CGTM NO. 176
DECEMBER 1976

THE SKOL PROGRAMMING LANGUAGE

REFERENCE MANUAL

CHARLES T. ZAHN, JR.

COMPUTATION RESEARCH GROUP
STANFORD LINEAR ACCELERATOR CENTER
STANFORD, CALIFORNIA 94305

TABLE OF CONTENTS

THE SKOL PROGRAMMING LANGUAGE REFERENCE MANUAL

SKOL: Summary and Genealogy

The design of the SKOL language was subject to two fairly important constraints. Firstly, all SKOL programs are translatable into standard FORTRAN (with one slight exception). Secondly, the translation from SKOL to FORTRAN is accomplished using the MORTRAN macro-translator [9, 10, 11, 12] and a set of text-substitution rules (macros) specifically designed to translate SKOL programs into FORTRAN.

As a natural consequence of the first constraint, SKOL has a FORTRAN "underbelly" consisting of the syntax and semantics of identifiers (called symbolic names in FORTRAN), logical and arithmetic expressions, specifications of the types of variables, and the bounds of arrays, assignment statements, input-output statements, formats for conversion between binary and character representation of data, subprograms and parameter communication. Some of this underbelly is described in the following sections, but the user is urged to have FORTRAN documentation available to resolve questions at this level of language. Errors made at this level will most likely be reported by the FORTRAN compiler rather than the SKOL pre-compiler, so the user will have to understand these diagnostic messages. As a result of the second constraint, some of the syntactic aspects of SKOL are somewhat awkward and "strong" type-checking (as in PASCAL) cannot be performed with complete consistency.

The major advantages which accrue as reward for accepting these two constraints have been discussed by Cook and Shustek [9, 10], but we shall briefly mention them here:

1) Standard FORTRAN compilers exist for many computers and, therefore, a language translatable to standard FORTRAN, by a translator imple-

mented in standard FORTRAN*, inherits a wide portability.

2) Many larger computer installations have substantial libraries of programs including general utilities as well as application packages written in FORTRAN or in machine-language but callable from FORTRAN.

3) Considerable effort has been invested by some major computer vendors to produce optimizing compilers for FORTRAN.

4) Because the MORTRAN macro-translator is based on a general parameterized text-substitution mechanism, any language L translated by it to FORTRAN can be extended by the user in the same way that the macro-translator extends FORTRAN to L.

Given SKOL's FORTRAN underbelly and the extensibility inherited from the translation technique (i.e., MORTRAN), the remainder of the language is a hopefully coherent assembly of features borrowed from existing languages or suggested in the literature, plus several features or modifications which appear to be novel. The following list includes the most characterizing features of SKOL and their origin:

Features	Origin
Expressions, formats, subprograms	FORTRAN
Named constants, text-substitution	PASCAL, MORTRAN
Nested blocks of statements	ALGOL-60
Record structures and references	ALGOL-W, PASCAL, PL/1
User-defined scalar types	PASCAL
Character data and string variables	PL/1, ALGOL-W, PASCAL
Flexible text output facility	PL/1, PASCAL, SKOL
Keyword-parameter macro-procedures	Hardgrave [5]
IF...THEN...ELSEIF...ELSE...ENDIF	LISP, ALGOL-68
Scalar CASE statement	PASCAL

*MORTRAN is so implemented

Situation CASE statement	Zahn [6, 7, 8]
LOOP...WHILE...ENDLOOP	Dahl (see [6])
Infinite open-ended FOR statement	SKOL
Iteration statement for linked lists	SKOL
Hierarchical scalar types and subtypes	SKOL
User-defined character data type	SKOL
ELSE block in scalar CASE	Hoare [13]
Character substring contexts and replacement	SKOL
Coroutine processes	Conway [3], Dahl and Hoare [4]
Recursive routines	ALGOL-60

For the convenience of those readers familiar with the borrowed features, we include here a brief description and discussion of the features thought to be novel. The idea to make the character data type CHAR user-defined rather than built into SKOL was an example of the cliché "Necessity is the mother of invention". The way that FORTRAN treats input/output of characters to and from text files necessitates additional processing to generate an internal form of character represented by a small integer. Otherwise, character CASE statements would be impossible. Since each character must be so processed, it costs little extra to allow the user to define the allowable set of character constants as well as their ordering within the scalar type CHAR. The only things built in are the name CHAR and the form (i.e., quote-brackets) used to denote most constants of the scalar type CHAR. It is natural to decompose a character type into subtypes like ALPHABET, DIGIT, ARITHMETIC, RELATIONAL, LOGICAL, PUNCTUATION, BRACKETS, QUOTES, SPECIAL. A lexical scanner for a language translator might find it convenient to combine ALPHABET, DIGIT and the underbar character into a subtype identified as NAME_SYMBOL, and to further combine ARITHMETIC, RELATIONAL, LOGICAL, PUNCTUATION, BRACKETS and QUOTES into a subtype DELIMITER, etc. Because of the naturalness of this example as well as others, it was decided to generalize the scalar type idea to include nested subtypes and to in-

tegrate this idea into the scalar CASE statement (see Section on "Scalar types...").

