **varargs**(5) argument list.

Miscellaneous Functions

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ctermid	**ctermid** (3S)	Return file name for controlling terminal.
cuserid	**cuserid** (3S)	Return login name for owner of current process.
system	**system** (3S)	Execute shell command.

tempnam	**tempnam** (3S)	Create temporary file name using directory and prefix.
tmpnam	**tmpnam** (3S)	Create temporary file name.
tmpfile	**tmpfile** (3S)	Create temporary file.

String Manipulation Functions

These functions are used to locate characters within a string, copy, concatenate, and compare strings. These functions are automatically located and loaded during the compiling of a C language program. No command line request is needed since these functions are part of the C library. The string manipulation functions are declared in a header file that may be included in the program being compiled. This is accomplished by including the line:

```
#include <string.h>
```

near the beginning of each file that uses one of these functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
strcat	**string** (3C)	Concatenate two strings.
strchr	**string** (3C)	Search string for character.
strcmp	**string** (3C)	Compares two strings.
strcpy	**string** (3C)	Copy string.
strcspn	**string** (3C)	Length of initial string not containing set of characters.

strlen	**string** (3C)	Length of string.
strncat	**string** (3C)	Concatenate two strings with a maximum length.
strncmp	**string** (3C)	Compares two strings with a maximum length.
strncpy	**string** (3C)	Copy string over string with a maximum length.
strpbrk	**string** (3C)	Search string for any set of characters.
strrchr	**string** (3C)	Search string backwards for character.
strspn	**string** (3C)	Length of initial string containing set of characters.
strtok	**string** (3C)	Search string for token separated by any of a set of characters.

Character Manipulation

The following functions and declarations are used for testing and translating ASCII characters. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The declarations associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <ctype.h>
```

near the beginning of the file being compiled.

Character Testing Functions

These functions can be used to identify characters as uppercase or lowercase letters, digits, punctuation, etc.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
isalnum	**ctype** (3C)	Is character alphanumeric?
isalpha	**ctype** (3C)	Is character alphabetic?
isascii	**ctype** (3C)	Is integer ASCII character?
iscntrl	**ctype** (3C)	Is character a control character?
isdigit	**ctype** (3C)	Is character a digit?
isgraph	**ctype** (3C)	Is character a printable character?
islower	**ctype** (3C)	Is character a lowercase letter?
isprint	**ctype** (3C)	Is character a printing character including space?
ispunct	**ctype** (3C)	Is character a punctuation character?
isspace	**ctype** (3C)	Is character a white space character?
isupper	**ctype** (3C)	Is character an uppercase letter?
isxdigit	**ctype** (3C)	Is character a hex digit?

Character Translation Functions

These functions provide translation of uppercase to lowercase, lowercase to uppercase, and integer to ASCII.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
toascii	**conv** (3C)	Convert integer to ASCII character.
tolower	**conv** (3C)	Convert character to lowercase.
toupper	**conv** (3C)	Convert character to uppercase.

Time Functions

These functions are used for accessing and reformatting the systems idea of the current date and time. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <time.h>
```

near the beginning of any file using the time functions.

These functions (except **tzset**) convert a time such as returned by **time**(2).

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
asctime	**ctime** (3C)	Return string representation of date and time.
ctime	**ctime** (3C)	Return string representation of date and time, given integer form.
gmtime	**ctime** (3C)	Return Greenwich Mean Time.
localtime	**ctime** (3C)	Return local time.
tzset	**ctime** (3C)	Set time zone field from environment variable.

Miscellaneous Functions

These functions support a wide variety of operations. Some of these are numerical conversion, password file and group file access, memory allocation, random number generation, and table management. These functions are automatically located and included in a program being compiled. No command line request is needed since these functions are part of the C library.

Some of these functions require declarations to be included. These are described following the descriptions of the functions.

Numerical Conversion

The following functions perform numerical conversion.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
a64l	**a64l**(3C)	Convert string to base 64 ASCII.
atof	**atof**(3C)	Convert string to floating.
atoi	**atof**(3C)	Convert string to integer.
atol	**atof**(3C)	Convert string to long.
frexp	**frexp**(3C)	Split floating into mantissa and exponent.
l3tol	**l3tol**(3C)	Convert 3-byte integer to long.
ltol3	**l3tol**(3C)	Convert long to 3-byte integer.
ldexp	**frexp**(3C)	Combine mantissa and exponent.
l64a	**a64l**(3C)	Convert base 64 ASCII to string.
modf	**frexp**(3C)	Split mantissa into integer and fraction.

DES Algorithm Access

The following functions allow access to the Data Encryption Standard (DES) algorithm used on the UNIX operating system. The DES algorithm is implemented with variations to frustrate use of hardware implementations of the DES for key search.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
crypt	**crypt** (3C)	Encode string.
encrypt	**crypt** (3C)	Encode/decode string of 0s and 1s.
setkey	**crypt** (3C)	Initialize for subsequent use of **encrypt**.

Group File Access

The following functions are used to obtain entries from the group file. Declarations for these functions must be included in the program being compiled with the line:

```
#include <grp.h>
```

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
endgrent	**getgrent** (3C)	Close group file being processed.
getgrent	**getgrent** (3C)	Get next group file entry.
getgrgid	**getgrent** (3C)	Return next group with matching gid.
getgrnam	**getgrent** (3C)	Return next group with matching name.
setgrent	**getgrent** (3C)	Rewind group file being processed.
fgetgrent	**getgrent** (3C)	Get next group file entry from a specified file.

Password File Access

These functions are used to search and access information stored in the password file (/etc/passwd). Some functions require declarations that can be included in the program being compiled by adding the line:

```
#include <pwd.h>
```

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
endpwent	**getpwent** (3C)	Close password file being processed.
getpw	**getpw** (3C)	Search password file for uid.
getpwent	**getpwent** (3C)	Get next password file entry.
getpwnam	**getpwent** (3C)	Return next entry with matching name.
getpwuid	**getpwent** (3C)	Return next entry with matching uid.
putpwent	**putpwent** (3C)	Write entry on stream.
setpwent	**getpwent** (3C)	Rewind password file being accessed.
fgetpwent	**getpwent** (3C)	Get next password file entry from a specified file.

Parameter Access

The following functions provide access to several different types of paramenters. None require any declarations.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
getopt	**getopt** (3C)	Get next option from option list.

getcwd	**getcwd** (3C)	Return string representation of current working directory.
getenv	**getenv** (3C)	Return string value associated with environment variable.
getpass	**getpass** (3C)	Read string from terminal without echoing.
putenv	**putenv** (3C)	Change or add value of an environment variable.

Hash Table Management

The following functions are used to manage hash search tables. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <search.h>
```

near the beginning of any file using the search functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
hcreate	**hsearch** (3C)	Create hash table.
hdestroy	**hsearch** (3C)	Destroy hash table.
hsearch	**hsearch** (3C)	Search hash table for entry.

Binary Tree Management

The following functions are used to manage a binary tree. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <search.h>
```

near the beginning of any file using the search functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
tdelete	**tsearch**(3C)	Deletes nodes from binary tree.
tfind	**tsearch**(3C)	Find element in binary tree.
tsearch	**tsearch**(3C)	Look for and add element to binary tree.
twalk	**tsearch**(3C)	Walk binary tree.

Table Management

The following functions are used to manage a table. Since none of these functions allocate storage, sufficient memory must be allocated before using these functions. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

```
#include <search.h>
```

near the beginning of any file using the search functions.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
bsearch	**bsearch**(3C)	Search table using binary search.

lfind	**lsearch**(3C)	Find element in library tree.
lsearch	**lsearch**(3C)	Look for and add element in binary tree.
qsort	**qsort**(3C)	Sort table using quick-sort algorithm.

Memory Allocation

The following functions provide a means by which memory can be dynamically allocated or freed.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
calloc	**malloc**(3C)	Allocate zeroed storage.
free	**malloc**(3C)	Free previously allocated storage.
malloc	**malloc**(3C)	Allocate storage.
realloc	**malloc**(3C)	Change size of allocated storage.

The following is another set of memory allocation functions available.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
calloc	**malloc**(3X)	Allocate zeroed storage.
free	**malloc**(3X)	Free previously allocated storage.
malloc	**malloc**(3X)	Allocate storage.
mallopt	**malloc**(3X)	Control allocation algorithm.

mallinfo	**malloc**(3X)	Space usage.
realoc	**malloc**(3X)	Change size of allocated storage.

Pseudorandom Number Generation

The following functions are used to generate pseudorandom numbers. The functions that end with **48** are a family of interfaces to a pseudorandom number generator based upon the linear congruent algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions provide an interface to a multiplicative congruential random number generator with period of 232.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
drand48	**drand48**(3C)	Random double over the interval [0 to 1).
lcong48	**drand48**(3C)	Set parameters for **drand48**, **lrand48**, and **mrand48**.
lrand48	**drand48**(3C)	Random long over the interval [0 to 2^{31}).
mrand48	**drand48**(3C)	Random long over the interval [-2^{31} to 2^{31}).
rand	**rand**(3C)	Random integer over the interval [0 to 32767).
seed48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48**.
srand	**rand**(3C)	Seed the generator for **rand**.
srand48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48** using a long.

Signal Handling Functions

The functions **gsignal** and **ssignal** implement a software facility similar to **signal**(2) in the **2UNIX Programmer's Manual—Volume 2: System Calls and Library Routines**. This facility enables users to indicate the disposition of error conditions and allows users to handle signals for their own purposes. The declarations associated with these functions can be included in the program being complied by the line

```
#include <signal.h>
```

These declarations define ASCII names for the 15 software signals.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
gsignal	**ssignal**(3C)	Send a software signal.
ssignal	**ssignal**(3C)	Arrange for handling of software signals.

Miscellaneous

The following functions do not fall into any previously described category.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
abort	**abort**(3C)	Cause an IOT signal to be sent to the process.
abs	**abs**(3C)	Return the absolute integer value.
ecvt	**ecvt**(3C)	Convert double to string.
fcvt	**ecvt**(3C)	Convert double to string using Fortran Format.

gcvt	**ecvt** (3C)	Convert double to string using Fortran F or E format.
isatty	**ttyname** (3C)	Test whether integer file descriptor is associated with a terminal.
mktemp	**mktemp** (3C)	Create file name using template.
monitor	**monitor** (3C)	Cause process to record a histogram of program counter location.
swab	**swab** (3C)	Swap and copy bytes.
ttyname	**ttyname** (3C)	Return pathname of terminal associated with integer file descriptor.

THE OBJECT AND MATH LIBRARIES

GENERAL

This chapter describes the Object and Math Libraries that are supported on the UNIX operating system. A library is a collection of related functions and/or declarations that simplify programming effort. All of the functions described are also described in Part 3 of the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines.* Most of the declarations described are in Part 5 of the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines.* The three main libraries on the UNIX system are:

C library — This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described in the **C Libraries** chapter in this volume.

Object file — This library provides functions for the access and manipulation of object files. This library is described later in this chapter.

Math library — This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is also described later in this chapter.

THE OBJECT FILE LIBRARY

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. For a description of the format of an object file, see **The Common Object File Format** in this volume.

This library consists of several portions. The functions reside in */usr/lib/libld.a* and are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

cc file **-lld**

which causes the link editor to search the object file library. The argument **-lld** must appear after all files that reference functions in *libld.a.*

In addition, various header files must be included. This is accomplished by including the line:

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ldaclose	**ldclose** (3X)	Close object file being processed.
ldahread	**ldahread** (3X)	Read archive header.
ldaopen	**ldopen** (3X)	Open object file for reading.
ldclose	**ldclose** (3X)	Close object file being processed.
ldfhread	**ldfhread** (3X)	Read file header of object file being processed.
ldgetname	**ldgetname** (3X)	Retrieve the name of an object file symbol table entry.

ldlinit	**ldlread** (3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	**ldlread** (3X)	Read line number entry from object file after **ldlinit**.
ldlread	**ldlread** (3X)	Read line number entry from object file.
ldlseek	**ldlseek** (3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	**ldlseek** (3X)	Seeks to the line number entries of the object file being processed given the name of a section.
ldnrseek	**ldrseek** (3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	**ldshread** (3X)	Read section header of the named section of the object file being processed.
ldnsseek	**ldsseek** (3X)	Seeks to the section of the object file being processed given the name of a section.
ldohseek	**ldohseek** (3X)	Seeks to the optional file header of the object file being processed.
ldopen	**ldopen** (3X)	Open object file for reading.

ldrseek	**ldrseek** (3X)	Seeks to the relocation entries of the object file being processed.
ldshread	**ldshread** (3X)	Read section header of an object file being processed.
ldsseek	**ldsseek** (3X)	Seeks to the section of the object file being processed.
ldtbindex	**ldtbindex** (3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	**ldtbread** (3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	**ldtbseek** (3X)	Seeks to the symbol table of the object file being processed.
sgetl	**sputl** (3X)	Access long integer data in a machine independant format.
sputl	**sputl** (3X)	Translate a long integer into a machine independant format.

Common Object File Interface Macros (ldfcn.h)

The interface between the calling program and the object file access routines is based on the defined type **LDFILE** which is defined in the header file **ldfcn.h** (see **ldfcn**(4)). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen**(3X) allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through the following macros: the **type** macro returns the magic number of the file, which is used to distinguish between archive files and simple object files. The **IOPTR** macro returns the file pointer which was opened by **ldopen**(3X) and is used by the input/output functions of the C library. The **OFFSET** macro returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file. The **HEADER** macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an **LDFILE** structure into a reference to its file descriptor field. The available macros are described in **ldfcn**(4) in the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines.*

THE MATH LIBRARY

The math library consists of functions and a header file. The functions are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

cc file **-lm**

which causes the link editor to search the math library. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

#include <math.h>

near the beginning of the (first) file being compiled.

The functions are grouped into the following categories:

- Trigonometric functions

- Bessel functions
- Hyperbolic functions
- Miscellaneous functions.

Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double precision.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
acos	**trig**(3M)	Return arc cosine.
asin	**trig**(3M)	Return arc sine.
atan	**trig**(3M)	Return arc tangent.
atan2	**trig**(3M)	Return arc tangent of a ratio.
cos	**trig**(3M)	Return cosine.
sin	**trig**(3M)	Return sine.
tan	**trig**(3M)	Return tangent.

Bessel Functions

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel**(3M).

Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
cosh	**sinh** (3M)	Return hyperbolic cosine.
sinh	**sinh** (3M)	Return hyperbolic sine.
tanh	**sinh** (3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double precision numbers.

FUNCTION	*REFERENCE*	*BRIEF DESCRIPTION*
ceil	**floor** (3M)	Returns the smallest integer not less than a given value.
exp	**exp** (3M)	Returns the exponential function of a given value.
fabs	**floor** (3M)	Returns the absolute value of a given value.
floor	**floor** (3M)	Returns the largest integer not greater than a given value.
fmod	**floor** (3M)	Returns the remainder produced by the division of two given values.

gamma	**gamma** (3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	**hypot** (3M)	Return the square root of the sum of the squares of two numbers.
log	**exp** (3M)	Returns the natural logarithm of a given value.
log10	**exp** (3M)	Returns the lorarithm base ten of a given value.
matherr	**matherr** (3M)	Error-handling function.
pow	**exp** (3M)	Returns the result of a given value raised to another given value.
sqrt	**exp** (3M)	Returns the square root of a given value.

COMPILER AND C LANGUAGE

This chapter describe the UNIX System's C compiler, **cc**, and the C programming language that the compiler translates. The compiler is part of the UNIX System Software Generation System (SGS).

The SGS is a package of tools used to create and test programs for UNIX Systems. These tools allow high-level program coding and source-level testing of code. The C language is implemented for high-level programming; it contains many control and structuring facilities that greatly simplify the task of algorithm construction. Within the SGS, a C compiler converts C programs into assembly language programs that are ultimately translated into object files by the assembler, **as**. The link editor, **ld**, collects and merges object files into executable load modules. Each of these tools preserves all symbolic information necessary for meaningful symbolic testing at C-language source level. In addition, a utility package aids in testing and debugging.

The current manual page for the C compiler can be obtained with the SGS command:

man cc

USE OF THE COMPILER

The main command of the SGS is **cc**; it operates much like the UNIX system **cc** command. To use the compiler, first creat a file (typically by using the UNIX system text editor) containing C source code. The name of the file created must have a special format; the last two characters of the file name must be **.c** as in *file1.c*.

Next, enter the SGS command

cc *options file.c*

to invoke the compiler on the C source file *file.c* with the appropriate *options* selected. The compilation process creates an absolute binary file named **a.out** that reflects the contents of *file.c* and any referenced library routines. The resulting binary file, **a.out**, can then be executed on the target system.

Options can control the steps in the compilation process. When none of the controlling options are used, and only one file is named, **cc** automatically calls the assembler, **as**, and the link editor, **ld**, thus resulting in an executable file, named **a.out**. If more than one file is named in a command,

cc *file1.c file2.c file3.c*

then the output will be placed on files *file1.o*, *file2.o*, and *file3.o*. These files can then be linked and executed through the **ld** command.

The **cc** compiler also accepts input file names with the last two characters **.s**. The **.s** signifies a source file in assembly language. The **cc** compiler passes this type of file directly to **as**, which assembles the file and places the output on a file of the same name with **.o** substituted for **.s**.

Cc is based on a portable C compiler and translates C source files into assembly code. Whenever the command **cc** is used, the standard C preprocessor (which resides on the file **/lib/cpp**) is called. The preprocessor performs file inclusion and macro substitution. The preprocessor is always invoked by **cc** and need not be called directly by the programmer. Then, unless the appropriate flags are set, **cc** calls the assembler and the link editor to produce an executable file.

COMPILER OPTIONS

All options recognized by the **cc** command are listed below:

Option	*Argument*	*Description*
-c	none	Suppress the link-editing phase of compilation and force an object file to be produced even if only one file is compiled.
-g	none	Produce symbolic debugging information.
-p	none	Reserved for invoking a profiler.

-D	*identifier[=constant]*	Define the external symbol *identifier* to the preprocessor, and give it the value *constant* (if specified).
-E	none	Same as the **-P** option except output is directed to the standard output.
-I	*directory*	Change the algorithm that searches for **#include** files whose names do not begin with / to look in the named *directory* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" are searched for first in the directory of the file being compiled, then in directories named by the **-I** options, and last in directories on the standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.
-O	none	Invoke an object code optimizer.
-P	none	Suppress compilation and loading; i.e., invoke only the preprocessor and leave out the output on corresponding files suffixed **.i**.
-U	*identifier*	Undefine the named *identifier* to the preprocessor.
-V	none	Print the version of the assembler that is invoked.
-W	*c,arg1[,arg2...]*	Pass along the argument(s) *argi* to pass *c*, where *c* is one of **[p012all]**, indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively.

This part provides additional information for those options not completely described above.

By using appropriate options, compilation can be terminated early to produce one of several intermediate translations such as relocatable object files (**-c** option), assembly source expansions for C code (**-S** option), or the output of the preprocessor (**-P** option). In general, the intermediate files may be saved and later resubmitted to the **cc** command, with other files or libraries included as necessary.

When compiling C source files, the most common practice is to use the **-c** option to save relocatable files. Subsequent changes to one file do not then require that the others be recompiled. A separate call to **cc** without the **-c** option then creates the linked executable **a.out** file. A relocatable object file created under the **-c** option is named by adding a **.o** suffix to the source file name.

The **-W** option provides the mechanism to specify options for each step that is normally invoked from the **cc** command line. These steps are preprocessing, the first pass of the compiler, the second pass of the compiler, optimization, assembly, and link editing. At this time, only assembler and link editor options can be used with the **-W** option. The most common example of use of the **-W** option is "**-W***a,-m*", which passes the **-m** option to the assembler. Specifying "**-wl,-m**" passes the **-m** option to the link editor.

When the **-P** option is used, the compilation process stops after only preprocessing, with output left on *file.i*. This file will be unsuitable for subsequent processing by **cc**.

The **-O** option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. However, line numbers used for symbolic debugging may be transposed when the optimizer is used.

The **-g** option produces information for a symbolic debugger. The SGS currently supports the SDB symbolic debugger.

A C PROGRAM CHECKER—"lint"

GENERAL

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions which nevertheless are legal. The **lint** program accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form:

lint [options] files ... library-descriptors ...

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

-a Suppress messages about assignments of long values to variables that are not long.

-b Suppress messages about break statements that cannot be reached.

-c Only check for intra-file bugs; leave external information in files suffixed with **.ln**.

-h Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).

-n Do not check for compatibility with either the standard or the portable **lint** library.

-0 *name*	Create a lint library from input files named **llib-l***name***.ln**.
-p	Attempt to check portability to other dialects of C language.
-u	Suppress messages about function and external variables used and not defined or defined and not used.
-v	Suppress messages about unused arguments in functions.
-x	Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as, **-ab** or **-xha**.

The names of files that contain C language programs should end with the suffix **.c** which is mandatory or **lint** and the C compiler.

The **lint** program accepts certain arguments, such as:

```
−ly
```

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions.

The **lint** library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file but are not used on a source file do not result in messages. The **lint** program does not simulate a full library search algorithm and will print messages if the source

files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the **−p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **−n** option can be used to suppress all library checking.

TYPES OF MESSAGES

The following paragraphs describe the major categories of messages printed by **lint**.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The **lint** program prints messages about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit **extern** statements but are never referenced; thus the statement

```
extern double sin();
```

will evoke no comment if **sin** is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest and can be discovered by using the **−x** option with the **lint** command.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **−v** option is available to suppress the printing of messages

about unused arguments. When **−v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the program before the function. This has the effect of the **-v** option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

There is one case where information about unused or undefined variables is more distracting than helpful. This is when **lint** is applied to some but not all files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used. Conversely, many functions and variables defined elsewhere may be used. The **−u** option may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

The **lint** program attempts to detect cases where a variable is used before it is set. The **lint** program detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use", since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print messages about some programs which are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables which are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

The **lint** program attempts to detect unreachable portions of the programs which it processes. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. The **lint** program also prints messages about loops which cannot be entered at the top. Some valid programs may have such loops which are considered to be bad style at best and bugs at worst.

The **lint** program has no way of detecting functions which are called and never returned. Thus, a call to **exit** may cause an unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program cannot be reached but it is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached.

The **lint** program will not print a message about unreachable **break** statements. Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements. The **−O** option in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance. There is typically nothing the user can do

about them, and the resulting messages would clutter up the **lint** output. If these messages are desired, **lint** can be invoked with the **−b** option.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function "values" that have never been returned. The **lint** program addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; the **lint** program will give the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a message from **lint**. If *g*, like **exit**, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

The **lint** program enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- Across certain binary operators and implied assignments
- At the structure selection operators
- Between the definition and uses of functions
- In the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the **->** be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the program immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. The **lint** program will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his intentions. It seems harsh for **lint** to continue to print messages about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On some systems, characters are signed quantities with a range from −128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar()) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, which are truncated. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **−a** option.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **−h** option is used to supress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message "null effect" from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. The **lint** program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message "constant in conditional context" since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statement

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Finally, when the **−h** option has not been used, **lint** prints messages about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal but is considered to be bad style, usually unnecessary, and frequently a bug.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes - assignment operators and initialization.

The older forms of assignment operators (e.g., **=+**, **=-**, ...) could cause ambiguous expressions, such as:

```
a =−1 ;
```

which could be taken as either

```
a =− 1 ;
```

or

```
a = −1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., **+=**, **-=**, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( −1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

and the compiler must read past *x* in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. The **lint** program tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

The **lint** program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

SYMBOLIC DEBUGGING PROGRAM—"sdb"

GENERAL

This chapter describes the symbolic debugger **sdb**(1) as implemented for C language and Fortran 77 programs on the UNIX operating system. The **sdb** program is useful both for examining "core images" of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and to be displayed in the correct format.

Breakpoints may be placed at selected statements or the program may be single stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines which provided formatted printout of structured data.

USAGE

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **-g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of **shell** commands for debugging a core image is

```
$ cc -g prgm.c -o prgm
$ prgm
Bus error - core dumped
$ sdb prgm
main:25:      x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred which caused a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function *main* at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an * indicating that it awaits a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to *main* and "25", respectively.

In the above example, **sdb** was called with one argument, *prgm*. In general, it takes three arguments on the command line. The first is the name of the executable file which is to be debugged; it defaults to *a.out* when not specified. The second is the name of the core file, defaulting to *core*; and the third is the name of the directory containing the source of the program being debugged. The **sdb** program currently requires all source to reside in a single directory. The default is the working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a function which was not compiled with the **-g** option. In this case, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in *main*. The **sdb** program will print an error message if *main* was not compiled with the **-g** option, but debugging can continue for those routines compiled with the **-g** option. Figure 1 shows a typical example of **sdb** usage.

Printing a Stack Trace

It is often useful to obtain a listing of the function calls which led to the error. This is obtained with the **t** command. For example:

```
*t
sub(x=2,y=3)     [prgm.c:25]
inter(i=16012)    [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c)[prgm.c:15]
```

This indicates that the error occurred within the function *sub* at line 25 in file *prgm.c*. The *sub* function was called with the arguments x=2 and y=3 from *inter* at line 96. The *inter* function was called from *main* at line 15. The *main* function is always called by the **shell** with three arguments often referred to as *argc*, *argv*, and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflag/
```

causes **sdb** to display the value of variable *errflag*. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the form

```
*sub:i/
```

to display variable *i* in function *sub*. F77 users can specify a common block variable in the same manner.

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol * is used to match any sequence of characters of a variable name and ? to match any single character. Consider the following commands

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with *x*, the second prints the values of all two letter variables in function *sub* beginning with *y*, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command

```
**:*/
```

displays the variables for each function on the call stack.

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

b One byte

h Two bytes (half word)

l Four bytes (long word).

The lengths are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A numeric length specifier may be used for the **s** or **a** commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies how many characters should be printed.

There are a number of format specifiers available:

c Character.

d Decimal.

u Decimal unsigned.

o	Octal.
x	Hexadecimal.
f	32-bit single-precision floating point.
g	64-bit double-precision floating point.
s	Assume variable is a string pointer and print characters starting at the address pointed to by the variable until a null is reached.
a	Print characters starting at the variable's address until a null is reached.
p	Pointer to function.
i	Interpret as a machine-language instruction.

For example, the variable *i* can be displayed with

```
*i/x
```

which prints out the value of *i* in hexadecimal.

The **sdb** program also knows about structures, arrays, and pointers so that all of the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Depending on your machine, accessing arrays may be limited to 1-dimensional arrays. Note that as a special case:

```
*psym->/d
```

displays the location pointed to by *psym* in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command

```
*1024/
```

displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both

```
*02000/
```

and

```
*0x400/
```

It is possible to mix numbers and variables so that

```
*1000.x/
```

refers to an element of a structure starting at address 1000, and

```
*1000->x/
```

refers to an element of a structure whose address is at 1000. For commands of the type *1000.x/ and *1000->x/, the **sdb** program uses the structure template of the last structured referenced.

The address of a variable is printed with the =, so

```
*i=
```

displays the address of *i*. Another feature whose usefulness will become apparent later is the command

```
*./
```

which redisplays the last variable typed.

SOURCE FILE DISPLAY AND MANIPULATION

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided which perform context searches within the source files of the program being debugged and to display selected portions of the source files. The commands are similar to those of the UNIX system text editor **ed**(1). Like the editor, **sdb** has a notion of current file and line within the file. The **sdb** program also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are:

p	Prints the current line.
w	Window; prints a window of ten lines around the current line.
z	Prints ten lines starting at the current line. Advances the current line by ten.
control-d	Scrolls; prints the next ten lines and advances the current line by ten. This command is used to cleanly display long segments of the program.

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file but is also used as input by some **sdb** commands.

Changing the Current Source File or Function

The **e** command is used to change the current source file. Either of the forms

```
*e function
*e file.c
```

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current function and file named to be printed.

Changing the Current Line in the Source File

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

There are two commands for searching for instances of regular expressions in source files. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression and the second searches backwards. The trailing **/** and **?** may be omitted from these commands. Regular expression matching is identical to that of **ed**(1).

The **+** and **-** commands may be used to move the current line forwards or backwards by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that

```
*+15z
```

advances the current line by 15 and then prints ten lines.

A CONTROLLED ENVIRONMENT FOR PROGRAM TESTING

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, certain lines in the source program may be specified to be *breakpoints*. The program is then started with a **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. Now, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single stepping. The **sdb** program can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single step through a function which has not been compiled with the **-g** option, execution proceeds until a statement in a function compiled with the **-g** option is reached. It is also possible to have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the **-g** option.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function which contains executable code. The command format is:

```
*12b
*proc:12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source file display commands. The second form sets a breakpoint at line 12 of function *proc*, and the third sets a breakpoint at the first line of *proc*. The last sets a breakpoint

at the current line.

Breakpoints are deleted similarly with the commands

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or **d**, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command, and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command.

```
*12b t;x/
```

causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

Running the Program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the **shell**. The command

```
*r args
```

runs the program with the given arguments as if they had been typed on the **shell** command line. If no arguments are specified, then the arguments from the last execution of the program are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. In all cases after an appropriate message is printed, control returns to **sdb**.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **c** command which continues but passes the signal which stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example:

```
*17 g
```

continues at line 17 of the current function. A use for this command is to avoid executing a section of code which is known to be bad. The user should not attempt to continue execution in a function different than that of the breakpoint.

The **s** command is used to run the program for a single line. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command but does not stop within called functions. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

The **i** command is used to run the program one machine level instruction at a time while ignoring the signal which stopped the program. Its uses are similar to the **s** command. There is also an **I** command which causes the program to execute one machine level instruction at a time, but also passes the signal which stopped the program back to the program.

Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a function which prints structured data in a nice way. There are two ways to call a function:

```
*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...)/m
```

The first simply executes the function. The second is intended for calling functions (it executes the function and prints the value that it returns). The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or values of variables which are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function which formats data from a dump.

MACHINE LANGUAGE DEBUGGING

The **sdb** program has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function *main*, use the command

```
*main:25?
```

The **?** command is identical to the **/** command except that it displays from text space. The default format for printing text space is the **i** format which interprets the machine language instruction. The **control-d** command may be

used to print the next ten instructions.

Absolute addresses may be specified instead of line numbers by appending a : to them so that

```
*0x1024:?
```

displays the contents of address *0x1024* in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line *0x1024* in the current function. It is also possible to set or delete a breakpoint by specifying its absolute address;

```
*0x1024:b
```

sets a breakpoint at address *0x1024*.

Manipulating Registers

The **x** command prints the values of all the registers. Also, individual registers may be named instead of variables by appending a % to their name so that

```
*r3%
```

displays the value of register *r3*.

OTHER COMMANDS

To exit **sdb**, use the **q** command.

The ! command is identical to that in **ed**(1) and is used to have the **shell** execute a command.

It is possible to change the values of variables when the program is stopped at a breakpoint. This is done with the command

*variable!value

which sets the variable to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type float or double, the value can also be a floating-point constant.

Figure 1

```
$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp; {
  int i;
  i = div2(-1);
  printf("-1/2 = %d\n", i);
}
div2(i) {
  int j;
  j = i>>1;
  return(j);
}
$ cc -g testdiv2.c
$ a.out
-1/2 = -1
$ sdb
No core image      # Warning message from sdb
*/^div2            # Search for function "div2"
7: div2(i) {       # It starts on line 7
*z                 # Print the next few lines
7: div2(i) {
8:    int j;
9:    j = i>>1;
10:   return(j);
11: }
*div2:b      # Place breakpoint at beginning of "div2"
div2:9 b     # Sdb echoes proc name and line number
*r           # Run the function
a.out        # Sdb echoes command line executed
Breakpoint at # Executions stops just before line 9
```

```
div2:9:    j = i>>1;
*t         # Print trace of subroutine calls
div2(i=-1)   [testdiv2.c:9]
main(argc=1,argv=0x7fffff50,envp=0x7fffff58)[testdiv2.c:4]
*i/        # Print i
-1
*s         # Single step
div2:10: return(j);   # Execution stops before line 10
*j/        # Print j
-1
*9d        # Delete the breakpoint
*div2(1)/ # Try running "div2" with different arguments
0
*div2(-2)/
-1
*div2(-3)/
-2
*q
$
```

FORTRAN UNIX SYSTEM COMMANDS

A UNIX system Fortran 77 user should be familiar with the following commands:

- **f77** [options] files - This command invokes the UNIX system Fortran 77 compiler

- **ratfor** [options] [files] - This command invokes the Ratfor preprocessor

- **efl** [options] [files] - This command compiles a program written in Extended Fortran Language (EFL) into clean Fortran

- **asa** [files] - This command interprets the output of Fortran programs that utilize ASA carriage control characters

- **fsplit** options files - This command splits the named file(s) into separate files, with one procedure per file.

For more information about the above commands, see the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.*

FORTRAN 77

This chapter describes the compiler and run-time system for Fortran 77 as implemented on the UNIX system. This chapter also describes the interfaces between procedures and the file formats assumed by the I/O system.

USAGE

The command to run the compiler is

f77 options file

The **f77**(1) command is a general purpose command for compiling and loading Fortran and Fortran-related files into an executable module. Ratfor (preprocessor) source files will be translated into Fortran before being presented to the Fortran compiler. The **f77** command invokes the C compiler to translate C source files and invokes the assembler to translate assembler source files. Object files will be link edited. [The **f77**(1) and **cc**(1) commands have slightly different link editing sequences. Fortran programs need two extra libraries (libI77.a, libF77.a) and an additional startup routine.] The following file name suffixes are understood:

.f	Fortran source file
.r	Ratfor source file
.c	C language source file
.s	Assembler source file
.o	Object file.

LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the Fortran 77 American National Standard, this compiler implements a few extensions. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C language procedures or to permit compilation of old (1966 Standard Fortran) programs.

Double Complex Data Type

The data type double complex is added. Each datum is represented by a pair of double-precision real variables. A double complex version of every complex built-in function is provided.

Internal Files

The Fortran 77 American National Standard introduces *internal files* (memory arrays) but restricts their use to formatted sequential I/O statements. This I/O system also permits internal files to be used in direct and unformatted reads and writes.

Implicit Undefined Statement

Fortran has a rule that the type of a variable that does not appear in a type statement is integer if its first letter is *i*, *j*, *k*, *l*, *m* or *n*. Otherwise, it is real. Fortran 77 has an implicit statement for overriding this rule. An additional type statement, *undefined*, is permitted. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **-u** compiler option is equivalent to beginning each procedure with this statement.

Recursion

Procedures may call themselves directly or through a chain of other procedures.

Automatic Storage

Two new keywords recognized are **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

Variable Length Input Lines

The Fortran 77 American National Standard expects input to the compiler to be in a 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the first 72 are ignored.) In order to make it easier to type Fortran programs, this compiler also accepts input in variable length lines. An ampersand (**&**) in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Fortran 77 Standard, there are only 26 letters

— Fortran is a one-case language. Consistent with ordinary system usage, the new compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if the **-U** compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them and to have distinct variables differing only in case. Regardless of the setting of the option, keywords will only be recognized in lowercase.

Include Statement

The statement

```
include "stuff"
```

is replaced by the contents of the file *stuff*. Includes may be nested to a reasonable depth, currently ten.

Binary Initialization Constants

A **logical, real,** or **integer** variable may be initialized in a data statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is *b*, the string is binary, and only zeroes and ones are permitted. If the letter is *o*, the string is octal with digits *zero* through *seven*. If the letter is z or x, the string is hexadecimal with digits *zero* through *nine*, *a* through *f*. Thus, the statements

```
integer a(3)
data a/b'1010',o'12',z'a'/
```

initialize all three elements of a to ten.

Character Strings

For compatibility with C language usage, the following backslash escapes are recognized:

\n	New-line
\t	Tab
\b	Backspace
\f	Form feed
\0	Null
\'	Apostrophe (does not terminate a string)
\"	Quotation mark (does not terminate a string)

\\ \

\x Where **x** is any other character.

Fortran 77 only has one quoting character — the apostrophe ('). This compiler and I/O system recognize both the apostrophe and the double quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an integer word boundary. Each character string constant appearing outside a data statement is followed by a null character to ease communication with C language routines.

Hollerith

Fortran 77 does not have the old Hollerith **(nh)** notation though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data may be used in place of character string constants and may also be used to initialize non character variables in data statements.

Equivalence Statements

This compiler permits single subscripts in **equivalence** statements under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

One-Trip DO Loops

The Fortran 77 American National Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs though they were in violation of the 1966 Standard, the **-onetrip** compiler option causes nonstandard

loops to be generated.

Commas in Formatted Input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
-345,.05e-3,12
```

correctly.

Short Integers

On machines that support half word integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C language type **long int;** half word integers are of C language type **short int.**) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-I2** flag, all small integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **-I2** command flag is in effect). When the **-I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and, xor,** and **not**) and for accessing the command arguments (**getarg** and **iargc**).

The following lists the Fortran intrinsic function library plus some additional functions. These functions are automatically available to the Fortran programmer and require no special invocation of the compiler. The asterisk (*) beside some of the commands indicate they are not part of standard F77. In parenthesis beside each function description listed below is the location for the command in the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines*. These functions are as follows:

abort* Terminate program (ABORT(3F))
abs Absolute value (MAX(3F)
acos Arccosine (ACOS(3F))
aimag Imaginary part of complex argument (AIMAG(3F))
aint Integer part (AINT(3F))
alog Natural logarithm (LOG(3F))
alog10 Common logarithm (ALOG10(3F))
amax0 Maximum value (MAX(3F))
amax1 Maximum value (MAX(3F))
amin0 Minimum value (MIN(3F))
amin1 Minimum value (MIN(3F))
amod Remaindering (MOD(3F))
and* Bitwise boolean (BOOL(3F))
anint Nearest integer (ROUND(3F))
asin Arcsine (ASIN(3F))
atan Arctangent (ATAN(3F))
atan2 Arctangent (ATAN2(3F))
cabs Complex absolute value (ABS(3F))
ccos Complex cosine (COS(3F))
cexp Complex exponential (EXP(3F))
char Explicit type conversion (FTYPE(3F))
clog Complex natural logarithm (LOG(3F))
cmplx Explicit type conversion (FTYPE(3F))
conjg Complex conjugate (CONJG(3F))
cos Cosine (COS(3F))
cosh Hyperbolic cosine (COSH(3F))
csin Complex sine (SIN(3F))
csqrt Complex square root (SQRT(3F))
dabs Absolute value (ABS(3F))

dacos Arccosine (ACOS(3F))
dasin Arcsine (ASIN(3F))
datan Arctangent (ATAN(3F))
datan2 Double precision arctangent (ATAN2(3F))
dble Explicit type conversion (FTYPE(3F))
dcmplx* Explicit type conversion (FTYPE(3F))
dconjg* Complex conjugate (CONJG(3F))
dcos Cosine (DCOS(3F))
dcosh Hyperbolic cosine (COSH(3F))
ddim Positive difference (DIM(3F))
dexp Exponential (EXP(3F))
dim Positive difference (DIM(3F))
dimag* Imaginary part of complex argument ((AIMAG(3F))
dint Integer part (AINT(3F))
dlog Natural logarithm (LOG(3F))
dlog10 Common logarithm (LOG10(3F))
dmax1 Maximum value (MAX(3F))
dmin1 Minimum value (MIN(3F))
dmod Remaindering (DMOD(3F))
dnint Nearest integer (ROUND(3F))
dprod Double precision product (DPROD(3F))
dsign Transfer of sign (SIGN(3F))
dsin Sine (SIN(3F))
dsinh Hyperbolic sine (SINH(3F))
dsqrt Square root (SQRT(3F))
dtan Tangent (TAN(3F))
dtanh Hyperbolic tangent (TANH(3F))
exp Exponential (EXP(3F))
float Explicit type conversion (FTYPE(3F))
getarg* Return command-line argument (GETARG(3F))
getenv* Return environment variable (GETENV(3F))
iabs Absolute value (ABS(3F))
iargc Return number of arguments (IARGC(3F))
ichar Explicit type conversion (FTYPE(3F))
idim Positive difference (DIM(3F))
idint Explicit type conversion (FTYPE(3F))
idnint Nearest integer (ROUND(3F))
ifix Explicit type conversion (FTYPE(3F))
index Return location of substring (INDEX(3F))
int Explicit type conversion (FTYPE(3F))
irand* Random number generator
isign Transfer of sign (SIGN(3F))
len Return location of string (LEN(3F))

lge String comparison (STRCMP(3F))
lgt String comparison (STRCMP(3F))
lle String comparison (STRCMP(3F))
llt String comparison (STRCMP(3F))
log Natural logarithm (LOG(3F))
log10 Common logarithm (LOG10(3F))
lshift* Bitwise boolean (BOOL(3F))
max Maximum value (MAX(3F))
max0 Maximum value (MAX(3F))
max1 Maximum value (MAX(3F))
mclock* Return Fortran time accounting (MCLOCK(3F))
min Minimum value (MIN(3F))
min0 Minimum value (MIN(3F))
min1 Minimum value (MIN(3F))
mod Remaindering (MOD(3F))
nint Nearest integer (BOOL(3F))
not* Bitwise boolean (BOOL(3F))
or* Bitwise boolean (BOOL(3F))
rand* Random number generator (RAND(3F))
real Explicit type conversion (FTYPE(3F))
rshift* Bitwise boolean (BOOL(3F))
sign Transfer of sign (SIGN(3F))
signal* Specify action on receipt of system signal (SIGNAL(3F))
sin Sine (SINE(3F))
sinh Hyperbolic sine (SINH(3F))
sngl Explicit type conversion (FTYPE(3F))
sqrt Square root (SQRT(3F))
srand* Random number generator (RAND(3F))
system* Issue a shell command (SYSTEM(3F))
tan Tangent (TAN(3F))
tanh Hyperbolic tangent (TANH(3F))
xor* Bitwise boolean (BOOL(3F))
zabs* Complex absolute value (ABS(3F)).

For more information on the Fortran intrinsic function commands, see the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines.*

VIOLATIONS OF THE STANDARD

The following paragraphs describe only three known ways in which the UNIX system implementation of Fortran 77 violates the new American National Standard.

Double Precision Alignment

The Fortran 77 American National Standard permits **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines require that double precision quantities be on double word boundaries; other machines run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double-precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use two separate operations. The first operation would be to move the upper and lower halves into the halves of an aligned temporary. The second would be to load that double-precision temporary. The reverse would be needed to store a result. All double-precision real and complex quantities are required to fall on even word boundaries on machines with corresponding hardware requirements and to issue a diagnostic if the source code demands a violation of the rule.

Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The implementation uses "seeks"; so if the unit is not one which allows seeks (such as a terminal) the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

INTERPROCEDURE INTERFACE

To be able to write C language procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C language procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

Data Representations

The following is a table of corresponding Fortran and C language declarations:

Fortran	C Language
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory.

Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C language function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C language routine with an additional initial argument that points to the place where the return value is to be stored. Thus, the following:

```
complex function f( . . . )
```

is equivalent to

```
struct { float r, i; } temp;
f_(&temp, . . .)
 . . .
```

A character-valued function is equivalent to a C language routine with two extra initial arguments - a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
char result[ ];
long int length;
g_(result, length, . . .)
 . . .
```

and could be invoked in C language by

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
. . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
. . .
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C language arrays are stored in row-major order.

FILE FORMATS

Structure of Fortran Files

Fortran requires four kinds of external files: *sequential formatted* and *unformatted*, and *direct formatted* and *unformatted*. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records." When a direct file is opened in a Fortran program, the record length of the records must be given; and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records but are treated as byte-addressable byte strings; i.e., as ordinary files on the UNIX system. (A read or write request on such a file keeps consuming bytes until satisfied rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Fortran 77 American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought was written will be treated as more than one record when being read or backspaced over.

Preconnected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist nor will they be created unless their units are used without first executing an **open.** The default connection is for sequential formatted I/O.

The Fortran 77 Standard does not specify where a file which has been explicitly **opened** for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end. A **write** will append to the file and a **read** will result in an "end of file" indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

RATFOR

GENERAL

This chapter describes the **ratfor**(1) preprocessor. It is assumed that the user is familiar with the current implementation of Fortran 77 on the UNIX system.