Although we borrowed from PL/1 the "varying-length character string with fixed maximum size," the PL/1 notation for substrings and associated pseudo-variables has never caught our fancy. The verbose notation "SUBSTR(CH,K,1)" to indicate the K-th character of string CH is especially unappealing. After considerable searching and discussion, we settled on a compact yet simple notation for denoting intervals of a sequence which allows empty intervals to be interpreted as positions before or after elements of the sequence. When used for string intervals, we call this notation a string-context, and an arbitrary string insertion, deletion or replacement can be uniformly specified as the replacement of a string-context by a string expression. The following string-contexts and associated meaning reflect the generality and compactness of the notation. The '|' denotes substring length.

Notation	Meaning
CH(K)	CH(K)
CH(1...3)	CH(1), CH(2), CH(3)
CH(3...|2)	CH(3), CH(4)
CH(3|...K)	CH(K-2), CH(K-1), CH(K)
CH(2...|0)	before CH(2)
CH(0|...2)	after CH(2)
CH(0|...LENGTH(CH))	after last character of string CH

SKOL contains a text OUTPUT statement which is a combination of ideas from FORTRAN, PL/1 and PASCAL. In PL/1, there are three flavors of text output possible -- edit-directed, in which conversion formats must be supplied explicitly by the programmer; list-directed, in which the conversion format is implicit but dependent on the type of each variable; data-directed, in which the symbolic name of each variable is output before the value (under type-dependent

format). These three kinds of output cannot, however, be mixed in a single output statement, and the association between a variable and its explicitly supplied format is not textually apparent, the data and format lists being segregated rather than merged. The formatted output of FORTRAN shares this flaw. PASCAL has a WRITE statement in which each data item may optionally be followed by an explicit format, but the very useful data-directed output is not available and control formats are not as flexible as in FORTRAN or PL/1.

The OUTPUT statement of SKOL requires a sequence of data and control items which will be processed in order, the control items causing some specific modification of the current output position and the data items causing character output after formatting in any of the three ways discussed above. For example, OUTPUT($PAGE,:10X,I:I2,')',X(I),:/,:20X,P(I):,' #'); causes the following to happen on file $OUTPUT:

```
Page eject; Indent 10 spaces
Print integer I in field of width 2; Print ')'
Print '  X(I)= '; Print X(I) with G12.5 format; Print ';'
Skip to next line; Indent 20 spaces
Print P(I) with G12.5 format; Print ' #'
```

The infinite open-ended FOR statement allows iterations in which a scalar control-variable takes on an arithmetic progression of values, the termination of the iteration being accomplished via a situation exit within the iterated block. Appendix E contains a prime-generating program exhibiting the usefulness of this feature.

When sequences are represented by linked-lists implemented via records and reference fields, it is often required to scan through such linked-lists in a fashion analogous to the way a normal FOR statement can scan through the indices of an array. For this purpose a LINK iteration statement is included in SKOL; it causes a reference variable to take on a succession of reference

values defined by a field and terminating when a NIL reference is encountered.

Syntax Notation and Syntax Flow Graphs

To describe the syntax of the SKOL language, we employ an extension of BNF defined as follows:

1) Reserved words and other terminal symbol strings of the language are enclosed in string quotes (e.g., 'IF', '+').

2) Syntactic constructs are named by identifying words sometimes including hyphens or operators, but no blanks (e.g., command, segment-body).

3) The notation $\alpha_1\ \alpha_2\ \ldots\ \alpha_n$ means α_1 followed by α_2, followed by α_3, etc.

4) The notation $[\alpha_1\ |\ \alpha_2\ |\ \ldots\ |\ \alpha_n]$ means one of the α_k.

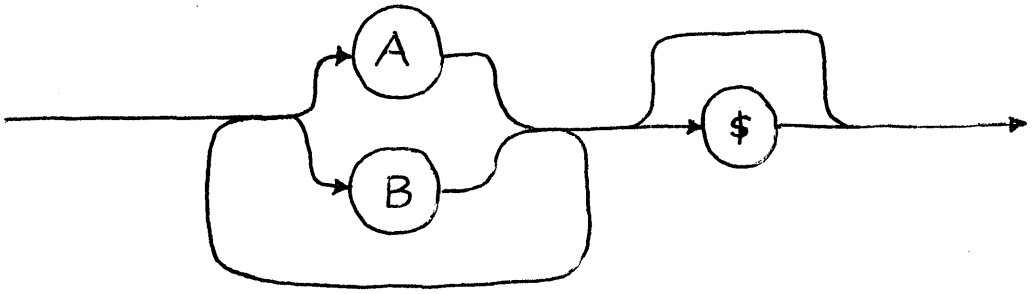
5) The notation $\{\beta\}_{\sigma}^{\text{count}}$ indicates a number of repetitions of β separated by σ, where count specifies a restriction on the possible number of β. If σ is omitted, then the βs are juxtaposed without an extra separator. The count specification indicates a range of non-negative integers; we have found frequent need for "zero or one" which we write as 0,1 and "n or more" which we write as $\geq n$.

For example, a rule for constructing identifiers which specifies one or more occurrences of letters A or B, followed by an optional $, can be described by:

$$\{['A'|'B']\}^{\geq 1}\ \{'\$'\}^{0,1}$$

Most of the syntax rules we will encounter can be very nicely and compactly described in the form of syntax flow graphs and we shall so describe

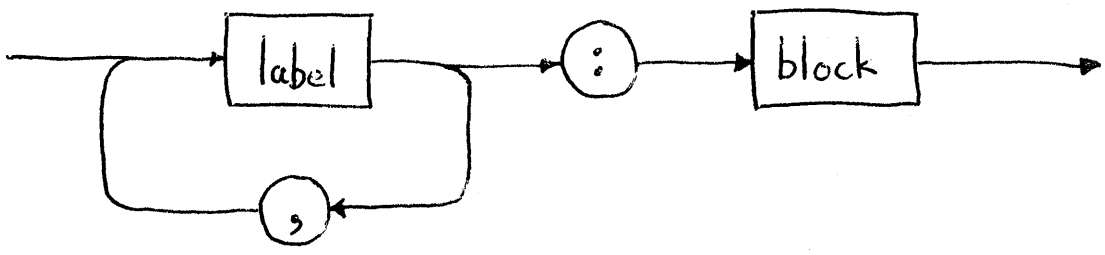
the syntax of SKOL in Appendix B. A syntax flow graph is a directed graph with unique start and finish nodes, terminal strings enclosed in oval nodes, and named syntactic constructs enclosed in rectangular nodes. Any valid directed path through the graph, from start to finish, represents a valid symbol sequence for the defining flow graph. The following flow graph* is equivalent to the above identifier rule:



Another example is the syntax described by:

$$\{\text{label}\}_{,}^{\geq 1}\ ':'\ \text{block}$$

and by the flow graph:



*These diagrams are not graphs in the strict sense but each such diagram corresponds to a proper directed graph whose edges correspond to "smooth" paths between nodes of the diagram.

Basic Format Rules for Program Text and Comments

Program text is essentially free-form in SKOL with no significance assigned to ends-of-line or particular columns within lines. The single exception to this is that any line with a % in the first column is interpreted as a special control line, and no characters on this line are considered to be part of the program text (see Appendix C).

The normal comment convention is described by:

'"' {non-"-symbol}$^{\geq 0}$ '"'

but this can be changed (via control line) to the safer:

'"' {non-"-symbol}$^{\geq 0}$ ['"'|end-of-line]

so that comments never extend over line boundaries.

Programs, Segments, Specifications and Actions

A SKOL program consists of a number of program-segments followed by a terminator-line which contains %% in the first two columns. This can be described by:

{program-segment}$^{\geq 1}$ terminator-line

where each program-segment is:

['BLOCKDATA' ':' {specification}$^{\geq 1}$ 'ENDBLOCKDATA' ';' |

'MAIN' ':' segment-body 'ENDMAIN' ';' |

'SUBROUTINE' Fident {parameters}^{0,1} ':' segment-body 'ENDSUBROUTINE' ';' |

'FUNCTION' Fident parameters Ftype ':' segment-body 'ENDFUNCTION' ';']

and segment-body is:

{specification}$^{\geq 0}$ {statement-function}$^{\geq 0}$ {command}$^{\geq 1}$

{routine definition}$^{\geq 0}$

and statement-function is:

Fident '(' {Fident}$^{\geq 1}_{,}$ ')' '=' Aexpr ';'

and command is:

[action | definition | format-declaration | pragmat]

and parameters are:

'(' {Fident}$^{\geq 1}_{,}$ ')'

An example of a statement function is:

```
ROUND(X) = INT(X+SIGN(.5,X));
```

It should be prefaced by declarations:

```
REAL X;  INTEGER ROUND;
```

The syntax for routine-definitions is described in the section on "Routines, Coroutines,...". Specifications and actions are precisely defined in Appendix A. An Ftype is defined by:

['REAL' | 'INTEGER' | 'LOGICAL' | 'COMPLEX']

and a definition is any CONSTANT, DEFINE or MACRO statement as described in the sections on "Constants, Definitions and Text Substitution" and "Macro Procedures...".

A format-declaration is a FORMAT statement as described in the section on "General Formatted Input and Output" and a pragmat is a RUNCHECK or TRACE statement as described in the section on "Run-time Error Checks and Variable Traces".

Statements, Blocks and Sequential Control

All statements in SKOL (specifications and commands) are terminated by a semicolon. A block is a sequence of commands. Formally, it has the form:

{command}$^{\geq 0}$

Notice that it may be an empty sequence of commands having no effect. A command is an action, a definition, a format-declaration or a pragmat (see Appendix A).

The most basic control statement is the "if" statement with the form:

'IF' {Lexpr ':' block}$^{\geq 1}_{\text{'ELSEIF'}}$ {'ELSE' ':' block}^{0,1} 'ENDIF' ';'

where Lexpr is a FORTRAN logical expression.

The execution of this statement is performed by testing the sequence of one or more Lexprs until one of them is true and then executing the statements of the corresponding block. If all Lexpr are false, then the block after ELSE is executed; when no ELSE phrase is explicitly specified, it is just as if the empty block has been specified.

Example:

```
IF A < 0 :
     J := J+1; P(J) := A;
ELSEIF A>0 :
     J := J-1; R(J) := A;
ELSE:
     OUTPUT (J,S(J));
ENDIF;
```

Another basic control statement of rather recent vintage [6, 7, 8] is the "situation" case statement which has the basic form:

'UNTIL' {ident}$^{\geq 2}_{\text{'OR'}}$ ':' block 'THENCASE' ':'

{{situation}$^{\geq 1}_{\text{','}}$ ':' 'BEGIN' block 'END'}$^{\geq 1}$ 'ENDUNTIL' ';'

where each situation is one of the idents in the UNTIL phrase and every ident appears exactly once as a situation. Within the block before THENCASE, "situation" statements of the form:

situation ';'

will cause immediate termination of the block and then execution of whichever block is associated to that particular situation in the THENCASE part.

Example:

```
UNTIL MATCH OR NO_MATCH:
  FOR I = 1 TO N:
     IF X = TABLE(I) : MATCH ; ENDIF ;
  ENDFOR;
  NO_MATCH;
THENCASE:
  MATCH : BEGIN COUNT(I)  := COUNT(I) +1; END
  NO_MATCH : BEGIN N  := N+1; TABLE(N)  := X;
                   COUNT(N)  := 1;
             END
ENDUNTIL;
```

This example shows how multiple-exit loops can be handled using the situation case.

An auxiliary form of this statement allows the suppression of the case part when only one situation can occur; the abbreviated form is:

'UNTIL' ident ':' block 'ENDUNTIL' ';'

Example:

```
UNTIL NON_BLANK :
   FOR I = 1 TO 81 :
      IF CH(I) ¬= ' ' : NON_BLANK ; ENDIF;
   ENDFOR;
ENDUNTIL;
```

This program delivers the index of the first non-blank character in array CH on assumption that CH(81) ≠ ' '.

The most basic repetetive statement in SKOL is the repeat statement of the form:

'REPEAT' {Iexpr 'TIMES'}^{0,1} ':' block 'ENDREPEAT' ';'

where Iexpr is an integer expression whose value should be non-negative. If the optional TIMES phrase is absent, the repetition is infinite and, therefore, the programmer must satisfy himself that eventually some "situation" statement within the repeated block will terminate an outer block enclosing the entire REPEAT statement.