The Ratfor language allows users to write Fortran programs in a fashion similar to C language. The Ratfor program is implemented as a preprocessor that translates this "simplified" language into Fortran. The facilities provided by Ratfor are:

- Statement grouping
- if–else and switch for decision making
- while, for, do, and repeat–until for looping
- break and next for controlling loop exits
- Free form input such as multiple statements/lines and automatic continuation
- Simple comment convention
- Translation of >, >=, etc., into .gt., .ge., etc.
- return statement for functions
- define statement for symbolic parameters
- include statement for including source files.

USAGE

The Ratfor program takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **-6x**, which uses x as a continuation character in column 6 (the UNIX system uses *&* in column 1), **-h**, which causes quoted strings to be turned into *n*H constructs and **-C**, which causes Ratfor comments to be copied into the generated Fortran.

STATEMENT GROUPING

The Ratfor language provides a statement grouping facility. A group of statements can be treated as a unit by enclosing them in the braces { and }. For example, the Ratfor code

```
if (x > 100)
   { call error("x>100"); err = 1; return }
```

will be translated by the Ratfor preprocessor into **Fortran** equivalent to

```
      if (x .le. 100) goto 10
          call error(5hx>100)
          err = 1
          return
10    ...
```

which should simplify programming effort. By using { and }, a group of statements can be used instead of a single statement.

Also note in the previous Ratfor example that the character > was used instead of **.GT.** in the **if** statement. The Ratfor preprocessor translates this C language type operator to the appropriate Fortran operator. More on relationship operators later.

In addition, many Fortran compilers permit character strings in quotes (like "*x>100*"). But others, like ANSI Fortran 66, do not. Ratfor converts it into the right number of *H*s.

The Ratfor language is free form. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The previous example could also be written as

```
if (x > 100) {
    call error("x>100")
    err = 1
    return
}
```

which shows grouped statements spread over several lines. In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement, no braces are needed.

THE "if-else" CONSTRUCTION

The Ratfor language provides an **else** statement. The syntax of the **if-else** construction is:

if (*legal Fortran condition*)
    ratfor statement
else
    ratfor statement

where the **else** part is optional. The legal Fortran condition is anything that can legally go into a Fortran Logical **IF** statement. The Ratfor preprocessor does not check this clause since it does not know enough Fortran to know what is permitted. The "ratfor" statement is any Ratfor or Fortran statement or any collection of them in braces. For example:

```
if (a <= b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

is a valid Ratfor **if-else** construction. This writes out the smaller of a and b, then the larger, and sets sw appropriately.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed.

Nested "if" Statements

The statement that follows an **if** or an **else** can be any Ratfor statement including another **if** or **else** statement. In general, the structure

```
if (condition) action
else if (condition) action
else action
```

provides a way to write a multibranch in Ratfor. (The Ratfor language also provides a **switch** statement which could be used instead, under certain conditions.) The last **else** handles the "default" condition. If there is no default action, this final **else** can be omitted. Thus, only the actions associated with the valid condition are performed. For example:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

will ensure that x is not less than 0 and not greater than 100.

Nested **if** and **else** statements could result in ambiguous code. In Ratfor when there are more **if** statements than **else** statements, **else** statements are associated with the closest previous **if** statement that currently does not have an associated **else** statement. For example:

```
if (x > 0)
if (y > 0)
write(6,1) x, y
else
write(6,2) y
```

is interpreted by the Ratfor preprocessor as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

in which the braces are assumed. If the other association is desired it must be written as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

with the braces specified.

THE "switch" STATEMENT

The **switch** statement provides a way to express multiway branches which branch on the value of some *integer*-valued expression. The syntax is

```
switch (expression) {
    case expr1 :
    statements
    case expr2, expr3 :
    statements
    ...
    default:
    statements
}
```

where each **case** is followed by an integer expression (or several integer expressions separated by commas). The **switch** *expression* is compared to each **case** *expr* until a match is found. Then the *statements* following that **case** are executed. If no **cases** match *expression*, then the *statements* following **default** are executed. The **default** section of a **switch** is optional.

When the *statements* associated with a **case** are executed, the entire **switch** is exited immediately. This is different from C language.

THE "do" STATEMENT

The **do** statement in Ratfor is quite similar to the **DO** statement in Fortran except that it uses no statement number (braces are used to mark the end of the **do** instead of a statement number). The syntax of the **ratfor do** statement is

```
do legal-Fortran-DO-text {
  ratfor statements
}
```

The *legal-Fortran-DO-text* must be something that can legally be used in a Fortran **DO** statement. Thus if a local version of Fortran allows **DO** limits to be expressions (which is not currently permitted in ANSI Fortran 66), they can be used in a **ratfor do** statement. The *ratfor statements* are enclosed in braces; but as with the **if**, a single statement need not have braces around it. For example, the following code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

and the code

```
do i = 1, n
  do j = 1, n
    m(i, j) = 0
```

sets the entire array *m* to zero.

THE "break" AND "next" STATEMENTS

The Ratfor **break** and **next** statements provide a means for leaving a loop early and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect, it is a branch to the statement *after* the **do**. The **next** is a branch to the bottom of the loop, so it causes the next iteration to be done.

For example, this code skips over negative values in an array

```
do i = 1, n {
  if (x(i) < 0.0)
    next
  process positive element
}
```

The **break** and **next** statements will also work in the other Ratfor looping constructions and will be discussed with each looping construction.

The **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

```
break 2
```

exits from two levels of enclosing loops, and

```
break 1
```

is equivalent to **break**. The

```
next 2
```

iterates the second enclosing loop.

THE "while" STATEMENT

The Ratfor language provides a **while** statement. The syntax of the **while** statement is

while (*legal-Fortran-condition*)
  ratfor statement

As with the **if**, legal–Fortran–condition is something that can go into a Fortran Logical **IF**, and ratfor statement is a single statement which may be multiple statements enclosed in braces.

For example, suppose nextch is a function which returns the next input character both as a function value and in its argument. Then a **while** loop to find the first nonblank character could be

```
while (nextch(ich) == iblank)
    ;
```

where a semicolon by itself is a null statement (which is necessary here to mark the end of the **while**). If the semicolon were not present, the **while** would control the next statement. When the loop is exited, ich contains the first nonblank.

THE "for" STATEMENT

The **for** statement is another Ratfor loop. A **for** statement allows explicit initialization and increment steps as part of the statement.

The syntax of the **for** statement is

for (*init* ; *condition* ; *increment*)
  ratfor statement

where *init* is any single Fortran statement which is executed once before the loop begins. The increment is any single Fortran statement that is executed at the end of each pass through the loop before the test. The *condition* is again anything that is legal in a Fortran Logical **IF**. Any of init, condition, and increment may be omitted although the semicolons must always be present. A nonexistent condition is treated as always true, so

```
for (;;)
```

is an infinite loop.

For example, a Fortran **DO** loop could be written as

```
for (i = 1; i <= n; i = i + 1) ...
```

which is equivalent to

```
i = 1
while (i <= n) {
    ...
    i = i + 1
}
```

The initialization and increment of *i* have been moved into the **for** statement.

The **for**, **do**, and **while** versions have the advantage that they will be done zero times if *n* is less than 1. In addition, the **break** and **next** statements work in a **for** loop.

The *increment* in a **for** need not be an arithmetic progression. The program

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

steps through a list (stored in an integer array *ptr*) until a zero pointer is found while adding up elements from a parallel array of values. Notice that the code also works correctly if the list is empty.

THE "repeat-until" STATEMENT

There are times when a test needs to be performed at the bottom of a loop after one pass through. This facility is provided by the **repeat-until** statement. The syntax for the **repeat-until** statement is

repeat
  ratfor statement
until (*legal-Fortran-condition*)

where ratfor–statement is done once, then the *condition* is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is an infinite loop. A **repeat-until** loop can be exited by the use of a **stop**, **return**, or **break** statement or an implicit stop such as running out of input with a **READ** statement.

As stated before, a **break** statement causes an immediate exit from the enclosing **repeat-until** loop. A **next** statement will cause a skip to the bottom of a **repeat-until** loop (i.e., to the **until** part).

THE "return" STATEMENT

The standard Fortran mechanism for returning a value from a routine uses the name of the routine as a variable. This variable can be assigned a value. The last value stored in it is the value returned by the function. For example, in a Fortran routine named *equal*, the statements

```
equal = 0
return
```

cause *equal* to return zero.

The Ratfor language provides a **return** statement similar to the C language **return** statement. In order to return a value from any routine, the **return** statement has the syntax

return (*expression*)

where *expression* is the value to be returned.

If there is no parenthesized expression after **return**, no value is returned.

THE "define" STATEMENT

The Ratfor language provides a **define** statement similar to the C language version. Any string of alphanumeric characters can be defined as a name. Whenever that name occurs in the input (delimited by nonalphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off.) A defined name can be arbitrarily long and must begin with a letter.

Usually the **define** statement is used for symbolic parameters. The syntax of the **define** statement is

define name value

where *name* is a symbolic name that represents the quantity of *value*. For example:

```
define ROWS 100
define CLOS 50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

causes the preprocessor to replace the name *ROWS* with the value *100* and the name *COLS* with the value *50*. Alternately, definitions may be written as

```
define(ROWS, 100)
```

in which case the defining text is everything after the comma up to the right parenthesis. This allows multiple-line definitions.

THE "include" STATEMENT

The Ratfor language provides an **include** statement similar to the **#include** <...> statement in C language. The syntax for this statement is

include *file*

which inserts the contents of the named file into the Ratfor input file in place of the **include** statement. The standard usage is to place **COMMON** blocks on a file and use the **include** statement to include the common code whenever needed.

FREE-FORM INPUT

In Ratfor, statements can be placed anywhere on a line. Long statements are continued automatically as are long conditions in **if**, **for**, and **until** statements. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
= + - * , | & ( _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and placed in columns 1 through 5 upon output. Thus:

```
write(6, 100); 100 format("hello")
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

TRANSLATIONS

When the **-h** option is chosen, text enclosed in matching single or double quotes is converted to *nH...* but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character; i.e., the next character is taken literally. This provides a way to get quotes and the backslash itself into quoted strings. For example:

```
"\'"
```

is a string containing a backslash and an apostrophe. (This is not the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character % is left absolutely unaltered except for stripping off the % and moving the line one position to the left. This is useful for inserting control cards and other things that should not be preprocessed (like an existing Fortran program). Use % only for ordinary statements not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made (except within single or double quotes or on a line beginning with a %):

```
==   .eq.
!=   .ne.
>    .gt.
>=   .ge.
<    .lt.
<=   .le.
&    .and.
|    .or.
!    .not.
```

In addition, the following translations are provided for input devices with restricted character sets:

```
[   {
]   }
$(  {
$)  }
```

WARNINGS

The Ratfor preprocessor catches certain syntax errors (such as missing braces), **else** statements without **if** statements, and most errors involving missing parentheses in statements.

All other errors are reported by the Fortran compiler. Unfortunately, the Fortran compiler prints messages in terms of generated Fortran code and not in terms of the Ratfor code. This makes it difficult to locate Ratfor statements that contain errors.

The keywords are deserved. Using **if**, **else**, **while**, etc., as variable names will cause considerable problems. Likewise, spaces within keywords and use of the Arithmetic **IF** will cause problems.

The Fortran *n*H convention is not recognized by Ratfor. Use quotes instead.

EXAMPLE OF RATFOR CONVERSION

As an example of how to use the Ratfor program, the following program **prog.r** (where the **.r** indicates a Ratfor source program), is written in the Ratfor language:

```
  ICNT=0
10 WRITE(6,31)
31 FORMAT("INPUT FIRST NUMBER")
  READ(5,32) A
32 FORMAT(F10.2)
  WRITE(6,33)
33 FORMAT("INPUT SECOND NUMBER")
  READ(5,34) B
34 FORMAT(F10.2)
  IF(A<B)
    WRITE(6,36) A,B
  ELSE WRITE(6,37)A,B
36 FORMAT(F10.2," < ",F10.2)
37 FORMAT(F10.2," >= ",F10.2)
  ICNT=ICNT+1
  IF(ICNT.EQ.5)
    GOTO 100
  GOTO 10
100 END
```

The command

```
ratfor prog.r > prog.f
```

causes the Fortran translation program **prog.f** to be produced. (The Ratfor program **prog.r** remains intact.) The Fortran program **prog.f** follows:

```
      icnt=0
10    write(6,31)
31    format("INPUT FIRST NUMBER")
      read(5,32) a
32    format(f10.2)
      write(6,33)
33    format("INPUT SECOND NUMBER")
      read(5,34) b
34    format(f10.2)
      if(.not.(a.lt.b))goto 23000
      write(6,36) a,b
      goto 23001
23000 continue
      write(6,37)a,b
23001 continue
36    format(f10.2," < ",f10.2)
37    format(f10.2," >= ",f10.2)
      icnt=icnt+1
      if(.not.(icnt.eq.5))goto 23002
      goto 100
23002 continue
      goto 10
100   end
```

The Fortran program **prog.f** is compiled using the command

```
f77 prog.f
```

An object program file **prog.o** and a final output file **a.out** are produced. Since the output file **a.out** is an executable file, the command

```
a.out
```

causes the program to run.

The Ratfor program **prog.r** can also be translated and compiled with the single command

```
f77 prog.r
```

where the **.r** indicates a Ratfor source program. An object file **prog.o** and a final output file **a.out** are produced.

THE PROGRAMMING LANGUAGE EFL

INTRODUCTION

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

The name EFL originally stood for "Extended Fortran Language." The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions.

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

[[*item* **]]**

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

LEXICAL FORM

Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m**
	n o p q r s t u v w x y z
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + − * /
	= < > & ~ | $

Letter case (upper or lower) is ignored except within strings, so "**a**" and "**A**" are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ("!") may be used in place of a tilde ("~"). Square brackets ("[" and "]") may be used in place of braces ("{" and "}").

Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Include**s may be nested at least ten deep.

Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of a line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

'hello there'
"ain't misbehavin'"

Integer Constants

An integer constant is a sequence of one or more digits.

0
57
123456

Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }

comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+  -  *  /  **
<  <=  >  >=  ==  ~=
&&  ||  &  |
+=  -=  /=  **=
&&=  ||=  &=  |=
->  .  $
```

A dot (".") is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see "ATAVISMS") in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.*, **.lt.**).

Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

```
n += 1
```

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

PROGRAM FORM

Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in "PROCEDURES."

Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See "Blocks" under "EXECUTABLE STATEMENTS.") An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *K* is defined throughout that block and in all deeper nested levels in which that

name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#   block 0
procedure george
real x
x = 2
. . .
if(x > 2)
            {           # new block
            integer x   # a different variable
            do x = 1,7
                        write(,x)
            . . .
            }           # end of block
end         # end of procedure, return to block 0
```

Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in "COMPILER OPTIONS". The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statement and finishes with an **end** statement; these are discussed in "PROCEDURES". Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
        read(, x)
        if(x < 3) goto error
        . . .
error:  fatal("bad input")
```

DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

Basic Types

The basic types are

logical
integer
field(*m*:*n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values *true* and *false*. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A "real" quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
−94
+6
0

A long real ("double precision") constant is a floating point constant containing an exponent field that begins with the letter **d.** A real ("single precision") constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
−.4
7.9e−6 ($= 7.9\times10^{-6}$)
14e9 ($= 1.4\times10^{10}$)

The following are valid **long real** constants

7.9d−6 ($= 7.9\times10^{-6}$)
5d3

A character constant is a quoted string.

Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression*)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in "Unary Operators" and "Binary Operators" under "EXPRESSIONS".

```
-> .
**
* /    unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **=   &= |= &&= ||=
```

Examples of expressions are

```
a<b & & b<c
-(a + sin(x)) / (5+cos(x))**2
```

Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

Constants

Constants are described in "Constants" under "DATA TYPES AND VARIABLES".

Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may appear only as procedure arguments and in input/output lists.

Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6, -3, 4)
```

Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename ( )
procedurename (*expression*)
procedurename (*expression-1*, ..., *expression-n*)

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see "Known Functions" under "PROCEDURES"), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied

in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See "PROCEDURES".

Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See "Input/Output Statements" under "EXECUTABLE STATEMENTS".

Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = 5+3i

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

Arithmetic

Unary + has no effect. A unary − yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator −− subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

Logical

The only logical unary operator is complement (~). This operator is defined by the equations

~ true = false
~ false = true

Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8−2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

Type of A	Type of B				
	i	r	l r	c	l c
i	i	r	l r	c	l c
r	r	r	l r	c	l c
l r	l r	l r	l r	l c	l c
c	c	c	l c	c	l c
l c	l c	l c	l c	l c	l c

i = integer
r = real
l r = long real
c = complex
l c = long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

Logical

The two binary logical operations in EFL, **and** and **or,** are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise, the expression has the value of **b.** The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise, the expression has the value of **b.** The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator		Meaning
<	$<$	less than
<=	$\leqslant$	less than or equal to
==	$=$	equal to
~=	$\neq$	not equal to
>	$>$	greater than
>=	$\geqslant$	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op= b* is equivalent to *a = a op b.* (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds

2 to n. The location of the left side is evaluated only once.

Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside **−>** *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) −> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place.**

Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two forms:

attributes variable-list
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

Attributes

Basic Types

The following are basic types in declarations

logical
integer
field(*m*:*n*)
character(*k*)
real
complex

In the above, the quantities *k*, *m*, and *n* denote integer constant expressions with the properties $k > 0$ and $n > m$.

Arrays

The dimensionality may be declared by an **array** attribute

array(b_1,...,b_n)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to *n*, the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper*−*lower*+1 is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

array(5)
array(5, 1:5, −3:0)
array(5, *)
array(0:m−1, m)

Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
        {
        integer a, b
        real x(5)
        }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'*s* and a character string.

Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables

in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external [[*name*]]

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial [[*var* = *val*]]

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements, otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

Expression Statements

Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see "Input/Output Statements" under "EXECUTABLE STATEMENTS") resemble procedure invocations but do not yield a value. If an error occurs the program stops.

Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i  # this variable is unknown
        # outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression*) □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

If-Else

A more general statement is of the form

if (*logical-expression*) □ *statement-1* □
else □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise, *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a

completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-**else**d **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression* **)** □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case [[*constant*]] **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case**

or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not "fall through" to the next case.

Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest **(while)** form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

While Statement

This construct has the form

while (*logical-expression*) □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* , □ *logical-expression* ,
□ *iteration-statement*) □ *body-statement*

Except for the behavior of the **next** statement (see "Branch Statement" under

"EXECUTABLE STATEMENTS"), this construct is equivalent to

initial-statement
while (*logical-expression*)
{
body-statement
iteration-statement
}

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
        n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
        ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

Repeat ... Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until** (*logical-expression*)

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise, control returns to the *statement.* Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until.** In practice, this appears to be the least frequently used looping construct.

Do Loop

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable* = *expression-1, expression-2, expression-3*
statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable=expression-1, variable<=t2, variable+=t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```

Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg("ouch")
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
    {
    do a computation
    if ( finished )
        break
    }
```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while,** the *iteration-statement* of a **for,** the body of a **repeat,** the test of a **repeat...until,** or the increment of a **do.** Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

Input/Output Units

Each I/O statement refers to a "unit," identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit*. These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records*. These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file*.

Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* **,** *binary-output-list* **)**
readbin(*unit* **,** *binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* **,** *formatted-output-list* **)**
read(*unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

Formats

The following are permissible *format-specifiers*. The quantities *w*, *d*, and *k* must be integer constant expressions.

i(*w*)	integer with *w* digits
f(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(*w*)	logical field of width *w* characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	character string of width equal to the length of the datum
c(*w*)	character string of width *w*
s(*k*)	skip *k* lines
x(*k*)	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted

in column-major order, the same order used for array initializations.

Manipulation Statements

The three input/output statements

backspace(*unit*)
rewind(*unit*)
endfile(*unit*)

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

Procedures Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* **( )**
attributes **procedure** *procedurename* **([[** *name* **]])**

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each

call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

End Statement

Each procedure terminates with a statement

end

Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real.** Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, −3.20)
max(i, z)
```

Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

Elementary Functions

The following generic functions take arguments of **real, long real,** or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base e) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x)=\tan^{-1}x$
atan2	$atan2(x,y)=\tan^{-1}\frac{x}{y}$

Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**$(x,y) = sgn(y)|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ("%") is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

Call Statement

A subroutine call may be preceded by the keyword **call.**

```
call joe
call work(17)
```

Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list*) *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a-h, o-z) real
implicit (i-n) integer
```

Computed Goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto ([[*label*]]), *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

Goto Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

go to xyz

Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see "COMPILER OPTIONS") which forces the compiler to recognize the forms in the second column below:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
~=	**.ne.**
&	**.and.**
|	**.or.**
& &	**.andand.**
||	**.oror.**
~	**.not.**
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt, le,** etc. The readable forms in the left column are always recognized.

Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

Function Values

The preferred way to return a value from a function in EFL is the **return**(*value*) construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

Equivalence

A statement of the form

equivalence v_1, v_2, ..., v_n

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	*Argument Type*	*Result Type*
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option [[*opt*]]

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (**&**) in the first column of the continued lines instead.

Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	*Type*
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	*Size Option*	*Alignment Option*
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
                write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program

keeps reading and writing until the input is exhausted.

Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
        {
        c(i,j) = 0
        do k = 1,n
                c(i,j) += a(i,k) * b(k,j)
        }
end
```

Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST             0
define NOTFOUND     -1

integer procedure val(list, first, x)

#   list is an array of structures.
#   Each structure contains a thread index value,
#   an x, and a y value.
struct
            {
            integer nextindex
            integer x, y
            } list(*)
integer first, p, arg

for(p = first , p~=LAST & & list(p).x<=x ,
   p = list(p).nextindex)
            if(list(p).x == x)
                        return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **& &** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it

is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
        print its value
otherwise
        print a left parenthesis
        print the left node
        print the operator
        print the right node
        print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)       # print an expression tree

integer first               # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
        {
        character(1) op
        integer leftp, rightp
        real val
        } tree(100)         # array of structures

struct
        {
        integer nextstate
        integer nodep
        } stackframe(100)

define NODE     tree(currentnode)
define STACK    stackframe(stackdepth)

#  nextstate values
define DOWN     1
define LEFT     2
define RIGHT    3
```

```
#  initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

while( stackdepth > 0 )
        {
        currentnode = STACK.nodep
        select(STACK.nextstate)
                {
                case DOWN:
                        if(NODE.op == " ")  # a leaf
                                {
                                outval( NODE.val )
                                stackdepth -= 1
                                }
                        else    { # a binary operator node
                                outch( "(" )
                                STACK.nextstate = LEFT
                                stackdepth += 1
                                STACK.nextstate = DOWN
                                STACK.nodep = NODE.leftp
                                }

                case LEFT:
                        outch( NODE.op )
                        STACK.nextstate = RIGHT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.rightp

                case RIGHT:
                        outch( ")" )
                        stackdepth -= 1
                }
        }
end
```

PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

Character String Copying

The subroutine **eflasc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a.**

Character String Comparisons

The function **eflcmc** is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb.** It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

DIFFERENCES BETWEEN RATFOR AND EFL

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the "ATAVISMS" are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

COMPILER

Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers.

Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (See "EXAMPLES".)

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
         do  2 j = 1, p
            c(i, j) = 0
            do  1 k = 1, n
               c(i, j) = c(i, j) +a(i, k)*b(k, j)
1              continue
2           continue
3        continue
      end
```

The following is the procedure for the tree walk:

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
   1  if (stapth .le. 0) goto 9
         curode = staame(2, stapth)
         goto 7
   2        if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(tree1(4, curode))
c a leaf
               stapth = stapth-1
               goto 4
   3           call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
   4        goto 8
   5        call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto 8
   6        call outch(1h))
            stapth = stapth-1
            goto 8
   7        if (staame(1, stapth) .eq. 3) goto 6
            if (staame(1, stapth) .eq. 2) goto 5
```

```
            if (staame(1, stapth) .eq. 1) goto  2
8       continue
        goto  1
9   continue
    end
```

CONSTRAINTS ON EFL

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

A PROGRAM FOR MAINTAINING COMPUTER PROGRAMS (make)

GENERAL

In a programming project, a common practice is to divide large programs into smaller pieces that are more manageable. The pieces may require several different treatments such as being processed by a macro processor or sophisticated program generators (e.g., **Yacc** or **Lex**). The project continues to become more complex as the output of these generators are compiled with special options and with certain definitions and declarations. A sequence of code transformations develops which is difficult to remember. The resulting code may need further transformation by loading the code with certain libraries under control of special options. Related maintenance activities also complicate the process further by running test scripts and installing validated modules. Another activity that complicates program development is a long editing session. A programmer may lose track of the files changed and the object modules still valid especially when a change to a declaration can make a dozen other files obsolete. The programmer must also remember to compile a routine that has been changed or that uses changed declarations.

The "make" is a software tool that maintains, updates, and regenerates groups of computer programs.

A programmer can easily forget

- Files that are dependent upon other files.
- Files that were modified recently.
- Files that need to be reprocessed or recompiled after a change in the source.
- The exact sequence of operations needed to make an exercise a new version of the program.

The many activities of program development and maintenance are made simpler by the **make** program.

The **make** program provides a method for maintaining up-to-date versions of programs that result from many operations on a number of files. The **make** program can keep track of the sequence of commands that create certain files and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of a program, the **make** command creates the proper files simply, correctly, and with a minimum amount of effort. The **make** program also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

The basic operation of **make** is to

- Find the name of the needed target file in the description.
- Ensure that all of the files on which it depends exit and are up to date.
- Create the target file if it has not been modified since its generators were modified.

The descriptor file really defines the graph of dependencies. The **make** program determines the necessary work by performing a depth-first search of the graph of dependencies.

If the information on interfile dependencies and command sequences is stored in a file, the simple command

make

is frequently sufficient to update the interesting files regardless of the number edited since the last **make**. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the **make** command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think − edit − **make** − test . . .

The **make** program is most useful for medium-sized programming projects. The **make** program does not solve the problems of maintaining multiple source versions or of describing huge programs.

As an example of the use of **make**, the description file used to maintain the **make** command is given. The code for **make** is spread over a number of C language source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

p = lp
FILES = Makefile version.c defs main.c doname.c misc.c
        files.c dosys.c gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS): defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print: $(FILES)         # print recently changed files
        pr $? | $P
        touch print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap
```

```
lint : dosys.c doname.c files.c main.c misc.c version.c
       gram.c
       $(LINT) dosys.c doname.c files.c main.c misc.c
       version.c gram.c

arch:
       ar uv /sys/source/s2/make.a $(FILES)
```

The **make** program usually prints out each command before issuing it.

The following output results from typing the simple command **make** in a directory containing only the source and description files:

```
cc -O -c version.c
cc -O -c main.c
cc -O -c doname.c
cc -O -c misc.c
cc -O -c files.c
cc -O -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -O -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o
   gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, **make** found them using its suffix rules and issued the needed commands. The string of digits results from the **size** **make** command. The printing of the command line itself was suppressed by an @ sign. The @ sign on the **size** command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files changed since the last **make print** command. A zero-length file *print* is maintained to keep track of the time of the printing. The **$?** macro in the command line then picks up only the names of the files changed since *print* was touched.

The printed output can be sent to a different printer or to a file by changing the definition of the **P** macro as follows:

```
make print "P= cat >zap"
```

BASIC FEATURES

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up to date. The target file is created if it has not been modified since the dependents were modified. The **make** program does a depth-first search of the graph of dependencies. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, consider a simple example in which a program named *prog* is made by compiling and loading three C language files *x.c*, *y.c*, and *z.c* with the **IS** library. By convention, the output of the C language compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
       cc x.o y.o z.o -lS -o prog

x.o y.o : defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

The **make** program operates using the following three sources of information:

- A user-supplied description file
- File names and "last-modified" times from the file system
- Built-in rules to bridge some of the gaps.

In the example, the first line states that *prog* depends on three "*.o*" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line states that *x.o* and *y.o* depend on the file *defs*. From the file system, **make** discovers that there are three "*.c*" files corresponding to the needed "*.o*" files and uses built-in information on how to generate an object from a source file (i.e., issue a "cc −c" command).

By not taking advantage of **make**'s innate knowledge, the following longer descriptive file results.

```
prog : x.o y.o z.o
       cc x.o y.o z.o −lS −o prog
x.o : x.c defs
       cc −c x.c
y.o : y.c defs
       cc −c y.c
z.o : z.c
       cc −c z.c
```

If none of the source or object files have changed since the last time *prog* was made, all of the files are current, and the command

```
make
```

announces this fact and stops. If, however, the *defs* file has been edited, *x.c* and *y.c* (but not *z.c*) is recompiled; and then *prog* is created from the new "*.o*" files. If only the file *y.c* had changed, only it is recompiled; but it is still necessary to reload *prog*. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions. A method, often useful to programmers, is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files. In other cases, one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

The **make** program has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. A $$ is a dollar sign.

The **$***, **$@**, **$?**, and **$<** are four special macros which change values during the execution of the command. (These four macros are described in the part "DESCRIPTION FILES AND SUBSTITUTIONS".) The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

The **make** command loads the three object files with the lS library. The command

```
make  "LIBES= -ll -lS"
```

loads them with both the Lex (-ll) and the standard (-lS) libraries since macro definitions on the command line override definitions in the description. Remember to quote arguments with embedded blanks in UNIX software commands.

DESCRIPTION FILES AND SUBSTITUTIONS

A description file contains the following information:

- macro definitions
- dependency information
- executable commands.

The comment convention is that a sharp (#) and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp (#) are totally ignored. If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following

are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as the macro's value.

Macro definitions may also appear on the **make** command line while other lines give information about target files. The general form of an entry is

```
target1 [target2 . .] :[:] [dependent1 . .] [; commands] [# . .]
[(tab) commands] [# . . .]
. . .
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as "*" and "?" are expanded. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp (#) except when the sharp is in quotes or not including a new line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the usual single-colon case, a command sequence may be associated with at most one dependency line. If the target is out of date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default creation rule may be invoked. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode or if the command line begins with an @ sign. **Make** normally stops if any command signals an error by returning a nonzero error code. Errors are

ignored if the **−i** flags have been specified on the **make** command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX software commands return meaningless status. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The **$?** macro is set to the string of names that were found to be younger than the target. The **$?** macro is evaluated when explicit rules from the *makefile* are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; and the **$*** macro is the prefix shared by the current and the dependent file names. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, **make** prints a message and stops.

COMMAND USAGE

The **make** command takes macro definitions, flags, description file names, and target file names as arguments in the form:

make [flags] [macro definitions] [targets]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the flag arguments are examined. The permissible flags are as follows:

−i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE"

appears in the description file.

−s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.

−r Do not use the built-in rules.

−n No execute mode. Print commands, but do not execute them. Even lines beginning with an "@" sign are printed.

−t Touch the target files (causing them to be up to date) rather than issue the usual commands.

−q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.

−p Print out the complete set of macro definitions and target descriptions.

−d Debug mode. Print out detailed information on files and times examined.

−f Description file name. The next argument is assumed to be the name of a description file. A file name of "−" denotes the standard input. If there are no "*−f*" arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present.

Finally, the remaining arguments are assumed to be the names of targets to be made, and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

SUFFIXES AND TRANSFORMATION RULES

The **make** program does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the **−r** flag is used, the internal table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES". The **make** program searches for a file with any of the suffixes on the list. If such a file exists and if there is a transformation rule for that combination, **make** transforms a file with one suffix into a file with another suffix. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a *.r* file to a *.o* file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **$*** is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro **$<** is the name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for ".SUFFIXES" in his own description file. The dependents are added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed. The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC = yacc
YACCR = yacc -r
YACCE = yacc -e
YFLAGS =
LEX = lex
LFLAGS =
CC = cc
AS = as -
CFLAGS =
RC = ec
RFLAGS =
EC = ec
EFLAGS =
FFlags =
.c.o :
        $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
        $(AS) -o $@ $<
.y.o :
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c :
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

IMPLICIT RULES

The **make** program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	Object file
.c	C source file

.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar.

If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

If the file *x.o* were needed and there were an *x.c* in the description or directory, the *x.o* file would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, **make** would discard the intermediate C language file and use the direct link.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros **AS**, **CC**, **RC**, **EC**, **YACC**, **YACCR**, **YACCE**, and **LEX**. The command

```
make CC=newcc
```

will cause the **newcc** command to be used instead of the usual C language compiler. The macros **CFLAGS**, **RFLAGS**, **EFLAGS**, **YFLAGS**, and **LFLAGS** may be set to cause these commands to be issued with optional flags. Thus

```
make "CFLAGS=-O"
```

causes the optimizing C language compiler to be used.

SUGGESTIONS AND WARNINGS

The most common difficulties arise from **make**'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on **defs**; the source file *x.c* does not. If **defs** is changed, nothing is done to the file *x.c* while file *x.o* must be recreated.

To discover what **make** would do, the **−n** option is very useful. The command

make −n

orders **make** to print out the commands which **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a new definition to an include file), the **−t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

make −ts

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag (**−d**) causes **make** to print out a very detailed description of what it is doing including the file times. The output is verbose and recommended only as a last resort.

AUGMENTED VERSION OF make

GENERAL

This section describes an augmented version of the **make** command of the UNIX operating system. The augmented version is upward compatible with the old version. This section describes and gives examples of only the additional features. Further possible developments for **make** are also discussed. Some justification will be given for the chosen implementation, and examples will demonstrate the additional features.

The **make** command is an excellent program administrative tool used extensively in at least one project for over 2 years. However, **make** had the following shortcomings:

- Handling of libraries was tedious.
- Handling of the Source Code Control System (SCCS) file name format was difficult or impossible.
- Environment variables were completely ignored by **make**.
- The general lack of ability to maintain files in a remote directory.

These shortcomings hindered large scale use of **make** as a program support tool.

The augmented version of **make** is modified to handle the above problems. The additional features are within the original syntactic framework of **make** and few if any new syntactical entities are introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of **make**.

The following paragraphs describe with examples the additional features of the **make** program. In general, the examples are taken from existing *makefiles*. Also, the illustrations are examples of working *makefiles*.

THE ENVIRONMENT VARIABLES

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A new macro, **MAKEFLAGS**, is maintained by **make**. The new macro is defined as the collection of all input flag arguments into a string (without minus signs). The new macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the *makefile* update **MAKEFLAGS**. Thus, to describe how the environment interacts with **make**, the **MAKEFLAGS** macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable. If it is not present or null, the internal **make** variable **MAKEFLAGS** is set to the null string. Otherwise, each letter in **MAKEFLAGS** is assumed to be an input flag argument and is processed as such. (The only exceptions are the **−f**, **−p**, and **−r** flags.)

2. Read and set the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.

3. Read macro definitions from the command line. These are made *not resettable*. Thus, any further assignments to these names are ignored.

4. Read the internal list of macro definitions. These are found in the file *rules.c* of the source for **make**. Figure 2 contains the complete makefile that represents the internally defined macros and rules of the current version of **make**. Thus, if **make −r ...** is typed and a *makefile* includes the *makefile* in Figure 2, the results would be identical to excluding the **−r** option and the *include* line in the *makefile*. The Figure 2 output can be reproduced by the following:

   ```
   make −fp − < /dev/null 2>/dev/null
   ```

   The output appears on the standard output.
   They give default definitions for the C language compiler
   (CC=cc), the assembler (AS=as), etc.

5. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). However, since **MAKEFLAGS*** is not an internally defined variable (in *rules.c*), this has the effect of doing the same assignment twice. The exception to this is when **MAKEFLAGS** is assigned on the command line. (The reason it was read previously was to turn the debug flag on before anything else was done.)

6. Read the *makefile*(*s*). The assignments in the *makefile(s)* overrides the environment. This order is chosen so that when a *makefile* is read and executed, you know what to expect. That is, you get what is seen unless the **−e** flag is used. The **−e** is an additional command line flag which tells **make** to have the environment override the *makefile* assignments. Thus, if **make −e ...** is typed, the variables in the environment override the definitions in the *makefile*†. Also **MAKEFLAGS** override the environment if assigned. This is useful for further invocations of **make** from the current *makefile*.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions (from *rules.c*)
2. environment
3. *makefile(s)*
4. command line.

The **−e** flag has the effect of changing the order to:

---

* **MAKEFLAGS** are read and set again.

† There is no way to override the command line assignments.

1. internal definitions (from *rules.c*)
2. *makefile(s)*
3. environment
4. command line.

This order is general enough to allow a programmer to define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

Figure 2

Example of Internal Definitions (Sheet 1 of 4).

LIST OF SUFFIXES
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~
# PRESET VARIABLES
MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-o
AS=as
ASFLAGS=
GET=get
GFLAGS=

Example of Internal Definitions (Sheet 2 of 4).

#	SINGLE SUFFIX RULES
.c:	
	$(CC) -n -o $< -o $@
.c~:	
	$(GET) $(GFLAGS) -p $< > $*.c $(CC) -n -o $* .c -o $* -rm -f $*.c
.sh:	
	cp $< @
.sh~:	
	$(GET) &(GFLAGS) -p $< > .sh cp $* .sh $* -rm -f $* .sh
#	DOUBLE SUFFIX RULES
.c.o:	
	$(CC) $(CFLAGs) -c $<
.c~.o:	

Example of Internal Definitions (Sheet 3 of 4).

	$(GET) $(CFLAGS) -p $< > $*.c $(CC) $(CFLAGS) -c $*.c -rm -f $*.c
.c~ .c:	
	$(GET) $(GFLAGS) -p $< >$*.c
.s.o:	
	$(AS) $(ASFLAGS) -o $@ $<
.s~ .o:	
	$(GET) $(GFLAGS) -p $< > $*.s $(AS) $(ASFLAGS) -o $* .o $* .s -rm -f $*.s
.y.o:	
	$(YACC) $(YFLAGS) $< $(CC) $(CFLAGS) -c y.tab.c rm y.tab.o$@
.y~.o:	
	$(GET) $(GFLAG) -p $< > $*.y $(YACC) $(YFLAGS) $*.y $(CC) $(CFLAG) -c y.tab.c rm -f y.tab $*.y mv y.tab.o $*.o
.l.o:	
	$(LEX) $(LFLAGS) $< $(CC) $(CFLAGS) -c lex.yy.c rm lex.yy.c mv lex.yy.o $@

Example of Internal Definitions (Sheet 4 of 4).

.l~.o:	
	$(GET) $(GFLAGS) -p $< > $*.l $(LEX) $(GFLAG) $*.l $(CC) $(CFLAGS) -c lex.yy.c rm -f lex.yy.c $*.l mv lex.yy.o $*.o
.y.c:	
	$(YACC) $(YFLAGS) $< mv y.tab.c $@
.y~.c:	
	$(GET) $(GFLAGS) -p $< > $*.y $(YACC) $(YFLAGS) $*.y mv -f $*.c -rm -f $*.y
.l.c:	
	$(LEX) $< mv lex.yy.c$@
.c.a:	
	$(CC) -c $(FLAGS) $< ar rv $@ $*.o rm -f $*.o
.c~.a:	
	$(GET) $(GFLAGS) -p $< > $*.c $(CC) -c $(CFLAGS) $*.c ar rv $@ $*.o
.s~.a:	
	$(GET) $(GFLAGS) -p $< > $*.s $(AS) $(ASFLAGS) -o $*.o $*.s ar rv $@ $*.o -rm -f $*.[so]
.h~.h	
	$(GET) $(GFLAGS) -p $< > $*.h

RECURSIVE MAKEFILES

Another feature was added to **make** concerning the environment and recursive invocations. If the sequence "$(MAKE)" appears anywhere in a shell command line, the line is executed even if the **−n** flag is set. Since the **−n** flag is exported across invocations of **make** (through the **MAKEFLAGS** variable), the only thing that actually gets executed is the **make** command itself. This feature is useful when a hierarchy of *makefile(s)* describes a set of software subsystems. For testing purposes, **make −n ...** can be executed and everything that would have been done will get printed out including output from lower level invocations of **make**.

FORMAT OF SHELL COMMANDS WITHIN make

The **make** program remembers embedded newlines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when **make** prints it out, it retains the indentation and backslashes. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new function is gained.

ARCHIVE LIBRARIES

The **make** program has an improved interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax of addressing members of archive libraries. The previous version of **make** allows a user to name a member of a library in the following manner:

```
lib(object.o)
    or
lib((_localtime))
```

where the second method actually refers to an entry point of an object file within the library. (**Make** looks through the library, locates the entry point, and translates it to the correct object file name.)

To use this procedure to maintain an archive library, the following type of *makefile* is required:

```
lib:: lib(ctime.o)
        $(CC) -c -O ctime.c
        ar rv lib ctime.o
        rm ctime.o
lib:: lib(fopen.o)
        $(CC) -c -O fopen.c
        ar rv lib fopen.o
        rm fopen.o
...and so on for each object ...
```

This is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the file name being the only difference each time. (This is true in most cases.)

The current version gives the user access to a rule for building libraries. The handle for the rule is the ".a" suffix. Thus, a ".c.a" rule is the rule for compiling a C language source file, adding it to the library, and removing the ".o" cadaver. Similarly, the ".y.a", the ".s.a", and the ".l.a" rules rebuild YACC, assembler, and LEX files, respectively. The current archive rules defined internally are ".c.a", ".c{.a", and ".s{.a". [The tilde (~) syntax will be described shortly.] The user may define in makefile other rules needed.

The above 2-member library is then maintained with the following shorter makefile:

```
lib:    lib(ctime.o) lib(fopen.o)
        echo lib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual ".c.a" rules are as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the ".a" target (lib); the **$<** and **$*** macros are set to the out-of-date C language file; and the file name scans the suffix, respectively (*ctime.c* and *ctime*). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction

```
lib:    lib(ctime.o)
        @echo lib up-to-date
```

Assume the object in the library is out-of-date with respect to *ctime.c*. Also, there is no *ctime.o* file.

1. Do *lib*.
2. To do *lib*, do each dependent of *lib*.
3. Do *lib*(*ctime.o*).
4. To do *lib*(*ctime.o*), do each dependent of *lib*(*ctime.o*). (There are none.)
5. Use internal rules to try to build *lib*(*ctime.o*). (There is no explicit rule.) Note that *lib*(*ctime.o*) has a parenthesis in the name to identify the target suffix as ".a". This is the key. There is no explicit ".a" at the end of the *lib* library name. The parenthesis forces the ".a" suffix. In this sense, the ".a" is hard wired into **make**.
6. Break the name *lib*(*ctime.o*) up into *lib* and *ctime.o*. Define two macros, **$@** (=*lib*) and **$*** (=*ctime*).
7. Look for a rule ".X.a" and a file **$*.X**. The first ".X" (in the .SUFFIXES list) which fulfills these conditions is ".c" so the rule is ".c.a", and the file is *ctime.c*. Set **$<** to be *ctime.c* and execute the rule. In fact, **make** must then do *ctime.c*. However, the search of the current directory yields no other candidates, and the search ends.
8. The library has been updated. Do the rule associated with the "lib:" dependency; namely:

```
        echo lib up-to-date
```

It should be noted that to let *ctime.o* have dependencies, the following syntax is required:

```
lib(ctime.o):       $(INCDIR)/stdio.h
```

Thus, explicit references to *.o* files are unnecessary. There is also a new macro for referencing the archive member name when this form is used. The **$%** macro is evaluated each time **$@** is evaluated. If there is no current archive member, **$%** is null. If an archive member exists, then **$%** evaluates to the expression between the parenthesis.

An example *makefile* for a larger library is given in Figure 3.

Figure 3

Example of Library Makefile (Sheet 1 of 3).

```
#                   @(#)/usr/src/cmd/make/make.tm 3.2
LIB =lsxlib
PR=lp
INSDIR = /rl/flopO/
INS = eval
lsx:            $(LIB) low.o mch.o
                ld -x low.o mch.o $(LIB)
                mv a.out lsx
                @size lsx
#                Here, $(INS) as either "." or "eval".
lsx:
                $(INS)'cp lsx $(INSDIR)lsx . .
                    strip $(INSDIR)lsx . .
                    ls -l $(INSDIR)lsx'
print:
                $(PR) header.slow.smch.s*.h*.c Makefile
```

Example of Library Makefile (Sheet 2 of 3).

$(LIB):	
	$(LIB)(CLOCK.o)
	$(LIB)(main.o)
	$(LIB)(tty.o)
	$(LIB)(trap.o)
	$(LIB)(sysent.o)
	$(LIB)(sys2.o)
	$(LIB)(syn3.o)
	$(LIB)(syn4.o)
	$(LIB)(sys1.o)
	$(LIB)(sig.o)
	$(LIB)(fio.o)
	$(LIB)(kl.o)
	$(LIB)(alloc.o)
	$(LIB)(nami.o)
	$(LIB)(iget.o)
	$(LIB)(rdwri.o)
	$(LIB)(subr.o)

Example of Library Makefile (Sheet 3 of 3).

	$(LIB)(bio.o)
	$(LIB)(decfd.o)
	$(LIB)(sip.o)
	$(LIB)(space.o)
	$(LIB)(puts.o)
	@echo $(LIB) now up to date.
.s.o:	
	as -o $*.o header.s $*.s
.o.a:	
	ar rv $@ $<
	rm -f $<
.s.a:	
	as -o $*.o header.s $*.s ar rv $@ $*.o rm -f $*.o
.PRECIOUS:	$(LIB)

The reader will note also that there are no lingering "*.o" files left around. The result is a library maintained directly from the source files (or more generally from the SCCS files).

SOURCE CODE CONTROL SYSTEM FILE NAMES: THE TILDE

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX operating system machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, "s." precedes the file name part of the complete pathname.

To allow **make** easy access to the prefix "s." requires either a redefinition of the rule naming syntax of **make** or a trick. The trick is to use the tilde (~) as an

identifier of SCCS files. Hence, ".c~.o" refers to the rule which transforms an SCCS C language source file into an object. Specifically, the internal rule is

```
.c~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Obviously, the user can define other rules and suffixes which may prove useful. The tilde gives him a handle on the SCCS file name format so that this is possible.

THE NULL SUFFIX

In the UNIX system source code, there are many commands which consist of a single source file. It was wasteful to maintain an object of such files for **make**. The current implementation supports single suffix rules (a null suffix). Thus, to maintain the program *cat*, a rule in the *makefile* of the following form is needed:

```
.c:
        $(CC) -n -O $< -o $@
```

In fact, this ".c:" rule is internally defined so no *makefile* is necessary at all. The user only needs to type

```
make cat dd echo date
```

(these are notable single file programs) and all four C language source files are passed through the above shell command line associated with the ".c:" rule. The internally defined single suffix rules are

```
.c:
.c~:
.sh:
.sh~:
```

Others may be added in the *makefile* by the user.

INCLUDE FILES

The **make** program has an include file capability. If the string *include* appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the string is assumed to be a file name which the current invocation of **make** will read. The file descriptors are stacked for reading *include* files so that no more than about 16 levels of nested *includes* are supported.

INVISIBLE SCCS MAKEFILES

The SCCS *makefiles* are invisible to **make**. That is, if **make** is typed and only a file named *s.makefile* exists, **make** will do a **get** on the file, then read and remove the file. Using the **−f**, **make** will get, read, and remove arguments and *include* files.

DYNAMIC DEPENDENCY PARAMETERS

A new dependency parameter has been defined. The parameter has meaning only on the dependency line in a makefile. The **$$@** refers to the current "thing" to the left of the colon (which is **$@**). Also the form **$$(@F)** exists which allows access to the file part of **$@**. Thus, in the following:

```
cat:    $$@.c
```

the dependency is translated at execution time to the string "cat.c". This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX software command directory could have a *makefile* like:

```
CMDS = cat dd echo date cc cmp comm ar ld chown

$(CMDS):        $$@.c
        $(CC) −O $? −o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate *makefile* is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is **$$(@F)**. It represents the file name part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the */usr/include* directory from a makefile in the */usr/src/head* directory. Thus, the */usr/src/head/makefile* would look like

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCIDR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the */usr/include* directory whenever one of the above files in */usr/src/head* was updated.

EXTENSIONS OF $*, $@, AND $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The "D" refers to the directory part of the single letter macro. The "F" refers to the file name part of the single letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a shell command can be

```
cd $(<D); $(MAKE) $(<F)
```

The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is *ucb*. The FRC is a convention for *FoRC*ing **make** to completely rebuild a target starting from scratch.

OUTPUT TRANSLATIONS

Macros in shell commands can now be translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of *string1* in the evaluated macro, *string2* is substituted. The meaning of finding *string1* in **$(macro)** is that the evaluated **$(macro)** is considered as a bunch of strings each delimited by white space (blanks or tabs). Thus, the occurrence of *string1* in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. A more general regular expression match could be implemented if the need arises. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script which can handle all the C language programs (i.e., those files ending in ".c"). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffices) which define the archive library. These translations are added in an effort to make more general use of the wealth of information which **make** generates.

SOURCE CODE CONTROL SYSTEM USER GUIDE

GENERAL

The Source Code Control System (SCCS) is a collection of the UNIX software commands that help individuals or projects control and account for changes to files of text. The source code and documentation of software systems are typical examples of files of text to be changed. The SCCS is a collection of programs that run under the UNIX operating system. It is convenient to conceive of SCCS as a custodian of files. The SCCS provides facilities for

- Storing files of text
- Retrieving particular versions of the files
- Controlling updating privileges to files
- Identifying the version of a retrieved file
- Recording when, where, and why the change was made and who made each change to a file.

These types of facilities are important when programs and documentation undergo frequent changes because of maintenance and/or enhancement work. It is often desirable to regenerate the version of a program or document as it existed before changes were applied to it. This can be done by keeping copies (on paper or other media), but this method quickly becomes unmanageable and wasteful as the number of programs and documents increases. The SCCS provides an attractive solution because the original file is stored on disk. Whenever changes are made to the file, the SCCS stores only the changes. Each set of changes is called a "delta".

This chapter, together with relevant portions of the *UNIX Programmer's Manual—Volume1: Commands and Utilities* is a complete user's guide to SCCS. The following topics are covered:

- SCCS for Beginners: How to make an SCCS file, how to update it, and how to retrieve a version thereof.

- How Deltas Are Numbered: How versions of SCCS files are numbered and named.
- SCCS Command Conventions: Conventions and rules generally applicable to all SCCS commands.
- SCCS Commands: Explanation of all SCCS commands with discussions of the more useful arguments.
- SCCS Files: Protection, format, and auditing of SCCS files including a discussion of the differences between using SCCS as an individual and using it as a member of a group or project. The role of a "project SCCS administrator" is introduced.

Neither the implementation of SCCS nor the installation procedure for SCCS is described in this section.

Throughout this section, each reference of the form **name** (1M), **name** (7), or **name** (8) refers to entries in the *UNIX Programmer's Manual—Volume 3: System Administration Facilities.* All other references to entries of the form **name**(N), where "N" is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section N of the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.*

SCCS FOR BEGINNERS

It is assumed that the reader knows how to log onto a UNIX system, create files, and use the text editor. A number of terminal-session fragments are presented. All of them should be tried since the best way to learn SCCS is to use it.

To supplement the material in this section, the detailed SCCS command descriptions in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.* should be consulted.

A. Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file, with each set of changes usually depending on all previous sets. Each set of changes is called a "delta" and is assigned a name, called the *S*CCS *ID*entification string (SID). The SID is composed of at most four components. The first two components are the "release" and "level" numbers which are separated by a period. Hence, the first delta (for the original file) is called "1.1", the second "1.2", the third "1.3", etc. The release number can also be changed allowing, for example, deltas "2.1", "3.1", etc. The change in the release number usually indicates a major change to the file.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file, obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

B. Creating an SCCS File via "admin"

Consider, for example, a file called *lang* that contains a list of programming languages.

```
c
pl/i
fortran
cobol
algol
```

Custody of the *lang* file can be given to SCCS. The following **admin** command (used to "administer" SCCS files) creates an SCCS file and initializes delta 1.1 from the file *lang*:

```
admin -ilang s.lang
```

All SCCS files *must* have names that begin with "s.", hence, *s.lang*. The **-i** keyletter, together with its value *lang*, indicates that **admin** is to create a new SCCS file and "initialize" the new SCCS file with the contents of the file *lang*. This initial version is a set of changes (delta 1.1) applied to the null SCCS file.

The **admin** command replies

No id keywords (cm7)

This is a warning message (which may also be issued by other SCCS commands) that is to be ignored for the purposes of this section. Its significance is described under the **get** command in the part "SCCS COMMANDS." In the following examples, this warning message is not shown although it may actually be issued by the various commands. The file *lang* should now be removed (because it can be easily reconstructed using the **get** command) as follows:

rm lang

C. Retrieving a File via "get"

The *lang* file can be reconstructed by using the following **get** command:

get s.lang

The command causes the creation (retrieval) of the latest version of file *s.lang* and prints the following messages:

1.1
5 lines

This means that **get** retrieved version 1.1 of the file, which is made up of five lines of text. The retrieved text is placed in a file whose name is formed by deleting the "s." prefix from the name of the SCCS file. Hence, the file *lang* is created.

The "get s.lang" command simply creates the file *lang* (read-only) and keeps no information regarding its creation. On the other hand, in order to be able to subsequently apply changes to an SCCS file with the **delta** command, the **get** command must be informed of your intention to do so. This is done as follows:

```
get −e s.lang
```

The **−e** keyletter causes **get** to create a file *lang* for both reading and writing (so it may be edited) and places certain information about the SCCS file in another new file. The new file, called the *p-file,* will be read by the **delta** command. The **get** command prints the same messages as before except that the SID of the version to be created through the use of **delta** is also issued. For example,

```
get −e s.lang
1.1
new delta 1.2
5 lines
```

The file *lang* may now be changed, for example, by

```
ed lang
27
$a
snobol
ratfor
.
w
41
q
```

D. Recording Changes via "delta"

In order to record within the SCCS file the changes that have been applied to *lang*, execute the following command:

```
delta s.lang
```

Delta prompts with

```
comments?
```

The response should be a description of why the changes were made. For example,

comments? added more languages

The **delta** command then reads the *p-file* and determines what changes were made to the file *lang*. The **delta** command does this by doing its own **get** to retrieve the original version and by applying the **diff**(1) command to the original version and the edited version.

When this process is complete, at which point the changes to *lang* have been stored in *s.lang*, **delta** outputs

1.2
2 inserted
0 deleted
5 unchanged

The number "1.2" is the name of the delta just created, and the next three lines of output refer to the number of lines in the file *s.lang*.

E. Additional Information About "get"

As shown in the previous example, the command

get s.lang

retrieves the latest version (now 1.2) of the file *s.lang*. This is done by starting with the original version of the file and successively applying deltas (the changes) in order until all have been applied.

In the example chosen, the following commands are all equivalent:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the **-r** keyletter are SIDs. Note that omitting the level number of the SID (as in "get -r1 s.lang") is equivalent to specifying the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case, also 1.2.

Whenever a truly major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal automatic numbering of deltas proceeds by incrementing the level number (second component of the SID), the user must indicate to SCCS the need to change the release number. This is done with the **get** command.

```
get -e -r2 s.lang
```

Because release 2 does not exist, **get** retrieves the latest version *before* release 2. The **get** command also interprets this as a request to change the release number of the delta which the user desires to create to 2, thereby causing it to be named 2.1, rather than 1.3. This information is conveyed to **delta** via the *p-file*. The **get** command then outputs

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version **delta** will create. If the file is now edited, for example, by

```
ed lang
41
/cobol/d
w
35
q
```

and **delta** executed

```
delta s.lang
comments? deleted cobol from list of languages
```

the user will see by **delta**'s output that version 2.1 is indeed created.

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas may now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release may be created in a similar manner. This process may be continued as desired.

F. The "help" Command

If the command

```
get abc
```

is executed, the following message will be output:

```
ERROR [abc]: not an SCCS file (co1)
```

The string "co1" is a code for the diagnostic message and may be used to obtain a fuller explanation of that message by use of the **help** command.

```
help col
```

This produces the following output:

```
col:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

Thus, **help** is a useful command to use whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages may be found in this manner.

DELTA NUMBERING

It is convenient to conceive of the deltas applied to an SCCS file as the nodes of a tree in which the root is the initial version of the file. The root delta (node) is normally named "1.1" and successor deltas (nodes) are named "1.2", "1.3", etc. The components of the names of the deltas are called the "release" and the "level" numbers, respectively. Thus, normal naming of successor deltas proceeds by incrementing the level number, which is performed automatically by SCCS whenever a delta is made. In addition, the user may wish to change the release number when making a delta to indicate that a major change is being made. When this is done, the release number also applies to all successor deltas unless specifically changed again. Such a structure may be termed the "trunk" of the SCCS tree.

However, there are situations in which it is necessary to cause a branching in the tree in that changes applied as part of a given delta are *not* dependent upon all previous deltas. As an example, consider a program which is in production use at version 1.3 and for which development work on release 2 is already in progress. Thus, release 2 may already have some deltas. Assume that a production user reports a problem in version 1.3 and that the nature of the problem is such that it cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.).

The new delta is a node on a branch of the tree. Its name consists of four components; the release number and the level number (as with trunk deltas) plus the "branch" number and the "sequence" number. The delta name appears as follows:

release.level.branch.sequence

The branch number is assigned to each branch that is a descendant of a particular trunk delta with the first such branch being 1, the next one 2, etc. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.2 identifies the second delta of the first branch that derives from delta 1.3.

The concept of branching may be extended to any delta in the tree. The naming of the resulting deltas proceeds in the manner just illustrated.

Two observations are of importance with regard to naming deltas. First, the names of trunk deltas contain exactly two components, and the names of branch deltas contain exactly four components. Second, the first two components of the name of branch deltas are always those of the ancestral trunk delta, and the branch component is assigned in the order of creation of the branch independently of its location relative to the trunk delta. Thus, a branch delta may always be identified as such from its name. Although the ancestral trunk delta may be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta. For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n. The only information that may be derived from the name of delta 1.3.2.2 is that it is the chronologically second delta on the chronologically second branch whose trunk ancestor is delta 1.3. In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all the deltas between it and trunk ancestor 1.3.

It is obvious that the concept of branch deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible because comprehension of its structure becomes extremely difficult as the tree becomes more complex.

SCCS COMMAND CONVENTIONS

This part discusses the conventions and rules that apply to SCCS commands. These rules and conventions are generally applicable to all SCCS commands with exceptions indicated. The SCCS commands accept two types of arguments:

- Keyletter arguments
- File arguments.

Keyletter arguments (hereafter called simply "keyletters") begin with a minus sign (−), followed by a lowercase alphabetic character, and in some cases, followed by a value. These keyletters control the execution of the command to which they are supplied.

File arguments (names of files and/or directories) specify the file(s) that the given SCCS command is to process. Naming a directory is equivalent to naming *all* the SCCS files within the directory. Non-SCCS files and unreadable files [because of permission modes via **chmod**(1)] in the named directories are silently ignored.

In general, file arguments may not begin with a minus sign. However, if the name "−" (a lone minus sign) is specified as an argument to a command, the command reads the standard input for lines and takes each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the **find**(1) or **ls**(1) commands. Again, names of non-SCCS files and of unreadable files are silently ignored.

All keyletters specified for a given command apply to all file arguments of that command. All keyletters are processed before any file arguments with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Somewhat different argument conventions apply to the **help**, **what**, **sccsdiff**, and **val** commands.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussed in this part. For a complete description of all such flags, see **admin**(1) section in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.*

The distinction between the real user [see **passwd**(1)] and the effective user of a UNIX system is of concern in discussing various actions of SCCS commands. For the present, it is assumed that both the real user and the effective user are one and the same (i.e., the user who is logged into a UNIX system). This subject is discussed further in "SCCS FILES."

All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file.* This file ensures that the SCCS file is not damaged if processing should terminate abnormally. The name of the *x-file* is formed by replacing the "s." of the SCCS file name with "x.". When processing is complete, the old SCCS file is removed and the *x-file* is renamed to be the SCCS file. The *x-file* is created in the directory containing the SCCS file, given the same mode [see **chmod**(1)] as the SCCS file, and owned by the effective user.

To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a *lock-file,* called the *z-file,* whose name is formed by replacing the "s." of the SCCS file name with "z.". The *z-file* contains the process number of the command that creates it, and its existence is an indication to other commands that the SCCS file is being updated. Thus, other commands that modify SCCS files do not process an SCCS file if the corresponding *z-file* exists. The *z-file* is created with mode 444 (read-only) in the directory containing the SCCS file and is owned by the effective user. This file exists only for the duration of the execution of the command that creates it. In general, users can ignore *x-files* and *z-files.* The files may be useful in the event of system crashes or similar situations.

The SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [name-of-file-being-processed]: message text (code)

The code in parentheses may be used as an argument to the **help** command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to terminate processing of that file and to proceed with the next file, in order, if more than one file has been named.

SCCS COMMANDS

This part describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities* and should be consulted for further information. The discussion below covers only the more common arguments of the various SCCS commands.

The commands follow in approximate order of importance. The following is a summary of all the SCCS commands and of their major functions:

get	Retrieves versions of SCCS files.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
admin	Creates SCCS files and applies changes to parameters of SCCS files.
prs	Prints portions of an SCCS file in user specified format.
help	Gives explanations of diagnostic messages.
rmdel	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
cdc	Changes the commentary associated with a delta.
what	Searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the **get** command.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
val	Validates an SCCS file.

A. The "get" Command

The **get** command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file*. The *g-file* name is formed by removing the "s." from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the **get** command is invoked.

The most common invocation of **get** is

```
get s.abc
```

which normally retrieves the latest version on the trunk of the SCCS file tree and produces (for example) on the standard output

```
1.3
67 lines
No id keywords (cm7)
```

which indicates that

1. Version 1.3 of file "s.abc" was retrieved (1.3 is the latest trunk delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file.

The generated *g-file* (file "abc") is given mode 444 (read-only). This particular way of invoking **get** is intended to produce *g-files* only for inspection, compilation, etc. It is not intended for editing (i.e., not for making deltas).

In the case of several file arguments (or directory-name arguments), similar information is given for each file processed, but the SCCS file name precedes it. For example,

```
get s.abc s.def
```

produces

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.def:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a *g-file* to be used for compilation, it is useful and informative to record the date and time of creation, the version retrieved, the module's name, etc. within the *g-file*. This information appears in a load module when one is eventually created. The SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of these ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%). For example,

```
%I%
```

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly, **%H%** is defined as the ID keyword for the current date (in the form "mm/dd/yy"), and **%M%** is defined as the name of the *g-file*. Thus, executing **get** on an SCCS file that contains the PL/I declaration,

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

gives (for example) the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/07/77');
```

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get**, although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately 20 ID keywords provided, see **get**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.*

Retrieval of Different Versions

Various keyletters are provided to allow the retrieval of other than the default version of an SCCS file. Normally, the default version is the most recent delta of the highest-numbered release on the trunk of the SCCS file tree. However, if the SCCS file being processed has a **d** (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the **−r** keyletter of **get**.

The **−r** keyletter is used to specify an SID to be retrieved, in which case the **d** (default SID) flag (if any) is ignored. For example,

```
get −r1.3 s.abc
```

retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output

```
1.3
64 lines
```

A branch delta may be retrieved similarly,

```
get −r1.5.2.3 s.abc
```

which produces (for example) on the standard output

```
1.5.2.3
234 lines
```

When a 2- or 4-component SID is specified as a value for the **−r** keyletter (as above) and the particular version does not exist in the SCCS file, an error message results. Omission of the level number, as in

```
get −r3 s.abc
```

causes retrieval of the trunk delta with the highest level number within the given release if the given release exists. Thus, the above command might output,

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assuming release 9 does not exist in file *s.abc* and that release 7 is actually the highest-numbered release below 9, execution of

```
get −r9 s.abc
```

might produce

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file *s.abc* below release 9. Similarly, omission of the sequence number, as in

```
get −r4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

The **−t** keyletter is used to retrieve the latest (top) version in a particular release (i.e., when no **−r** keyletter is supplied or when its value is simply a release number). The latest version is defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
get -r3 -t s.abc
```

might produce

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce

```
3.2.1.5
46 lines
```

Retrieval With Intent to Make a Delta

Specification of the **-e** keyletter to the **get** command is an indication of the intent to make a delta, and as such, its use is restricted. The presence of this keyletter causes **get** to check

1. The user list (a list of login names and/or group IDs of users allowed to make deltas) to determine if the login name or group ID of the user executing **get** is on that list. Note that a null (empty) user list behaves as if it contained all possible login names.

2. The release (R) of the version being retrieved satisfies the relation:

   floor is < or = to R which is
   < or = to ceiling

   to determine if the release being accessed is a protected release. The "floor" and "ceiling" are specified as flags in the SCCS file.

3. The R is not locked against editing. The "lock" is specified as a flag in the SCCS file.

4. Whether or not multiple concurrent edits are allowed for the SCCS file as specified by the **j** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, the **−e** keyletter causes the creation of a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable *g-file* already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are not substituted by **get** (when the **−e** keyletter is specified) because the generated *g-file* is subsequently used to create another delta. Replacement of ID keywords cause them to be permanently changed within the SCCS file. In view of this, **get** does not need to check for the presence of ID keywords within the *g-file,* so the message

```
No id keywords (cm7)
```

is never output when **get** is invoked with the **−e** keyletter.

In addition, the **−e** keyletter causes the creation (or updating) of a *p-file* which is used to pass information to the **delta** command.

The following is an example of the use of the **−e** keyletter:

```
get −e s.abc
```

which produces (for example) on the standard output

```
1.3
new delta 1.4
67 lines
```

If the **−r** and/or **−t** keyletters are used together with the **−e** keyletter, the version retrieved for editing is as specified by the **−r** and/or **−t** keyletters.

The keyletters **−i** and **−x** may be used to specify a list [see **get**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities* for the syntax of such a list] of deltas to be included and excluded, respectively, by **get**. Including a delta means forcing the changes that constitute the particular delta

to be included in the retrieved version. This is useful if one wants to apply the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo (in the version of the SCCS file to be created) the effects of a previous delta. Whenever deltas are included or excluded, **get** checks for possible interference between such deltas and those deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists and to take whatever corrective measures (if any) are deemed necessary (e.g., edit the file).

Warning: The −i and −x keyletters should be used with extreme care.

The **−k** keyletter is provided to facilitate regeneration of a *g-file* that may have been accidentally removed or ruined subsequent to the execution of **get** with the **−e** keyletter or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed. Thus, a *g-file* generated by the **−k** keyletter is identical to one produced by **get** and executed with the **−e** keyletter. However, no processing related to the *p-file* takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows a number of deltas to be "in progress" at any given time. This means that a number of **get** commands with the **−e** keyletter may be executed on the same file provided that no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The *p-file* (created by the **get** command invoked with the **−e** keyletter) is named by replacing the "s." in the SCCS file name with "p.". It is created in the directory containing the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The *p-file* contains the following information for each delta that is still "in progress":

- The SID of the retrieved version.
- The SID that is given to the new delta when it is created.

- The login name of the real user executing **get**.

The first execution of **get** **−e** causes the creation of the *p-file* for the corresponding SCCS file. Subsequent executions only update the *p-file* with a line containing the above information. Before updating, however, **get** checks to assure that no entry (already in the *p-file)* specifies that the SID (of the version to be retrieved) is already retrieved (unless multiple concurrent edits are allowed).

If both checks succeed, the user is informed that other deltas are in progress and processing continues. If either check fails, an error message results. It is important to note that the various executions of **get** should be carried out from different directories. Otherwise, only the first execution succeeds since subsequent executions would attempt to overwrite a writable *g-file,* which is an SCCS error condition. In practice, such multiple executions are performed by different users so that this problem does not arise since each user normally has a different working directory. See "Protection" under the part "SCCS FILES" for a discussion of how different users are permitted to use SCCS commands on the same files.

Figure 4 shows, for the most useful cases, the version of an SCCS file retrieved by **get**, as well as the SID of the version to be eventually created by **delta**, as a function of the SID specified to **get**.

Figure 4

Determination of New SID (Sheet 1 of 3).

SID SPECIFIED*	−b KEY-LETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DATA TO BE CREATED
none†	no	R default to mR	mRmL	mR(mL+1)
none‡	yes	R default to mR	mRmL	mRmL.(mB+1)
R	no	R > mR	mRmL	R.1§
R	no	R == mR	mRmL	mR.(mL+1)
R	yes	R > mR	mRmL	mR.mL.(mB+1).1
R	yes	R == mR	mR.mL	mR.mL.(mB+1).1
R	−	R<mR		
R	−	R< mR and does not exist	hR.mL**	hR.mL.(mB+1).1
R	−	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1

See footnotes on sheet 3 of 3.

Determination of New SID (Sheet 2 of 3).

SID SPECIFIED*	-b KEY-LETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DELTA TO BE CREATED
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunks successor	R.L	R.L.(mB+1).1
R.L	–	Trunk in release >= R	R.L	R.L.(mS+1).1
R.L.b	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	no	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1

See footnotes on sheet 3 of 3.

Determination of New SID (Sheet 3 of 3).

Footnotes:

* "R", "L", "B", and "S" are "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL"means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the (i.e., maximum branch number plus 1) of level L within release R". Also note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components must exist.

† The **−b** keyletter is effective only if the **b** flag [see **admin(1)**] is present in the file. In this state, an entry of "−" means "irrelevant".

‡ This case applies if the **d** (default SID) flag is not present in the file. If the **d** flag is present in the file, the SID obtained from the **d** flag is interrupted as if it had been specified on the command line. Thus, one of the other cases in this figure applies.

§ This case is used to force the creation of the first delta in the new release.

** "hR" is the highest existing release that is lower than the specified, nonexisting, release R.

Concurrent Edits of Same SID

Under normal conditions, **gets** for editing (**−e** keyletter is specified) based on the same SID are not permitted to occur concurrently. That is, **delta** must be executed before a subsequent **get** for editing is executed at the same SID as the previous **get**. However, multiple concurrent edits (defined to be two or more successive executions of **get** for editing based on the same retrieved SID) are allowed if the **j** flag is set in the SCCS file. Thus:

```
get −e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by

```
get −e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening execution of **delta**. In this case, a **delta** command corresponding to the first **get** produces delta 1.2 [assuming 1.1 is the latest (most recent) trunk delta], and the **delta** command corresponding to the second **get** produces delta 1.1.1.1.

Keyletters That Affect Output

Specification of the **−p** keyletter causes **get** to write the retrieved text to the standard output rather than to a *g-file*. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. This may be used, for example, to create *g-files* with arbitrary names.

```
get −p s.abc > arbitrary-file-name
```

The **−p** keyletter is particularly useful when used with the "!" or "$" arguments of the **send**(1C) command. For example,

```
send MOD=s.abc REL=3 compile
```

given that file *compile* contains

```
//plicomp job job-card-information
//step1 exec plickc
//pli.sysin dd *
 −s
 !get −p −rREL MOD
/*
//
```

will **send** the highest level of release 3 of file *s.abc*. Note that the line "~−s" (that causes **send** to make ID keyword substitutions before detecting and interpreting control lines) is necessary if **send** is to substitute "s.abc" for MOD and "3" for REL in the line "~!get −p −rREL MOD".

The **−s** keyletter suppresses all output that is normally directed to the standard output. Thus, the SID of the retrieved version, the number of lines retrieved, etc., are not output. This does not, however, affect messages to the diagnostic output. This keyletter is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used in conjunction with the **−p** keyletter to "pipe" the output of **get**, as in

```
get −p −s s.abc | nroff
```

The **−g** keyletter is supplied to suppress the actual retrieval of the text of a version of the SCCS file. This may be useful in a number of ways. For example, to verify the existence of a particular SID in an SCCS file, one may execute

```
get −g −r4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file or it generates an error message if it does not. Another use of the **−g** keyletter is in regenerating a *p-file* that may have been accidentally destroyed.

```
get −e −g s.abc
```

The **−l** keyletter causes the creation of an *l-file*, which is named by replacing the "s." of the SCCS file name with "l.". This file is created in the current directory with mode 444 (read-only) and is owned by the real user. It contains a table [whose format is described in **get**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*] showing the deltas used in constructing a particular version of the SCCS file. For example,

```
get −r2.3 −l s.abc
```

generates an *l-file* showing the deltas applied to retrieve version 2.3 of the SCCS file. Specifying a value of "p" with the **−l** keyletter, as in

```
get −lp −r2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*. The **−g** keyletter may be used with the **−l** keyletter to suppress the

actual retrieval of the text.

The **-m** keyletter is of use in identifying, line by line, the changes applied to an SCCS file. Specification of this keyletter causes each line of the generated *g-file* to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **-n** keyletter causes each line of the generated *g-file* to be preceded by the value of the **sccs1** ID keyword and a tab character. The **-n** keyletter is most often used in a pipeline with **grep**(1). For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

```
get -p -n -s directory | grep pattern
```

If both the **-m** and **-n** keyletters are specified, each line of the generated *g-file* is proceded by the value of the **%M%** ID keyword and a tab (this is the effect of the **-n** keyletter) and followed by the line in the format produced by the **-m** keyletter. Because use of the **-m** keyletter and/or the **-n** keyletter causes the contents of the *g-file* to be modified, such a *g-file* must *not* be used for creating a delta. Therefore, neither the **-m** keyletter nor the **-n** keyletter may be specified together with the **-e** keyletter.

See **get**(1) in the *UNIX Programmer's Manual—Volume1: Commands and Utilities* for a full description of additional **get** keyletters.

B. The "delta" Command

The **delta** command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta, and therefore, a new version of the file.

Invocation of the **delta** command requires the existence of a *p-file*. The **delta** command examines the *p-file* to verify the presence of an entry containing the user's login name. If none is found, an error message results. The **delta** command performs the same permission checks that **get** performs when invoked by the **-e** keyletter. If all checks are successful, **delta** determines what has been changed in the *g-file* by comparing it via **diff**(1) with its own temporary copy of the *g-file* as it was before editing. This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the "s." of the SCCS file name with "d.") and is obtained by performing an internal **get** at the SID

specified in the *p-file* entry.

The required *p-file* entry is the one containing the login name of the user executing **delta** because the user who retrieved the *g-file* must be the one who creates the delta. However, if the login name of the user appears in more than one entry, the same user has executed **get** with the **−e** keyletter more than once on the same SCCS file. The **−r** keyletter must then be used with **delta** to specify the SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

In practice, the most common invocation of **delta** is

```
delta s.abc
```

which prompts on the standard output (but only if it is a terminal)

```
comments?
```

to which the user replies with a description of why the delta is being made, terminating the reply with a newline character. The user's response may be up to 512 characters long with newlines (not intended to terminate the response) escaped by backslashes "\".

If the SCCS file has a **v** flag, **delta** first prompts with

```
MRs? (Modification Requests)
```

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR numbers, separated by blanks and/or tabs, terminated in the same manner as the response to the prompt "comments?". In a tightly controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc., collectively called [MRs]. It is desirable (or necessary) to record such MR number(s) within each delta.

The **−y** and/or **−m** keyletters may be used to supply the commentary (comments and MR numbers, respectively) on the command line rather than through the standard input.

delta −y"descriptive comment" −m"mrnum1 mrnum2" s.abc

In this case, the corresponding prompts are not printed, and the standard input is not read. The **−m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when **delta** is executed from within a shell procedure [see **sh**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*].

The commentary (comments and/or MR numbers), whether solicited by **delta** or supplied via keyletters, is recorded as part of the entry for the delta being created and applies to all SCCS files processed by the same invocation of **delta**. This implies that (if **delta** is invoked with more than one file argument and the first file named has a **v** flag) all files named must have this flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, **delta** outputs (on the standard output) the SID of the created delta (obtained from the *p-file* entry) and the counts of lines inserted, deleted, and left unchanged by the delta. Thus, a typical output might be

1.4
14 inserted
7 deleted
345 unchanged

It is possible that the counts of lines reported as inserted, deleted, or unchanged by **delta** do not agree with the user's perception of the changes applied to the *g-file*. The reason for this is that there usually are a number of ways to describe a set of such changes, especially if lines are moved around in the *g-file*, and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited *g-file*.

If (in the process of making a delta) **delta** finds no ID keywords in the edited *g-file*, the message

No id keywords (cm7)

is issued after the prompts for commentary but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by creating a delta from a *g-file* that was created by a **get** without the **−e** keyletter (recall that ID keywords are replaced by **get** in that case). This could also be caused by accidentally deleting or changing the ID keywords during the editing of the *g-file*. Another possibility is that the file had no ID keywords. In any case, it is left up to the user to determine what remedial action is necessary. However, the delta is made unless there is an **i** flag in the SCCS file indicating that this should be treated as a fatal error. In this last case, the delta is not created.

After the processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*. All updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file* which is described in the part "SCCS COMMAND CONVENTIONS". If there is only one entry in the *p-file*, then the *p-file* itself is removed.

In addition, **delta** removes the edited *g-file* unless the **−n** keyletter is specified. Thus:

```
delta −n s.abc
```

will keep the *g-file* upon completion of processing.

The **−s** (silent) keyletter suppresses all output that is normally directed to the standard output, other than the prompts "comments?" and "MRs?". Thus, use of the **−s** keyletter together with the **−y** keyletter (and possibly, the **−m** keyletter) causes **delta** neither to read the standard input nor to write the standard output.

The differences between the *g-file* and the *d-file* (see above), constitute the delta and may be printed on the standard output by using the **−p** keyletter. The format of this output is similar to that produced by **diff**(1).

C. The "admin" Command

The **admin** command is used to administer SCCS files, that is, to create new SCCS files and to change parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of keyletters or are assigned default values if no keyletters are supplied. The same keyletters are used to change the parameters of existing files.

Two keyletters are supplied for use in conjunction with detecting and correcting "corrupted" SCCS files (see "Auditing" in part "SCCS FILES"). Newly created SCCS files are given mode 444 (read-only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command upon that file.

Creation of SCCS Files

An SCCS file may be created by executing the command

```
admin -ifirst s.abc
```

in which the value "first" of the **-i** keyletter specifies the name of a file from which the text of the initial delta of the SCCS file *s.abc* is to be taken. Omission of the value of the **-i** keyletter indicates that **admin** is to read the standard input for the text of the initial delta. Thus, the command

```
admin -i s.abc < first
```

is equivalent to the previous example. If the text of the initial delta does not contain ID keywords, the message

```
No id keywords (cm7)
```

is issued by **admin** as a warning. However, if the same invocation of the command also sets the **i** flag (not to be confused with the **-i** keyletter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using the **-i** keyletter.

When an SCCS file is created, the release number assigned to its first delta is normally "1", and its level number is always "1". Thus, the first delta of an SCCS file is normally "1.1". The **-r** keyletter is used to specify the release

number to be assigned to the first delta. Thus:

```
admin -ifirst -r3 s.abc
```

indicates that the first delta should be named "3.1" rather than "1.1". Because this keyletter is only meaningful in creating the first delta, its use is only permitted with the **-i** keyletter.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may choose to supply commentary stating the reason for creation of the file. This is done by supplying comments (**-y** keyletter) and/or MR numbers (**-m** keyletter) in exactly the same manner as for **delta**. The creation of an SCCS file may sometimes be the direct result of an MR. If comments (**-y** keyletter) are omitted, a comment line of the form

```
date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically generated.

If it is desired to supply MR numbers (**-m** keyletter), the **v** flag must also be set (using the **-f** keyletter described below). The **v** flag simply determines whether or not MR numbers must be supplied when using any SCCS command that modifies a "delta commentary" [see **sccsfile**(4) in the **UNIX Programmer's Manual—Volume 1: Commands and Utilties**] in the SCCS file. Thus:

```
admin -ifirst -mmrnum1 -fv s.abc
```

Note that the **-y** and **-m** keyletters are only effective if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

The portion of the SCCS file reserved for descriptive text may be initialized or changed through the use of the **-t** keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file.

When an SCCS file is being created and the **−t** keyletter is supplied, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

```
admin −ifirst −tdesc s.abc
```

specifies that the descriptive text is to be taken from file *desc*;.

When processing an *existing* SCCS file, the **−t** keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
admin −tdesc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of *desc*; omission of the file name after the **−t** keyletter as in

```
admin −t s.abc
```

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized, changed, or deleted through the use of the **−f** and **−d** keyletters, respectively. The flags of an SCCS file are used to direct certain actions of the various commands. See **admin**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities* for a description of all the flags. For example, the **i** flag specifies that the warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. Also the **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command. The **−f** keyletter is used to set a flag and, possibly, to set its value. For example,

```
admin −ifirst −fi −fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The value "modname" specified for the **m** flag is the value that the **get** command will use to replace the **%M%** ID keyword. (In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the **%M%** ID keyword.) Note that several **−f** keyletters may be supplied on a single invocation of **admin** and that **−f** keyletters may be supplied whether the command is creating a new SCCS file or processing an

existing one.

The **-d** keyletter is used to delete a flag from an SCCS file and may only be specified when processing an existing file. As an example, the command

```
admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several **-d** keyletters may be supplied on a single invocation of **admin** and may be intermixed with **-f** keyletters.

The SCCS files contain a list (user list) of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default which implies that anyone may create deltas. To add login names and/or group IDs to the list, the **-a** keyletter is used. For example,

```
admin -axyz -awql -a1234 s.abc
```

adds the login names "xyz" and "wql" and the group ID "1234" to the list. The **-a** keyletter may be used whether **admin** is creating a new SCCS file or processing an existing one and may appear several times. The **-e** keyletter is used in an analogous manner if one wishes to remove (erase) login names or group IDs from the list.

D. The "prs" Command

The **prs** command is used to print on the standard output all or parts of an SCCS file in a format, called the output "data specification," supplied by the user via the **-d** keyletter. The data specification is a string consisting of SCCS file data keywords (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values according to their definitions. For example,

```
:I:
```

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the SCCS file name currently

being processed, and **:C:** is defined as the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list of the data keywords, see **prs**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.*

There is no limit to the number of times a data keyword may appear in a data specification. Thus, for example,

```
prs −d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output

```
2.1 this is the top delta for s.abc 2.1
```

Information may be obtained from a single delta by specifying the SID of that delta using the **−r** keyletter. For example,

```
prs −d":F:: :I: comment line is: :C:" −r1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the **−r** keyletter is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained by specifying the **−l** or **−e** keyletters. The **−e** keyletter substitutes data keywords for the SID designated via the **−r** keyletter and all deltas created earlier. The **−l** keyletter substitutes data keywords for the SID designated via the **−r** keyletter and all deltas created later. Thus, the command

```
prs −d:I: −r1.4 −e s.abc
```

may output

1.4
1.3
1.2.1.1
1.2
1.1

and the command

```
prs -d:I: -r1.4 -1 s.abc
```

may produce

3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both the **-e** and **-l** keyletters.

E. The "help" Command

The **help** command prints explanations of SCCS commands and of messages that these commands may print. Arguments to **help**, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, **help** prompts for one. The **help** command has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example,

```
help ge5 rmdel
```

produces

```
ge5:
"nonexistent sid"
The specified sid does not exist in the
given file.
Check for typos.

rmdel:
  rmdel -rSID name ...
```

F. The "rmdel" Command

The **rmdel** command is provided to allow removal of a delta from an SCCS file. Its use should be reserved for those cases in which incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be a "leaf" delta. That is, it must be the latest (most recently created) delta on its branch or on the trunk of the SCCS file tree.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed or be the owner of the SCCS file and its directory.

The **-r** keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a trunk delta and four components for a branch delta). Thus:

```
rmdel -r2.3 s.abc
```

specifies the removal of (trunk) delta "2.3" of the SCCS file. Before removal of the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation.

```
floor <= R <= ceiling
```

The **rmdel** command also checks that the SID specified is not that of a version for which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's "user list", or the "user list" must be empty. Also, the release specified cannot be locked against editing. That is, if the **l** flag is set [see **admin**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*], the release specified *must* not be contained in the list. If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the "delta table" of the SCCS file is changed from "D" ("delta") to "R" ("removed").

G. The "cdc" Command

The **cdc** command is used to change a delta's commentary that was supplied when that delta was created. Its invocation is analogous to that of the **rmdel** command, except that the delta to be processed is not required to be a leaf delta. For example,

```
cdc −r3.4 s.abc
```

specifies that the commentary of delta "3.4" of the SCCS file is to be changed.

The new commentary is solicited by **cdc** in the same manner as that of **delta**. The old commentary associated with the specified delta is kept, but it is preceded by a comment line indicating that it has been changed (i.e., superseded), and the new commentary is entered ahead of this comment line. The "inserted" comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the deletion of selected MR numbers associated with the specified delta. This is specified by preceding the selected MR numbers by the character "!". Thus:

```
cdc −r1.4 s.abc
MRs? mrnum3 !mrnum1
comments? deleted wrong MR number and inserted
  correct MR number
```

inserts "mrnum3" and deletes "mrnum1" for delta 1.4.

H. The "what" Command

The **what** command is used to find identifying information within any UNIX system file whose name is given as an argument to **what**. Directory names and a name of "–" (a lone minus sign) are not treated specially as they are by other SCCS commands and no keyletters are accepted by the command.

The **what** command searches the given file(s) for all occurrences of the string "@(#)", which is the replacement for the @(#) ID keyword [see **get**(1)], and prints (on the standard output) the balance following that string until the first double quote ("), greater than (>), backslash (\), newline, or (nonprinting) NUL character. For example, if the SCCS file *s.prog.c* (a C language program) contains the following line:

```
char id[] "@(#)sccs2:5.1";
```

and then the command

```
get -r3.4 s.prog.c
```

is executed, the resulting *g-file* is compiled to produce "prog.o" and "a.out". Then the command

```
what prog.c prog.o a.out
```

produces

```
prog.c:
  prog.c:3.4
prog.o:
  prog.c:3.4
a.out:
  prog.c:3.4
```

The string searched for by **what** need not be inserted via an ID keyword of **get**; it may be inserted in any convenient manner.

I. The "sccsdiff" Command

The **sccsdiff** command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified by using the **−r** keyletter, whose format is the same as for the **get** command. The two versions must be specified as the first two arguments to this command in the order they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the **pr**(1) command (which actually prints the differences) and must appear before any file names. The SCCS files to be processed are named last. Directory names and a name of "−" (a lone minus sign) are not acceptable to **sccsdiff**.

The differences are printed in the form generated by **diff**(1). The following is an example of the invocation of **sccsdiff**:

```
sccsdiff −r3.4 −r5.6 s.abc
```

J. The "comb" Command

The **comb** command generates a "shell procedure" [see **sh**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*] which attempts to reconstruct the named SCCS files so that the reconstructed files are smaller than the originals. The generated shell procedure is written on the standard output. Named SCCS files are reconstructed by discarding unwanted deltas and combining other specified deltas. The SCCS files that contain deltas no longer useful should be discarded. It is not recommended that **comb** be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file.

In the absence of any keyletters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the "shape" of the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Some of the keyletters are summarized as follows:

The **−p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.

The **−c** keyletter specifies a list [see **get**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities* for the syntax of such a list] of deltas to be preserved. All other deltas are discarded.

The **−s** keyletter causes the generation of a shell procedure, which when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that **comb** be run with this keyletter (in addition to any others desired) before any actual reconstructions.

It should be noted that the shell procedure generated by **comb** is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

K. The "val" Command

The **val** command is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The **val** command checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **−r** keyletter. The string following the **−y** or **−m** keyletter is used to check the value set by the **t** or **m** flag, respectively [see **admin**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities* for a description of the flags].

The **val** command treats the special argument "**−**" differently from other SCCS commands. This argument allows **val** to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end of file. This capability allows for one invocation of **val** with different values for the keyletter and file arguments. For example,

```
val −
−yc −mabc s.abc
−mxyz −ypl1 s.xyz
```

first checks if file *s.abc* has a value "c" for its "type" flag and value "abc" for the "module name" flag. Once processing of the first file is completed, **val** then processes the remaining files, in this case, *s.xyz*, to determine if they meet the

characteristics specified by the keyletter arguments associated with them.

The **val** command returns an 8-bit code; each bit set indicates the occurrence of a specific error [see **val**(1) for a description of possible errors and the codes]. In addition, an appropriate diagnostic is printed unless suppressed by the **−s** keyletter. A return code of "0" indicates all named files met the characteristics specified.

SCCS FILES

This part discusses several topics that must be considered before extensive use is made of SCCS. These topics deal with the protection mechanisms relied upon by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

A. Protection

The SCCS relies on the capabilities of the UNIX software for most of the protection mechanisms required to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). The only protection features provided directly by SCCS are the "release lock" flag, the "release floor" and "ceiling" flags, and the "user list".

New SCCS files created by the **admin** command are given mode 444 (read-only). It is recommended that this mode *remain unchanged* as it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files be given mode 755 which allows only the owner of the directory to modify its contents.

The SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.

The SCCS files must have only one link (name) because the commands that modify SCCS files do so by creating a copy of the file (the *x-file,* see "SCCS COMMAND CONVENTIONS"). Upon completion of processing, remove the old file and rename the *x-file.* If the old file has more than one link, this would break such additional links. Rather than process such files, SCCS commands

produce an error message. All SCCS files *must* have names that begin with "s.".

When only one user uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

However, in those situations in which several users with unique user IDs are assigned responsibility for one SCCS file (e.g., in large software development projects), one user (equivalently, one user ID) must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Because other users of SCCS do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the "set user ID on execution" bit "on" [see **chmod**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*]. This assures that the effective user ID is the user ID of the administrator. This program invokes the desired SCCS command and causes it to inherit the privileges of the interface program for the duration of that command's execution. Thus, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the "user list" for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Other users are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc**. The project-dependent interface program, as its name implies, must be custom-built for each project.

B. Formatting

The SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	A line containing the "logical" sum of all the characters of the file (*not* including this checksum itself).
Delta Table	Information about each delta, such as type, SID, date and time of creation, and commentary.

User Names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of various SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.
Body	Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

Detailed information about the contents of the various sections of the file may be found in **sccsfile**(5). The checksum is the only portion of the file that is of interest below.

It is important to note that because SCCS files are ASCII files they may be processed by various UNIX software commands, such as **ed**(1), **grep**(1), and **cat**(1). This is very convenient in those instances in which an SCCS file must be modified manually (e.g., when the time and date of a delta was recorded incorrectly because the system clock was set incorrectly) or when it is desired to simply look at the file.

Caution: Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

C. Auditing

On rare occasions, perhaps due to an operating system or hardware malfunction, an SCCS file or portions of it (i.e., one or more "blocks") can be destroyed. The SCCS commands (like most UNIX software commands) issue an error message when a file does not exist. In addition, SCCS commands use the checksum stored in the SCCS file to determine whether a file has been corrupted since it was last accessed [possibly by having lost one or more blocks or by having been modified with **ed**(1)]. No SCCS command will process a corrupted SCCS file except the **admin** command with the **−h** or **−z** keyletters, as described below.

It is recommended that SCCS files be audited for possible corruptions on a regular basis. The simplest and fastest way to perform an audit is to execute the **admin** command with the **−h** keyletter on all SCCS files.