Example:

```
REPEAT 5 TIMES : OUTPUT('*****') ; ENDREPEAT;
```

An extremely useful repetitive statement is the "Dahl-loop" which subsumes the familiar "while-do" and "repeat-until" statements from structured programming. Its form is:

'LOOP' ':' block 'WHILE' Lexpr ':' block 'ENDLOOP' ';'

The first command sequence is executed and if Lexpr is false, the repetition is terminated; if Lexpr is true, then the second command sequence is executed, followed immediately by the first sequence and the test and possible termination, etc.

Example:

```
SUM := 0.0; COUNT := 0;
LOOP : INPUT (I:I5, X:F10.5);
WHILE I > 0 :
   SUM := SUM + X;  COUNT := COUNT + I;
ENDLOOP;
OUTPUT ($SKIP2, COUNT, SUM);
```

SKOL has two forms of "for" statements, one infinite and one finite described by:

'FOR' Ivar '=' Iexpr 'BY' Iexpr ':' block 'ENDFOR' ';'

or

'FOR' Ivar '=' Iexpr {'BY' Iexpr}^{0,1} 'TO' Iexpr ':' block 'ENDFOR' ';'

where Ivar and Iexpr are any integer variables or expressions, respectively. Ivar may be a subscripted variable and the increment expression may be negative. In this context, integer includes any programmer defined scalar types as described later.

The second form may fail to execute block even once if the iteration phrase specifies an empty arithmetic progression of integer values as in I = 1 BY 1 to 0. In this case, the value of Ivar will be unchanged. If a non-empty arithmetic progression terminates normally, then Ivar will have the exact terminal value at completion of the FOR statement. A runtime error may occur if the terminal expression does not differ from the initial expression by an exact multiple of the increment expression. For example, I = 11 BY -2 TO 0 is considered to be in error.

Example:

```
FOR P(J) = 0 BY -(INC+1) : ... ENDFOR;
FOR IND = 2 BY DELTA TO N-1 : X(2,IND) := 0; ENDFOR;
FOR K = 1 TO 100 : P(K) := A(K) + B(K); ENDFOR;
```

The default increment value is +1 when the BY phrase is omitted.

The infinite form of FOR carries the same warning concerning termination as was given for the analogous infinite REPEAT.

To cater for simple iterations in the most efficient way, a FORTRAN-like "DO" statement of the following form may be used:

'DO' simpleIvar '=' simpleIexpr 'TO' simpleIexpr ':' block 'ENDDO' ';'

where simpleIvar means non-subscripted integer variable and simpleIexpr means a positive integer constant or non-subscripted integer variable.

Constants, Definitions and Text Substitution

The programmer can declare that certain names are to be considered equivalent to constant values using a constant-definition of the form:

'CONSTANT' {ident '=' value}$^{\geq 1}_{,}$ ';'

Whenever such an ident occurs subsequently in the program text (preceded and followed by blanks!), it will be replaced by the corresponding value.

Example:

```
CONSTANT PI = 3.14159, CM_PER_INCH = 2.54;
X := (2.0* PI * RADIUS)* CM_PER_INCH ;
CONSTANT $INPUT = 5, $OUTPUT = 10, $GENFORM = G20.7;
CONSTANT  LIMIT = 50;
REAL A( LIMIT , LIMIT ), B(2, LIMIT );
FOR I = 1 TO LIMIT : B(1,I) := 0; ENDFOR;
```

The constant-definition facility is really a special case of a more general definition statement whose form is:

'DEFINE' '''' pattern '''' '=' '''' replacement '''' ';'

where pattern and replacement are sequence of characters and special "operators" as defined in the user documentation for the MORTRAN2 macro-preprocessor [12]. Rather than repeat that description here, we will simply give several simple examples of the use of this text substitution facility.

In its simplest form, a pattern is just a sequence of characters (with ' and # and @ represented by '', ## and @@, respectively) and replacement is similar.

Example:

```
DEFINE ';INITIALIZE;' = ';A := 0; B := 1; P(2) := 3;' ;
```

Every subsequent instance of the pattern will be replaced by a copy of the replacement. In this form, the DEFINE is a parameterless macro facility.

By placing # at various places in the pattern, and by placing #δ where δ is a digit in the replacement, one can create parameterized text substitution rules. Indeed, SKOL is translated into FORTRAN by just such rules.

Example:

```
DEFINE ';SWAP(#,#);'  =  ';R99999 := #1 ; #1 := #2 ; #2 := R99999;' ;
SWAP(A(I,J),A(J,I));
```

The swap statement will be translated to:

```
R99999 := A(I,J) ; A(I,J) := A(J,I) ; A(J,I) := R99999;
```

Each # in pattern will match any character sequence which is properly parenthesized and contains no semicolon. Each #δ in replacement means the actually matching text for the δ-th # in pattern.