```
admin -h s.file1 s.file2 ...
      or
admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

```
corrupted file (co6)
```

is produced for that file. This process continues until all the files have been examined. When examining directories (as in the second example above), the process just described will not detect missing files. A simple way to detect whether any files are missing from a directory is to periodically execute the **ls**(1) command on that directory and compare the outputs of the most current and the previous executions. Any file whose name appears in the previous output but not in the current one has been removed by some means.

Whenever a file has been corrupted, the manner in which the file is restored depends upon the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request that the file be restored from a backup copy. In the case of minor damage, repair through use of the editor **ed**(1) may be possible. In the latter case after such repair, the following command must be executed:

```
admin -z s.file
```

The purpose of this is to recompute the checksum to bring it into agreement with the actual contents of the file. After this command is executed on a file, any corruption that existed in the file will no longer be detectable.

AN SCCS INTERFACE PROGRAM

A. General

In order to permit UNIX system users [with different user identification numbers (user IDs)] to use SCCS commands upon the same files, an SCCS interface program is provided. It temporarily grants the necessary file access permissions to these users. This part discusses the creation and use of such an interface program. The SCCS interface program may also be used as a preprocessor to SCCS commands since it can perform operations upon its arguments.

B. Function

When only one user uses SCCS, the real and effective user IDs are the same; and that user's ID owns the directories containing SCCS files. However, there are situations (e.g., in large software development projects) in which it is practical to allow more than one user to make changes to the same set of SCCS files. In these cases, one user must be chosen as the "owner" of the SCCS files and be the one who will "administer" them (e.g., by using the **admin** command). This user is termed the "SCCS administrator" for that project. Since other users of SCCS do not have the same privileges and permissions as the SCCS administrator, the other users are not able to execute directly those commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get**, **delta**, and if desired, **rmdel**, **cdc**, and **unget** commands. Other SCCS commands either do not require write permission in the directory containing SCCS files or are (generally) reserved for use only by the administrator.

The interface program

- Must be owned by the SCCS administrator
- Must be executable by the new owner
- Must have the " set user on execution" bit "on" [see **chmod**(1) in the UNIX Programmer's Manual—Volume 1: Commands and Utilities].

Then when executed, the effective user ID is the user ID of the administrator. This program's function is to invoke the desired SCCS command and to cause it to inherit the privileges of the SCCS administrator for the duration of that command's execution. In this manner, the owner of an SCCS file (the administrator) can modify it at will. Other users whose login names are in the user list for that file (but who are not its owners) are given the necessary permissions only for the duration of the execution of the interface program. They are thus able to modify the SCCS files only through the use of **delta** and, possibly, **rmdel** and **cdc**.

C. Basic Program

When a UNIX system program is executed, the program is passed as argument 0, which is the name that invoked the program, and followed by any additional user-supplied arguments. Thus, if a program is given a number of links (names), the program may alter its processing depending upon which link invokes the program. This mechanism is used by an SCCS interface program to determine the SCCS command it should subsequently invoke [see **exec**(2) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*].

D. Linking and Use

In general, the following demonstrates the steps to be performed by the SCCS administrator to create the SCCS interface program. It is assumed, for the purposes of the discussion, that the interface program **inter.c** resides in directory "/x1/xyz/sccs". Thus, the command sequence

```
cd /x1/xyz/sccs
cc ... inter.c -o inter ...
```

compiles **inter.c** to produce the executable module **inter** (the "..." represents other arguments that may be required). The proper mode and the "set user ID on execution" bit are set by executing

```
chmod 4755 inter
```

For example, new links are created by

```
ln inter get
ln inter delta
ln inter rmdel
```

The names of the links may be arbitrary if the interface program is able to determine from them the names of SCCS commands to be invoked. Subsequently, any user whose shell parameter PATH [see **sh**(1) in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*] specifies directory "/x1/xyz/sccs" as the one to be searched first for executable commands may execute

```
get −e /x1/xyz/sccs/s.abc
```

from any directory to invoke the interface program (via its link "get"). The interface program then executes "/usr/bin/get" (the actual SCCS **get** command) upon the named file. As previously mentioned, the interface program could be used to supply the pathname "/x1/xyz/sccs" so that the user would only have to specify

```
get −e s.abc
```

to achieve the same results.

THE M4 MACRO PROCESSOR

GENERAL

The M4 macro processor is a front end for rational Fortran (Ratfor) and the C programming languages. The "#define" statement in C language and the analogous "define" in Ratfor are examples of the basic facility provided by any macro processor.

At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The compiler will then replace later unquoted occurrences of the symbolic name with the corresponding string. Besides the straightforward replacement of one string of text by another, the M4 macro processor provides the following features:

- arguments
- arithmetic capabilities
- file manipulation
- conditional macro expansion
- string and substring functions.

The basic operation of M4 is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The user also has the capability to define new macros. Built-ins and user-defined macros work exactly the same way except that some of the built-in macros have side effects on the state of the process. A list of 21 built-in macros provided by the M4 macro processor can be found in Figure 5.

Figure 5

Built-in Macros (Sheet 1 of 4).

Macro Name	Function
changequote	Restores original characters or makes new quote characters the left and right brackets.
changescom	Changes left and right comment markers from the default # and new line.
deer	Returns the value of its argument decremented by 1.
define	Defines new macros.
defn	Returns the quoted definition of its argument(s).
divert	Diverts output to 1-out-of-10 diversions.

Built-in Macros (Sheet 2 of 4).

Macro Name	Function
divnum	Returns the number of the currently active diversion.
dnl	Reads and discards characters up to and including the next new line.
dumpdef	Dumps the current names and definitions of items named as arguments.
errprint	Prints its arguments on the standard error file.
eval	Prints arbitrary arithmetic on integers.
ifdef	Determines if a macro is currently defined.
ifelse	Performs arbitrary conditional testing.
include	Returns the contents of the file named in the argument. A fatal error occurs if the file name cannot be accessed.

Built-in Macros (Sheet 3 of 4).

Macro Name	Function
iner	Returns the value of its argument incremented by 1.
index	Returns the position where the second argument begins in the first argument pf index.
len	Returns the number of characters that makes its argument.
m4exit	Causes immediate exit from M4.
m4wrap	Pushes the exit code back at final EOF.
maketemp	Facilitates making unique file names.
popdef	Removes current definition of its argument(s) exposing any previous definitions.
pushdef	Defines new macros but saves any previous definition.

Built-in Macros (Sheet 4 of 4).

Macro Name	Function
shift	Returns all arguments of shift except the first argument.
sinclude	Returns the contents of the file named in the arguments. The macro remains silent and continues if the file is inaccessible.
substr	Produces substrings of strings.
syscmd	Executes the UNIX System command given in the first argument.
traceoff	Turns macro trace off.
traceon	Turns the macro trace on.
translit	Performs character transliteration.
undefine	Removes user-defined or built-in macro definitions.
undivert	Discards the diverted text.

To use the M4 macro processor, input the following command:

m4 [optional files]

Each argument file is processed in order. If there are no arguments or if an argument is "–", the standard input is read at that point. The processed text is written on the standard output which may be captured for subsequent

processing with the following input:

```
m4 [files] >outputfile
```

DEFINING MACROS

The primary built-in function of M4 is **define**. **Define** is used to define new macros. The following input:

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter (the underscore counts as a letter). *Stuff* is any text that contains balanced parentheses. Use of a slash may stretch *stuff* over multiple lines. Thus, as a typical example,

```
define(N, 100)
...
if (i > N)
```

defines *N* to be 100 and uses the symbolic constant *N* in a later **if** statement.

The left parenthesis must immediately follow the word **define** to signal that **define** has arguments. If a user-defined macro or built-in name is not followed immediately by "(", it is assumed to have no arguments. Macro calls have the following general form:

```
name(arg1,arg2,...argn)
```

A macro name is only recognized as such if it appears surrounded by nonalphanumerics. Using the following example:

```
define(N, 100)
...
if (NNN > 100)
```

the variable *NNN* is absolutely unrelated to the defined macro *N* even though the variable contains a lot of *N*s.

Macros may be defined in terms of other names. For example,

```
define(N, 100)
define(M, N)
```

defines both *M* and *N* to be 100. If *N* is redefined and subsequently changes, *M* retains the value of 100 not *N*.

The M4 macro processor expands macro names into their defining text as soon as possible. The string *N* is immediately replaced by 100. Then the string *M* is also immediately replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now *M* is defined to be the string *N*, so when the value of *M* is requested later, the result is the value of *N* at that time (because the *M* will be replaced by *N* which will be replaced by 100).

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off. The value of a quoted string is the string stripped of the quotes. If the input is

```
define(N, 100)
define(M, 'N')
```

the quotes around the *N* are stripped off as the argument is being collected. The results of using quotes is to define *M* as the string *N*, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
'define' = 1;
```

Another example of using quotes is redefining *N*. To redefine *N*, the evaluation must be delayed by quoting

```
define(N, 100)
...
define('N', 200)
```

In M4, it is often wise to quote the first argument of a macro. The following example will not redefine *N*:

```
define(N, 100)
...
define(N, 200)
```

The *N* in the second definition is replaced by 100. The result is equivalent to the following statement:

```
define(100, 200)
```

This statement is ignored by M4 since only things that look like names can be defined.

If left and right single quotes are not convenient for some reason, the quote characters can be changed with the following built-in macro:

```
changequote([, ])
```

The built-in **changequote** makes the new quote characters the left and right brackets. The original characters can be restored by using **changequote** without arguments as follows:

```
changequote
```

There are two additional built-ins related to **define**. The **undefine** macro removes the definition of some macro or built-in as follows:

```
undefine('N')
```

The macro removes the definition of *N*. Built-ins can be removed with **undefine**, as follows:

```
undefine('define')
```

But once removed, the definition cannot be reused.

The built-in **ifdef** provides a way to determine if a macro is currently defined. Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('pdp11', 'define(wordsize,16)')
ifdef('u3b', 'define(wordsize,32)')
```

Remember to use the quotes.

The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not

defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null. If the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef('unix', on UNIX, not on UNIX)
```

ARGUMENTS

So far the simplest form of macro processing has been discussed which is replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of **$n** is replaced by the **n**th argument when the macro is actually used. Thus, the macro **bump** defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1. The 'bump(x)' statement is equivalent to 'x = x + 1.'

A macro can have as many arguments as needed, but only the first nine are accessible (**$1** through **$9**). The macro name is **$0** although that is less commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, 'cat(x, y, z)' is equivalent to 'xyz'. Arguments **$4** through **$9** are null since no corresponding arguments were provided. Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a,   b   c)
```

defines 'a' to be 'b c'.

Arguments are separated by commas; however, when commas are within parentheses, the argument is not terminated nor separated. For example,

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second is literally **(b,c)**. A bare comma or parenthesis can be inserted by quoting it.

ARITHMETIC BUILT-INS

The M4 provides three built-in functions for doing arithmetic on integers (only). The simplest is **incr** which increments its numeric argument by 1. The built-in **decr** decrements by 1. Thus to handle the common programming situation where a variable is to be defined as "one more than *N*", use the following:

```
define(N, 100)
define(N1, 'incr(N)')
```

Then *N1* is defined as one more than the current value of *N*.

The more general mechanism for arithmetic is a built-in called **eval** which is capable of arbitrary arithmetic on integers. The operators in decreasing order of precedence are

```
unary + and -
** or ^ (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
!       (not)
& or &&   (logical and)
| or ||   (logical or).
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1 and false is 0. The precision in **eval** is 32 bits under the UNIX operating system.

As a simple example, define *M* to be "2==N+1" using **eval** as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

The defining text for a macro should be quoted unless the text is very simple. Quoting the defining text usually gives the desired result and is a good habit to get into.

FILE MANIPULATION

A new file can be included in the input at any time by the built-in function **include**. For example,

```
include(filename)
```

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (**include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc.

A fatal error occurs if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used. The built-in **sinclude** (silent include) says nothing and continues if the file named cannot be accessed.

The output of M4 can be diverted to temporary files during processing, and the collected material can be output upon command. The M4 maintains nine of these diversions, numbered 1 through 9. If the built-in macro

```
divert(n)
```

is used, all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by the **divert** or **divert(0)** command which resumes the normal output process.

Diverted text is normally output all at once at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded. The built-in **undivert** brings back all diversions in numerical order. The built-in **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted text (as does diverting) into a diversion whose number is not between 0 and 9, inclusive.

The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros. The built-in **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

SYSTEM COMMAND

Any program in the local operating system can be run by using the **syscmd** built-in. For example,

```
syscmd(date)
```

on the UNIX system runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**. To facilitate making unique file names, the built-in **maketemp** is provided with specifications identical to the system function *mktemp*. The **maketemp** macro fills in a string of XXXXX in the argument with the process id of the current process.

CONDITIONALS

Arbitrary conditional testing is performed via built-in **ifelse**. In the simplest form

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, **ifelse** returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called **compare** can be defined as one which compares two strings and returns "yes" or "no" if they are the same or different as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes which prevents evaluation of **ifelse** occurring too early. If the fourth argument is missing, it is treated as empty.

The built-in **ifelse** can actually have any number of arguments and provides a limited form of multiway decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

STRING MANIPULATION

The built-in **len** returns the length of the string (number of characters) that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and **len((a,b))** is 5.

The built-in **substr** can be used to produce substrings of strings. Using input, **substr(s, i, n)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. Inputting

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time.
```

If *i* or *n* are out of range, various actions occur.

The built-in **index(s1, s2)** returns the index (position) in *s1* where the string *s2* occurs or −1 if it does not occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration and has the general form

```
translit(s, f, t)
```

which modifies *s* by replacing any character found in *f* by the corresponding character of *t*. Using input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So

```
translit(s, aeiou)
```

would delete vowels from *s*.

There is also a built-in called **dnl** that deletes all characters that follow it up to and including the next new line. The **dnl** macro is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. Using input

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. So the new line is copied into the output where it may not be wanted. If the built-in **dnl** is added to each of these lines, the newlines will disappear. Another method of achieving the same results is to input

```
divert(-1)
define(...)
...
divert.
```

PRINTING

The built-in **errprint** writes its arguments out on the standard error file. An example would be

```
errprint('fatal error')
```

The built-in **dumpdef** is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.

NOTES

THE awk PROGRAMMING LANGUAGE

GENERAL

The *awk* is a file-processing programming language designed to make many common information and retrieval text manipulation tasks easy to state and perform. The *awk:*

- Generates reports
- Matches patterns
- Validates data
- Filters data for transmission.

PROGRAM STRUCTURE

The *awk* program is a sequence of statements of the form

```
pattern {action}
pattern {action}
  ...
```

The *awk* program is run on a set of input files. The basic operation of *awk* is to scan a set of input lines, in order, one at a time. In each line, *awk* searches for the pattern described in the *awk* program, then if that pattern is found in the input line, a corresponding action is performed. In this way, each statement of the *awk* program is executed for a given input line. When all the patterns are tested, the next input line is fetched; and the *awk* program is once again executed from the beginning.

In the *awk* command, either the pattern or the action is omitted, but not both. If there is no action for a pattern, the matching line is simply printed. If there is no pattern for an action, then the action is performed for every input line.

The null *awk* program does nothing. Since patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

For example, this *awk* program

```
/x/      {print}
```

prints every input line that has an "x" in it.

An *awk* program has the following structure:

- a <BEGIN> section
- a <record> or main section
- an <END> section.

The <BEGIN> section is run before any input lines are read, and the <END> section is run after all the data files are processed. The <record> section is data driven. That is, it is the section that is run over and over for each separate line of input.

Values are assigned to variables from the *awk* command line. The <BEGIN> section is run before these assignments are made.

The words "BEGIN" and "END" are actually patterns recognized by *awk*. These are discussed further in the pattern section of this guide.

LEXICAL CONVENTION

All *awk* programs are made up of lexical units called tokens. In *awk* there are eight token types:

1. *numeric constants*
2. *string constants*
3. *keywords*
4. *identifiers*
5. *operators*
6. *record and file tokens*
7. *comments*
8. *separators.*

Numeric Constants

A *numeric constant* is either a decimal constant or a floating constant. A decimal constant is a nonnull sequence of digits containing at most one decimal point as in **12, 12., 1.2,** and **.12**. A floating constant is a decimal constant followed by e or E followed by an optional + or − sign followed by a nonnull sequence of digits as in **12e3, 1.2e3, 1.2e−3,** and **1.2E+3**. The maximum size and precision of a numeric constant are machine dependent.

String Constants

A *string constant* is a sequence of zero or more characters surrounded by double quotes as in **","** **"a", "ab", and "12"**. A double quote is put in a string by proceeding it with \ as in "He said, \ Sit! \"". A newline is put in a string by using \n in its place. No other characters need to be escaped. Strings can be (almost) any length.

Keywords

Strings used as keywords are shown in Figure 6.

Figure 6

Strings Used As Keywords.

Keywords		
begin	break	length
end	close	log
FILENAME	continue	next
FS	close	number
NF	exit	print
NR	exp	printf
OFS	for	split
ORS	getline	sprintf
OFMT	if	sqrt
RS	in	string
	index	substr
	int	while

Identifiers

Identifiers in *awk* serve to denote variables and arrays. An identifier is a sequence of letters, digits, and underscores, beginning with a letter or an underscore. Uppercase and lowercase letters are different.

Operators

The *awk* has assignment, arithmetic, relational, and logical operators similar to those in the C programming language and regular expression pattern matching operators similar to those in the UNIX operating system program *egrep* and *lex*.

Assignment operators are shown in Figure 7.

Figure 7

Symbols and Descriptions for Assignment Operators.

Assignment Operators		
Symbol	**Usage**	**Description**
=	assignment	
+=	plus-equals	X += Y is similar to X = X+Y
-=	minus-equals	X-=Y is similar to X = X-Y
*=	times-equals	X *= Y is similar to X = X*Y
/=	divide-equals	X = Y is similar to X = X/Y
%=	mod-equals	X %= Y is similar to X = X%Y
++	prefix and postfix increments	++X and FBX++ are similar to X=X+1
—	prefix and postfix decrements	— and X similar to X = X − 1

Arithmetic operators are shown in Figure 8.

Figure 8

Symbols and Descriptions for Arithmetic Operators.

Arithmetic Operators	
Symbol .R	**Description**
+	unary binary plus
−	unary and binary minus
*	multiplication
/	division
%	modulus
(...)	grouping

Relational operators are shown in Figure 9.

Figure 9

Symbols and Descriptions for Relational Operators.

Relational Operators	
Symbol	**Description**
<	less than
< =	less than or equal to
= =	equal to
!=	not equal to
> =	greater than or equal to
>	greater than

Logical operators are shown in Figure 10.

Figure 10

Symbols and Descriptions for Logical Operators.

Logical Operators	
Symbol	**Description**
& & ! ! !	and or not

Regular expression matching operators are shown in the Figure 11.

Figure 11

Symbols and Descriptions for Regular Expression Pattern.

Regular Expression Pattern Matching Operators	
Symbol	**Description**
~ !~	matches does not match

Record and Field Tokens

The **$0** is a special variable whose value is that of the current input record. The **$1, $2...** are special variables whose values are those of the first field, the second field , . . . , respectively, of the current input record. The keyword **NF** (Number of Fields) is a special variable whose value is the number of fields in the current input records. Thus **$NF** has, as its value, the value of the last field of the current input records. Notice that the field of each record is numbered 1 and that the number of fields can vary from record to record.

None of these variables is defined in the action associated with a BEGIN or END pattern, where there is no current input record.

The keyword **NR** (Number of Records) is a variable whose value is the number of input records read so far. The first input record read is **1**.

Record Separators

The keyword **RS** (Record Separators) is a variable whose value is the current record separator. The value of **RS** is initially set to newline, indicating that adjacent input records are separated by a newline. Keyword **RS** is changed to any character **c** by including the assignment statement **RS** = "**c**" in an action.

Field Separator

The keyword **FS** (Field Separator) is a variable indicating the current field separator. Initially, the value of **FS** is a blank, indicating that fields are separated by white space, i.e., any nonnull sequence of blanks and tabs. Keyword **FS** is changed to any single character **c** by including the assignment statement **F** = "**c**" in an action or by using the optional command line argument **−Fc**. Two values of c have special meaning, **space** and **t**. The assignment statement **FS** = " " makes white space in field separator; and on the command line, **−Ft** makes tab the field separator.

If the field operator is not a blank, then there is a field in the record on each side of the separator. For instance, if the field separator is 1, the record **1XXX1** has three fields. The first and last are null. If the field separator is blank, then fields are separated by white space, and none of the **NF** fields are null.

Multiline Records

The assignment **RS** = " " makes an empty line the record separator and makes a nonnull sequence (consisting of blanks, tabs, and possibly a newline) the field separator. With this setting, none of the first **NF** fields of any record are null.

Output Record and Field Separators

The value of **OFS** (Output Field Separator) is the output field separator. It is put between fields by print. The value of **ORS** (Output Record Separators) is put after each record by print. Initially , **ORS** is set to a newline and **OFS** to a space. These values may change to any string by assignments such as **ORS = "abc"** and **OFS = "xyz"**.

Comments

A comment is introduced by a # and terminated by a newline. For example:

```
# part of the line is a comment
```

A comment can be appended to the end of any line of an *awk* program.

Separators and Brackets

Tokens in *awk* are usually separated by nonnull sequences of blank, tabs, and newlines, or by other punctuation symbols such as commas and semicolons. Braces {...} surround actions, slashes /.../ surround regular expression patterns, and double quotes "..." surround strings.

PRIMARY EXPRESSIONS

In *awk,* patterns and actions are made up of expressions. The basic building blocks of expressions are the *primary expressions*:

numeric constants
string constant
var
function

Each expression has both a numeric and a string value, one of which is usually preferred. The rules for determining the preferred value of an expression are explained below.

Numeric Constants

The format of a numeric constant was defined previously in **LEXICAL CONVENTIONS**. Numeric values are stored as floating point numbers. Both the numeric and string value of a numeric constant is the decimal number represented by the constant. The preferred value is the numeric value. Numeric values for string constants are in Figure 12.

Figure 12

Numeric Values for String Constants.

Numeric Constants		
Numeric Constant	**Numeric Value**	**String Value**
0	0	0
1	1	1
.5	0.5	.5
.5e2	50	50

String Constants

The format of a string constant was defined previously in **LEXICAL CONVENTIONS**. The numeric value of a string constant is **0** unless the string is a numeric constant enclosed in double quotes. In this case, the numeric value is the number represented. The preferred value of a string constant is its string value. The string value of a string constant is always the string itself. String values for string constants are in Figure 13.

Vars

A *var* is one of the following:

identifier
identifier{expression}
$term

Figure 13

String Values for String Constants.

String Constants		
String Constant	**Numeric Value**	**String Value**
""	0	empty space
"a"	0	a
"XYZ"	0	xyz
"0"	0	0
"1"	1	1
".5"	0.5	.5
".5e2"	0.5	.5e2a

The numeric value of any uninitialized *var* is **0**, and the string value is the empty string.

An *identifier* by itself is a simple variable. A *var* of the form *identifier* {expression} represents an element of an associative array named by *identifier*. The string value of *expression* is used as the index into the array. The preferred value of *identifier* or *identifier* {*expression*} is determined by context.

The *var* **$0** refers to the current input record. Its string and numeric values are those of the current input record. If the current input record represents a number, then the numeric value of **$0** is the number and the string value is the literal string. The preferred value of **$0** is string unless the current input record is a number. The **$0** cannot be changed by assignment.

The *var* **$1, $2, . . .** refer to fields 1, 2, . . . of the current input record. The string and numeric value of **$i** for **1<=i<=NF** are those of the ith field of the current input record. As with **$0**, if the ith field represents a number, then the numeric value of **$i** is the number and the string value is the literal string. The preferred value of **$i** is string unless the ith field is a number. The **$i** is changed by assignment. The **$0** is then changed accordingly.

In general, *$term* refers to the input record if *term* has the numeric value 0 and to field i if the greatest integer in the numeric value of *term* is i. If **i<0** or if **i>=100**, then accessing **$i** causes *awk* to produce an error diagnostic. If **NF<i<=100**, then **$i** behaves like an uninitialized *var*. Accessing **$i** for **i > NF** does not change the value of **NF**.

Function

The *awk* has a number of built-in functions that perform common arithmetic and string operations. The arithmetic functions are in Figure 14.

Figure 14

Built-in Functions for Arithmetic and String Operations.

Functions	
exp	*(expression)*
int	*(expression)*
log	*(expression)*
sqrt	*(expression)*

These functions (exp, int, log, and sqrt) compute the exponential, integer part, natural logarithm, and square root, respectively, of the numeric value of *expression*. The *(expression)* may be omitted; then the function is applied to **$0**. The preferred value of an arithmetic function is numeric. String functions are shown in Figure 15.

The function getline causes the next input record to replace the current record. It returns 1 if there is a next input record or a 0 if there is no next input record. The value of **NR** is updated.

The function index (e1,e2) takes the string value of expressions e1 and e2 and returns the first position of where e2 occurs as a substring in e1. If e2 does not occur in e1, index returns 0. For example, **index ("abc", "bc") =2** and **index ("abc", "ac") =0**.

Figure 15

Expressions for String Functions.

String Functions	
getline	
index	(expression1, expression2)
length	(expression)
split	(expression, identifier, expression2)
split	(expression, identifier)
sprintf	(format, expression1, expression2...)
substr	(expression1, expression2)
substr	(expression1, expression2, expression3)

The function length without an argument returns the number of characters in the current input record. With an expression argument, length (e) returns the number of characters in the string value of e. For example, **length ("abc") = 3** and **length (17) = 2**.

The function split *(e array, sep)* splits the string value of expression e into fields that are then stored in *array [1], array [2],..., array [n]* using the string value of *sep* as the field separator. Split returns the number of fields found in *e*. The function split *(e, array)* uses the current value of **FS** to indicate the field separator. For example, after invoking n = split (**$0**), **a**[1], **a**[2,..., **a**[n] is the same sequence of values as **$l, $2 . . ., $NF**.

The function splitf (**f, e1, e2 . . .**) produces the value of expressions **e1, e2 . . .** in the format specified by the string value of the expression **f**. The format control conventions are those of the printf statement in the C programming language **[KR]**.

The function substr *(string, pos)* returns the suffix of *string* starting at position *pos*. The function substr *(string, pos, length)* returns the substring of *string* that begins at position *pos* and is *length* characters long. If pos + length is greater than the length of *string* then substr *(string, pos, length)* is equivalent to substr *(string, pos)*. For example, substr (**"abc", 2, 1**) = **"b"**, substr (**"abc", 2, 2**) = **"bc"**, and subtr (**"abc", 2, 3**) = **"bc"**. Positions less than 1 are taken as 1. A negative or zero length produces a null result.

The preferred value of sprintf and substr is string. The preferred value of the remaining string functions is numeric.

TERMS

Various arithmetic operators are applied to primary expressions to produce larger syntactic units called *terms*. All arithmetic is done in floating point. A term has one of the following forms:

primary expression
term binop term
unop term
incremented var
(term)

Binary Terms

In a *term* of the form

term1
binop
term2

binop can be one of the five binary arithmetic operators +, −, * (multiplication), /(division), % (modulus). The binary operator is applied to the numeric value of the operand *term1* and *term2,* and the result is the usual numeric value. This numeric value is the preferred value, but it can be interpreted as a string value (see **Numeric Constants**). The operators * , /, and % have higher precedence than + and −. All operators are left associative.

Unary Term

In a *term* of the form

unop term

unop can be unary + or −. The unary operator is applied to the numeric value of *term,* and the result is the usual numeric value which is preferred. However,

it can be interpreted as a string value. Unary + and − have higher precedence than *, /, and %

Incremented Vars

An *incremented var* has one of the forms

+ + var
− − var
var + +
var − −

The *+ + var* has the value *var + 1* and has the effect of *var = var + 1*. Similarly, *− − var* has the value *var − 1* and has the effect of *var = var − 1*. Therefore, *var + +* has the same value as *var* and has the effect of *var = var + 1*. Similarly, *var − −* has the same value as *var* and has the effect of *var = var − 1*. The preferred value of an *incremented var* is numeric.

Parenthesized Terms

Parentheses are used to group terms in the usual manner.

EXPRESSIONS

awk expression is one of the following:

term
term term ...
var asgnop expression

Concatenation of Terms

In an expression of the form *term1 term2 ...*, the string value of the terms are concatenated. The preferred value of the resulting expression is a string value that can be interpreted as a numeric value. Concatenation of terms has lower precedence than binary + and −. For example, 1+2 3=4 has the string (and numeric) value 37.

Assignment Expressions

An *assignment expression* is one of the forms

var asgnop expression

where *asgnop* is one of the six assignment operators:

=
+=
−=
*=
/=
%=

The preferred value of *var* is the same as that of *expression.*

In an expression of the form

var = expression

the numeric and string value of *var* becomes those of *expression.*

var op = expression

is equivalent to

var = var op expression

where *op* is one of; +, −, *, /, %. The *asgnops* are right associative and have the lowest precedence of any operator. Thus, a += b *= c−2 is equivalent to the sequence of assignments

```
b = b * (0−2)
a = a+2
```

USING *awk*

There are two ways in which to present your *awk* program of pattern-action statements to *awk* for processing:

1. If the program is short (a line or two), it is often easiest to make the program the first argument on the command line:

   awk ' program ' files

   where "files" is an optional list of input files and "program" is your *awk* program. Note that there are single quotes around the program in order for the shell to accept the entire string (program) as the first argument to *awk*. For example, write to the shell

   awk ' /x/ {print } ' files

   to run the *awk* script /x/ {print} on the input file "files". If no input files are specified, *awk* takes input from the standard input **stdin**. You can also specify that input comes from **stdin** by using "-" (the hyphen) as one of the files. The pattern-action statement

   awk 'program' files -

   looks for input from "files" and from **stdin** and processes first from "files" and then from **stdin**.

2. Alternately, if your *awk* program is long, it is more convenient to put the program in a separate file, awkprog, and tell *awk* to fetch it from there. This is done by using the "–f" option after the *awk* command as follows:

   awk –f awkprog files

   where "files" is an optional list of input files that may include **stdin** as is indicated by a hyphen (-).

For example:

```
awk ' BEGIN {
                    print "hello, world"
                    exit
            }
```

prints

```
hello, world
```

on the standard output when given to the shell. Recall that the word "BEGIN" is a special pattern indicating that the action following in braces is run before any data is read. Words "print" and "exit" are both discussed in later sections.

This *awk* program could be run by putting

```
BEGIN {
        print "hello, world"
        exit
        }
```

in a file named awkprog , and then the command

```
awk −f awkprog
```

given to the shell. This would have the same effect as the first procedure.

INPUT: RECORDS AND FIELDS

The *awk* reads its input one record at a time unless changed by you. A record is a sequence of characters from the input ending with a newline character or with an end of file. Thus, a record is a line of input. The *awk* program reads in characters until it encounters a newline or end of file. The string of characters, thus read, is assigned to the variable **$0**. You can change the character that indicates the end of a record by assigning a new character to the special variable **RS** (the record separator). Assignment of values to variables and these special variables such as **RS** are discussed later.

Once *awk* has read in a record, it then splits the record into "fields". A field is a string of characters separated by blanks or tabs, unless you specify otherwise. You may change field separators from blanks or tabs to whatever characters you choose in the same way that record separators are changed. That is, the special variable FS is assigned a different value.

As an example, let us suppose that the file "countries" contains the area in thousands of square miles, the population in millions, and the continent for the ten largest countries in the world. (Figures are from 1978; Russia is placed in Asia.)

Sample Input File "countries":

Russia	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	68	14	Australia
India	1269	637	Asia
Argentina	72	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa

The wide spaces are tabs in the original input and a single blank separates North and South from America. We use this data as the input for many of the *awk* programs in this guide since it is typical of the type of material that *awk* is best at processing (a mixture of words and numbers separated into fields or columns separated by blanks and tabs).

Each of these lines has either four or five fields if blanks and/or tabs separate the fields. This is what *awk* assumes unless told otherwise. In the above example, the first record is

```
Russia 8650   262    Asia
```

When this record is read by *awk*, it is assigned to the variable **$0**. If you want to refer to this entire record, it is done through the variable, **$0**.

For example, the following input:

```
{print $0}
```

prints the entire record. Fields within a record are assigned to the variables $1, $2, $3, and so forth; that is, the first field of the present record is referred to as $1 by the *awk* program. The second field of the present record is referred to as $2 by the *awk* program. The ith field of the present record is referred to as $i by the *awk* program. Thus, in the above example of the file countries, in the first record;

$1 is equal to the string "Russia"
$2 is equal to the integer 8650
$3 is equal to the integer 262
$4 is equal to the string "Asia"
$5 is equal to the null string
... and so forth.

To print the continent, followed by the name of the country, followed by its population, use the following *awk* script:

```
{print $4, $1, $3}
```

Note that *awk* does not require type declarations.

INPUT: FROM THE COMMAND LINE

It is possible to assign values to variables from within an *awk* program. Because you do not declare types of variables, a variable is created simply by referring to it. An example of assigning a value to a variable is:

```
x=5
```

This statement in an *awk* program assigns the value 5 to the variable x. It is also possible to assign values to variables from the command line. This provides another way to supply input values to *awk* programs.

For example

awk ' {print x }' x=5 −

will print the value 5 on the standard output. The minus sign at the end of this command is necessary to indicate that input is coming from **stdin** instead of a file called "x=5". Similarly if the input comes from a file named "file", the command is

awk '{print x}' file

It is *not* possible to assign values to variables used in the BEGIN section in this way.

If it is necessary to change the record separator and the field separator, it is useful to do so from the command line as in the following example:

awk −f awk.program RS=":" file

Here, the record separator is changed to the character ":". This causes your program in the file "awk.program" to run with records separated by the colon instead of the newline character and with input coming from the file, "file". It is similarly useful to change the field separator from the command line.

This operation is so common that there is yet another way to change the field separator from the command line. There is a separate option "−Fx" that is placed directly after the command *awk*. This changes the field separator from blank or tab to the character "x".

For example

awk −F: −f awk.program file

changes the field separator FS to the character ":". Note that if the field separator is specifically set to a tab, (that is, with the −F option or by making a direct assignment to FS) then blanks are recognized by *awk* as separating fields. However, even if the field separator is specifically set to a blank, tabs are STILL recognized by *awk* as separating fields.

An exercise:

Using the input file ("countries" described earlier) write an *awk* script that prints the name of a country followed by the continent that it is on. Do this in such a way that continents composed of two words (e. g., North America) are processed as only one field and not two.

OUTPUT: PRINTING

An action may have no pattern; in this case, the action is executed for all lines as in the simple printing program

```
{print}
```

This is one of the simplest actions performed by *awk*. It prints each line of the input to the output. More useful is to print one or more fields from each line. For instance, using the file "countries", that was used earlier,

```
awk '{ print $1, $3 }' countries
```

prints the name of the country and the population:

```
Russia 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 14
Sudan 19
Algeria 18
```

Note that the use of a semicolon at the end of statements in *awk* programs is optional. *Awk* accepts

```
{print $1 }
```

and

```
{print $1; }
```

equally and takes them to mean the same thing. If you want to put two *awk* statements on the same line of an *awk* script, the semicolon is necessary. For example, the following semicolon is necessary if you want the number 5 printed:

```
{x=5; print x }
```

Parentheses are also optional with the print statement.

```
print $3, $2
```

is the same as

```
print ($3, $2 )
```

Items separated by a comma in a print statement are separated by the current output field separators (normally spaces, even though the input is separated by tabs) when printed. The **OFS** is another special variable that can be changed by you. These special variables are summarized in a later section.

An exercise:

Using the input file, "countries", print the continent followed by the country followed by the population for each input record. Then pipe the output to the UNIX operating system command "sort" so that all countries from a given continent are printed together.

Print also prints strings directly from your programs with the *awk* script

```
{print "hello, world" }
```

from an earlier section.

An exercise:

Print a header to the output of the previous exercise that says "Population of Largest Countries" followed by headers to the columns that follow describing what is in that column, for example, Country or Population.

As we have already seen, *awk* makes available a number of special variables with useful values, for example, **FS** and **RS**. We now introduce another special variable in the next example. **NR** and **NF** are both integers that contain the number of the present record and the number of fields in the present record, respectively. Thus,

```
{print NR, NF, $0}
```

prints each record number and the number of fields in each record followed by the record itself. Using this program on the file, "countries" yields:

```
1 4 Russia      8650   262   Asia
2 5 Canada      3852   24    North America
3 4 China       3692   866   Asia
4 5 USA         3615   219   North America
5 5 Brazil      3286   116   South America
6 4 Australia   2968   14    Australia
7 4 India       1269   637   Asia
8 5 Argentina   1072   26    South America
9 4 Sudan       968    19    Africa
10 4 Algeria    920    18    Africa
```

and the program

```
{print NR, $1 }
```

prints

```
1 Russia
2 Canada
3 China
4 USA
5 Brazil
6 Australia
7 India
8 Argentina
9 Sudan
10 Algeria
```

This is an easy way to supply sequence numbers to a list. Print, by itself, prints the input record. Use

```
print ""
```

to print the empty line.

Awk also provides the statement printf so that you can format output as desired. Print uses the default format "%.6g" for each variable printed.

```
printf format, expr, expr, ...
```

formats the expressions in the list according to the specification in the string, format, and prints them. The format statement is exactly that of the printf in the C library. For example,

```
{ printf "%10s %6d0, $1, $2, $3 }
```

prints $1 as a string of 10 characters (right justified). The second and third fields (6-digit numbers) make a neatly columned table.

```
    Russia   8650   262
    Canada   3852   244
     China   3692   866
       USA   3615   219
    Brazil   3286   116
 Australia   2968    14
     India   1269   637
 Argentina   1072    26
     Sudan    968    19
   Algeria    920    18
```

With printf, no output separators or newlines are produced automatically. You must add them as in this example. In the C library version of printf, the various escape characters "**\n**", "**\t**", "**\b**" (backspace) and "**\r**" (carriage return) are valid with the *awk* printf.

There is a third way that printing can occur on standard output when a pattern is specified but there is no action to go with it. In this case, the entire record **$0** is printed. For example, the program

```
/x/
```

prints any record that contains the character "**x**".

There are two special variables that go with printing, **OFS** and **ORS**. These are by default set to blank and the newline character, respectively. The variable **OFS** is printed on the standard output when a comma occurs in a print statement such as

```
{ x="hello"; y="world"
print x,y
}
```

which prints

```
hello world
```

However, without the comma in the print statement as

```
{ x="hello"; y="world"
print x y
}
```

you get

```
helloworld
```

To get a comma on the output, you can either insert it in the print statement as in this case

```
{ x="hello"; y="world"
print x"," y
}
```

or you can change **OFS** in a BEGIN section as in

```
BEGIN {OFS=", "}
{ x="hello"; y="world"
print x, y
}
```

both of these last two scripts yields

```
hello, world
```

Note that the output field separator is not used when **$0** is printed.

OUTPUT: TO DIFFERENT FILES

The UNIX operating system shell allows you to redirect standard output to a file. The *awk* program also lets you direct output to many different files from within your *awk* program. For example, with our input file "countries", we want to print all the data from countries of Asia in a file called "ASIA", all the data from countries in Africa in a file called "AFRICA", and so forth. This is done with the following *awk* program:

```
{ if ($4 == "Asia") print > "ASIA"
  if ($4 == "Europe") print > "EUROPE"
  if ($4 == "North") print > "NORTH_AMERICA"
  if ($4 == "South") print > "SOUTH_AMERICA"
  if ($4 == "Australia") print > "AUSTRALIA"
  if ($4 == "Africa") print > "AFRICA"
}
```

The flow of control statements (for example, "if") are discussed later.

In general, you may direct output into a file after a print or a printf statement by using a statement of the form

```
print > "FILE"
```

where FILE is the name of the file receiving the data, and the print statement may have any legal arguments to it.

Notice that the file names are quoted. Without quotes, the file names are treated as uninitialized variables and all output then goes to the same file.

If > is replaced by >>, output is appended to the file rather than overwriting it.

Users should also note that there is an upper limit to the number of files that are written in this way. At present it is ten.

OUTPUT: TO PIPES

It is also possible to direct printing into a pipe instead of a file. For example,

```
{
        if ($2 == "XX") print | "mail mary"
}
```

where "mary" is someone's login name, any record is sent (with the second field equal to "XX") to the user, mary, as mail. Awk waits until the entire program is run before it executes the command that was piped to, in this case the "mail" command.

For example:

```
{
print $1 | "sort"
}
```

takes the first field of each input record, sorts these fields, and then prints them. The command in parentheses is any UNIX operating system command.

An exercise:

Write an *awk* script that uses the input file to

- List countries that were used previously
- Print the name of the countries
- Print the population of each country
- Sort the data so that countries with the largest population appear first
- Mail the resulting list to yourself.

Another example of using a pipe for output is the following idiom which guarantees that its output always goes to your terminal:

```
print ... | "cat -u > /dev/tty"
```

Only one output statement to a pipe is permitted in an *awk* program. In all output statements involving redirection of output, the files or pipes are identified by their names but they are created and opened only once in the entire run.

COMMENTS

Comments are placed in *awk* programs; they begin with the character # and end with the end of the line as in

```
print x, Y      # this is a comment
```

PATTERNS

A pattern in front of an action acts as a selector that determines if the action is to be executed. A variety of expressions are used as patterns:

- Regular expressions
- Arithmetic relational expressions
- String valued expressions
- Combinations of these.

BEGIN and END

The special pattern, BEGIN, matches the beginning of the input before the first record is read. The pattern, END, matches the end of the input after the last line is processed. BEGIN and END thus provide a way to gain control before and after processing for initialization and wrapping up.

An example:

As you have seen, you can use BEGIN to put column headings on the output

```
BEGIN {print "Country", "Area", "Population", "Continent"}
      {print}
```

which produces

Country Area Population Continent

Russia	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	2968	14	Australia
India	1269	637	Asia
Argentina		1072	26South America
Sudan	968	19	Africa
Algeria	920	18	Africa

Formatting is not very good here; printf would do a better job and is usually mandatory if you really care about appearance.

Recall also, that the BEGIN section is a good place to change special variables such as **FS** or **RS**.

Example:

```
BEGIN { FS= "   "
      print "Countries", "Area", "Population", "Continent"
      }
      {print}
END     {print "The number of records is", NR}
```

In this program, **FS** is set to a tab in the BEGIN section and as a result all records (in the file countries) have exactly four fields.

Note that if BEGIN is present it is the first pattern; END is the last if it is used.

Relational Expressions

An *awk* pattern is any expression involving comparisons between strings of characters or numbers. For example, if you want to print only countries with more than 100 million population, use

```
$3 >100
```

This tiny *awk* program is a pattern without an action so it prints each line whose third field is greater than 100 as follows:

Russia	8650	262	Asia
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
India	1269	637	Asia

To print the names of the countries that are in Asia, type

```
$4 == "Asia" {print $1}
```

which produces

```
Russia
China
India
```

The conditions tested are <, <=, ==, !=, >=, and >. In such relational tests if both operands are numeric, a numerical comparison is made. Otherwise, the operands are compared as strings. Thus,

```
$1 >= "S"
```

selects lines that begin with S, T, U, and so forth which in this case is

```
USA    3615   219    North America
Sudan  968    19     Africa
```

In the absence of other information, fields are treated as strings, so the program

```
$1 == $4
```

compares the first and fourth fields as strings of characters and prints the single line

```
Australia      2968      14 Australia
```

If fields appear as numbers, the comparisons are done numerically.

Regular Expressions

Awk provides more powerful capabilities for searching for strings of characters than were illustrated in the previous section. These are regular expressions. The simplest regular expression is a literal string of characters enclosed in slashes.

```
/Asia/
```

This is a complete *awk* program that prints all lines which contain any occurrence of the name "Asia". If a line contains "Asia" as part of a larger word like "Asiatic", it is also printed (but there are no such words in the countries file.)

Awk regular expressions include

- Regular expression forms found in the text editor
- **ed** and the pattern finder
- *grep* in which certain characters have special meanings.

For example, we could print all lines that begin with A with

```
/^A/
```

or all lines that begin with A, B, or C with

```
/^[ABC]/
```

or all lines that end with "ia" with

```
/ia$/
```

In general, the circumflex (^) indicates the beginning of a line. The dollar sign ($) indicates the end of the line and characters enclosed in brackets ,{}, match any one of the characters enclosed. In addition, *awk* allows parentheses for grouping, the pipe (|) for alternatives, + for "one or more" occurrences, and ? for "zero or one" occurrences. For example,

```
/x|y/  {print}
```

prints all records that contain either an "x" or a "y".

```
/ax+b/        {print}
```

prints all records that contain an "a" followed by one or more "x's" followed by a "b". For example, axb, Paxxxxxxxb, QaxxbR.

```
/ax?b/        {print}
```

prints all records that contain an "a" followed by zero or one "x" followed by a "b". For example: ab, axb, yaxbPPP, CabD.

The two characters "." and "*" have the same meaning as they have in *ed*: namely, "." can stand for any character and "*" means zero or more occurrences of the character preceding it. For example,

```
/a.b/
```

matches any record that contains an "a" followed by any character followed by a "b". That is, the record must contain an "a" and a "b" separated by exactly one character. For example, /a.b/ matches axb, aPb and xxxxaXbxx, but NOT ab, axxb.

```
/ab*c/
```

matches a record that contains an "a" followed by zero or more "b"'s followed by a "c". For example, it matches

```
ac
abc
pqrabbbbbbbbbc901
```

Just as in *ed,* it is possible to turn off the special meaning of these metacharacters such as "^" and "*" by preceding these characters with a backslash. An example of this is the pattern

```
//.*//
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) by using the operators ~ or !~. For example, with the input file countries as before, the program

```
$1 ~ /ia$/      {print $1}
```

prints all countries whose name ends in "ia":

```
Russia
Australia
India
Algeria
```

that is indeed different from *lines* which end in "ia".

Combinations of Patterns

A pattern is made up of similar patterns combined with the operators || (OR), && (AND), ! (NOT), and parentheses. For example,

```
$2 >= 3000 && $3 >=100
```

selects lines where both area AND population are large. For example,

Russia	8650	262	Asia
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America

while

```
$4 == "Asia" || $4 == "Africa"
```

selects lines with Asia or Africa as the fourth field. An alternate way to write this last expression is with a regular expression:

```
$1 ~ /^(Asia|Africa))$/
```

&& and || guarantee that their operands are evaluated from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma as in

```
pattern1, pattern2      { ... }
```

In this case, the action is performed for each line between an occurrence of pattern1 and the next occurrence of pattern2 (inclusive). As an example with no action

```
/Canada/,/Brazil/
```

prints all lines between the one containing "Canada" and the line containing "Brazil". For example,

```
Canada    3852    24     North America
China     3692    866    Asia
USA       3615    219    North America
Brazil    3286    116    South America
```

while

```
NR == 2, NR == 5 { ... }
```

does the action for lines 2 through 5 of the input. Different types of patterns are mixed as in

```
/Canada/, $4 == "Africa"
```

and prints all lines from the first line containing "Canada" up to and including the next record whose fourth field is "Africa".

Users should note that patterns in this form occur OUTSIDE of the action parts of the *awk* programs (outside of the braces that define *awk* actions). If you need to check patterns inside an awk action (inside the braces), use a flow

of control statement such as an "if" statement or a "while" statement. Flow of control statements are discussed in the part "BUILT-IN FUNCTIONS".

ACTIONS

An *awk* action is a sequence of action statements separated by newlines or semicolons. These action statements do a variety of bookkeeping and string manipulating tasks.

Variables, Expressions, and Assignments

The *awk* provides the ability to do arithmetic and to store the results in variables for later use in the program. However, variables can also store strings of characters. You cannot do arithmetic on character strings, but you can stick them together and pull them apart as shown. As an example, consider printing the population density for each country in the file countries.

```
{print $1, (1000000 * $3)/($2 * 1000) }
```

(Recall that in this file the population is in millions and the area in thousands.) The result is population density in people per square mile.

```
Russia 30.289
Canada 6.23053
China 234.561
USA 60.5809
Brazil 35.3013
Australia 4.71698
India 501.97
Argentina 24.2537
Sudan 19.6281
Algeria 19.5652
```

The formatting is bad; so using printf instead gives the program

```
{printf "%10s %6.1f0, $1, (1000000 * $3)/($2 * 1000) }
```

and the output

Russia	30.3
Canada	6.2
China	234.6
USA	60.6
Brazil	35.3
Australia	4.7
India	502.0
Argentina	24.3
Sudan	19.6
Algeria	19.6

Arithmetic is done internally in floating point. The arithmetic operators are +. −, *, / and % (mod or remainder).

To compute the total population and number of countries from Asia, we could write

```
/Asia/    { pop = pop + $3; n = n + 1 }
END       {print "total population of", n, "Asian countries is", pop }
```

which produces total population of three Asian countries is 1765.

Actually, no experienced programmer would write

```
{pop = pop + $3; n = n + 1 }
```

since both assignments are written more clearly and concisely. The better way is

```
{pop += $3; ++n }
```

Indeed, these operators, ++, −−, −=, /=, * =, +=, and %= are available in *awk* as they are in C. Operator x += y has the same effect as x = x + y but

+= is shorter and runs faster. The same is true of the ++ operator; it adds one to the value of a variable. The increment operators ++ and −− (as in C) is used as prefix or as postfix operators. These operators are also used in expressions.

Initialization of Variables

In the previous example, we did not initialize pop nor n; yet, everything worked properly. This is because (by default) variables are initialized to the null string which has a numerical value of 0. This eliminates the need for most initialization of variables in BEGIN sections. We can use default initialization to advantage in this program which finds the country with the largest population.

```
maxpop < $3 {
        maxpop = $3
        country = $1
        }
END     {print country, maxpop}
```

which produces

```
China 866
```

Field Variables

Fields in *awk* share essentially all of the properties of variables. They are used in arithmetic and string operations and may be assigned to and initialized to the null string. Thus, divide the second field by 1000 to convert the area to millions of square miles by

```
{ $2 /= 1000; print }
```

or process two fields into a third with

```
BEGIN   { FS = "        "}
        { $4 = 1000 * $3 / $2; print }
```

or assign strings to a field as in

```
/USA/   { $1 = "United States" ; print }
```

which replaces USA by United States and prints the effected line

```
United States 3615 219 North America
```

Fields are accessed by expressions; thus, $NF is the last field and $(NF−1) is the second to the last. Note that the parentheses are needed since $NF−1 is 1 less than the values in the last field.

String Concatenation

Strings are concatenated by writing them one after the other as in the following example:

```
{ x = "hello"
  x = x ", world"
  print x
}
```

prints the usual

```
hello, world
```

With input from the file "countries", the following program:

```
/A/       { s = s " " $1 }
END         { print s }
```

prints

Australia Argentina Algeria

Variables, string expressions, and numeric expressions may appear in concatenations; the numeric expressions are treated as strings in this case.

Special Variables

Some variables in *awk* have special meanings. These are detailed here and the complete list given.

NR	Number of the current record.
NF	Number of fields in the current record.
FS	Input field separator, by default it is set to a blank or tab.
RS	Input record separator, by default it is set to the newline character.
$i	The ith input field of the current record.
$0	The entire current input record.
OFS	Output field separator, by default it is set to a blank.
ORS	Output record separator, by default it is set to the newline character.
OFMT	The format for printing numbers, with the print statement, by default is "%.6g".
FILENAME	The name of the input file currently being read. This is useful because *awk* commands are typically of the form

awk −f program file1 file2 file3 ...

Type

Variables (and fields) take on numeric or string values according to context. For example, in

```
pop += $3
```

pop is presumably a number, while in

```
country = $1
```

country is a string. In

```
maxpop < $3
```

the type of maxpop depends on the data found in $3. It is determined when the program is run.

In general, each variable and field is potentially a string or a number or both at any time. When a variable is set by the assignment

```
v = expr
```

its type is set to that of expr. (Assignment also includes +=, ++, −=, and so forth.) An arithmetic expression is of the type, "number"; a concatenation of strings is of type "string". If the assignment is a simple copy as in

```
v1 = v2
```

then the type of v1 becomes that of v2.

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to strings if necessary and the comparison is made on strings.

The type of any expression is coerced to numeric by subterfuges such as

```
expr + 0
```

and to string by

```
expr ""
```

This last expression is string concatenated with the null string.

Arrays

As well as ordinary variables, *awk* provides 1-dimensional arrays. Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value including non-numeric strings.

As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input line to the NRth element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the following *awk* program:

```
        { x[NR] = $0 }
END     { ... program ... }
```

The first line of this program records each input line into the array x. In particular, the following program

```
{ x[NR] = $1}
```

(when run on the file countries) produces an array of elements with

```
x[1] = "Russia"
x[2] = "Canada"
x[3] = "China"
        ... and so forth.
```

Arrays are also indexed by non-numeric values that give *awk* a capability rather like the associative memory of Snobol tables. For example, we can write

```
/Asia/    {pop["Asia"] += $3 }
/Africa/{pop[Africa] += $3 }
END       print "Asia=" pop["Asia"], "Africa="pop["Africa"] }
```

which produces

```
Asia=1765 Africa=37
```

Notice the concatenation. Also, any expression can be used as a subscript in an array reference. Thus,

```
area[$1] = $2
```

uses the first field of a line (as a string) to index the array area.

BUILT IN FUNCTIONS

The function

```
length
```

is provided by *awk* to compute the length of a string of characters. The following program prints each record preceded by its length:

```
{print length, $0 }
```

In this case (the variable) length means length($0), the length of the present record. In general, length(x) will return the length of x as a string.

Example:

With input from the file countries, the following *awk* program will print the longest country name:

```
length($1) > max  {max = length($1); name = $1 }
END               {print name}
```

The function

```
split
```

split (s, array) assigns the fields of the string "s" to successive elements of the array, "array".

For example;

split("Now is the time", w)

assigns the value "Now" to w[1], "is" to w[2], "the" to w[3] and "time" to w[4]. All other elements of the array w[], if any, are set to the null string. It is possible to have a character other than a blank as the separator for the elements of w. For this, use split with three elements.

n = split(s, array, sep)

This splits the string s into array[1], ..., array[n]. The number of elements found is returned as the value of split. If the sep argument is present, its first character is used as the field separator; otherwise, FS is used. This is useful if in the middle of an *awk* script, it is necessary to change the record separator for one record.

Also provided by the *awk* are the

Math Functions

sqrt,
log,
exp
int,

They provide the square root function, the base e logarithm function, exponential and integral part functions. This last function returns the greatest integer less than or equal to its argument. These functions are the same as those of the C library (*int* corresponds to the libc *floor* function) and so they have the same return on error as those in libc. (See the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines.*)

The subtract function

substr

substr(s,m,n) produces the substring of s that begins at position m and is at most n characters long. If the third argument (n in this case) is omitted, the substring goes to the end of s. For example, we could abbreviate the country names in the file countries by

```
{ $1 = substr($1, 1, 3); print }
```

which produces

```
Rus     8650   262   Asia
Can     3852    24   North America
Chi     3692   866   Asia
USA     3615   219   North America
Bra     3286   116   South America
Aus     2968    14   Australia
Ind     1269   637   Asia
Arg     1072    26   South America
Sud      968    19   Africa
Alg      920    18   Africa
```

If s is a number, substr uses its printed image; substr(123456789,3,4)=3456.

The function

index:

index (s1,s2) returns the leftmost position where the string s2 occurs in s1 or zero if s2 does not occur in s1.

The function

sprintf

formats expressions as the printf statement does but will assign the resulting expression to a variable instead of sending the results to stdout. For example,

```
x = sprintf( "%10s %6d ", $1, $2 )
```

sets x to the string produced by formatting the values of $1 and $2. The x is then used in subsequent computations.

The function

```
getline
```

immediately reads the next input record. Fields NR and $0 are all set but control is left at exactly the same spot in the *awk* program. Getline returns 0 for the end of file and a 1 for a normal record.

FLOW OF CONTROL

The *awk* provides the basic flow of control statements

- **if-else**
- **while/fR**
- **for**

with statement grouping as in C language.

The **if** statement is used as follows:

```
if ( condition ) statement1 else statement2
```

The condition is evaluated; and if it is true, statement1 is executed; otherwise, statement2 is executed. The else part is optional. Several statements enclosed in braces ({,}) are treated as a single statement. Rewriting the maximum population computation from the pattern section with an if statement results in

```
{       if (maxpop < $3) {
                maxpop= $3
                country= $1
                }       }
END     { print country, maxpop }
```

There is also a while statement in *awk*.

```
while ( condition ) statement
```

The condition is evaluated; if it is true, the statement is executed. The condition is evaluated again, and if true, the statement is executed. The cycle repeats as long as the condition is true. For example, the following prints all input fields one per line:

```
{       i = 1
        while (i <= NF) {
                pint $i
                ++i
                }
        }
```

Another example is the Euclidean algorithm for finding the greatest common divisor of $1 and $2:

```
{printf "the greatest common divisor of " $1 "and ", $2, "is"
while ($1 != $2) {
                                        if ($1 > $2) $1 = $1 - $2
                                        else    $2 = $2 - $1
                                        }
printf $1 "0
}
```

The **for** statement is like that of C.

```
for ( expression1 ; condition ; expression2 ) statement
```

has the same effect as

```
expression1
while (condition) {
        statement
        expression2
        }
```

so

```
{       for (i=1 ; i <= NF; i++)
                print $i
}
```

is another *awk* program that prints all input fields one per line.

This is an alternate form of the **or** statement that is suited for accessing the elements of an associative array as is in *awk*.

```
for (i in array) statement
```

executes statement with the variable i set in turn to each subscript of array. The subscripts are each accessed once but in random order. Chaos will ensue if the variable i is altered or if any new elements are created within the loop. For example, you could use the "for" statement to print the record number followed by the record of all input records after the main program is executed.

```
        { x[NR] = $0 }
END     { for(i in x) { print i, x[i] }
```

A more practical example is the following use of strings to index arrays to add the populations of countries by continents:

```
BEGIN   {FS=""}
        {population[$4] =+ $3}
END     {for(i in population)
             print i, population[i]
        }
```

In this program, the body of the **for** loop is executed for *i* equal to the string "Asia", then for *i* equal to the string "North America", and so forth until all the possible values of i are exhausted; that is, until all the strings of names of countries are used. Note, however, the order the loops are executed is not specified. If the loop associated with "Canada" is executed before the loop associated with the string "Russia", such a program produces

```
South America 26
Africa 16
Asia 637
Australia 14
North America 219
```

Note that the expression in the condition part of an **if, while**, or, **for** statement can include relational operators like <, <=, >, >=, ==, and !=; it can include regular expressions that are used with the "matching" operators ~ and !~; it can include the logical operators |, &&, and !; and it also include parentheses for grouping.

The **break** statement (when it occurs within a **while** or **for** loop) causes an immediate exit from the **while** or **for** loop.

The **continue** statement (when it occurs within a **while** or **for** loop) causes the next iteration of the loop to begin.

The **next** statement in an *awk* program causes awk to skip immediately to the next record and begin scanning patterns from the top of the program. (Note the difference between getline and next. Getline does not skip to the top of the *awk* program.)

If an **exit** statement occurs in the BEGIN section of an *awk* program, the program stops executing and the END section is not executed (if there is one).

An **exit** that occurs in the main body of the *awk* program causes execution of the main body of the *awk* program to stop. No more records are read, and the END section is executed.

An **exit** in the END section causes execution to terminate at that point.

REPORT GENERATION

The flow of control statements in the last section are especially useful when *awk* is used as a report generator. *Awk* is useful for tabulating, summarizing, and formatting information. We have seen an example of *awk* tabulating in the last section with the tabulation of populations. Here is another example of this. Suppose you have a file "prog.usage" that contains lines of three fields; name, program, and usage:

```
Smith    draw    3
Brown    eqn     1
Jones    nroff   4
Smith    nroff   1
Jones    spell   5
Brown    spell   9
Smith    draw    6
```

The first line indicates that Smith used the draw program three times. If you want to create a program that has the total usage of each program along with the names in alphabetical order and the total usage, use the following program, called *list.a*:

```
      { use[$1 "" $2] += $3}
END   {for (np in use)
          print np "       " use[np] | "sort +0 +2nr" }
```

This program produces the following output when used on the input file, *prog.usage*.

```
Brown    eqn     1
Brown    spell   9
Jones    nroff   4
Jones    spell   5
Smith    draw    9
Smith    nroff   1
```

If you would like to format the previous output so that each name is printed only once, pipe the output of the previous *awk* program into the following program, called "format.a:

```
{       if ($1 != prev) {
                print $1 ":"
                rev = $1
                }
        print " " $2 " " $3
}
```

The variable *prev* prints the unique values of the first field. The command

awk −f list.a prog.usage | *awk* −f format.a

gives the output

```
Brown:
        eqn     1
        spell   9
Jones:
        nroff   4
        spell   5
Smith:
        draw    9
        nroff   1
```

It is often useful to combine different *awk* scripts and other shell commands such as *sort* as was done in the last script.

COOPERATION WITH THE SHELL

Normally, an *awk* program is either contained in a file or enclosed within single quotes as in

```
awk '{print $1}' ...
```

Awk uses many of the same characters that the shell does, such as $ and the double quote. Surrounding the program by ' ... ' ensures that the shell passes the *awk* program to *awk* intact.

Consider writing an *awk* program to print the nth field, where n is a parameter determined when the program is run. That is, we want a program called field such that

```
field n
```

runs the *awk* program

```
awk '{print $n}'
```

How does the value of n get into the *awk* program?

There are several ways to do this. One is to define field as follows:

```
awk '{print $'$1'}'
```

Spaces are critical here: as written there is only one argument, even though there are two sets of quotes. The $1 is outside the quotes, visible to the shell, and therefore substituted properly when field is invoked.

Another way to do this job relies on the fact that the shell substitutes for $ parameters within double quotes.

```
awk "{print $1}"
```

Here the trick is to protect the first $ with a \\; the $1 is again replaced by the number when field is invoked.

This kind of trickery is extended in remarkable ways, but it is hard to understand quickly.

MISCELLANEOUS HINTS

You can simulate the effect of multidimensional arrays by creating your own subscripts. For example,

```
for ( i = 1; i <= 10; i++)
        for ( j = 1; j <= 10; j++)
                mult[i "," j] = . . .
```

creates an array whose subscripts have the form i,j; that is, 1,1; 1,2; and so forth and thus simulate a 2-dimensional array.

THE LINK EDITOR

GENERAL

The link editor [*ld*(1)*] is a UNIX system support tool used on the VAX processor and on all processors in the 3B Computer family. The *ld* creates executable object files by combining object files, performing relocation, and resolving external references. The *ld* also processes symbolic debugging information. The inputs to *ld* are relocatable object files produced either by the compiler [*cc*(1)], the assembler [*as*(1)], or by a previous ld run. The *ld* combines these object files to form either a relocatable or an absolute (i.e., executable) object file.

The *ld* also supports a command language that allows users to control the *ld* process with great flexibility and precision. The UNIX system *ld* shares most of its source with other lds in-use on other processors and operating systems. Therefore, the UNIX system ld provides many powerful features that may or may not be useful on a UNIX system.

Although the link edit process is controlled in detail through use of the *ld* command language described later, most users do *not* require this degree of flexibility, and the manual page obtained by typing

```
man ld
```

is sufficient instruction in the use of ld.

The command language (described later) supports the ability to

- Specify the memory configuration of the machine
- Combine object file sections in particular fashions

---

* Section 1 of the *UNIX Programmer's Manual—Volume 1: Commands and Utilities.*

- Cause the files to be bound to specific addresses or within specific portions of memory
- Define or redefine global symbols at link edit time.

There are several concepts and definitions with which you should familiarize yourself before proceeding further.

Memory Configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into *configured* and *unconfigured* memory. The default condition is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of RAM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K-1 are then not configured. Unconfigured memory is treated as "reserved" or "unusable" by the ld. *Nothing can ever be linked into unconfigured memory.* Thus, specifying a certain memory range to be unconfigured is one way of marking the addresses (in that range) "illegal" or "nonexistent" with respect to the linking process. Memory configurations other than the default must be explicitly specified by you (the user).

Unless otherwise specified, all discussion in this document of memory, addresses, etc. are with respect to the *configured* sections of the address space.

Section

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in "section headers" at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be "holes" or gaps between input sections and between output sections, storage is allocated contiguously *within* each output section and may not overlap a hole in memory.

Addresses

The *physical address* of a section or symbol is the relative offset from address zero of the address space. The *physical address* of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called "binding", and the section in question is said to be "bound to" or "bound at" the required address. While binding is most commonly relevant to output sections, it is also possible to bind global symbols with an assignment statement in the *ld* command language.

Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by the ld. The *ld* accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to the *ld* can also be absolute files.

Files produced from the compiler/assembler always contain three sections, called *.text*, *.data*, and *.bss*. The *.text* section contains the instruction text (for example, executable instructions), *.data* contains initialized data variables, and *.bss* contains uninitialized data variables. For example, if a C program contained the global (i.e., not inside a function) declarations

```
int i = 100;
char abc[200];
```

and the assignment

```
abc[i] = 0;
```

then compiled code from the C assignment is stored in *.text*. The variable i is located in *.data*, and abc is located in *.bss*. There is an exception to the rule however; both initialized and uninitialized statics are allocated into the *.data*

section. The value of an uninitialized static in a *.data* section is zero.

USING THE LINK EDITOR

The *ld* is called by the command

ld [options] filename1 filename2 . . .

Files passed to the *ld* must be object files, archive libraries containing object files, or text source files containing *ld* directives. The *ld* uses the "magic number" (in the first two bytes of the file) to determine which type of file is encountered. If the *ld* does not recognize the magic number, it assumes the file is a text file containing *ld* directives and attempts to parse it.

Input object files and archive libraries of object files are linked together to form an output object file. If there are no unresolved references, this file is executable *on the target machine.* An input file containing directives is referred to as an *ifile* in this document. Object files have the form "name.o" throughout the examples in this chapter. The names of actual input object files need not follow this convention.

If you merely want to link the object files file1.o and file2.o, the following command is sufficient:

ld file1.o file2.o

No directives to the *ld* are needed. If no errors are encountered during the link edit, the output is left on the default file a.out. The sections of the input files are combined in order. That is, if file1.o and file2.o each contain the standard sections *.text*, *.data*, and *.bss*, the output object file also contains these three sections. The output *.text* section is a concatenation of *.text* from file1.o and *.text* from file2.o. The *.data* and *.bss* sections are formed similarly. The output *.text* section is then bound (with the exception of 3B5 Computers) at address 0X000000. The output *.data* and *.bss* sections are link edited together into contiguous addresses (the particular address depending on the particular processor).

Instead of entering the names of files to be link edited (as well as *ld* options on the *ld* command line), this information can be placed into an ifile, and just the ifile passed to ld. For example, if you are going to frequently link the object files file1.o, file2.o, and file3.o with the same options f1 and f2, then enter the command

```
ld −f1 −f2 file1.o file2.o file3.o
```

each time it is necessary to invoke ld. Alternatively, an ifile containing the statements

```
−f1
−f2
file1.o
file2.o
file3.o
```

could be created, and then the following UNIX system command would serve:

```
ld ifilename
```

Note that it is perfectly permissible to specify some of the object files to be link edited in the ifile and others on the command line—as well as some *options* in the ifile and others on the command line. Input object files are link edited in the order they are encountered, whether this occurs on the command line or in an ifile. As an example, if a command line were

```
ld file1.o ifile file2.o
```

and the ifile contained

```
file3.o
file4.o
```

then the order of link editing would be: file1.o, file3.o, file4.o, and file2.o. Note from this example that an ifile is read *and processed* immediately upon being encountered in the command line.

Options may be interspersed with file names both on the command line and in an ifile. The ordering of options is not significant, except for the "l" and "L" options for specifying libraries. The "l" option is a shorthand notation for specifying an archive library, and an archive library is just a collection of object files. Thus, as is the case with any object file, libraries are searched as they are encountered. The "L" specifies an alternative directory for searching for libraries. Therefore, to be effective, a "–L" option must appear before any "–l" options.

All options for *ld* must be preceded by a hyphen (-) whether in the ifile or on the *ld* command line. Options that have an argument (except for the "–l" and "–L" options) are separated from the argument by white space (blanks or tabs). The following options (in alphabetical order) are supported, though not all options are available on each processor.

–a Produces an absolute, executable file. Messages are issued when undefined symbols are found, and several special symbols (such as "_end") are defined. Unless overridden by the "–r" option, relocation information is stripped from the output file. If neither "–r" nor "–a" is specified, "–a" is assumed. This flag applies only to the 3B5 and 3B2 Computers.

–e ss Defines the primary entry point of the output file to be the symbol given by the argument "ss". See "Changing the Entry Point" in "NOTES AND SPECIAL CONSIDERATIONS" for a discussion of how the option is used.

–f bb Sets the default fill value. This value is used to fill "holes" formed within output sections. Also, it is used to initialize input *.bss* sections when they are combined with other non-*.bss* input sections. The argument "bb" is a 2-byte constant. If the "–f" option is not used, the default fill value is zero.

–lx Specifies a UNIX system archive library file as *ld* input. The argument is a character string (less than 10 characters) immediately following the "–l" without any intervening white space. As an example, –lc refers to libc.a, –lC to libC.a, etc. The given archive library must contain valid object files as its members.

–m Produces a map or listing of the input/output sections (including "holes") on the standard output.

−o name Names the output object file. The argument "name" is the name of the UNIX system file to be used as the output file. The default output object file name is "a.out". The "name" can be a full or partial UNIX system pathname.

−r Retains relocation entries in the output object file. Relocation entries must be saved if the output file is to be used as an input file in a subsequent *ld* call. If the −r option is used, unresolved references do not prevent the creation of an output object file.

−s Strips line number entries and symbol table information from the output object file. Relocation entries ("−r" option) are meaningless without the symbol table, hence use of "−s" precludes the use of "−r". All symbols are stripped, including global and undefined symbols.

−t Disables checking that all instances of a multiply defined symbol are the same size.

−u sym Introduces an unresolved external symbol into the output file's symbol table. The argument "sym" is the name of the symbol. This is useful for linking entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the linking of an initial routine from the library.

−x Does not preserve any local (nonglobal) symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file.

−H Changes the type of all global symbols to "static". This option can be used to "hide" symbols since static symbols have different accessing rules from global symbols.

−Ldir Changes the algorithm for searching for libraries to look in *dir* before looking in the default location. This option is for *ld* libraries as the −I option is for compiler #include files. The "−L" option is useful for finding libraries that are not in the standard library directory. To be useful, this option must appear before the "−l" option.

−M Prints a warning message for all external variables that are multiply defined.

−N Places the data section immediately following the text section in memory and stores the magic number 0407 in the UNIX system header. This prevents the text from being shared (the default).

−S Requests a "silent" *ld* run. All error messages resulting from errors that do not immediately stop the *ld* run are suppressed.

−V Prints on the standard error output a "version id" identifying the *ld* being run.

−VS num Takes *num* as a decimal version number identifying the *a.out* file that is produced. The version stamp is stored in the UNIX system header.

LINK EDITOR COMMAND LANGUAGE

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 17, "SYNTAX DIAGRAM FOR INPUT DIRECTIVES".) Constants are as in C with a number recognized as decimal unless preceded with "0" for octal or "0x" for hexadecimal. All numbers are treated as long ints. Symbol names may contain uppercase or lowercase letters, digits, and the underscore ('_'). Symbols within an expression have the value of the *address* of the symbol only. The *ld* does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

The *ld* uses a lex-generated input scanner to identify symbols, numbers, operators, etc. The current scanner design makes the following names *reserved* and unavailable as symbol names or section names:

ALIGN	DSECT	MEMORY	PHY	SECTIONS
ASSIGN	GROUP	NOLOAD	RANGE	SPARE
BLOCK	LENGTH	ORIGIN	REGION	TV

align	group	length	origin	spare
assign	l	o	phy	
block	len	org	range	

The operators that are supported, in order of precedence from high to low, are shown in Figure 16:

Figure 16

Symbols and Functions of Operators.

symbol
!~–(UNARY Minus)
* / %
+ -(BINARY Minus)
>> <<
== != > < <= >=
&
|
&&
||
= += -= *= /=

The above operators have the same meaning as in the C language. Operators on the same line have the same precedence.

Assignment Statements

External symbols may be defined and assigned addresses via the assignment statement. The syntax of the assignment statement is

symbol = expression;

or

symbol op= expression;

where *op* is one of the operators +, −, *, *or /.*

Assignment statements must be terminated by a semicolon.

All assignment statements (with the exception of the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are appropriately relocated but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address *in the output object file.* References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are input to ld.

Assignment statements are normally placed outside the scope of section-definition directive (see "Section Definition Directive" under "LINK EDITOR COMMAND LANGUAGE"). However, there exists a special symbol, called ".", that can occur only *within* a section-definition directive. This symbol refers to the *current R address of the ld's location counter. Thus, assignment expressions involving "." are evaluated during the allocation phase of ld.* Assigning a value to the "." symbol within a section-definition directive increments/resets ld's location counter and can create "holes" within the section, as described in "Section Definition Directives". Assigning the value of the "." symbol to a conventional symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

Align is provided as a shorthand notation to allow alignment of a symbol to an n-byte boundary within an output section, where *n* is a power of 2. For example, the expression

align(n)

is equivalent to

(. + n − 1) &~(n − 1)

Link editor expressions may have either an absolute or a relocatable value. When the *ld* creates a symbol through an assignment statement, the symbol's value takes on that type of expression. That type depends on the following rules:

- An expression with a *single* relocatable symbol (and zero or more constants or absolute symbols) is relocatable. The value is in relation to the section of the referenced symbol.
- All other expressions have absolute values.

Specifying a Memory Configuration

MEMORY directives are used to specify

a. The total size of the virtual space of the target machine.

b. The configured and unconfigured areas of the virtual space.

If no directives are supplied, the *ld* assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

By means of MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically *named* memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters '$', '.', or '_'. Names of memory ranges are used by *ld* only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, *all* virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in the ld's allocation process, and hence nothing can be link edited, bound, or assigned to any address within unconfigured memory.

As an option on the MEMORY directive, *attributes* may be associated with a named memory area. This restricts the memory areas (with specific attributes) to which an output section can be bound. The attributes assigned to output sections in this manner are recorded in the appropriate section headers in the output file to allow for possible error checking in the future. For example, putting a text section into writable memory is one potential error condition. Currently, error checking of this type is not implemented.

The attributes currently accepted are

a. R : readable memory.

b. W : writable memory.

c. X : executable, i.e., instructions may reside in this memory.

d. I : initializable, i.e., stack areas are typically not initialized.

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of W, R, I, and X.

The syntax of the MEMORY directive is

MEMORY
{
    name1 (attr) :   *origin* = n1, *length* = n2
    name2 (attr) :   *origin* = n3, *length* = n4
    etc.
}

The keyword "origin" (or "org" or "o") must precede the origin of a memory range, and "length" (or "len" or "l") must precede the length as shown in the above prototype. The origin operand refers to the *virtual* address of the memory range. Origin and length are entered as long integer *constants* in either decimal, octal, or hexadecimal (standard C syntax). Origin and length specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

By specifying MEMORY directives, the *ld* can be told that memory is configured in some manner other than the default. For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this.

```
MEMORY
{
        valid : org = 0x10000, len = 0xFE0000
}
```

Section Definition Directives

The purpose of the SECTIONS directive is to describe how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. For example, if a number of object files from the compiler are linked, each containing the three sections *.text*, *.data*, and *.bss*, the output object file also contains three sections, *.text*, *.data*, and *.bss*. If two object files are linked (one that contains sections s1 and s2 and the other containing sections s3 and s4), the output object file contains the four sections s1, s2, s3, and s4. The *order* of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the SECTIONS directive is

```
SECTIONS
{
        secname1 :
        {
                file_specifications,
                assignment_statements*
        }
        secname2 :
        {
                file_specifications,
                assignment_statements*
        }
etc.
}
```

The various types of section definition directives are discussed in the remainder of this section.

File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by

filename (secname)

or

filename (secnam1 secnam2 . . .)

---

* These may be intermixed.

Sections of an input file are separated either by white space or commas as are the file specifications themselves.

If a file name appears with no sections listed, then *all* sections from the file are linked into the current output section. For example,

```
SECTIONS
{
        outsec1:
        {
                file1.o (sec1)
                file2.o
                file3.o (sec1, sec2)
        }
}
```

The order in which the input sections appears in the output section "outsec1" is given by

a. Section sec1 from file file1.o

b. All sections from file2.o, in the order they appear in the file

c. Section sec1 from file file3.o, and then section sec2 from file file3.o.

If there are any additional input files that contained input sections also named "outsec1", these sections are linked following the last section named in the definition of "outsec1". If there are any other input sections in file1.0 or file3.0, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

Load a Section at a Specified Address

Bonding of an output section to a specific virtual address is accomplished by an *ld* option as shown on the following SECTIONS directive example:

```
SECTIONS
{
        outsec addr:
        {
                . . .
        }
        etc.
}
```

The "addr" is the bonding address expressed as a C constant. If "outsec" does not fit at "addr" (perhaps because of holes in the memory configuration or because "outsec" is too large to fit without overlapping some other output section), ld issues an appropriate error message.

So long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The SECTIONS directives defining output sections need not be given to *ld* in any particular order.

The *ld* does not ensure that each section's size consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4. The *ld* directives can be used to force a section to start on an odd byte boundary although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, the ld issues a warning message.

Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an n-byte boundary, where n is a power of 2. The ALIGN option of the SECTIONS directive performs this function, so that the option

```
ALIGN(n)
```

is equivalent to specifying a bonding address of

```
( . + n − 1) &~(n − 1)
```

For example

```
SECTIONS
{
        outsec  ALIGN(0x20000) :
        {
                . . .
        }
        etc.
}
```

The output section "outsec" is not bound to any given address but is linked to some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, etc.).

Grouping Sections Together

The default allocation algorithm for ld

a. Links all input *.text* sections together into one output section. This output section is called *.text* and is bound to an address of 0x0.

b. Links all input *.data* sections together into one output section. This output section is called *.data* and (with the exception of 3B5 Computers) is bound to an address aligned to a machine dependent constant.

c. Links all input *.bss* sections together into one output section. This output section is called *.bss* and is allocated so as to immediately follow the output section *.data*. Note that the output section *.bss* is *not given any particular address alignment.*

Specifying any SECTIONS directives results in this default allocation *not* being performed.

The default allocation of *ld* is equivalent to supplying the following directive:

```
SECTIONS
{
        .text   : { }
        GROUP ALIGN( align_value ) :
        {
                .data   : { }
                .bss    : { }
        }
}
```

where *align_value* is a machine dependent constant. The GROUP command ensures that the two output sections, .data and .bss, are allocated (e.g., "grouped") together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If *.text*, *.data*, and *.bss* are to be placed in the same segment, the following SECTIONS directive is used:

```
SECTIONS
{
        GROUP           :
        {
                .text       : { }
                .data       : { }
                .bss        : { }
        }
}
```

Note that there are still three output *sections* (*.text*, *.data*, and *.bss*), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive. To bind to 0xC0000, use

```
GROUP 0xC0000 : {
```

To align to 0x10000, use

```
GROUP ALIGN(0x10000) : {
```

With this addition, first the output section *.text* is bound at 0xC0000 (or is aligned to 0x10000); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the GROUP directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
        .text   : { }
        .data ALIGN(0x20000)  : { }
        .bss    : { }
}
```

The *.text* section starts at virtual address 0x0 and the *.data* section at a virtual address aligned to 0x20000. The *.bss* section follows immediately after the *.text* section *if there is enough space.* If there is not, it follows the *.data* section.

The order in which output sections are defined to the ld *cannot* be used to force a certain allocation order in the output file.

Creating Holes Within Output Sections

The special symbol dot (.) appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, "." causes the ld's location counter to be incremented or reset and a "hole" left in the output section. "Holes" built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the "–f" option in "USING THE LINK EDITOR" and the discussion of filling holes in "Initialized Section Holes or .bss Sections" under "LINK EDITOR COMMAND LANGUAGE".

Consider the following section definition:

```
outsec:
{
        . += 0x1000;
        f1.o (.text)
        . += 0x100;
        f2.o (.text)
        . = align (4);
        f3.o (.text)
}
```

The effect of this command is as follows:

a. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input file f1.o(.text) is linked after this hole.

b. The text of input file f2.o begins at 0x100 bytes following the end of f1.o(.text).

c. The text of f3.o is linked to start at the next full word boundary following the text of f2.o with respect to the beginning of "outsec".

For the purposes of allocating and aligning addresses *within an output section,* the *ld* treats the output section as if it began at address zero. As a result, if, in the above example, "outsec" ultimately is linked to start at an odd address, then the part of "outsec" built from f3.o(.text) also starts at an odd address—even though f3.o(.text) is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {
```

It should be noted that the assembler, **as**, always pads the sections it generates to a full word length making explicit alignment specifications unnecessary. This also holds true for the compiler.

Expressions that decrement "." are illegal. For example, subtracting a value from the location counter is not allowed since overwrites are not allowed. The most common operators in expressions that assign a value to "." are "+=" and "align".

Creating and Defining Symbols at Link-Edit Time

The assignment instruction of the *ld* can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

a. Use of "." to adjust ld's location counter during allocation

b. Use of "." to assign an allocation-dependent value to a symbol

c. Assigning an allocation-independent value to a symbol.

Case a) has already been discussed in the previous section.

Case b) provides a means to assign addresses (known only after allocation) to symbols. For example

```
SECTIONS
{
        outsc1: {...}
        outsc2:
        {
                file1.o (s1)
                s2_start = . ;
                file2.o (s2)
                s2_end = . − 1;
        }
}
```

The symbol "s2_start" is defined to be the address of file2.o(s2), and "s2_end" is the address of the last byte of file2.o(s2).

Consider the following example:

```
SECTIONS
{
        outsc1:
        {
                file1.o (.data)
                mark = .;
                . += 4;
                file2.o (.data)
        }
}
```

In this example, the symbol "mark" is created and is equal to the address of the first byte beyond the end of file1.o's *.data* section. Four bytes are reserved for a future run-time initialization of the symbol mark. The type of the symbol is a long integer (32 bits).

Assignment instructions involving "." must appear within SECTIONS definitions since they are evaluated during *allocation.* Assignment instructions that do not involve "." can appear within SECTIONS definitions but typically do not. Such instructions are evaluated *after* allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within .data is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address. The *ld* issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section Into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific *named* memory (as previously specified on a MEMORY directive). (The ">" notation is borrowed from the UNIX system concept of "redirected output".)

For example

```
MEMORY
{
        mem1:        o=0x000000   l=0x10000
        mem2 (RW):   o=0x020000   l=0x40000
        mem3 (RW):   o=0x070000   l=0x40000
        mem1:        o=0x120000   l=0x04000
}

SECTIONS
{
        outsec1: { f1.o(.data) } > mem1
        outsec2: { f2.o(.data) } > mem3
}
```

This directs *ld* to place "outsec1" anywhere within the memory area named "mem1" (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FF). The "outsec2" is to be placed somewhere in the address range 0x70000-0xAFFFF.

Initialized Section Holes or BSS Sections

When "holes" are created within a section (as in the example in "LINK EDITOR COMMAND LANGUAGE"), the *ld* normally puts out bytes of zero as "fill". By default, *.bss* sections are not initialized at all; that is, no initialized data is generated for any *.bss* section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a SECTIONS directive to set such "holes" or output *.bss* sections to an arbitrary 2-byte pattern. *Such initialization options apply only to .bss* sections or "holes". As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the ".o" file or a "hole" in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized *.bss* section, if part of such a section is initialized, then the entire section is initialized. In other words, if a *.bss* section is to be combined with a *.text* or *.data* section (both of which are initialized) or if part of an output *.bss* section is to be initialized, then one of the following will hold:

a. Explicit initialization options must be used to initialize all *.bss* sections in the output section.

b. The *ld* will use the default fill value to initialize all *.bss* sections in the output section.

Consider the following *ld* ifile:

```
SECTIONS
{
        sec1:
        {
                f1.o
                . =+ 0x200;
                f2.o (.text)
        } = 0xDFFF
        sec2:
        {
                f1.o (.bss)
                f2.o (.bss) = 0x1234
        }
        sec3:
        {
                f3.o (.bss)
                . . .
        } = 0xFFFF
         sec4: { f4.o (.bss) }
}
```

In the example above, the 0x200 byte "hole" in section "sec1" is filled with the value 0xDFFF. In section "sec2", f1.o(.bss) is initialized to the default fill value of 0x00, and f2.o(.bss) is initialized to 0x1234. All *.bss* sections within "sec3" as well as all "holes" are initialized to 0xFFFF. Section "sec4" is not initialized; that is, no data is written to the object file for this section.

NOTES AND SPECIAL CONSIDERATIONS

Changing the Entry Point

The a.out header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

a. The value of the symbol specified with the "–e" option, if present, is used.

b. The value of the symbol "_start", if present, is used.

c. The value of the symbol "main", if present, is used.

d. The value zero is used.

Thus, an explicit entry point can be assigned to this a.out header field through the "–e" option or by using an assignment instruction in an ifile of the form

```
_start = expression;
```

If the *ld* is called through **cc**(1), a startup routine is automatically linked in. Then, when the program is executed, the routine **exit**(1) is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling the ld directly or when changing the entry point. The user must supply the startup routine or make sure that the program always calls exit rather than falling through the end. Otherwise, the program will dump core.

Use of Archive Libraries

Each member of an archive library (e.g., libc.a) is a complete object file typically consisting of the standard three sections: *.text*, *.data*, and *.bss*. Archive libraries are created through the use of the UNIX system "ar" command from object files generated by running the **cc** or **as**.

An archive library is always processed using *selective inclusion:* Only those members that resolve existing undefined-symbol references are taken from the library for link editing.

Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever

a. There exists a reference to a symbol defined in that member.

b. The reference is found by the *ld* prior to the actual scanning of the library.

When a library member is included by searching the library *inside* a SECTIONS directive, all input sections from the library member are included in the output section being defined. When a library member is included by searching the library *outside* of a SECTIONS directive, all input sections from the library member are included into the output section with the same name. That is, the .text section of the member goes into the output section named *.text*, the .data section of the member into *.data*, the *.bss* section of the member into *.bss*, etc. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that

a. Specific members of a library cannot be referenced explicitly in an ifile.

b. The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

The "–l" option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. *By convention,* such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the "–l" option by simply giving the (full or relative) UNIX system file path.

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference *that is known to be unresolved at the time the library is searched.* Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. The *ld* will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

a. The input files file1.o and file2.o each contain a reference to the external function FCN.

b. Input file1.o contains a reference to symbol ABC.

c. Input file2.o contains a reference to symbol XYZ.

d. Library liba.a, member 0, contains a definition of XYZ.

e. Library libc.a, member 0, contains a definition of ABC.

f. Both libraries have a member 1 that defines FCN.

If the *ld* command were entered as

```
ld file1.o -la file2.o -lc
```

then the FCN references are satisfied by liba.a, member 1, ABC is obtained from libc.a, member 0, and XYZ remains undefined (since the library liba.a is searched before file2.o is specified). If the *ld* command were entered as

```
ld file1.o file2.o -la -lc
```

then the FCN references is satisfied by liba.a, member 1, ABC is obtained from libc.a, member 0, and XYZ is obtained from liba.a, member 0. If the *ld* command were entered as

```
ld file1.o file2.o -lc -la
```

then the FCN references is satisfied by libc.a, member 1, ABC is obtained from libc.a, member 0, and XYZ is obtained from liba.a, member 0.

The "−u" option is used to force the linking of library members when the link edit run does not contain an actual external reference to the members. For example,

```
ld -u rout1 -la
```

creates an undefined symbol called "rout1" in the ld's global symbol table. If any member of library liba.a defines this symbol, it (and perhaps other members as well) is extracted. Without the "–u" option, there would have been no "trigger" to cause ld to search the archive library.

Dealing With Holes in Physical Memory

When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
        mem1:     o = 0x00000     l = 0x02000
        mem2:     o = 0x40000     l = 0x05000
        mem3:     o = 0x20000     l = 0x10000
}
```

Let the files f1.o, f2.o, . . . fn.o each contain the standard three sections *.text*, *.data*, and *.bss*, and suppose the combined *.text* section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the *.text* output section so *ld* may do allocation. For example,

```
SECTIONS
{
        txt1:
        {
                f1.o (.text)
                f2.o (.text)
                f3.o (.text)
        }
        txt2:
        {
                f4.o (.text)
                f5.o (.text)
                f6.o (.text)
        }
        etc.
}
```

Allocation Algorithm

An output section is formed either as a result of a SECTIONS directive or by combining input sections of the same name. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. Ld uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

a. Any output sections for which explicit bonding addresses were specified are allocated.

b. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the *first* available space within the (named) memory with any alignment taken into consideration.

c. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no SECTIONS directives are given, then output sections are allocated in the order they appear to the ld, normally *.text*, *.data*, *.bss*. Otherwise, output sections are allocated in the order they were defined or made known to the ld into the

first available space they fit.

Incremental Link Editing

As previously mentioned, the output of the ld can be used as an input file to subsequent *ld* runs *providing that the relocation information is retained* ("−r" option). Large applications may find it desirable to partition their C programs into "subsystems", link each subsystem independently, and then link edit the entire application. For example,

Step 1:

```
ld −r −o outfile1 ifile1

/* ifile1 */
SECTIONS
{
        ss1:
        {
                f1.o
                f2.o
                . . .
                fn.o
        }
}
```

Step 2:

```
ld −r −o outfile2 ifile2

/* ifile2 */
SECTIONS
{
        ss2:
        {
                g1.o
                g2.o
                . . .
                gn.o
        }
}
```

Step 3:

ld −a −m −o final.out outfile1 outfile2

By judiciously forming subsystems, applications may achieve a form of "incremental link editing" whereby it is necessary to relink only a portion of the total link edit when a few programs are recompiled.

To apply this technique, there are two simple rules

a. Intermediate link edits should contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.

b. All allocation and memory directives, as well as any assignment statements, are included only in the final ld call.

DSECT, COPY, and NOLOAD Sections

Sections may be given a "type" in a section definition as shown in the following example:

```
SECTIONS
{
      name1 0x200000 (DSECT)      : { file1.o }
      name2 0x400000 (COPY)       : { file2.o }
      name3 0x600000 (NOLOAD)     : { file3.o }

}
```

The DSECT option creates what is called a "dummy section". A "dummy section" has the following properties:

a. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map (the "−m" option) generated by the ld.

b. It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.

c. The global symbols defined within the "dummy section" *are relocated normally.* That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any members which define such symbols are link edited normally (i.e., not in the DSECT or as a DSECT).

d. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from file1.o are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A "copy section" created by the COPY option is similar to a "dummy section". The only difference between a "copy section" and a "dummy section" is that the contents of a "copy section" and all associated information is written to the output file.

A section with the "type" of NOLOAD differs in only one respect from a normal output section: *its text and/or data is not written to the output file.* A NOLOAD section is allocated virtual space, appears in the memory map, etc.

Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct *ld* to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

```
SECTIONS
{
        .text BLOCK(0x200) : { }
        .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

With this SECTIONS directive, *ld* assures that each section, *.text* and *.data*, is physically written at a file offset which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400,..., etc. in the file).

Nonrelocatable Input Files

If a file produced by the *ld* is intended to be used in a subsequent ld run, the first *ld* run has the "−r" option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent ld run.

When the *ld* detects an input file (that does not have relocation or symbol table information), a warning message is given. Such information can be removed by the *ld* (see the "−a" and "−s" options in the part USING THE LINK EDITOR) or by the *strip*(**1**) program. However, *the link edit run continues using the nonrelocatable input file.*

For such a link edit to be successful (i.e., to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, etc.), two conditions on the nonrelocatable input files must be met.

a. Each input file must have no unresolved external references.

b. Each input file must be bound to the exact same virtual address as it was bound to in the *ld* run that created it.

Note that if these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to the ld.

ERROR MESSAGES

Corrupt Input Files

The following error messages indicate that the input file is corrupt, nonexistent, or unreadable. The user should check that the file is in the correct directory with the correct permissions. If the object file is corrupt, try recompiling or reassembling it.

- Can't open *name*
- Can't read archive header from archive *name*
- Can't read file header of archive *name*
- Can't read 1st word of file *name*
- Can't seek to the beginning of file *name*
- Fail to read file header of *name*
- Fail to read lnno of section *sect* of file *name*
- Fail to read magic number of file *name*
- Fail to read section headers of file *name*
- Fail to read section headers of library *name* member *number*
- Fail to read symbol table of file *name*
- Fail to read symbol table when searching libraries
- Fail to read the aux entry of file *name*
- Fail to read the field to be relocated
- Fail to seek to symbol table of file *name*
- Fail to seek to symbol table when searching libraries

- Fail to seek to the end of library *name* member *number*
- Fail to skip aux entries when searching libraries
- Fail to skip the mem of struct of *name*
- Illegal relocation type
- No reloc entry found for symbol
- Reloc entries out of order in section *sect* of file *name*
- Seek to *name* section *sect* failed
- Seek to *name* section *sect* lnno failed
- Seek to *name* section *sect* reloc entries failed
- Seek to relocation entries for section *sect* in file *name* failed.

Errors During Output

These errors occur because the *ld* cannot write to the output file. This usually indicates that the file system is out of space.

- Cannot complete output file *name*. Write error.
- Fail to copy the rest of section *num* of file *name*
- Fail to copy the bytes that need no reloc of section *num* of file
- *name* I/O error on output file *name*.

Internal Errors

These messages indicate that something is wrong with the *ld* internally. There is probably nothing the user can do except get help.

- Attempt to free nonallocated memory
- Attempt to reinitialize the SDP aux space

- Attempt to reinitialize the SDP slot space
- Default allocation did not put *.data* and *.bss* into the same region
- Failed to close SDP symbol space
- Failure dumping an AIDFNxxx data structure
- Failure in closing SDP aux space
- Failure to initialize the SDP aux space
- Failure to initialize the SDP slot space
- Internal error: audit_groups, address mismatch
- Internal error: audit_group, finds a node failure
- Internal error: fail to seek to the member of *name*
- Internal error: in allocate lists, list confusion (*num num*)
- Internal error: invalid aux table id
- Internal error: invalid symbol table id
- Internal error: negative aux table *ld*
- Internal error: negative symbol table id
- Internal error: no symtab entry for DOT
- Internal error: split_scns, size of *sect* exceeds its new displacement.

Allocation Errors

These error messages appear during the allocation phase of the link edit. They generally appear if a section or group does not fit at a certain address or if the given MEMORY or SECTION directives in some way conflict. If you are using an ifile, check that MEMORY and SECTION directives allow enough room for the sections to ensure that nothing overlaps and that nothing is being placed in unconfigured memory. For more information, see "LINK EDITOR COMMAND LANGUAGE".

- Bond address *address* for *sect* is not in configured memory
- Bond address *address* for *sect* overlays previously allocated section *sect* at *address*
- Can't allocate output section *sect*, of size *num*
- Can't allocate section *sect* into owner *mem*
- Default allocation failed: *name* is too large
- GROUP containing section *sect* is too big
- Memory types *name1* and *name2* overlap
- Output section *sect* not allocated into a region
- *Sect* at *address* overlays previously allocated section *sect* at *address*
- *Sect*, bonded at *address*, won't fit into configured memory
- *Sect* enters unconfigured memory at *address*
- Section *sect* in file *name* is too big.

Misuse of Link Editor Directives

These errors arise from the misuse of an input directive. Please review the appropriate section in the manual.

- Adding *name*(*sect*) to multiple output sections.

The input section is mentioned twice in the SECTION directive.

- Bad attribute value in MEMORY directive: *c*.

An attribute must be one of "R", "W", "X", or "I".

- Bad flag value in SECTIONS directive, *option*.

Only the "−l" option is allowed inside of a SECTIONS directive

- Bad fill value.

The fill value must be a 2-byte constant.

- Bonding excludes alignment.

The section will be bound at the given address regardless of the alignment of that address.

- Cannot align a section within a group
- Cannot bond a section within a group
- Cannot specify an owner for sections within a group.

The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group may not be handled individually.

- DSECT *sect* can't be given an owner
- DSECT *sect* can't be linked to an attribute.

Since dummy sections do not participate in the memory allocation, it is meaningless for a dummy section to be given an owner or an attribute.

- Region commands not allowed

The UNIX system link editor does not accept the REGION commands.

- Section *sect* not built.

The most likely cause of this is a syntax error in the SECTIONS directive.

- Semicolon required after expression
- Statement ignored.

Caused by a syntax error in an expression.

- Usage of unimplemented syntax.

The UNIX system *ld* does not accept all possible ld commands.

Misuse of Expressions

These errors arise from the misuse of an input expression. Please review the appropriate section in the manual.

- Absolute symbol *name* being redefined.

An absolute symbol may not be redefined.

- ALIGN illegal in this context.

Alignment of a symbol may only be done within a SECTIONS directive.

- Attempt to decrement DOT
- Illegal assignment of physical address to DOT.
- Illegal operator in expression
- Misuse of DOT symbol in assignment instruction.

The DOT symbol (".") cannot be used in assignment statements that are outside SECTIONS directives.

- Symbol *name* is undefined.

All symbols referenced in an assignment statement must be defined.

- Symbol *name* from file *name* being redefined.

A defined symbol may not be redefined in an assignment statement.

- Undefined symbol in expression.

Misuse of Options

These errors arise from the misuse of options. Please review the appropriate section of the manual.

- Both −r and −s flags are set. The −s flag is turned off.

Further relocation requires a symbol table.

- Can't find library lib*x*.a
- −L path too long (*string*)
- −o file name too large (>128 char), truncated to (*string*)
- Too many −L options, seven allowed.

Some options require white space before the argument, some do not; see "USING THE LINK EDITOR". Including extra white space or not including the required white space is the most likely cause of the following messages.

- *option* flag does not specify a number
- *option* is an invalid flag
- −e flag does not specify a legal symbol name *name*
- −f flag does not specify a 2-byte number
- No directory given with −L
- −o flag does not specify a valid file name: *string*

- the −l flag (specifying a default library) is not supported
- −u flag does not specify a legal symbol name: *name.*

Space Restraints

The following error messages may occur if the *ld* attempts to allocate more space than is available. The user should attempt to decrease the amount of space used by the ld. This may be accomplished by making the ifile less complicated or by using the "−r" option to create intermediate files.

- Fail to allocate *num* bytes for slotvec table
- Internal error: aux table overflow
- Internal error: symbol table overflow
- Memory allocation failure on *num*-byte 'calloc' call
- Memory allocation failure on realloc call
- Run is too large and complex.

Miscellaneous Errors

These errors occur for many reasons. Refer to the error message for an indication of where to look in the manual.

- Archive symbol table is empty in archive *name*, execute 'ar ts *name*' to restore archive symbol table .

On systems with a random access archive capability, the link editor requires that all archives have a symbol table. This symbol table may have been removed by strip.

- Cannot create output file *name* .

The user may not have write permission in the directory where the output file is to be written.

- File **name** has no relocation information.

See "NOTES AND SPECIAL CONSIDERATIONS".

- File *name* is of unknown type, magic number = *num*
- Ifile nesting limit exceeded with file *name.*

Ifiles may be nested 16 deep.

- Library *name*, member has no relocation information.
- Line nbr entry (*num num*) found for nonrelocatable symbol.

Section *sect*, file *name*

This is generally caused by an interaction of *yacc*(**1**) and *cc*(**1**). Re-*yacc* the offending file with the "–l" option of *yacc.*

See the part "NOTES AND SPECIAL CONSIDERATIONS".

- Multiply defined symbol *sym*, in *name* has more than one size.

A multiply defined symbol may not have been defined in the same manner in all files.

- *name*(*sect*) not found.

An input section specified in a SECTIONS directive was not found in the input file.

- Section *sect* starts on an odd byte boundary!

This will happen only if the user specifically binds a section at an odd boundary.

- Sections *.text*, *.data*, or *.bss* not found. Optional header may be useless.

The UNIX system a.out header uses values found in the *.text*, *.data*, and *.bss* section headers.

- Undefined symbol *sym* first referenced in file *name* .

Unless the −r option is used, the *ld* requires that all referenced symbols are defined.

- Unexpected EOF (End Of File).

Syntax error in the ifile.

Figure 17

Syntax Diagram for Input Directives (Sheet 1 of 4).

directives	->	expanded directives
<file>	->	{ <cmd> }
<cmd>	->	<memory>
	->	<sections>
	->	<assignment>
	->	<filename>
	->	<flags>
<memory	->	**MEMORY** { <memory_spec> { [,] <memory_spec> }}
<memory_spec>	->	<name> [<attributes>] : <origin_spec> [,] <length_spec>
<attributes>	->	({ **R** | **W** | **X** | **I** })
<origin_spec>	->	<origin> = <long>
<lenth_spec>	->	<length> = <long>
<origin>	->	**ORIGIN** | **o** | **org** | **origin**
<length>	->	**LENGTH** | **l** | **len** | **length**
<sections>	->	**SECTIONS** { { <sec_or_group> } }
<sec_or_group>	->	<section> | <group> | <library>
<group>	->	**GROUP** <group_options> : { <section_list> } [<mem_spec>]
<section_list>	->	<section> { [,] <section> }

Syntax Diagram for Input Directives (Sheet 2 of 4).

directives	->	**expanded directives**
<section>	->	<name> <sec_options> : { <statement_list> } [<fill>] [<mem_spec>]
<group_options>	->	[<addr>] [<align_option>]
<sec_options>	->	[<addr>] [<align_option>] [<block_option>] [<type_option>]
<addr>	->	<long>
<align_option>	->	<align> (<long>)
<align>	->	**ALIGN** | **align**
<block_option>	->	<block> (<long>)
<block>	->	**BLOCK** | **block**
<type_option>	->	**(DSECT)** | **(NOLOAD)** | **(COPY)**
<fill>	->	= <long>
<mem_spec>	->	> <name>
	->	> <attributes>
<statement>	->	<file_name> [(<name_list>)] [<fill>] <library> <assignment>
<name_list>	->	<name> { [,] <name> }
<library>	->	-l<name>
<assignment>	->	<lside> <assign_op> <expr> <end>
<lside>	->	<name> | .
<assign_op>	->	= | += | -= | *= |/ =
<end>	->	; | ,
<expr>	->	<expr> <binary_op> <expr>
	->	<term>
<binary_op>	->	* | / | %
	->	+ | -
	->	>> | <<

Syntax Diagram for Input Directives (Sheet 3 of 4).

directives	->	expanded directives
	->	== | != | > | < | <= | >=
	->	&
	->	|
	->	&&
	->	||
<term>	->	<long>
	->	<name>
	->	<align> (<term>)
	->	(<expr)
	->	<unary_op> <term>
<unary_op>	->	! | -
<flags>	->	**-e**<wht_space><name>
	->	**-f**<wht_space><long>
	->	**-h**<wht_space><long>
	->	**-l**<name>
	->	**-m**
	->	**-o**<wht_space><filename>
	->	**-r**
	->	**-s**
	->	**-t**
	->	**-u**<wht_space><name>
	->	**-z**
	->	**-H**
	->	**-L**<pathname>
	->	**-M**
	->	**-N**
	->	**-S**
	->	**-V**
	->	**-VS**<wht_space><long>
	->	−a
	->	−x

Syntax Diagram for Input Directives (Sheet 4 of 4).

directives	->	**expanded directives**
<name>	->	Any valid symbol name
<long>	->	Any valid long integer constant
<wht_space>	->	Blanks, tabs, and newlines
<filename>	->	Any valid UNIX operating system filename. This may include a full or partial pathname.
<pathname>	->	Any valid UNIX operating system pathname (full or partial)

THE COMMON OBJECT FILE FORMAT

GENERAL

This chapter describes the Common Object File Format (COFF) used on several processors and operating systems, including the AT&T 3B Computer family and the UNIX operating system. The COFF is simple enough to be easily incorporated into existing projects, yet flexible enough to meet the needs of most projects. The COFF is the output file produced on some *UNIX systems* by the assembler (*as*) and the link editor (*ld*). This format is also used by other operating systems; hence, the word *common* is both descriptive and widely recognized. Currently, this object file format is used for the AT&T 3B Computer, including the 3B20D, the 3B20S, the 3B5 and 3B2 Computers, and on the VAX*-11/780 and 11/750 UNIX operating systems. Some key features of COFF are

- Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information used by debuggers and other applications
- Users may make some modifications in the object file construction at compile time.

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains

- A file header
- Optional header information
- A table of section headers
- Data corresponding to the section header

---

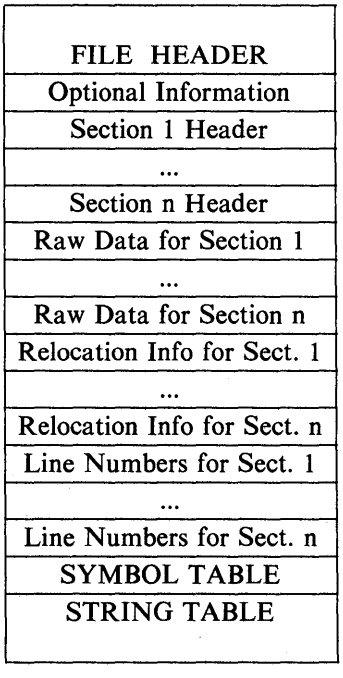
* Trademark of Digital Equipment Corporation

- Relocation information
- Line numbers
- A symbol table
- A string table.

Figure 18 shows the overall structure.

Figure 18

Object File Format.



The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **–s** option of the *UNIX system* link editor or if the line number information, symbol table, and string table are removed by the *strip* command. The line number information does not appear unless the program is compiled with the **–g** option of the compiler

(CC) command. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references can be executed on the target machine.

DEFINITIONS AND CONVENTIONS

Before proceeding further, you should become familiar with the following terms and conventions:

Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the default case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate multiple text or data segments, shared data segments, or user-specified sections. However, the UNIX operating system loads only the *.text*, *.data*, and *.bss* into memory when the file is executed.

Physical and Virtual Addresses

The *physical address* of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to the general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section heading contains two address fields, a physical address, and a virtual address; but in all versions of COFF on *UNIX* systems, the *physical address* is equivalent to the *virtual address.*

FILE HEADER

The file header contains the 20 bytes of information shown in Figure 19. The last 2 bytes are flags that are used by *ld* and object file utilities.

Figure 19

File Header Contents (Sheet 1 of 2).

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number, see Figure 20.
2-3	unsigned short	f_nscns	Number of section headers (equals the number of sections)
4-7	long int	f_timdat	Time and date stamp indicating when the file was created relative to the number of elapsed seconds since 00:00:00 GMT, January 1, 1970.

File Header Contents (Sheet 2 of 2).

Bytes	Declaration	Name	Description
8-11	long int	f_symptr	File pointer containing the starting address of the symbol table
12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags (see Figure 21)

The size of optional header information (**f_opthdr**) is used by all referencing programs that seek to the beginning of the section header table. This enables the same utility programs to work correctly on files targeted for different systems.

Magic Numbers

The magic number specifies the target machine on which the object file is executable. The currently defined magic numbers are in Figure 20.

Figure 20

Magic Numbers.

Mnemonic	Magic Number	System
N3B MAGIC	0550	3B20S Computers
FBOMAGIC	0560	3B2 and 3B5 Computers
VAXWRMAGIC	0570	VAX-11/750 and VAX-11/780 (writable text segments)
VAXROMAGIC	0575	VAX-11/750 and VAX-11780 (read-only text segments)
U370WRMAGIC	0530	IBM 370 (writable text segments)
U370ROMAGIC	0535	IBM 370 (read-only sharable text segments)

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. The currently defined flags are given in Figure 21.

Figure 21

File Header Flags (Sheet 1 of 2).

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_MINMAL	00020	Not used by the *UNIX system*
F_UPDATE	00040	Not used by the *UNIX system*
F_SWABD	00100	Not used by the *UNIX system*
F_AR16WR	00200	File has the byte ordering used by the PDP*-11/70 processor.

* Trademark of Digital Equipment Corporation

Filer Heade Flags (Sheet 2 of 2).

Mnemonic	Flag	Meaning
F_AR32WR	00400	File has the byte ordering used by the VAX-11/780 (i.e., 32 bits per word, least significant byte first).
F_AR32W	01000	File has the byte ordering used by the 3B computers (i.e., 32 bits per word, most significant byte first).
F_PATCH	02000	Not used by the *UNIX system*
F_BM32ID	0160000	WE 32000 processor ID field.

File Header Declaration"

The C structure declaration for the file header is given in Figure 22. This declaration may be found in the header file *filehdr.h*.

Figure 22

File Header Declaration.

```
struct filehdr {
   unsigned short   f_magic;   /* magic number */
   unsigned short   f_nscns;   /* number of section *

   long             f_timdat;  /* time and data stamp /*

   long             f_symptr;  /* file ptr to symbol table */

   long             f-nsyms;   /* number entries in the symbol table */

   unsigned short   f_opthdr;  /* size of optional header */

   unsigned short   f_flags;   /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
```

OPTIONAL HEADER INFORMATION

The template for optional information varies among different systems that use the COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, the disassembler, etc.) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header **f_opthdr**.

Standard *UNIX system* a.out Header

By default, files produced by the link editor for a *UNIX* system always have a standard *UNIX* system **a.out** header in the optional header field. The *UNIX* system a.out header is 28 bytes. There is one exception; files produced for the 3B20S Computers have an optional header of 36 bytes. The extra 8 bytes represent unused fields that are present for historical reasons. Therefore, the

two formats contain functionally equivalent information. The fields of the optional header are described in Figure 23 and 24.

Figure 23

Optional Header Contents (3B20S Computers Only).

Bytes	Declaration	Name	Description
0-1	short	magic	Magic number
2-3	short	vstamp	Version stamp
4-7	long int	tsize	Size of text in bytes
8-11	long int	dsize	Size of initialized data in bytes
12-15	long int	bsize	Size of uninitialized data in bytes
16-19	long int	dum1	Unused dummy field
20-23	long int	dum2	Unused dummy field
24-27	long int	entry	Entry point
27-31	long int	text_start	Base address of text
32-35	long int	data_start	Base address of data

Figure 24

Optional Header Contents (Processors other than the 3B20S Computer).

Bytes	Declaration	Name	Description
0-1	short	magic	Magic number
2-3	short	vstamp	Version stamp
4-7	long int	tsize	Size of text in bytes
8-11	long int	dsize	Size of initialized data in bytes
12-15	long int	bsize	Size of uninitialized data in bytes
16-19	long int	entry	Entry point
20-23	long int	text_start	Base address of text
24-37	long int	data_start	Base address of data

The magic number in the optional header supplies operating system dependent information about the object file. Whereas, the magic number in the file header specifies the machine on which the object file runs. The magic number in the optional header supplies information telling the operating system on that machine how that file should be executed.

The magic numbers recognized by the UNIX operating system are given in Figure 25.

Figure 25

UNIX System Magic Numbers.

Value	Meaning
0407	The text segment is not write-protected or sharable; the data segment is contiguous with the text segment.
0410	The data segment starts at the next segment following the text segment and the text segment is write protected.
0413	The data segment starts at a certain boundary within the next segment following the text segment. The text segment is write protected.

Optional Header Declaration

The C language structure declaration currently used for the *UNIX* system **a.out** file header is given in Figure 26. This declaration may be found in the header file *aouthdr.h*.

Figure 26

Aouthdr Declaration.

```
typedef struct aouthdr {
      short   magic;       /* magic number */
      short   vstamp;      /* version stamp */
      long    tsize;      /* text size in bytes, padded */

                          /* to full word boundary */

      long    dsize;      /* initialized data size */

      long    bsize;      /* uninitialized data size */

#if    u3b
      long    dum1;        /* unused dummy field */
      long    dum2;        /* unused dummy field */
#endif
      long    entry;      /* entry point */
      long    text_start; /* base of text for this file */

      long    data_start   /* base of data for this file */

} AOUTHDR;
```

SECTION HEADERS

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure 27.

Figure 27

Section Header Contents.

Bytes	Declaration	Name	Description
0-7	char	s_name	8-char null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File ptr to relocation entries
28-31	long int	s_lnnoptr	File ptr to line number entries
32-33	unsigned short	s_nreloc	Number of entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Figure 28)

The size of a section is padded to a multiple of 4 bytes.

File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the *UNIX* system function **fseek**(3S).

Flags

The lower 4 bits of the flag field indicate a section type. The flags are described in Figure 28.

Figure 28

Section Header Flags (Sheet 1 of 2).

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)

Section Header Flags (Sheet 2 of 2).

Mnemonic	Flag	Meaning
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data

Section Header Declaration

The C structure declaration for the section headers is described in Figure 29. This declaration may be found in the header file *scuhdr.h*.

Figure 29

Section Header Declaration.

```
struct scnhdr  {
    char      s_name[8];      /* section name */
    long      s_paddr;        /* physical address */
    long      s_vaddr;        /* virtual address */
    long      s_size;         /* section size */
    long      s_scnptr;       /* file ptr to section raw data */

    long      s_relptr;       /* file ptr to relocation */

    long      s_lnnoptr;      /* file ptr to line number */

unsigned short  s_nreloc;     /* number of relocation entries */

unsigned short  s_nlnno;      /* number of line number entries */

    long      s_flags;        /* flags */

};

#define  SCNHDR  struct scnhdr
#define  SCNHSZ  sizeof(SCNHDR)
```

.bss Section Header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are 0.

SECTIONS

Figure 18 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a full word boundary in the file.

Files produced by the cc and the as always contain three sections, called **.text**, **.data**, and **.bss**. The **.text** section contains the instruction text (i.e., executable code), the **.data** section contains initialized data variables, and the **.bss** section contains uninitialized data variables.

The link editor "SECTIONS directives" (see The **LINK EDITOR** chapter) allows users to

- Describe how input sections are to be combined.
- Direct the placement of output sections.
- Rename output sections.

If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files from the "cc" are linked together (each containing the three sections **.text**, **.data**, and **.bss**), the output object file contains three sections, **.text**, **.data**, and **.bss**.

RELOCATION INFORMATION

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure 30.

Figure 30

Relocation Section Contents.

Bytes	Declaration	Name	Description
0-3	long int	r_symndx	(Virtual) address of reference
4-7	long int	r_symndx	Symbol table index
8-9	unsigned short	r_type	Relocation type

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

The currently recognized relocation types are given in Figures 31 through 33.

Figure 31

3B20S Computers Relocation Types.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR24	04	Direct 24-bit reference to the symbol's virtual address.
R_REL24	05	A "PC-relative" 24-bit reference to the symbol's virtual address. Actual address is calculated by adding a constant to the PC value.

Figure 32

3B2 and 3B5 Computer Relocation Types.

Mnemonic	Flag	Meaning
R_BS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address
R_DIR32S	012	Direct 32-bit reference to the symbol's virtual address, with the 32-bit value stored in the reverse order in the object file.

Figure 33

VAX Relocation Types.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_RELBYTE	017	Direct 8-bit reference to the symbol's virtual address.
R_RELWORD	020	Direct 16-bit reference to the symbol's virtual address.
R_RELLONG	021	Direct 32-bit reference to the symbol's virtual address.
R_PCRBYTE	022	A "PC_relative" 8-bit reference to the symbol's virtual address.
R_PCRWORD	023	A "PC_relative" 16-bit reference to the symbol's virtual address.
R_PCRLONG	024	A "PC_relative" 32-bit reference to the symbol's virtual address.

On the VAX processors, relocation of a symbol index of −1 indicates that the amount by which the section is being relocated is added to the relocatable address.

The *as* automatically generates relocation entries which are then used by the link editor. The link editor uses this information to resolve external references in the file.

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 34. This declaration may be found in the header file *reloc.h*.

Figure 34

Relocation Entry Declaration.

```
struct reloc {
    long            r_vaddr;    /* virtual address of reference */

    long            r_symndx;   /* index into symbol table */

    unsigned short  r_type;     /* relocation type */
};

    #define RELOC    struct reloc

    #define RELSZ    10
0
```

LINE NUMBERS

When invoked with the −g option, *UNIX system* ccs (*cc*, *f77*) generates an entry in the object file for every C language source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like *sdb*. All line numbers in a section are grouped by function as shown in Figure 35.

Figure 35

Line Number Grouping.

symbol index	0
physical address	line number
physical address	line number
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries appear in increasing order of address.

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 36.

Figure 36

Line Number Entry Declaration.

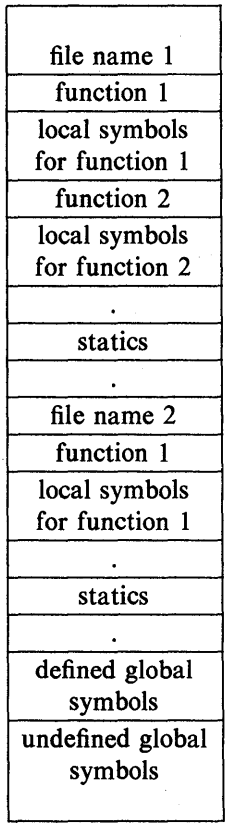
```
struct lineno {
        union
        {
                long    l_symndx;   /* symtbl index of func name */

                long    l_paddr;    /* paddr of line number */
        } l_addr;
        unsigned short  l_lnno;     /* line number */
};

#define LINENO      struct lineno

#define LINESZ      6
0
```

SYMBOL TABLE

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 37.

Figure 37

COFF Global Symbol Table.



The word "statics" in Figure 37 means symbols defined in the C language storage class *static* outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Special Symbols

The symbol table contains some special symbols that are generated by the *cc*, *as*, and other tools. These symbols are given in Figure 38.

Figure 38

Special Symbols in the Symbol Table (Sheet 1 of 2).

Symbol	Meaning
.file	file name
.text	address of **.text** section
.data	address of **.data** section
.bss	address of **.bss** section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.*x***fake**	dummy tag name for structure, union, or enumeran

Special Symbols in the Symbol Table (Sheet 2 of 2).

Symbol	Meaning
.eos	end of members of structure, union, or enumeration
_etext,etext	next available address after the end of the output section *.text*
_edata,edata	next available address after the end of the output section *.data*
_end,end	next available address after the end of the output section *.bss*.

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks. A **.bf** and **.ef** pair brackets each function; and a *.x***fake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the *cc* invents a name to be used in the symbol table. The name chosen for the symbol table is *.x***.fake**, where "x" is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are ".0fake", ".1fake", and ".2fake".

Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entry.

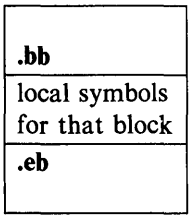
Inner Blocks

The C language defines a *block* as a compound statement that begins and ends with braces ({ and }). An *inner block* is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol **.bb** is put in the symbol table immediately before the first local symbol of that block. Also a special symbol, **.eb** is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 39.

Figure 39

Special Symbols (.bb and .eb).



Because inner blocks can be nested by several levels, the **.bb-.eb** pairs and associated symbols may also be nested. See Figure 40.

Figure 40

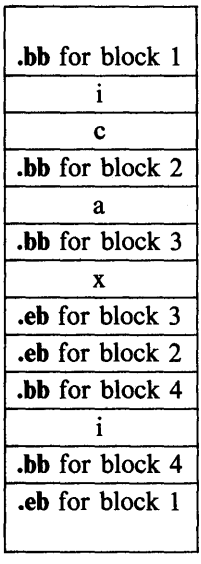
Nested Blocks.

```
{                               /* block 1 */
        int i;
        char c;
        ...
        {                       /* block 2 */
                long a;
                ...

        {                       /* block 3 */
                        int x;
                        ....
        }                       /* block 3 */
        }                       /* block 2 */

    {                           /* block 4 */
                long i;
                ...
    }                           /* block 4 */
}                               /* block 1 */
```

The symbol table would look like Figure 41.

Figure 41

Example of the Symbol Table.

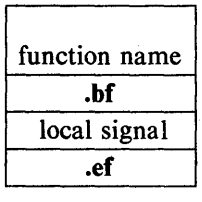


Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 42.

Figure 42

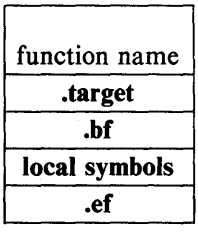
Symbols for Functions.



If the return value of the function is a structure or union, a special symbol **.target** is put between the function name and the **.bf**. The sequence is shown in Figure 43.

Figure 43

Special Symbol .Target.



The **cc** invents **.target** to store the function-return structure or union. The symbol **.target** is an automatic variable with "pointer" type. Its value field in the symbol is always 0.

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain the 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 44.

It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Figure 44

Symbol Table Entry Format.

Bytes	Declaration	Name	Description
0-7	(see text below)	_n	These 8 bytes contain either the name of a pointer or the name of a symbol.
8-11	long int	n_value	Symbol value; storage class dependent
12-13	short	n_scnum	Section number of symbol
14-15	unsigned short	n_type	Basic and derived type specification
16	char	n_sclass	Storage class of symbol
17	char	n_numaux	Number of auxiliary entries.

Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 45.

Figure 45

Name Field.

Bytes	Declaration	Name	Description
0-7	char	n_name	8-character null-padded symbol name
0-3	long	n_zeroes	Zero in this field indicates the name is in the string table
4-7	long	n_offset	Offset of the name in the string table

Some special symbols are generated by the *cc* and link editor as discussed in "special symbols". The VAX "cc" prepends an underscore ('_') to all the user defined symbols it generates.

Storage Classes

The storage class field has one of the values described in Figure 46. These "defines" may be found in the header file *storclass.h*.

Figure 46

Storage Classes (Sheet 1 of 2).

Mnemonic	Value	Storage Class
C_EFCN	−1	physical end of a function
C_NULL	0	−
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field

Storage Classes (Sheet 2 of 2).

Mnemonic	Value	Storage Class
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	file name
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

All of these storage classes except for C_ALIAS and C-HIDDEN are generated by the "cc" or "as". The compress utility, **cprs**, generates the C_ALIAS mnemonic. This utility (described in the *UNIX Programmer's Manual—Volume 1: Commands and Utilities*) removes duplicated structure, union, and enumeration definitions and puts ALIAS entries in their places. The storage class C-HIDDEN is not used by any *UNIX system* tools.

Some of these storage classes are used only internally by the "cc" and the "as". These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Figure 47.

Figure 47

Storage Class by Special Symbols.

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
*.x***fake**	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Also some storage classes are used only for certain special symbols. They are summarized in Figure 48.

Figure 48

Restricted Storage Classes.

Storage Class	Special Symbol
C_BLOCK	**.bb, .eb**
C_FCN	**.bf, .ef**
C_EOS	**.eos**
C_FILE	**.file**

Symbol Value Field

The meaning of the "value" of a symbol depends on its storage class. This relationship is summarized in Figure 49.

Figure 49

Storage Class and Value.

Storage Class	Meaning
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next **.file** symbol. That is, the **.file** entries form a 1-way linked list in the symbol table. If there are no more **.file** entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Figure 50.

Figure 50

Section Number.

Mnemonic	Section Number	Meaning
N_DEBUG	−2	Special symbolic debugging symbol
N_ABS	−1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol was defined

A special section number (−2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of −1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and **.eos** symbols. The **.text**, **.data**, and **.bss** symbols default to section numbers 1, 2, and 3, respectively.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply defined external symbol (i.e., FORTRAN common or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0 and the value of the symbol is a positive number giving the size of the symbol. When the files are combined, the link editor combines all the input symbols into one symbol with the section number of the **.bss** section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Figure 51.

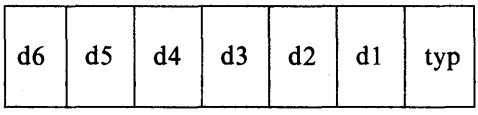
Figure 51

Section Number and Storage Class.

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the "cc". The VAX "cc" generates this information only if the **−g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is



Bits 0 through 3, called "typ", indicate one of the fundamental types given in Figure 52.

Figure 52

Fundamental Types.

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Bits 4 through 15 are arranged as six 2-bit fields marked "d1" through "d6." These "d" fields represent levels of the derived types given in Figure 53.

Figure 53

Derived Types.

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

The following examples demonstrate the interpretation of the symbol table entry representing type.

*char *func();*

Here *func* is the name of a function that returns a pointer to a character. The fundamental type of *func* is 2 (character), the d1 field is 2 (function), and the d2 field is 1 (pointer). Therefore, the type word in the symbol table for *func* contains the hexadecimal number 0x62, which is interpreted to mean "function that returns a pointer to a character."

*short *tabptr[10][25][3];*

Here *tabptr* is a 3-dimensional array of pointers to short integers. The fundamental type of *tabptr* is 3 (short integer); the d1, d2, and d3 fields each contains a 3 (array), and the d4 field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a "3-dimensional array of pointers to short integers."

Type Entries and Storage Classes

Figure 54 shows the type entries that are legal for each storage class.

Figure 54

Type Entries by Storage Class (Sheet 1 of 2)

Storage Class	----------"d" entry----------			"typ" entry
	Function?	**Array?**	**Pointer?**	**Basic Type**
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION

Type Entries by Storage Class (Sheet 2 of 2).

Storage Class	----------"d" entry----------			"typ" entry
	Function?	Array?	Pointer?	Basic Type
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION<, T_ENUM

Conditions for the "d" entries apply to d1 through d6, except that it is impossible to have two consecutive derived types of "function."

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have "array" as its first derived type.

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 55. This declaration may be found in the header file *syms.h*.

Figure 55

Symbol Table Entry Declaration.

```
struct syment
{
  union
  {
       char          _n_name[SYMNMLEN];
                    /* symbol name*/
       struct
       {
            long     _n_zeroes;
                    /* symbol name */

            long     _n_offset;
                    /* location in string table */
       } _n_n;
       char          _n_nptr[2];
                    /* allows overlaying */
       } _n;
       long           n_value;
                     /* value of symbol */

       short          n_scnum;
                     /* section number */

       unsigned short   n_type;
                     /* type and derived */

       char           n_sclass;
                     /* storage class */

       char           n_numaux;
                     /* number of aux entries */
};

  #define  n_name        _n._n_name
  #define  n_zeroes      _n._n_n._n_zeroes
  #define  n_offset      _n._n_n._n_offset
  #define  n_nptr        _n._n_nptr[1]

  #define  SYMNMLEN  8
  #define  SYMESZ    18    /* size of a symbol table entry */
```

Auxiliary Table Entries

Currently, there is at most one auxiliary entry per symbol. The auxiliary table entry contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Figure 56.

In Figure 56, "tagname" means any symbol name including the special symbol .x**fake**, and "fcname" and "arrname" represent any symbol name.

Any symbol that satisfies more than one condition in Figure 56 should have a union format in its auxiliary entry. Symbols that do not satisfy any of the above conditions should **NOT** have any auxiliary entry.

File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are 0, regardless of the size of the entry.

Sections

The auxiliary table entries for sections have the format as shown in Figure 57.

Figure 56

Auxiliary Symbol Table Entries.

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	file name
.text,.data, .bss	C_STAT	DT_NON	T_NULL	section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
fcname	C_EXT C_STAT	DT_FCN	(Note 1)	function
arrname **.bb**	(Note 2) C_BLOCK	DT_ARY DT_NON	(Note 1) T_NULL	array beginning of block
.eb	C_BLOCK	DT_NON	T_NULL	end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure union, enumeration	(Note 2)	DT_PTR DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Notes:
1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

Figure 57

Format for Auxiliary Table Entries.

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-6	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-17	-	-	unused (filled with zeroes)

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 58.

Figure 58

Tag Names Table Entries.

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeros)
6-7	unsigned short	x_size	size of strucrt, union,and enumeration
8-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	-	-	unused (filled with zeroes)

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 59.

Figure 59

Table Entries for End of Structures.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of struct, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Functions

The auxiliary table entries for functions have the format shown in Figure 60.

Figure 60

Table Entries for Functions.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x-lnnoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this point
16-17	unsigned short	x_tvndx	index of the function's address in the transfer vector table (not used in *UNIX system*)

Arrays

The auxiliary table entries for arrays have the format shown in Figure 61.

Figure 61

Table Entries for Arrays.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_lnno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	-	-	unused (filled with zeroes)

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Figure 62.

Figure 62

End of Block and Function Entries.

Bytes	Declaration	Name	Description
0-3	-	-	used (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-17	-	-	unused (filled with zeroes)

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Figure 63.

Figure 63

Format for Beginning of Block and Function.

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	-	-	unused (filled with zeroes)

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumerations symbols have the format shown in Figure 64.

Figure 64

Entries for Structures, Unions, and Numerations.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union, or numeration
8-17	-	-	unused (filled with zeroes)

Names defined by "typedef" may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;

struct people {
        char name[20];
        long id;
        };

typedef struct people EMPLOYEE;
```

The symbol "EMPLOYEE" has an auxiliary table entry in the symbol table but symbol "STUDENT" will not.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 65. This declaration may be found in the header file *syms.h*.

Figure 65

Auxiliary Symbol Table Entry.

```
union auxent {
        struct {
                        long    x_tagndx;
         union {
        struct {
                unsigned short  x_lnno;
                unsigned short  x_size;
        } x_lnsz;
                        long    x_fsize;
        } x_misc;
         union {
        struct {
                        long    x_lnnoptr;
                        long    x_endndx;
        } x_fcn;
        struct {
                unsigned short  x_dimen[DIMNUM];
        } x_ary;
        } x_fcnary;
                unsigned short  x_tvndx;
        } x_sym;
        struct {
                        char    x_fname[FILNMLEN];
        } x_file;
        struct {
                        long    x_scnlen;
                unsigned short  x_nreloc;
                unsigned short  x_nlinno;
        } x_scn;
        struct {
                        long    x_tvfill;
                unsigned short  x_tvlen;
                unsigned short  x_tvran[2];
        } x_tv;
}
#define FILNMLEN  14
#define DIMNUM    4
#define AUXENT    union auxent
#define AUXESZ    18
```

STRING TABLE

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table therefore are greater than or equal to 4.

For example, given a file containing two symbols (with names longer then eight characters, *long_name_1* and *another_one*) the string table has the format as shown in Figure 66.

Figure 66

String Table.

28			
'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

The index of *long_name_1* in the string table is 4 and the index of *another_one* is 16.

ACCESS ROUTINES

Supplied with every standard *UNIX* system release is a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file. In this way, you can concern yourself with the section you are interested in without knowing all the object file details.

The access routines can be divided into four categories:

1. Functions that open or close an object file.

2. Functions that read header or symbol table information.

3. Functions that position an object file at the start of a particular section of the object file.

4. A function that returns the symbol table index for a particular symbol.

These routines can be found in the library *libld.a* and are listed in Section 3 of the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines*. A summary of what is available can be found in the *UNIX Programmer's Manual—Volume 2: System Calls and Library Routines* under LDFCN(4).

ARBITRARY PRECISION DESK CALCULATOR LANGUAGE (BC)

GENERAL

The arbitrary precision desk calculator language (BC) is a language and compiler for doing arbitrary precision arithmetic under the UNIX operating system. The output of the compiler is interpreted and executed by a collection of routines that can input, output, and do arithmetic on infinitely large integers and on scaled fixed-point numbers. These routines are based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The BC language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. A small collection of library functions is also available, including *sin, cos, arctan, log, exponential,* and *Bessel* functions of integer order.

The BC compiler was written to make conveniently available a collection of routines (called DC) that are capable of doing arithmetic on integers of arbitrary size. The compiler is not intended to provide a complete programming language. It is a minimal language facility.

Some of the uses of this compiler are:

- Compile large integers
- Compute accurately to many decimal places
- Convert numbers from one base to another base.

There is a scaling provision that permits the use of decimal point notation. Provision is also made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal eight.

The actual limit on the number of digits that can be handled depends on the amount of core storage available. This is possible even on the smallest versions of the UNIX operating system.

The syntax of BC is very similar to that of the C language. This enables users who are familiar with C language to easily work with BC.

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the addition of two numbers (with the + operator) such as

142857 + 285714

the program responds immediately with the sum

428571.

The operators −, *, /, %, and ^ can also be used. They indicate subtraction, multiplication, division, remaindering, and integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the **unary** minus sign). The expression

7+−3

is interpreted to mean that −3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in power, then *, %, and /, and finally, + and −. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right.

a^b^c and a^(b^c)

are equivalent as are the two expressions

a*b*c and (a*b)*c.

However, BC shares with Fortran and C language the undesirable convention that

```
a/b*c  is equivalent to  (a/b)*c.
```

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named *x*. When, as in this case, the outermost operator is an "=", the assignment is performed; but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (see the part on "SCALING"). Entering the lines

```
x = sqrt(191)
x
```

produces the printed result

```
13
```

BASES

There are two special internal quantities; **ibase** (input base) and **obase** (output base). The contents of **ibase**, initially set to 10 (decimal), determines the base used for interpreting numbers read in. For example, the input lines

```
ibase = 8
11
```

produces the output line

```
9
```

and the system is ready to do octal to decimal conversions. Beware, however, of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement has no effect. For dealing in hexadecimal notation, the characters A through F are permitted in numbers (regardless of what base is in effect) and are interpreted as digits having values 10 through 15, respectively. The statement

```
ibase = A
```

changes the base to decimal regardless of what the current input base is. Negative and large positive input bases are permitted but are useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The content of **obase**, initially 10 (decimal), is used as the base for output numbers. The input lines

```
obase = 16
1000
```

produces the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted and are sometimes useful. For example, large numbers can be output in groups of five digits by setting **obase** to 100000. Strange output bases (i.e., 1, 0, or negative) are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with a backslash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Nondecimal output conversion of a

100-digit number takes about 3 seconds.

The **ibase** and **obase** have no effect on the course of internal computation or on the evaluation of expressions. They only affect input and output conversions, respectively.

SCALING

A third special internal quantity called **scale** is used to determine the scale of calculated quantities. The number of digits after the decimal point of a number is referred to as its scale. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations.

The contents of **scale** must be no greater than 99 and no less than 0. It is initially set to 0. However, appropriate scaling can be arranged when more than 99 fraction digits are required.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

- Addition and subtraction—The scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result.

- Multiplication—The scale of the result is never less than the maximum of the two scales of the operands and never more than the sum of the scales of the operands. Subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity **scale**.

- Division—The scale of a quotient is the contents of the internal quantity **scale**. The scale of a remainder is the sum of the scales of the quotient and the divisor.

- Exponentiation—The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.

- Square root—The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale**.

All of the internal operations are actually carried out in terms of integers with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The internal quantities **scale**, **ibase**, and **obase** can be used in expressions just like other variables. The input line

```
scale = scale + 1
```

increases the value of **scale** by one, and the input line

```
scale
```

causes the current value of **scale** to be printed.

The value of **scale** retains its meaning as a number of decimal digits to be retained in internal computation even when **ibase** or **obase** are not equal to 10. The internal computations (which are still conducted in decimal regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal, octal, or any other kind of digits.

FUNCTIONS

The name of a function is a single lowercase letter. Function names are permitted to coincide with simple variable names. Twenty-six different defined functions are permitted in addition to the 26 variable names. The input line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements which make up the body of the function ending with a right brace (}). The general form of a function is

```
define a(x) {
        . . .
        . . .
        return
}
```

Return of control from a function occurs when a **return** statement is executed or when the end of the function is reached. The **return** statement can take either of the two forms:

```
return
return(x)
```

In the first case, the value of the function is 0; and in the second, the value of the function is the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one **auto** statement in a function, and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return (exit). The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function *a*, when called, is the product of its two arguments, "x" and "y".

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: ().

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

causes the result 21.98 to be printed, and the line

```
z = a(a(3,4),5)
```

causes the result 60 to be printed.

SUBSCRIPTED VARIABLES

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name, and the expression in brackets is called the subscript. Only 1-dimensional arrays are permitted. The names of arrays are permitted to coincide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to 0 and less than or equal to 2047.

Subscripted variables may be used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function and thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other contexts.

CONTROL STATEMENTS

The **if**, **while**, and **for** statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way:

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the following six relational operators:

```
<   less than
>   greater than
<=  less than or equal to
>=  greater than or equal to
==  equal to
!=  not equal to
```

Beware of using "=" instead of "==" as a relational operator. Unfortunately, both of these are legal, so there will be no diagnostic message, but "=" will not do a comparison.

The **if** statement causes execution of its range if and *only if* the relation is true. Then control passes to the next statement in sequence.

The **while** statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range; and if the relation is false, control passes to the next statement beyond the range of the **while** statement.

The **for** statement begins by executing **expression1**. Then the relation is tested; and if true, the statements in the range of the **for** are executed. Then **expression2** is executed. The relation is then tested, etc. The typical use of the **for** statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which prints the integers from one to ten. The following are some examples of the use of the control statements:

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The input line

```
f(a)
```

prints "a" factorial if "a" is a positive integer. The following is the definition of a function that computes values of the binomial coefficient (m and n are assumed to be positive integers):

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

ADDITIONAL FEATURES

There are some additional language features that every user should know.

Normally, statements are typed one to a line. It is also permissible, however, to type several statements on a line by separating the statements by semicolons.

If an assignment statement is parenthesized, it then has a value; and it can be used anywhere that an expression can. For example, the input line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

The following is an example of a use of the value of an assignment statement even when it is not parenthesized. The input line

```
x = a[i=i+1]
```

causes a value to be assigned to *x* and also increments *i* before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language.

```
x=y=z  is the same as   x=(y=z)
x =+ y      "           x = x+y
x =- y      "           x = x-y
x =* y      "           x = x*y
x =/ y      "           x = x/y
x =% y      "           x = x%y
x =^ y      "           x = x^y
x++         "           (x=x+1)-1
x--         "           (x=x-1)+1
++x         "           x = x+1
--x         "           x = x-1
```

Warning: In some of these constructions, spaces are significant. There is a real difference between x=−y and x= −y. The first replaces x by x−y and the second by −y.

The following are three important things to remember when using BC programs:

- To exit a BC program, type **quit**.
- There is a comment convention identical to that of the C language. Comments begin with /* and end with */.
- There is a library of math functions that may be obtained by typing at command level:

  bc −l

This command loads a set of library functions that includes sine (**s**), cosine (**c**), arctangent (**a**), natural logarithm (**l**), exponential (**e**), and Bessel functions of integer order [**j(n,x)**]. The library sets the scale to 20, but it can be reset to another value.

If you type

bc file ...

the BC program reads and executes the named file or files before accepting commands from the keyboard. In this way, programs and function definitions are loaded.

BC APPENDIX

NOTATION

In the following pages, syntactic categories are in *italics* and literals are in **bold**. Material in brackets "[]" is optional.

TOKENS

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs, or comments. Newline characters or semicolons separate statements.

Comments are introduced by the characters /* and terminated by */.

There are three kinds of identifiers-ordinary, array, and function. All three types consist of single lowercase letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict. A program can have a variable named **x**, an array named **x**, and a function named **x**; all of which are separate and distinct.

The following are reserved keywords:

ibase	if
obase	break
scale	define
sqrt	auto
length	return
while	quit
for	

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A through F are also recognized as digits with values 10 through 15, respectively.

EXPRESSIONS

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

Named Expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

identifiers

Simple identifiers are named expressions. They have an initial value of zero.

array-name[expression]

Array elements are named expressions. They have an initial value of zero.

scale, ibase, and obase

The internal registers **scale**, **ibase**, and **obase** are all named expressions. The **scale** register is the number of digits after the decimal point to be retained in arithmetic operations. It has an initial value of zero. The **ibase** and **obase** registers are the input and output number radix, respectively. Both **ibase** and **obase** have initial values of ten.

Function Calls

function name ([expression[,expression..]])

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by

value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a **return** statement, the value of the function is the value of the expression in the parentheses of the **return** statement or is zero if no expression is provided or if there is no **return** statement.

sqrt(expression)

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of **scale**, whichever is larger.

length(expression)

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale(expression)

The result is the scale of the expression. The scale of the result is zero.

Constants

Constants are primitive expressions.

Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

The unary operators bind right to left.

−expression

The result is the negative of the expression.

++named-expression

The named expression is incremented by one. The result is the value of the named expression after incrementing.

−−named-expression

The named expression is decremented by one. The result is the value of the named expression after decrementing.

named-expression++

The named expression is incremented by one. The result is the value of the named expression before incrementing.

named-expression−−

The named expression is decremented by one. The result is the value of the named expression before decrementing.

The exponentiation operator binds right to left.

expression ^ expression

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If *a* is the scale of the left expression and *b* is the absolute value of the right expression, then the scale of the result is

$$\min(a \times b, \max(\text{scale}, a))$$

The operators *, /, and % bind left to right.

expression * expression

The result is the product of the two expressions. If *a* and *b* are the scales of the two expressions, then the scale of the result is

```
min(a+b,max(scale,a,b))
```

expression / expression

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

expression % expression

The % operator produces the remainder of the division of the two expressions. More precisely, *a%b* is *a−a/b*b*.

The scale of the result is the sum of the scale of the divisor and the value of **scale**.

The additive operators bind left to right.

expression + expression

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

expression − expression

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

The assignment operators bind right to left.

named-expression = expression

This expression results in assigning the value of the expression on the right to the named expression on the left.

named-expression =+ *expression*
named-expression =− *expression*
named-expression =* *expression*
named-expression =/ *expression*
named-expression =% *expression*
named-expression =^ *expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

RELATIONAL OPERATORS

Unlike all other operators, the relational operators are only valid as the object of an **if** or **while** statement or inside a **for** statement.

expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression == *expression*
expression != *expression*

STORAGE CLASSES

There are only two storage classes in BC-global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. The **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in C language. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

STATEMENTS

Statements must be separated by a semicolon or newline. Except where altered by control statements, execution is sequential.

When a statement is an expression unless the main operator is an assignment, the value of the expression is printed followed by a newline character.

Statements may be grouped together and used when one statement is expected by surrounding them with braces { }.

The following statement prints the string inside the quotes.

"any string"

if (*relation*) *statement*

The substatement is executed if the relation is true.

while (*relation*) *statement*

The **while** statement is executed while the relation is true. The test occurs before each execution of the statement.

for (*expression*; *relation*; *expression*) *statement*

The **for** statement is the same as

first-expression
while (*relation*) {
    statement
    last-expression
}

All three expressions must be present.

break

The **break** statement causes termination of a **for** or **while** statement.

auto *identifier*[,*identifier*]

The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name with empty square brackets. The **auto** statement must be the first statement in a function definition.

define ([*parameter*[,*parameter*...]]){
statements}

The **define** statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

return
return (*expression*)

The **return** statement causes the following:

- Termination of a function
- Popping of the auto variables on the stack
- Specifies the results of the function.

The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

The **quit** statement stops execution of a BC program and returns control to the UNIX system software when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

NOTES

INTERACTIVE DESK CALCULATOR (DC)

GENERAL

The **DC** program is an interactive desk calculator program implemented on the UNIX operating system to do arbitrary-precision integer arithmetic. It has provisions for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated by **DC** is limited only by available core storage. On typical implementations of the UNIX system, the size of numbers that can be handled varies from several hundred on the smallest systems to several thousand on the largest.

The **DC** program works like a stacking calculator using reverse Polish notation. Ordinarily, **DC** operates on decimal integers; but an input base, output base, and a number of fractional digits to be maintained can be specified.

A language called BC has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles the output which is interpreted by **DC**. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into **DC** are put on a pushdown stack. The **DC** commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then it is taken from the standard input.

DC COMMANDS

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number (e.g. 244)

The value of a number is pushed onto the stack. A number is an unbroken string of digits 0 through 9 and uppercase letters A through F (treated as digits with values 10 through 15, respectively). The number may be preceded by an underscore (_) to input a negative number and numbers may contain decimal points.

The top two values on the stack are added (+), subtracted (−), multiplied (*), divided (/), remaindered (%), or exponentiated (^) by using

+ − * / % ^

The two entries are popped off the stack, and the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. An exponent must not have any digits after the decimal point.

s*x*

The top of the main stack is popped and stored in a register named *x* (where *x* may be any character). If **s** is uppercase, *x* is treated as a stack; and the value is pushed onto it. Any character, even blank or newline, is a valid register name.

The value of register x is pushed onto the stack. Register x is not altered. If the l in

l*x*

is uppercase, register x is treated as a stack, and its top value is popped onto the main stack. All registers start with empty value which is treated as a zero by the command **l** and is treated as an error by the command **L**.

The following characters perform the stated tasks:

d

The top value on the stack is duplicated.

p

The top value on the stack is printed. The top value remains unchanged.

f

All values on the stack and in registers are printed.

x

Treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of **DC** commands.

[...]

Puts the bracketed character string onto the top of the stack.

q

Exits the program. If executing a string, the recursion level is popped by two. If **q** is uppercase, the top value on the stack is popped; and the string execution level is popped by that value.

<x >x =x !<x !>x !=x

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

v

Replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer.

!

Interprets the rest of the line as a UNIX software command. Control returns to **DC** when the command terminates.

c

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If **i** is uppercase, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If **o** is uppercase, the value of the output base is pushed onto the stack.

k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If **k** is uppercase, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

INTERNAL REPRESENTATION OF NUMBERS

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 to 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100s complement notation, which is analogous to twos complement notation for binary numbers. The high-order digit of a negative number is always −1 and all other digits are in the range 0 to 99. The digit preceding the high-order −1 digit is never a 99. The representation of −157 is 43,98,−1. This is called the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when it is convenient.

An additional byte is stored with each number beyond the high-order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high-order digit. The value of this extra byte is called the **scale factor** of the number.

THE ALLOCATOR

The **DC** program uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is through the allocator. Associated with each string in the allocator is a 4-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and **DC** is via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of two. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Leftover strings are put on the free list. If there are no larger strings, the allocator tries to combine smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long.

If a string of the proper length cannot be found, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in **DC**. If the allocator runs out of headers at any time in the process of trying to allocate a string, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end of string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

INTERNAL ARITHMETIC

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is

appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. The **scale** register limits the number of decimal places retained in arithmetic computations. The **scale** register may be set to the number on the top of the stack truncated to an integer with the **k** command. The **K** command may be used to push the value of **scale** on the stack. The value of **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations includes the exact effect of **scale** on the computations.

ADDITION AND SUBTRACTION

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

The addition is performed digit by digit from the low-order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers, replacing the high-order configuration 99,−1 by the digit −1. In any case, digits that are not in the range 0 through 99 must be brought into that range, propagating any carries or borrows that result.

MULTIPLICATION

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly follows the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low-order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

DIVISION

The scales are removed from the two operands. Zeros are appended, or digits are removed from the dividend to make the scale of the result of the integer division equal to the internal quantity **scale**. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise, the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. If it turns out to be one unit too low, the next trial quotient is larger than 99; and this is adjusted at the end of the process. The trial digit is multiplied by the divisor, the result subtracted from the dividend, and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form with propagation of carry as needed. The sign is set from the sign of the operands.

REMAINDER

The division routine is called, and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

SQUARE ROOT

The scale is removed from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand. The method used to compute the square root is Newton's method with successive approximations by the rule.

$$X_{n+1}=(X_n+Y/X_n)$$

The initial guess is found by taking the integer square root of the top two digits.

EXPONENTIATION

Only exponents with 0 scale factor are handled. If the exponent is 0, then the result is 1. If the exponent is negative, then it is made positive; and the base is divided into 1. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared, and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

INPUT CONVERSION AND BASE

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with an underscore (_). The hexadecimal digits A through F correspond to the numbers 10 through 15 regardless of input base. The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base (**ibase**) is initialized to 10 (decimal) but may, for example, be changed to 8 or 16 for octal or hexadecimal to decimal conversions. The command **I** pushes the value of the input base on the stack.

OUTPUT COMMANDS

The command **p** causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers are output by typing the command **f**. The **o** command is used to change the output base (**obase**). This command uses the top of the stack truncated to an integer as the base for all further output. The output base in initialized to 10 (decimal). It works correctly for any base. The command **O** pushes the value of the output base on the stack.

OUTPUT FORMAT AND BASE

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 are used for decimal-octal or decimal-hexadecimal conversions.

INTERNAL REGISTERS

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**. The command **s***x* pops the top of the stack and stores the result in register *x*. The *x* can be any character. The command **l***x* puts the contents of register *x* on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

STACK COMMANDS

The command **c** clears the stack. The command **d** pushes a duplicate of the number on the top of the stack onto the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

SUBROUTINE DEFINITIONS AND CALLS

Enclosing a string in brackets "[]" pushes the ASCII string on the stack. The **q** command quits or (in executing a string) pops the recursion levels by two.

INTERNAL REGISTERS—PROGRAMMING DC

The load and store commands, together with "[]" to store strings, the **x** command to execute, and the testing commands (**<**, **>**, **=**, **!<**, **!>**, **!=**), can be used to program **DC**. The **x** command assumes the top of the stack is a string of **DC** commands and executes it. The testing commands compare the top two elements on the stack and, if the relation holds, execute the register that follows the relation. For example, to print the numbers 0 through 9,

```
[lip1+ si li10>a]sa
0si lax
```

PUSHDOWN REGISTERS AND ARRAYS

These commands are designed for use by a compiler, not directly by programmers. They involve pushdown registers and arrays. In addition to the stack that commands work on, **DC** can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**. **S***x* pushes the top value of the main stack onto the stack for the register *x*. **L***x* pops the stack for register *x* and puts the result on the main stack. The commands **s** and **l** also work on registers but not as pushdown stacks. The command **l** does not affect the top of the register stack, but **s** destroys what was there before.

The commands to work on arrays are : and ;. The command :*x* pops the stack and uses this value as an index into the array *x*. The next element on the stack is stored at this index in *x*. An index must be greater than or equal to 0 and less than 2048. The command ;*x* loads the main stack from the array *x*. The value on the top of the stack is the index into the array *x* of the value to be loaded.

MISCELLANEOUS COMMANDS

The command ! interprets the rest of the line as a UNIX software command and passes it to the UNIX operating system to execute. One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

DESIGN CHOICES

The real reason for the use of a dynamic storage allocator is that a general purpose program can be used for a variety of other tasks. The allocator has some value for input and for compiling (i.e., the bracket [...] commands) where it cannot be known in advance how long a string will be. The result is that at a modest cost in execution time:

- All considerations of string allocation and sizes of strings are removed from the remainder of the program.
- Debugging is made easier.
- The allocation method used wastes approximately 25 percent of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5 percent in space debugging was made a great deal easier, and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all **DC** commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases is to provide an understandable means of proceeding after a change of base or scale (when numbers had already been entered). An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** is interpreted in the current input or output base, then a change of base or scale in the midst of a computation causes great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program. It is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The rationale for the choices for the scales of the results of arithmetic is that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give them the result 5.017 without requiring to unnecessarily specify rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands. It seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places, and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

NOTES

LEXICAL ANALYZER GENERATOR (LEX)

GENERAL

The **Lex** is a program generator that produces a program in a general purpose language that recognizes regular expressions. It is designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching. The regular expressions are specified by you (the user) in the source specifications given to **Lex.** The **Lex** program generator source is a table of regular expressions and corresponding program fragments. The table is translated to a program that reads an input stream, copies the input stream to an output stream, and partitions the input into strings that match the given expressions. As each such string is recognized, the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by **Lex.** The program fragments written by you are executed in the order in which the corresponding regular expressions occur in the input stream.

The user supplies the additional code beyond expression matching needed to complete the tasks, possibly including codes written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for your program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while your freedom to write actions is unimpaired.

The **Lex** written code is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages". Just as general purpose languages can produce code to run on different computer hardware, **Lex** can write code in different host languages. The host language is used for the output code generated by **Lex** and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes **Lex** adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is the C language, although Fortran (in the form of Ratfor) has been available in the past. The **Lex** generator exists on the UNIX operating system, but the codes generated by **Lex** may be taken anywhere the appropriate compilers exist.

The **Lex** program generator turns the user's expressions and actions (called **source**) into the host general purpose language; the generated program is named **yylex.** The **yylex** program recognizes expressions in a stream (called **input**) and performs the specified actions for each expression as it is detected.

For example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$        ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules. This rule contains a regular expression that matches one or more instances of the characters blank or tab (written for visibility, in accordance with the C language convention) and occurs prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the **$** indicates "end of line," as in **QED.** No action is specified, so the program generated by **Lex** *yylex()* ignores these characters. Everything else is copied. To change any remaining string of blanks or tabs to a single blank, add another rule.

```
%%
[ \t]+$        ;
[ \t]+         printf(" ");
```

The coded instructions (generated for this source) scans for both rules at once, observes (at the termination of the string of blanks or tabs) whether or not there is a newline character, and then executes the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

The **Lex** program generator can be used alone for simple transformations or for analysis and statistics gathering on a lexical level. The **Lex** generator can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface **Lex** and **yacc.** The **Lex** program recognizes only regular expressions; **yacc** writes parsers that accept a large class of context free grammars but requires a lower level analyzer to recognize input tokens. Thus, a combination of **Lex** and **yacc** is often appropriate. When used as a preprocessor for a later parser generator, **Lex** is used to partition the input stream; and the parser generator assigns structure to the resulting pieces.

Additional programs, written by other generators or by hand, can be added easily to programs written by **Lex.** You will realize that the name **yylex** is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **Lex** simplifies interfacing.

In the program written by **Lex,** the user's fragments (representing the **actions** to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions or to add subroutines outside this action routine.

The **Lex** program generator is not limited to a source that can be interpreted on the basis of one character look-ahead. For example, if there are two rules, one looking for "ab" and another for "abcdefg" and the input stream is "abcdefh," **Lex** recognizes "ab" and leaves the input pointer just before "cd ...". Such backup is more costly than the processing of simpler languages.

LEX SOURCE

The general format of **Lex** source is

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The first % % is required to mark the beginning of the rules, but the second % % is optional. The absolute minimum **Lex** program is

```
%%
```

(no definitions, no rules) which translates into a program that copies the input to the output unchanged.

In the outline of **Lex** programs shown above, the rules represent your control decisions. They are in a table containing

- A left column with regular expressions
- A right column with actions and program fragments to be executed when the expressions are recognized.

Thus an individual rule might be

```
integer    printf("found keyword INT");
```

to look for the string **integer** in the input stream and print the message "found keyword INT" whenever it appears. In this example, the host procedural language is C, and the C language library function **printf** is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C language expression, it can just be given on the right side of the line; if it is compound or takes more than a line, it should be enclosed in braces. As a more useful example, suppose you desire to change a number of words from British to American spelling. The **Lex** rules such as:

```
colour        printf("color");
mechanize       printf("mechanize");
petrol        printf("gas");
```

would be a start. These rules are not sufficient since the word "petroleum" would become "gaseum".

LEX REGULAR EXPRESSIONS

The definitions of regular expressions are very similar to those in **QED**. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; the regular expression

```
integer
```

matches the string "integer" wherever it appears, and the expression

```
a57D
```

looks for the string "a57D".

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator " indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus:

```
xyz"++"
```

matches the string **xyz++** when it appears. Note that a part of a string may be quoted. It is harmless, but unnecessary, to quote an ordinary text character; the expression

```
"xyz++"
```

is equivalent to the one above. Thus, by quoting every nonalphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters and is safe should further extensions to **Lex** lengthen the list.

An operator character may also be turned into a text character by preceding it with a backslash (\) as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C language escapes with

\ are recognized: **\n** is newline, **\t** is tab, and **\b** is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, **\n** must be used; it is not required to escape tab and backspace. Every character except blank, tab, newline, and the list of operator characters above is always a text character.

Character Classes

Classes of characters can be specified using the operator pair **[]**. The construction **[abc]** matches a single character which may be "a", "b", or "c". Within square brackets, most operator meanings are ignored. Only three characters are special; these are \, −, and ^. The − character indicates ranges. For example,

```
[a−z0−9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and gets a warning message (e.g., [0-z] in ASCII is many more characters than is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus:

```
[−+0−9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket to indicate that the resulting string is complemented with respect to the computer character set. Thus:

```
[^abc]
```

matches all characters except "a", "b", or "c", including all special or control characters; or

```
[^a−zA−Z]
```

is any character that is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character (dot)

is the class of all characters except newline. Escaping into octal is possible although nonportable.

```
[\40–\176]
```

matches all printable ASCII characters from octal 40 (blank) to octal 176 (tilde).

Optional Expressions

The operator ? indicates an optional element of an expression. Thus:

```
ab?c
```

matches either "ac" or "abc".

Repeated Expressions

Repetitions of classes are indicated by the operators * and +. For example,

```
a*
```

is any number of consecutive "a" characters, including zero; while

```
a+
```

is one or more instances of "a". For example,

```
[a-z]+
```

is all strings of lowercase letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator | indicates alternation

```
(ab|cd)
```

matches either "ab" or "cd". Note that parentheses are used for grouping; although they are not necessary on the outside level,

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions.

```
(ab|cd+)?(ef)*
```

matches such strings as "abefef", "efefef", "cdef", or "cddd"; but not "abc", "abcd", or "abcdef".

Context Sensitivity

The **Lex** program recognizes a small amount of surrounding context. The two simplest operators for this are ^ and **$.** If the first character of an expression is ^, the expression is only matched at the beginning of a line (after a newline character or at the beginning of the input stream). This never conflicts with the other meaning of ^ (complementation of character classes) since that only applies within the **[]** operators. If the very last character is **$**, the expression is only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the **/** operator character which indicates trailing context. The expression

```
ab/cd
```

matches the string "ab" but only if followed by "cd". Thus:

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in **Lex** by "start conditions" as explained later. If a rule is only to be executed when the **Lex** automaton interpreter is in start condition **x,** the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition **ONE**, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

Repetitions and Definitions

The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example,

```
{digit}
```

looks for a predefined string named "digit" and inserts it at that point in the expression. The definitions are given in the first part of the **Lex** input before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of "a".

Finally, initial % is special being the separator for **Lex** source segments.

LEX ACTIONS

When an expression written as above is matched, **Lex** executes the corresponding action. This part describes some features of **Lex** that aid in writing actions. Note that there is a default action that consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus, the **Lex** user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When **Lex** is being used with **yacc,** this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule that merely copies can be omitted. Also, a character combination that is omitted from the rules and that appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C language null statement, ; as an action causes this result. A frequent rule is

```
[ \t\n]    ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character | which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "    |
"\t"   |
"\n"   ;
```

with the same result although in different style. The quotes around **\n** and **\t** are not required.

In more complex actions, you may often want to know the actual text that matched some expression like "[a-z]+". The **Lex** program leaves this text in an external character array. Thus, to print the name found, a rule like

```
[a−z]+    printf("%s", yytext);
```

prints the string in *yytext[]*. The C language function **printf** accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and s indicating string type), and the data are the characters in *yytext[]*. This places the matched string on the output. This action is so common that it may be written as ECHO.

```
[a−z]+    ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule like this one which merely specifies the default action. Such rules are often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches **read,** it normally matches the instances of **read** contained in **bread** or **readjust.** To avoid this, a rule of the form "[a-z]+" is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence, **Lex** also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, write

```
[a−zA−Z]+    {words++; chars += yyleng;}
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng−1]
```

Occasionally, a **Lex** action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless(n)* may be called to indicate that not all the characters matched by the currently

successful expression are wanted right now. The argument "n" indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of look ahead offered by the / operator but in a different form.

Example:

Consider a language that defines a string as a set of characters between quotation (") marks and provides that to include a (") in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*  {
        if (yytext[yyleng-1] == '\\')
            yymore();
        else
            ... normal user processing
        }
```

will, when faced with a string such as **"abc\"def"**, first match the five characters **"abc**; then the call to *yymore()* will cause the next part of the string **"def** to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C language problem of distinguishing the ambiguity of "=-a ". Suppose it is desired to treat this as "=- a" but also to print a message: a rule might be

```
=-[a-zA-Z] {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=- ". Alternatively, it might be desired to treat this as "=-a ". To do this, just return the minus sign as well as the letter to the input.

```
=-[a-zA-Z] {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

performs the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case, and

```
=/-[A-Za-z]
```

in the second; no backup is required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=-3", however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, **Lex** also permits access to the I/O routines it uses. They are as follows:

1. *input()* returns the next input character.

2. *output(c)* writes the character "c" on the output.

3. *unput(c)* pushes the character "c" back onto the input stream to be read later by *input()*.

By default, these routines are provided as macro definitions; but the user can override them and supply private versions. These routines define the relationship between external files and internal characters and must all be retained or modified consistently. They may be redefined to cause input or output to be transmitted to or from strange places including other programs or

internal memory. The character set used must be consistent in all routines and a value of zero returned by *input* must mean end of file. The relationship between *unput* and *input* must be retained or the **Lex** look ahead will not work. The **Lex** program does not look ahead at all if it does not have to, but every rule ending in **+**, *****, **?**, or **$** or containing **/** implies look ahead. Look ahead is also necessary to match an expression that is a prefix of another expression. The standard **Lex** library imposes a 100-character limit on backup.

Another **Lex** library routine that you may sometimes want to redefine is *yywrap()* which is called whenever **Lex** reaches an end of file. If *yywrap* returns a 1, **Lex** continues with the normal wrap up on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs **Lex** to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc., at the end of a program. Note that it is not possible to write a normal rule that recognizes end of file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied, a file containing nulls cannot be handled since a value of 0 returned by *input* is taken to be end of file.

AMBIGUOUS SOURCE RULES

The **Lex** program can handle ambiguous specifications. When more than one expression can match the current input, **Lex** chooses as follows:

1. The longest match is preferred.

2. Among rules that matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer    keyword action ...;
[a-z]+     identifier action ...;
```

are to be given in that order. If the input is "integers", it is taken as an identifier because

"[a-z]+"

matches eight characters while "integer" matches only seven. If the input is "integer", both rules match seven characters; and the keyword rule is selected because it was given first. Anything shorter (e.g., "int") does not match the expression "integer" and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example:

```
'.*'
```

might appear to be a good way of recognizing a string in single quotes. However, it is an invitation for the program to read far ahead looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^\n]*'
```

which, on the above input, stops after **('first')**. The consequences of errors like this are mitigated by the fact that the dot (.) operator does not match newline. Thus expressions like .* stop on the current line. Do not try to defeat this with expressions like **[.\n]+** or equivalents; the **Lex** generated program tries to read the entire input file causing internal buffer overflows.

Note that **Lex** is normally partitioning the input stream not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count

occurrences of both "she" and "he" in an input text. Some **Lex** rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides "he" and "she". Remember that dot (.) does not include newline. Since "she" includes "he", **Lex** normally *does not* recognize the instances of "he" included in "she" since once it has passed a "she" those characters are gone.

Sometimes the user desires to override this choice. The action *REJECT* means "go do the next alternative". It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose you really want to count the included instances of "he". Use the following rule to change the previous example to accomplish the task.

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

After counting each expression, it is rejected; whenever appropriate, the other expression is then counted. In this example, you could note that "she" includes "he" but not vice versa and omit the *REJECT* action on "he". In other cases, it is not possible to state which input characters are in both classes.

Consider the two rules

```
a[bc]+      { ... ; REJECT;}
a[cd]+      { ... ; REJECT;}
```

If the input is "ab", only the first rule matches, and on "ad" only the second matches. The input string "accb" matches the first rule for four characters and then the second rule for three characters. In contrast, the input "accd" agrees with the second rule for four characters and then the first rule for three.

In general, *REJECT* is useful whenever the purpose of **Lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally, the digrams overlap, that is the word "the" is considered to contain both "th" and "he". Assuming a 2-dimensional array named *digram[]* to be incremented, the appropriate source is

```
%%
[a-z][a-z]    {digram[yytext[0]][yytext[1]]++; REJECT;}
.             ;
\n            ;
```

where the *REJECT* is necessary to pick up a letter pair beginning at every character rather than at every other character.

The action *REJECT* does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found and *REJECT* executed the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

LEX SOURCE DEFINITIONS

Recalling the format of the **Lex** source,

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far, only the rules have been described. You need additional options to define variables for use in the program and for use by **Lex**. Variables can go either in the definitions section or in the rules section.

Remember **Lex** is generating the rules into a program. Any source not intercepted by **Lex** is copied into the generated program. There are three classes of such things.

1. Any line not part of a **Lex** rule or action that begins with a blank or tab is copied into the **Lex** generated program. Such source input prior to the first %% delimiter is external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by **Lex** which contains the actions. This material must look like program fragments and should precede the first **Lex** rule.

   Lines that begin with a blank or tab and that contain a comment are passed through to the generated program. This can be used to include comments in either the **Lex** source or the generated code; the comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1 or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the **Lex** output.

Definitions intended for **Lex** are given before the first %% delimiter. Any line in this section not contained between %{ and %} and beginning in column 1 is assumed to define **Lex** substitution strings. The format of such lines is

name translation

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the **{name}** syntax in a rule. Using **{D}** for the digits and **{E}** for an exponent field, for example, abbreviate rules to recognize numbers

```
D                [0-9]
E                [DEde][-+]?{D}+
%%
{D}+               printf("integer");
{D}+"."{D}*({E})?  |
{D}*"."{D}+({E})?  |
{D}+{E}            printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field. The first requires at least one digit before the decimal point, and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as "35.EQ.I", which does not contain a real number, a context-sensitive rule such as:

```
[0-9]+/"."EQ     printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within **Lex** itself for larger source programs. These possibilities are discussed later.

USAGE

There are two steps in compiling a **Lex** source program. First, the **Lex** source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded usually with a library of **Lex** subroutines. The generated program is on a file named *lex.yy.c*. The I/O library is defined in terms of the C language standard library.

On the UNIX operating system, the library is accessed by the loader flag **−ll.** So an appropriate set of commands is

```
lex source
cc lex.yy.c −ll
```

The resulting program is placed on the usual file *a.out* for later execution. To use **Lex** with **yacc,** see part "LEX AND YACC". Although the default **Lex** I/O routines use the C language standard library, the **Lex** automata themselves do not do so; if private versions of *input, output,* and *unput* are given, the library is avoided.

LEX AND YACC

To use **Lex** with **yacc,** observe that **Lex** writes a program named *yylex()* (the name required by **yacc** for its analyzer). Normally, the default main program on the **Lex** library calls this routine; but if **yacc** is loaded and its main program is used, **yacc** calls *yylex()*. In this case, each **Lex** rule ends with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc**'s names for tokens is to compile the **Lex** output file as part of the **yacc** output file by placing the line

```
# include "lex.yy.c"
```

in the last section of **yacc** input. If the grammar is to be named "good" and the lexical rules are to be named "better", the UNIX software command sequence could be

```
yacc good
lex better
cc y.tab.c −ly −ll
```

The **yacc** library **(−ly)** should be loaded before the **Lex** library to obtain a main program that invokes the **yacc** parser. The generations of **Lex** and **yacc** programs can be done in either order.

EXAMPLES

As a problem, consider copying an input file while adding three to every positive number divisible by seven. A suitable **Lex** source program follows:

```
%%
        int k;
[0-9]+    {
        k = atoi(yytext);
        if (k%7 == 0)
           printf("%d", k+3);
        else
           printf("%d",k);
        }
```

The rule "[0-9]+" recognizes strings of digits; *atoi()* converts the digits to binary and stores the result in "k". The operator % (remainder) is used to check whether "k" is divisible by seven; if it is, "k" is incremented by three as it is written out. It may be objected that this program alters such input items as "49.63" or "X7". Furthermore, it increments the absolute value of all negative numbers divisible by seven. To avoid this, add a few more rules after the active one, as here:

```
%%
                    int k;
-?[0-9]+              {
                    k = atoi(yytext);
                    printf("%d", k%7 == 0 ? k+3 : k);
                    }
-?[0-9.]+             ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a dot (.) or preceded by a letter will be picked up by one of the last two rules and not changed. The "if-else" has been replaced by a C language conditional expression to save space; the form "a?b:c" means "if a then b else c".

For an example of statistics gathering, here is a program that histograms the lengths of words, where a word is defined as a string of letters:

```
            int lengs[100];
%%
[a-z]+      lengs[yyleng]++;
.           |
\n          ;
%%
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram while producing no output. At the end of the input, it prints the table. The final statement "return(1);" indicates that **Lex** is to perform wrap up. If *yywrap* returns zero (false), it implies that further input is available and the program is to continue reading and processing. Providing a *yywrap* (that never returns true) causes an infinite loop.

LEFT CONTEXT SENSITIVITY

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator recognizing immediately preceding left context just as **$** recognizes immediately following right context. Adjacent left context could be extended to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful since often the relevant left context appeared some time earlier such as at the beginning of a line.

This part describes three means of dealing with different environments: a simple use of flags (when only a few rules change from one environment to another), the use of "start conditions" on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules that

recognize the need to change the environment in which the following input text is analyzed and that set a parameter to reflect the change. This may be a flag explicitly tested by the user's action code; this is the simplest way of dealing with the problem since **Lex** is not involved at all. It may be more convenient, however, to have **Lex** remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It is only recognized when **Lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word "magic" to "first" on every line which began with the letter "a", changing "magic" to "second" on every line which began with the letter "b", and changing "magic" to "third" on every line which began with the letter "c". All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag.

```
  int flag.
%%
^a {flag = 'a'; ECHO;}
^b {flag = 'b'; ECHO;}
^c {flag = 'c'; ECHO;}
\n {flag =  0 ; ECHO;}
magic {
  switch (flag)
  {
  case 'a': printf("first"); break;
  case 'b': printf("second"); break;
  case 'c': printf("third"); break;
  default: ECHO; break;
  }
}
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to **Lex** in the definitions section with a line reading

%Start name1 name2 ...

where the conditions may be named in any order. The word "Start" may be abbreviated to "s" or "S". The conditions may be referenced at the head of a rule with <> brackets;

<name1>expression

is a rule that is only recognized when **Lex** is in the start condition **name1.** To enter a start condition, execute the action statement

BEGIN name1;

which changes the start condition to **name1.** To resume the normal state

BEGIN 0;

resets the initial condition of the **Lex** automaton interpreter. A rule may be active in several start conditions.

<name1,name2,name3>

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written as follows:

```
%START AA BB CC
%%
^a              {ECHO; BEGIN AA;}
^b              {ECHO; BEGIN BB;}
^c              {ECHO; BEGIN CC;}
\n              {ECHO; BEGIN 0;}
<AA>magic         printf("first");
<BB>magic         printf("second");
<CC>magic         printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but **Lex** does the work rather than the user's code.

CHARACTER SET

The programs generated by **Lex** handle character I/O only through the routines *input(), output(),* and *unput()*. Thus, the character representation provided in these routines is accepted by **Lex** and used to return values in *yytext()*. For internal use, a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter **a** is represented in the same form as the character constant **'a'**. If this interpretation is changed by providing I/O routines that translate the characters, **Lex** must be given a translation table that is in the definitions section and must be bracketed by lines containing only **%T**; the translation table contains lines of the form

{integer} {character string}

which indicate the value associated with each character.

SUMMARY OF SOURCE FORMAT

The general form of a **Lex** source file is

{definitions}
%%
{rules}
%%
{user subroutines}

The definitions section contains a combination of

1. Definitions in the form "name space translation".
2. Included code in the form "space code".

3. Included code in the form:

```
%{
code
%}
```

4. Start conditions given in the form:

```
%S name1 name2 ...
```

5. Character set tables in the form:

```
%T
number space character-string
...
%T
```

6. Changes to internal array sizes in the form:

```
%x  nnn
```

where "nnn" is a decimal integer representing an array size and "a" selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in **Lex** use the following operators:

x	the character "x".
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x–z]	the characters x, y, or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	m through n occurrences of x.

CAVEATS AND BUGS

There are pathological expressions that produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found and *REJECT* executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

YET ANOTHER COMPILER-COMPILER (yacc)

GENERAL

The **yacc** program provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification of the input process. This includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. The **yacc** program then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*. When one of these rules has been recognized, then user code (supplied for this rule, an **action**) is invoked. Actions have the ability to return values and make use of the values of other actions.

The **yacc** program is written in a portable dialect of the C language, and the actions and output subroutine are in the C language as well. Moreover, many of the syntactic conventions of **yacc** follow the C language.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

where "date", "month_name", "day", and "year" represent structures of interest in the input process; presumably, "month name", "day", and "year" are defined elsewhere. The comma is enclosed in single quotes. This implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in controlling the input. With proper definitions, the input

```
July 4, 1776
```

might be matched by the rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizes the lower-level structures,

and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a "terminal symbol", while the structure recognized by the parser is called a "nonterminal symbol". To avoid confusion, terminal symbols will usually be referred to as "tokens".

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

        ...

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer only needs to recognize individual letters, and "month name" is a nonterminal symbol. Such low-level rules tend to waste time and space and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a "month name" is seen. In this case, "month name" is a "token".

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

on input. In most cases, this new rule could be "slipped in" to a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructions which are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The **yacc** program has been extensively used in numerous practical applications, including **lint**, the Portable C Compiler, and a system for typesetting mathematics.

The remainder of this document describes the following subjects as they relate to **yacc**

- Basic process of preparing a **yacc** specification
- Parser operation
- Handling ambiguities
- Handling operator precedences in arithmetic expressions
- Error detection and recovery
- The operating environment and special features of the parsers **yacc** produces

- Suggestions to improve the style and efficiency of the specifications
- Advanced topics.

In addition, there are four appendices. In the **EXAMPLES** section, **A Simple Example** is a brief example and **YACC Input Syntax** is a summary of the **yacc** input syntax. **An Advanced Example** gives an example using some of the more advanced features of **yacc**, and Appendix 12.4 describes mechanisms and syntax no longer actively supported but provided for historical continuity with older versions of **yacc**.

BASIC SPECIFICATIONS

Names refer to either tokens or nonterminal symbols. The **yacc** program requires token names to be declared as such. In addition, it is often desirable to include the lexical analyzer as part of the specification file. It may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent (% %) marks. (The percent symbol is generally used in **yacc** specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

when each section is used.

The declaration section may be empty, and if the programs section is omitted, the second % % mark may also be omitted. The smallest legal **yacc** specification is

```
%%
rules
```

since the other two sections may be omitted.

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal. They are enclosed in **/* ... */**, as in C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form

```
A : BODY ;
```

where "A" represents a nonterminal name, and "BODY" represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of arbitrary length and may be made up of letters, dots, underscores, and noninitial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ('). As in C language, the backslash (\) is an escape character within literals, and all the C language escapes are recognized. Thus:

```
'\n'    newline
'\r'    return
'\''    single quote ( ' )
'\\'    backslash ( \ )
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  "xxx" in octal
```

are understood by **yacc**. For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar ⟨|⟩ can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to **yacc** as

```
A : B C D
  | E F
  | G
  ;
```

by using the vertical bar. It is not necessary that all grammar rules with the same left side appear together in the grammar rules section although it makes the input much more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by

```
empty : ;
```

which is understood by **yacc**.

Names representing tokens must be declared. This is most simply done by writing

```
%token name1 name2 ...
```

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the *start symbol* has particular importance. The parser is designed to recognize the start symbol. Thus, this symbol represents the largest, most general structure described by the grammar rules.

By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare the start symbol explicitly in the declarations section using the **%start** keyword

```
%start  symbol
```

to define the start symbol.

The end of the input to the parser is signaled by a special token, called the *end-marker*. If the tokens up to but not including the end-marker form a structure that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as "end of file" or "end of record".

ACTIONS

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements enclosed in curly braces ({} and {}). For example:

```
A   : '(' B ')'
    {
      hello( 1, "abc" );
    }
```

and

```
XXX  : YYY  ZZZ
     {
       printf("a message\n");
       flag = 25;
     }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign symbol (**$**) is used as a signal to **yacc** in this context.

To return a value, the action normally sets the pseudo-variable **$$** to some value. For example, the action

```
{ $$ = 1; }
```

does nothing but return the value of one.

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables **$1, $2, ...**, which refer to the values returned by the components of the right side of a rule, reading from left to right. If the rule is

```
A  : B  C  D  ;
```

then **$2** has the value returned by C, and **$3** the value returned by D.

The rule

```
expr  :  '('  expr  ')'  ;
```

provides a more concrete example. The value returned by this rule is usually the value of the "expr" in parentheses. This can be indicated by

```
expr  :   '(' expr ')'
      {
          $$ = $2 ;
      }
```

By default, the value of a rule is the value of the first element in it (**$1**). Thus, grammar rules of the form

```
A  :  B  ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully parsed. The **yacc** permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value accessible through the usual **$** mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A  :  B
      {
        $$ =1;
      }
      C
  {
      x = $2;
      y = $3;
  }
  ;
```

the effect is to set *x* to 1 and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by **yacc** by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. The **yacc** program actually treats the above example as if it had been written

```
$ACT  :  /* empty */
      {
         $$ = 1;
      }
      ;

A     :  B $ACT C
      {
         x = $2;
         y = $3;
      }
      ;
```

where **$ACT** is an empty action.

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node* written so that the call

```
node( L, n1, n2 )
```

creates a node with label L and descendants n1 and n2 and returns the index of the newly created node. Then parse tree can be built by supplying actions such as

```
expr  :  expr '+' expr
      {
         $$ = node( '+', $1, $3 );
      }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example:

```
%{  int variable = 0;  %}
```

could be placed in the declarations section making "variable" accessible to all of the actions. The **yacc** parser uses only names beginning with **yy**. The user should avoid such names.

In these examples, all the values are integers. A discussion of values of other types is found in the part "ADVANCED TOPICS".

LEXICAL ANALYSIS

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc** or the user. In either case, the **#define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like

```
yylex()
{
  extern int yylval;
  int c;
  ...
  c = getchar();
  ...
  switch( c )
  {
    ...
  case '0':
  case '1':
    ...
  case '9':
    yylval = c-'0';
    return( DIGIT );
    ...
  }
  ...
```

to return the appropriate token.

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling and should not be used naively.

As mentioned above, the token numbers may be chosen by **yacc** or the user. In the default situation, the numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately

followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the end-marker must have token number 0 or negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the **lex** program. These lexical analyzers are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **Lex** can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

PARSER OPERATION

The **yacc** program turns the specification file into a C language program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, however, is relatively simple and understanding how it works will make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *look-ahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

The machine has only four actions available—*shift*, *reduce*, *accept*, and *error*. A step of the parsor is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have

one, it calls *yylex* to obtain the next token.

2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look-ahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56 there may be an action

IF shift 34

which says, in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether to reduce or not (usually it is not necessary). In fact, the default action (represented by a dot) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The action

. reduce 18

refers to grammar rule 18, while the action

IF shift 34

refers to state 34.

Suppose the rule

```
A : x y z ;
```

is being reduced. The reduce action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z* and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the look-ahead token is cleared by a shift but is not affected by a goto. In any case, the uncovered state contains an entry such as

```
A goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the reduce action "turns back the clock" in the parse popping the states off the stack to go back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stacks. The uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable "yylval" is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable "yyval" is copied onto the value stack. The pseudo-variables **$1**, **$2**, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the

specification. This action appears only when the look-ahead token is the end-marker and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Consider:

```
%token DING DONG DELL
%%
rhyme  :  sound  place
       ;
sound  :  DING  DONG
       ;
place  :  DELL
       ;
```

as a **yacc** specification.

When **yacc** is invoked with the **-v** option, a file called *y.output* is produced with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_  (1)

        . reduce 1

state 5
        place : DELL_  (3)

        . reduce 3

state 6
        sound : DING DONG_  (2)

        . reduce 2
```

where the actions for each state are specified and there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen and what is yet to come in each rule. The following input

```
DING DONG DELL
```

can be used to track the operations of the parser. Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read and becomes the look-ahead token. The action in state 0 on *DING* is *shift 3*, state 3 is pushed onto the stack, and the look-ahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read and becomes the look-ahead token. The action in state 3 on the token *DONG* is *shift 6*, state 6 is pushed onto the stack, and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by

```
sound : DING DONG
```

which is rule 2. Two states, 6 and 3, are popped off of the stack uncovering state 0. Consulting the description of state 0 (looking for a goto on *sound*),

```
sound goto 2
```

is obtained. State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, *DELL*, must be read. The action is *shift 5*, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place* (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained indicated by **$end** in the *y.output* file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG*

DELL DELL, etc. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

AMBIGUITY AND CONFLICTS

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr  :  expr '—' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr — expr — expr
```

the rule allows this input to be structured as either

```
( expr — expr ) — expr
```

or as

```
expr — ( expr — expr )
```

(The first is called "left association", the second "right association".)

The **yacc** program detects such ambiguities when it is attempting to build the parser. Given the input

```
expr — expr — expr
```

consider the problem that confronts the parser. When the parser has read the second expr, the input seen

expr – expr

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to "expr" (the left side of the rule). The parser would then read the final part of the input

– expr

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, if the parser sees

expr – expr

it could defer the immediate application of the rule and continue reading the input until

expr – expr – expr

is seen. It could then apply the rule to the rightmost three symbols reducing them to "expr" which results in

expr – expr

being left. Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read

expr – expr

the parser can do one of two legal things, a shift or a reduction. It has no way of deciding between them. This is called a "shift/reduce conflict". It may also happen that the parser has a choice of two legal reductions. This is called a "reduce/reduce conflict". Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a "disambiguating rule".

The **yacc** program invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider

```
stat  :  IF '(' cond ')' stat
      |  IF '(' cond ')' stat ELSE stat
      ;
```

which is a fragment from a programming language involving an "if-then-else" statement. In these rules, "IF" and "ELSE" are tokens, "cond" is a

nonterminal symbol describing conditional (logical) expressions, and "stat" is a nonterminal symbol describing statements. The first rule will be called the "simple-if" rule and the second the "if-else" rule.

These two rules form an ambiguous construction since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways

```
IF (C1)
{
    IF (C2)
        S1
}
ELSE
    S2
```

or

```
IF (C1)
{
    IF (C2)
        S1
    ELSE
        S2
}
```

where the second interpretation is the one given in most programming languages having this construct. Each "ELSE" is associated with the last preceding "un-ELSE'd" IF. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the "ELSE". It can immediately reduce by the simple-if rule to get

IF (C1) stat

and then read the remaining input

ELSE S2

and reduce

IF (C1) stat ELSE S2

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the "ELSE" may be shifted, "S2" read, and then the right-hand portion of

IF (C1) IF (C2) S1 ELSE S2

can be reduced by the if-else rule to get

IF (C1) stat

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input which is usually desired.

Once again, the parser can do two valid things—there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, "ELSE", and particular inputs, such as

IF (C1) IF (C2) S1

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**-v**) option output file. For example, the output corresponding to the above conflict state might be

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

  stat : IF ( cond ) stat_          (18)
  stat : IF ( cond ) stat_ELSE stat

  ELSE    shift 45
  .       reduce 18
```

where the first line describes the conflict—giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is "ELSE", it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the "ELSE" will have been shifted in this state. In state 23, the alternative action [describing a dot (.)] is to be done if the input symbol is not mentioned explicitly in the actions. In this case, if the input symbol is not "ELSE", the parser reduces to

```
stat : IF '(' cond ')' stat
```

by grammar rule 18.

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule

numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there is reduce action possible in the state and this is the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

PRECEDENCE

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | NAME
     ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

in order to perform the correct precedence of operators. When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary "−". Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, **%prec**, changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules

```
%left '+' '-'
%left '*' '/'

%%

expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr     %prec '*'
     | NAME
     ;
```

might be used to give unary minus the same precedence as multiplication.

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in an essentially "cookbook" fashion until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

ERROR HANDLING

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides a simple, but reasonably general feature. The token name "error" is reserved for error handling. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that

caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat  :  error
```

means that on a syntax error the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as

```
stat  :  error  ';'
```

are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example

```
input   :  error  '\n'
              {
                printf( "Reenter last line: " );
              }
              input
        {
          $$ = $4;
        }
        ;
```

is one way to do this. There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable. For this reason, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example can be rewritten as

```
input   :  error  '\n'
              {
                yyerrok;
                printf( "Reenter last line: " );
              }
              input
        {
          $$ = $4;
        }
        ;
```

which is somewhat better.

As previously mentioned, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous

look-ahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by *yylex* is presumably the first token in a legal statement. The old illegal token must be discarded and the error state reset. A rule similar to

```
stat  :  error
      {
        resynch();
        yyerrok ;
        yyclearin;
      }
      ;
```

could perform this.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

THE "yacc" ENVIRONMENT

When the user inputs a specification to **yacc**, the output is a file of C language programs, called *y.tab.c* on most systems. (Due to local file system conventions, the names may differ from installation to installation.) The function produced by **yacc** is called *yyparse()*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex()*, the lexical analyzer supplied by the user (see "LEXICAL ANALYSIS"), to obtain input tokens. Eventually, an error is detected, *yyparse()* returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, *yyparse()* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a program called *main()* must be defined that eventually calls *yyparse()*. In addition, a routine called *yyerror()* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of *main()* and *yyerror()*. The name of this library is system dependent; on many systems, the library is accessed by a **−ly** argument to the loader. The source codes

```
main()
{
    return ( yyparse() );
}
```

and

```
# include <stdio.h>

yyerror(s)
        char *s;
{
        fprintf( stderr, "%s\n", s );
}
```

show the triviality of these default programs. The argument to *yyerror()* is a string containing an error message, usually the string "syntax error". The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the *main()* program is probably supplied by the user (to read arguments, etc.), the **yacc** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions

are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

HINTS FOR PREPARING SPECIFICATIONS

This part contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong".

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.

5. Indent rule bodies by two tab stops and action bodies by three tab stops.

The example in **A Simple Example** is written following this style, as are the examples in this section (where space permits). The user must make up his own mind about these stylistic questions. The central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the **yacc** parser encourages so called "left recursive" grammar rules. Rules of the form

```
name  :  name rest_of_rule ;
```

match this algorithm. These rules such as

```
list  :  item
      |  list ',' item
      ;
```

and

```
seq   :  item
      |  seq item
      ;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq   :  item
      |  item seq
      ;
```

the parser is a bit bigger; and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as

```
seq   :  /* empty */
      |  seq  item
      ;
```

using an empty rule. Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer and set by actions. For example,

```
%{
  int dflag;
%}
  ... other declarations ...

%%

prog  :  decls  stats
      ;

decls :  /* empty */
      {
           dflag = 1;
      }
      |  decls  declaration
      ;

stats :  /* empty */
      {
           dflag = 0;
      }
      |  stats  statement
      ;

   ... other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. The flag "dflag" is now 0 when reading statements and 1 when reading declarations, *except for the first token in the first statement*. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "back-door" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult if not impossible to do otherwise.

Reserved Words

Some programming languages permit you to use words like "if", which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it "this instance of *if* is a keyword and that instance is a variable". The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*, i.e., forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

ADVANCED TOPICS

This part discusses a number of advanced features of **yacc**.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros **YYACCEPT** and **YYERROR**. The **YYACCEPT** macro causes *yyparse()* to return the value 0; **YYERROR** causes the parser to behave as if the current input symbol had been a syntax error; *yyerror()* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context sensitive syntax checking.

Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit.

```
sent  :  adj  noun  verb  adj  noun
      {
         look at the sentence ...
      }
      ;
adj   :  THE
      {
            $$ = THE;
      }
      |  YOUNG
      {
            $$ = YOUNG;
      }
      ...
      ;
noun  :  DOG
      {
         $$ = DOG;
      }
      |  CRONE
      {
         if( $0 == YOUNG )
         {
            printf( "what?\n" );
         }
         $$ = CRONE;
      }
      ;
      ...
```

In this case, the digit may be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol "noun" in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. The **yacc** program can also support values of other types including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked. The **yacc** value stack is declared to be a *union* of the various types of values desired. The user declares the union and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a **$$** or **$n** construction, **yacc** will automatically insert the appropriate union name so that no unwanted conversions take place. In addition, type checking commands such as **lint** is far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes

```
%union
{
  body of union ...
}
```

in the declaration section. This declares the **yacc** value stack and the external variables *yylval* and *yyval* to have type equal to this union. If **yacc** was invoked with the **−d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might have said

```
typedef union
{
  body of union ...
}
YYSTYPE;
```

instead. The header file must be included in the declarations section by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, **%type**, is used to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

to associate the union member *nodetype* with the nonterminal symbols "expr" and "stat".

There remains a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first $. The example

```
rule : aaa
        {
          $<intval>$ = 3;
        }
        bbb
    {
      fun( $<intval>2, $<other>0 );
    }
    ;
```

shows this usage. This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in **An Advanced Example**. The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold *int*'s, as was true historically.

EXAMPLES

A Simple Example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers labeled "a" through "z" and accepts arithmetic expressions made up of the operators +, -, *,/,% (mod operator), & (bitwise and), | (bit wise or), and assignments. If an expression at the top level is an assignment, the value is printed; otherwise, the expression is printed. As in C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

```
%{
# includes<stdio.h>
# includes<ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
```

```
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /* supplies precedence for unary minus */

% %       /* beginning of rule section */

list     : /* empty */
         | list stat '\n'
         | list error '\n'

         {

            yyerrork;

         }
         ;

stat     : expr
         {
           printf( "%dn", $1 );
         }
         | LETTER '=' expr
         {
           regs[$1] = $3
         }
         ;

expr     : '(' expr ')'
           {
               $$ = $2;
           }
           | expr '+' expr
           {
               $$ = $1 + $3
           }
           | expr '-' expr
           {
               $$ = $1 - $3
           {
           | expr '*' expr
           {
               $$ = $1 * $3;
           }
```

```
        | expr '/' expr
        {
            $$ = $1/$3;
        }
        | exp '%' expr
        {
            $$ = $1 % $3
        }
        | expr '&' expr

        {
            $$ = $1 & $3;
        }
        | expr '|' expr
        {
            $$ = $1 | $3
        }
        | '-' expr  %prec UMINUS
        {
            $$ = - $2;
        }
        | LETTER
        {
            $$ = reg[$1];
        }
        | number
        ;

number      : DIGIT
        {
            $$ = $1; base = ($1==0) ? 8 ; 10;
        }
        | number DIGIT
        {
            $$ = bas * $1 + $2
        }
        ;

%% /* start of program */

yylex( )        /* lexical analysis routine */
{           /* return LETTER for lowercase letter,
           yylval = 0 through 25*/
```

```
        /* returns DIGIT for digit, yylval = 0 through 9*/
        /* all other characters are returned immediately */

        int c;
                            /*skip blanks*/
        while (c=getchar( ) )    = = ‘’)
              ;

        /* c is now nonblank */

        if( islower( c ))
        {
                yylval = c- ‘a’;
                return( LETTER );
        }
        if( isdigit( c ))
        }
                yylval = c-‘0’;
                return( DIGIT );

        }
        return( c );
}
```

YACC Input Syntax

This appendix has a description of the **yacc** input systax as a **yacc** specification. Contex dependencies, etc. are not considered. Ironically, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, and comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS but never as part of C_IDENTIFIERs.

```
/* grammar for the input to yacc */

/* basic entries */
```

```
%token    IDENTIFIER   /* includes identifiers and literals */
%token     C_IDENTIFIER  /* identifier (but not literal)
                  followed by a colon */
%token     NUMBER        /* [0-9]+ */

   /*    reserved words: %type=> TYPE %left=>LEFT,etc. */


%token     LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token     MARK    /* the %% mark */
%token     LCURL   /* the % { mark */
%token     RCURL   /* the % } mark */

   /*  ASCII character literals stand for themselves */

%token     spec

%%

spec   :    defs MARK rules tail
       ;

tail   :    MARK
       {
              In this action, eat up the rest of the file
       }

       |      /* empty: the second MARK is optional */
       ;

defs   :       /* empty */
       |       defs def
       ;

defs   :       START IDENTIFIER
       |       UNION
       {
                    Copy union definition to output
       }
       |  LCURL
       {
                    Copy C code to output file
                    RCURL
```

```
    }
    | ndefs rword tag nlist
    ;

rword : TOKEN
    | LEFT
    | RIGHT
    | NONASSOC
    | TYPE
    ;

tag   : /* empty: union tag is optional */
    | '<' IDENTIFIER '>'
    ;

nlist : nmno
    | nlist nmno
    | nlist',' nmno
    ;

nmno  : IDENTIFIER        /*Note: literal illegal with % type */
    | IDENTIFIER NUMBER   /* Note: illegal with % type */
    ;

  /* rule section */

rules : C_IDENTIFIER rbody proc
    | rules rule
    ;

rule : C_IDENTIFIER rbody prec
    | '|' rbody prec
    ;


rbody : /* empty */
    | rbody IDENTIFIER
    | rbody act
    ;

act   : '{'
            {
```

```
                Copy action translate $$' etc.
                    }
                '}'
    ;

Bprec : /* empty */
    | PREC IDENTIFIER
    | PREC IDENTIFIER act
    | prec';'
    ;
```

An Advanced Example

This appendix gives an example of a grammar using some of the advanced features. The desk calculator example in **A Simple Example** is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants; the arithmetic operations +, - *, /, unary - "a" through "z". Moreover, it also understands intervals written

(X,Y)

where X is less than or equal to Y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in **A Simple Example**; assignments returns no value and prints nothing while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C language. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, INTERVAL, by using *typedef*. The **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions—division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**—18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines.

```
2.5+(3.5-4.)
```

and

```
2.5 + ( 3.5,4 )
```

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time,, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator—one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive but not very general. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C language library routine *atof()* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar provoking a syntax error in the parser and thence

error recovery.

```
%{

#include<stdio.h>
#include<ctype.h>

typedef struct interval
{
    double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv( );

double atof( );

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start line

%union
{
  int ival;
  double dval;
  INTERVAL vval;
}

%token <ival> DREG VREG  /*indices into dreg, vreg arrays */


%token <dval> CONST      /* floating point constant */

%type <dval> dexp        /* expression */

%type <vval> vexp        /* interval expression */

  /* precedence information about the operators */

%left  '+' '-'
%left  '*' '/'
%left  UMINUS   /* precedence for unary minus */
```

```
%%

lines   : /* empty */
        | lines line
        ;
line    : dexp '\n'
        {
                printf( "%15.8f\n".$1 );
        }
        | vexp '\n'

        {
                printf( "(%15.8f , %15.8f )0,$1.1o,$1.hi );

        }
        | DREG '=' '\n'
        {

                dreg[$1] = $3;

        }
        | VREG '=' vexp '\n'
        {

                vreg[$1] = $3;

        }
        | error '\n'
        {

                yyerrork;

        }
        ;

dexp    : CONST
        | DREG
        {

                $$ = dreg[$1]

        }
        | dexp '+' dexp
```

```
        {
                $$ = $1 + $3
        }
        | dexp '-' dexp
        {
                $$ = $1 - $3
        }
        | dexp '*' dexp
        {
                $$ = $1 * $3
        }
        | dexp '/' dexp
        {
                $$ = $1 / $3
        }
        | '-' dexp    %prec UMINUS
        {
                $$ =- $2
        }
        | '(' dexp')'
        {
                $$ = $2
        }
        ;
vexpp   : dexp
        {
                $$.hi = $$.lo = $1;
        }
        | '(' dexp',' dexp')'
        {
```

```
            $$.lo = $2;
            $$.hi = $4;
            If( $$.lo > $$.hi )
            {
                printf( "interval out of order n" );
                YYERROR;
            }
    }
    | VREG
    {

            $$ = vreg[$1]

    }
    | vexp '+' vexp
    {

            $$.hi = $1.hi + $3.hi;
            $$.lo = $1.lo + $3.lo


    }
    | dexp '+' vexp
    {

            $$.hi = $1 + $3.hi;
            $$.lo = $1 + $3.lo
    }
    | vexp '=' vexp
    {

            $$.hi = $1.hi - $3.lo;
            $$.lo = $1.lo - $3.hi

    }
    | dvep '-' vdep

    {

            $$.hi = $1 - $3.lo;
            $$.lo = $1 - $3.hi

    }
```

```
    | vexp '*' vexp
    {

            $$ = vmul( $1.lo,$.hi,$3 )

    }
    | dexp '*' vexp
    {

            $$ = vmul( $1, $1, $3 )

    }
    | vexp '/' vexp
    {
            if( dcheck( $3 ) ) YYERROR;
            $$ = vdiv( $1.lo, $1.hi, $3 )

    }
    | dexp '/' vexp


            if( dcheck( $3 ) ) YYERROR;
            $$ = vdiv( $1.lo, $1.hi, $3 )
    }
    | '-' vexp    %prec UMINUS
    {
           $$.hi = -$2.lo;$$.lo =-$2.hi
    }
    | '(' vexp ')'
    }
             $$ = $2
    }
    ;

%%

# define BSZ 50   /* buffer size for floating point number */

       /* lexical analysis */

yylex( )
{
       register c;
```

```
                    /* skip over blanks */

        ;

if( isupper( c ) )
{
    yylval.ival = c - 'A'
    return( VREG );
}
if( islower( c ) )
{

    yylval.ival = c - 'a',
    return( DREG );
}

    /* gobble up digits. points, exponents */
if( idigit( c ) | c=='.' )
{

    char buf[BSZ+1], *cp = buf;
    int dot = 0, exp = 0;

    for( ; (cp-buf)<BSZ ; ++cp,c=getchar( ) )
{

        *cp = c;
        if( isdigit( c ) )
          continue;
        if( c =='.'
        {
        if( dot++ | exp )
              return( '.' );/* will cause syntax error */
          continue;
        }
        if( c == 'e')
        {

            if( exp++ )
                  return( 'e' ); */will cause syntax error */
            continue;
        }
            /* end of number */
```

```
                break;
        }
        *cp = '\0';
        if(cp-buff) >= BSZ )
          printcf( "constant too long truncated\n" );
        else
           ungetc( c, stdin ); /* push back last char read */
        yylval.dval = atof( buf );
        return( CONST );
        }
        return( c );

}

INTERVAL
hilo( a, b, c, d )
        double a, b, c, d;
{
        /* returns the smallest interval containing a, b, c, and d */

        /* used by *,/ routine */
        INTERVAL v;

        if( a>b )
        {
             v.hi = a;
             v.lo = b;
        }
        else
        {
             v.hi = b;
             v.lo = a;
        }
        if( c>d )
        {
             if( c>v.hi )
                v.hi = c;
             if( d<v.lo )
                v.lo = d;
        }
        else
        }
             if( d>v.hi )
                v.hi = d;
```

```
            if( c<v.lo )
                v.lo = c;
        }
        return( v );
}
INTERVAL vmul( a, b, v )
        double a, b;
        INTERVAL v;
{
         return( hilo( a*v.hi, a*v,lo, b*v.hi, b*v.lo ) );
}
dcheck( v )
        INTERVAL v;
{
        if( v.hi >=0.&& v.lo <=0. )
        {
             printf( "divisor internal contains 0.\n" );
             return( 1);
        }
        return( 0 );
{

INTERVAL vdiv( a, b, v )
        double a, b;
        INTERVAL v;
{
  return( hilo( a/v.hi, a/v,lo, b/v.hi, b/v.lo ) );

}
```

Old Features Supported But Not Encouraged

This appendix mentions synonyms and features that are supported for historical continuity but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes.

2. Literals may be more that one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literal.

The use of multicharacter literals is likely to mislead those unfamiliar with **yacc** since it suggests that **yacc** is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as % left, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Action may also have the form

   ={ ... }

   and the curly braces can be dropped if the action is a single C language statement.

6. The C language code between %{ and %} use to be permitted at the head of the rules section as well as in the declaration section.

NOTES

UNIX SYSTEM TO UNIX SYSTEM COPY (UUCP)

INTRODUCTION

The **uucp** network has provided a means of information exchange between UNIX systems over the direct distant dialing network for several years. This chapter provides you with the background to make use of the network.

The first half of the document discusses concepts. Understanding these basic principles helps the user make the best possible use of the **uucp** network. The second half explains the use of the user level interface to the network and provides numerous examples.

There are several major uses of the network. Some of the uses are:

- Distribution of software
- Distribution of documentation
- Personal communication (mail)
- Data transfer between closely sited machines
- Transmission of debugging dumps and data exposing bugs
- Production of hard copy output on remote printers.

THE UUCP NETWORK

The **uucp**(1) network is a network of UNIX systems that allows file transfer and remote execution to occur on a network of UNIX systems. The extent of the network is a function of both the interconnection hardware and the controlling network software. Membership in the network is tightly controlled via the software to preserve the integrity of all members of the network. You cannot use the **uucp** facility to send files to systems that are not part of the **uucp** network. The following parts describe the topology, services, operating rules, etc., of the network to provide a framework for discussing use of the network.

Network Hardware

The **uucp** was originally designed as a dialup network so that systems in the network could use the DDD network to communicate with each other. The three most common methods of connecting systems are:

1. Connecting two UNIX systems directly by cross-coupling (via a null modem) two of the computers ports. This means of connection is useful for only short distances (several hundred feet can be achieved although the RS232 standard specifies a much shorter distance) and is usually run at high speed (9600 baud). These connections run on asynchronous terminal ports.

2. Using a modem (a private line or a limited distance modem) to directly connect processors over a private line (using 103- or 212-type data sets).

3. Connecting a processor to another system through a modem, an automatic calling unit (ACU), and the DDD network. This is by far the most common interconnection method, and it makes available the largest number of connections.

The **uucp** could be extended to use higher speed media (e.g., HYPERchannel*, Ethernet†, etc.), and this possibility is being explored for future UNIX system releases. Some sites already support local modifications to **uucp** to allow the use of Datakit, X.25 (permanent virtual circuits), and calling through data switches.

Network Topology

A large number of connections between systems are possible via the DDD network. The topology of the network is determined by both the hardware connections and the software that control the network.

---

* Trademark of Network Systems Corporation.

† Trademark of Xerox Corporation.

Software Topology

The hardware capability of systems in the network defines the maximum number of connections in the network. The software at each node restricts the access by other systems and thereby defines the extent of the network. As part of the security mechanism used by **uucp**, the extent of access that other systems have can be controlled at each node.

The **uucp** uses the UNIX system password mechanism coupled with a system file (*/usr/lib/uucp/L.sys*) and a file system permission file (*/usr/lib/uucp/USERFILE*) to control access between systems. The password file entries for **uucp** (usually, **luucp**, **nuucp**, **uucp**, etc.) allow only those remote systems that know the passwords for these IDs to access the local system. (Great care should be taken in revealing the password for these **uucp** logins since knowing the password allows a system to join the network.) The system file (*/usr/lib/uucp/L.sys*) defines the remote systems that a local host knows about. This file contains all information needed for a local host to contact a remote system (including system name, password, login sequence, etc.) and as such is protected from viewing by ordinary users.

In summary, while the available hardware on a network of systems determines the connectivity of the systems, the combination of password file entries and the **uucp** system files determine the extent of the network.

Forwarding

One of the recent additions to **uucp** (for UNIX system 5.0) is a limited forwarding capability whereby systems that are part of the network can forward files through intermediate nodes. For security reasons, when forwarding, files may only be transmitted to the *public* area or fetched from the remote systems *public* area.

Security

The most critical feature of any network is the security that it provides. Users are familiar with the security that UNIX system provides in protecting files from access by other users and in accessing the system via passwords. In building a network of processors, the notion of security is widened because access by a wider community of users is granted. Access is granted on a system basis (that is, access is granted to all users on a remote system). This follows from the fact that the process of sending (receiving) a file to (from) another system is done via daemons that use one special user ID(s). This user ID(s) is granted (denied) access to the system via the **uucp** system file

(*/usr/lib/uucp/L.sys*) and the areas that the system has access to is controlled by another file (*/usr/lib/uucp/USERFILE*). For example, access can be granted to the entire file system tree or limited to specific areas.

Software Structure

The **uucp** network is a batch network. That is, when a request is made, it is spooled for later transmission by a daemon. This is important to users because the success or failure of a command is only known at some later time via **mail**(1) notification. For most transfers, there is little trouble in transmitting files between systems, however, transmissions are occasionally delayed or fail because a remote system cannot be reached.

Rules of the Road

There are several rules by which the network runs. These rules are necessary to provide the smooth flow of data between systems and to prevent duplicate transmissions and lost jobs. The following outline these rules and their influence on the network.

Queuing

Jobs submitted to the network are assigned a sequence number for transmission. Jobs are represented by a file (or files) in a common spool directory (*/usr/spool/uucp*). When a file transfer daemon (**uucico**) is started to transmit a job, it selects a system to contact and then transmits all jobs to that system. Before breaking off the conversation, any jobs to be received from that remote system are accepted. The system selected as the one to contact is randomly selected if there is work for more than one system. In releases of **uucp** prior to UNIX system 5.0, the first system appearing in the spool directory is selected so preference is given to the most recently spawned jobs. **Uucp** may be sending to or receiving from many systems simultaneously. The number of incoming requests is only limited by the number of connections on the system, and the number of outgoing transfers is limited by the number of ACUs (or direct connections).

Dialing and the DDD Network

In order to transfer data between processors that are not directly connected, an auto dialer is used to contact the remote system. There are several factors that can make contacting a remote system difficult.

1. All lines to the remote system may be busy. There is a mechanism within **uucp** that restricts contact with a remote system to certain times of the day (week) to minimize this problem.

2. The remote system may be down.

3. There may be difficulty in dialing the number (especially if a large sequence of numbers involving access through PBXs is involved). The dialing algorithm tries dialing a number twice and the algorithm used to dial remote systems is not perfect, particularly when intermediate dial tones are involved.

Scheduling and Polling

When a job is submitted to the network, an attempt to contact that system is made immediately. Only one conversation at a time can exist between the same two systems.

Systems that are polled can do nothing to force immediate transmission of data. Jobs will only be transmitted when the system is polled (hourly, daily, etc.) by a remote system.

Retransmissions and Hysteresis

The **uucp** network is fairly persistent in its attempt to contact remote systems to complete a transmission. To prevent **uucp** from continually calling systems that are unavailable, hysteresis is built into the algorithm used to contact other systems. This mechanism forces a minimum fixed delay (specifiable on a per system basis) to occur before another transmission can take place to that system.

Purging and Cleanup

Transfers that cannot be completed after a defined period of time (72 hours is the value that is set when the system is distributed) are deleted and the user is notified.

Special Places: The Public Area

In order to allow the transfer of files to a system for which a user does not have a login on, the *public* directory (usually kept in */usr/spool/uucppublic*) is available with general access privileges. When receiving files in the *public* area, the user should dispose of them quickly as the administrative portion of **uucp** purges this area on a regular basis.

Permissions

File Level Protection

In transferring files between systems, users should make sure that the destination area is writable by **uucp**. The **uucp** daemons preserve execute permission between systems and assign permission 0666 to transferred files.

System Level Protection

The system administrator at each site determines the global access permissions for that processor. Thus, access between systems may be confined by the administrator to only some sections of the file system.

Forwarding Permissions

The forwarding feature is a new addition to the **uucp** package. You should be aware that

1. When forwarding is attempted through a node that is running an old version of **uucp**, the transmission fails.

2. Nodes that allow forwarding can restrict the forwarding feature in several ways.

a. Forwarding is allowed for only certain users.

b. Forwarding to certain destination nodes (e.g., Australia) should be avoided.

c. Forwarding for selected source nodes is allowed.

3. The most important restriction is that forwarding is allowed only for files sent to or fetched from the *public* area.

NETWORK USAGE

The following parts discuss the user interface to the network and give examples of command usage.

Name Space

In order to reference files on remote systems, a syntax is necessary to uniquely identify a file. The notation must also have several defaults to allow the reference to be compact. Some restrictions must also be placed on pathnames to prevent security violations. For example, pathnames may not include ".." as a component because it is difficult to determine whether the reference is to a restricted area.

Naming Conventions

Uucp uses a special syntax to build references to files on remote systems. The basic syntax is

system-name!pathname

where the system-name is a system that **uucp** is aware of. The *pathname* part of the name may contain any of the following:

1. A fully qualified *pathname* such as

mhtsa!/usr/you/file

The *pathname* may also be a directory name as in

mhtsa!/usr/you/directory

2. The login directory on a remote may be specified by use of the ~ character. The combination ~user references the login directory of a user on the remote system. For example,

   mhtsa!~adm/file

   would expand to

   mhtsa!/usr/sys/adm/file

   if the login directory for user adm on the remote system is */usr/sys/adm*.

3. The *public* area is referenced by a similar use of the prefix ~/user preceding the pathname. For example,

   mhtsa!~/you/file

   would expand to

   mhtsa!/usr/spool/uucp/you/file

   if */usr/spool/uucp* is used as the spool directory.

4. Pathnames not using any of the combinations or prefixes discussed above are prefixed with the current directory (or the login directory on the remote). For example,

   mhtsa!file

   would expand to

   mhtsa!/usr/you/file

The naming convention can be used in reference to either the source or destination file names.

Forwarding Syntax

The newest feature of **uucp** is the ability to allow files to be passed between systems via intermediate nodes. This is done via a variation of the bang (!) syntax that describes the path to be taken to reach that file. For example, a user on system a wishing to transmit a file to system e might specify the transfer as

```
uucp file b!c!d!e!~/you/file
```

if the user desires the request to be sent through b, c, and d before reaching e. Note that the pathname is the path that the file would take to reach node e. Note also that the destination must be specified as the *public* area. Fetching a file from another system via intermediate nodes is done similarly. For example,

```
uucp b!c!d!e!~/you/file x
```

fetches file from system e and renames it x on the local system. The forwarding prefix is the path from the local system and not the path from the remote to the local system. The forwarding feature may also be used in conjunction with remote execution. For example,

```
uux mhtsa!uucp mhtsb!mhrtc!/usr/spool/uucppublic/file x
```

sends a request to mhtsa to execute the **uucp** command to copy a file from mhrtc to x on mhtsa.

Types of Transfers

Uucp has a very flexible command syntax for file transmission. The following give examples of different combinations of transfers.

Transmissions of Files to a Remote

Any number of files can be transferred to a remote system via **uucp**. The syntax supports the *, ? and [..] metacharacters. For example,

```
uucp *.[ch] mhtsa!dir
```

transfers all files whose name ends in c or h to the directory *dir* in the users login directory on mhtsa.

Fetching Files From a Remote

Files can be fetched from a remote system in a similar manner. For example,

```
uucp mhtsa!*.[ch] dir
```

will fetch all files ending in c or h from the users login directory on mhtsa and place the copies in the subdirectory *dir* on the local system.

Switching

Transmission of files can be arranged in such a way that the local system effectively acts as a switch. For example,

```
uucp mhtsb!files mhtsa!filed
```

will fetch files from the users login directory on mhtsb, rename it as *filed*, and place it in the login directory on mhtsa.

Broadcasting

Broadcast capability (that is, copying a file to many systems) is not supported by **uucp**, however, it can be simulated via a shell script as in

```
for i in mhtsa mhtsb mhtsd
do
        uucp file $i!broad
done
```

Unfortunately, one **uucp** command is spawned for each transmission so that it is not possible to track the transfer as a single unit.

Remote Executions

The remote execution facility allows commands to be executed remotely. For example,

```
uux "!diff mhtsa!/etc/passwd mhtsd!/etc/passwd > !pass.diff"
```

will execute the command **diff**(1) on the password file on mhtsa and mhtsd and place the result in pass.diff.

Spooling

To continue modifying a file while a copy is being transmitted across the network, the **−c** option should be used. This forces a copy of the file to be queued. The default for **uucp** is not to queue copies of the files since it is wasteful of both Central Processing Unit time and storage. For example, the following command forces the file work to be copied into the spool directory before it is transmitted.

```
uucp -c work mhtsa!~/you/work
```

Notification

The success or failure of a transmission is reported to users asynchronously via the **mail**(1) command. A new feature of **uucp** is to provide notification to the user in a file (of the users choice). The choices for notification are:

1. Notification returned to the requesters system (via the **−m** option). This is useful when the requesting user is distributing files to other machines. Instead of logging onto the remote machine to read mail, mail is sent to the requester when the copy is finished.

2. A variation of the **−m** option is to force notification in a file (using the **−m***file* option where *file* is a file name). For example,

```
uucp −mans /etc/passwd mhtsb!/dev/null
```

sends the file */etc/passwd* to system mhtsb and place the file in the bit bucket (*/dev/null*). The status of the transfer is reported in the file *ans* as,

```
uucp job 0306 (8/20-23:08:09) (0:31:23) /etc/passwd copy succeeded
```

3. **Uux**(1) always reports the exit status of the remote execution unless notification is suppressed (via the **−n** option). Notification can be sent to a different user on the remote system via the **−nuser** option.

Tracking and Status

The most pervasive change to the **uucp** package is revising the internal formatting of jobs so that each invocation of **uucp** or **uux**(1) corresponds to a single job. It is now possible to associate a single job number with each command execution so that the job can be terminated or its status obtained.

The Job ID

The default for the **uucp** and **uux** command is not to print the job number for each job. This was done for compatibility with previous versions of **uucp** and to prevent the many shell scripts built around **uucp** from printing job numbers. If the following environment variable

```
JOBNO=ON
```

is made part of the users environment and exported, **uucp** and **uux** prints the job number. Similarly, if the user wishes to turn the job numbers off, the environment variable is set as follows:

```
JOBNO=OFF
```

If you wish to force printing of job numbers without using the environment mechanism, use the **−j** option. For example,

```
uucp −j /etc/passwd mhtsb!/dev/null
uucp job 282
```

forces the job number (282) to be printed. If the **–j** option is not used, the IDs of the jobs (belonging to the user) are found by using the **uustat**(1) command. This provides the job number. For example,

```
uustat
0282 tom mhtsb 08/20-21:47 08/20-21:47 JOB IS QUEUED
0272 tom mhtsb 08/20-21:46 08/20-21:46 JOB IS QUEUED
```

shows that the user has two jobs (282 and 272) queued.

Job Status

The **uustat** command allows a user to check on one or all jobs that have been queued. The ID printed when a job is queued is used as a key to query status of the particular job. An example of a request for the status of a given job is

```
uustat –j0711

0711 tom mhtsb 07/30-02:18 07/30-02:18 JOB IS QUEUED
```

There are several status messages that may be printed for a given job; the most frequent ones are JOB IS QUEUED and JOB COMPLETED (meanings are obvious). The manual page for **uustat** lists the other status messages.

Network Status

The status of the last transfer to each system on the network is found by using the **uustat** command. For example,

```
uustat –mall
```

reports the status of the last transfer to all of the systems known to the local system. The output might appear as

mhb5c	08/10-12:35	CONVERSATION SUCCEEDED
resear	08/20-17:01	CONVERSATION SUCCEEDED
minimo	07/22-16:31	DIAL FAILED
austra	08/20-18:36	WRONG TIME TO CALL
ucbvax	08/20-20:37	LOGIN FAILED

where the status indicates the time and state of the last transfer to each system. When sending files to a system that has not been contacted recently, it is a good idea to use **uustat** to see when the last access occurred (because the remote system may be down or out of service).

Job Control

With the unique job ID generated for each **uucp** or **uux** command, it is possible to control jobs in the following ways.

Job Termination

A job that consists of transferring many files from several different systems can be terminated using the **−k** option of **uustat**. If any part of the job has left the system, then only the remaining parts of the job on the local system is terminated.

Requeuing a Job

The **uucp** package clears out its working area of jobs on a regular basis (usually every 72 hours) to prevent the buildup of jobs that cannot be delivered. The −r option is used to force the date of a job to be changed to the current date, thereby lengthening the time that **uucp** attempts to transmit the job. It should be noted that the **−r** option does not impart immortality to a job. Rather, it only postpones deleting the job during housekeeping functions until the next cleanup.

Network Names

Users may find the names of the systems on the network via the **uuname**(1) command. Only the names of the systems in the network are printed.

UTILITIES THAT USE UUCP

There are several utilities that rely on **uucp** or **uux**(1) to transfer files to other systems. The following parts outline the more important of these functions. This increases awareness of the extent of the use of the network.

The Stockroom

The UNIX system stockroom is a facility whereby a library of source can be maintained at a central location for distribution of source or bug fixes. Access to and distribution of library entries is controlled by shell scripts that use **uucp**.

Mail

The **mail**(1) command uses **uux** to forward mail to other systems. For example, when a user types

```
mail mhtsa!tom
```

the **mail** command invokes **uux** to execute **rmail** on the remote system (**rmail** is a link to the **mail** command). Forwarding mail through several systems (e.g., mail a!b!tom) does not use the **uucp** forwarding feature but is simulated by the **mail** command itself.

Netnews

The **netnews(1)** command that is locally supported on many systems uses **uux** in much the same way that **mail** does to broadcast network mail to systems subscribing to news categories.

Uuto

The **uuto**(1) command uses the **uucp** facility to send files while allowing the local system to control the file access. Suppose your login is emsgene and you are on system aaaaa. You have a friend (David) on system bbbbb with a login name of w1dmc. Also assume that both systems are networked to each other [See **uuname**(1)]. To send files using **uuto**, enter the following:

```
uuto filename aaaaa!w1dmc
```

where filename is the name of a file to be sent. The files are sent to a public directory defined in the uucp source. In this example, David will receive the following mail:

```
From nuucp Tue Jan 25 11:09:55 1983
/usr/spool/uucppublic/receive/w1dmc/aaaaa\
//filename from aaaaa!emsgene arrived
```

See **uuto**(1) for more details.

Other Applications

The Office Automation System (OAS) uses **uux** to transmit electronic mail between systems in a manner similar to the standard **mail** command. Some sites have replaced utilities such as **lpr**(1), **opr**(1), etc., with shell scripts that invoke **uux** or **uucp**. Other sites use the **uucp** ne. work as a backup for higher speed networks (e.g., PCL, NSC HYPERchannel, etc.).

Other Volumes of the UNIX* Programmer's Manual

Volume 1
Commands and Utilities, contains the manual pages for the commands and applications programs that can be invoked directly by the user or by command language procedures. Manual pages describe the purpose and use of the UNIX system commands, warn of potential problems, give examples, and tell where to find related information.

Volume 2
System Calls and Library Routines, describes the programming features of the UNIX system. Included are the descriptions of system calls, subroutines, libraries, file formats, macro packages, and character set tables.

Volume 3
System Administration Facilities, contains the commands used by UNIX system administrators. It describes system maintenance commands and application programs, special files, and system maintenance procedures.

Volume 4
Documentation Preparation, describes and explains the commands and macros needed to input and format a document. It provides examples of advanced UNIX system editing commands and the stream editor (**sed**), a non-interactive content editor. Also described are the text processors used to format text, **nroff** and **troff**, and the preprocessors, **tbl** and **eqn** used to prepare tables and typeset mathematics.

Select Code 320-035
ISBN 0-03-011204-4