Macro definitions may be placed in replacement text to create some very powerful effects.

Example:

```
DEFINE ';TRACE #;' =
   ';DEFINE'';#1:=##;''='''";" #1 := ##1 ;
      OUTPUT('''''*****TRACE '''',#1);'';' ;
TRACE Z;
Z := F(Y)*Z; ... Z := A(2,K);
```

The above 3-line macro definition essentially extends the language by adding a trace statement of the form:

'TRACE' variable ';'

This statement will cause all subsequent assignments to variable to be followed by a well-annotated dump of the newly assigned value. The statement TRACE Z; will be replaced by the following text:

```
DEFINE ';Z:=#;' =
    '";" Z := #1 ; OUTPUT(''*****TRACE '',Z);';
```

This macro-definition causes the statement Z := F(Y)*Z; to be replaced by

```
Z := F(Y)*Z;
OUTPUT('*****TRACE ',Z);
```

When Z := F(Y)*Z; is executed, a line like the following will be printed on $OUTPUT:

```
*****TRACE   Z = 114.72;
```

The double-quotes " around ; are merely to prevent an infinite recursion in the rescan mechanism of MORTRAN2.

This trace facility is actually included in the SKOL language and its implementation requires little more than the above 3-line macro.

Record Classes, References and Dynamic Allocation

A record is a structure consisting of a fixed number of components called fields, each identified by a field-identifier. Each field may be of any simple type or array thereof or may be a reference field pointing to another record (also possibly an array of such).

A record class consists of a fixed number of records, all of the same form used as a pool for the dynamic creation and release of record variables directly accessible to the programmer. Each record class is named and introduced via the specification:

'RECORD' 'CLASS' '(' +Iconst ')' 'OF' Fident4 ':' {field-group ';'}$^{\geq 1}$
'ENDRECORD' ';'

where each field-group is of the form:

['REF' | Ftype | 'CHAR'] ':' {ident {array-bounds}^{0,1}}$^{\geq 1}_{,}$

array-bounds is:

'(' {+Iconst}$^{\geq 1}_{,}$ ')'

and Fident4 is a FORTRAN symbolic name of 4 characters or less.

Example:

```
RECORD CLASS (100) OF PERS:
   REF : NEXT,FATHER;
   INTEGER : AGE, ID_NUMBER,LEAVE(12);
   REAL : PAY ;
   LOGICAL : MARRIED ;
   CHAR : NAME (15) ;
ENDRECORD;
```

A reference variable identifies a record of a particular class once such a record has been dynamically created and associated with the variable. Each reference variable is restricted to refer to records of only one class and is introduced by a specification:

'REF' 'TO' class ':' {Fident {array-bounds}$^{0,1}\}^{\geq 1}_{,}$ ';'

Example:

```
REF TO PERS : WORKER, FORMAN  (6), P, LAST;
```

Before any use can be made of a record class, it must be initialized by a statement of the form:

'MAKEAVAIL' class ';'

The effect of this statement is to return all records of the designated class to the available pool ready for re-use.

To allocate a new record to a reference variable requires a statement of the form:

'NEW' reference ';'

Analogously, a record is released by:

'FREE' reference ';'

In each case, the variable designated must have been declared as a reference to some record class; otherwise, a diagnostic message will ensue.

To access a field of a record associated with a reference variable requires a special form called a field-designator:

'@' '(' reference '.' field ')'

The reference and/or the field may be subscripted if that corresponds to the declarations. If the field is not among those declared for the record class to which the reference has been bound, then an error diagnostic is given.

Notice that designators like @(@(P.NEXT).VAL) are not legal and must be replaced by

```
Q := @(P.NEXT) ; ... @(Q.VAL) ...
```

where Q has been properly declared as REF to the class of records having a VAL field.

Example:

```
MAKEAVAIL PERS ; NEW WORKER ;
@(WORKER.AGE) := 25; @(WORKER.LEAVE(3)) := 2;
FORMAN(1) := WORKER ; @(WORKER.NAME(1)) := 'Z' ;
IF @(FORMAN(1).LEAVE(K)) >2 : ... ;
FREE WORKER;
```

When a portion of program text concentrates its attention on a particular record, it is possible to abbreviate the field-designators by employing a "WITH" statement of the form:

'WITH' reference ':' block 'ENDWITH' ';'

Inside the block, any fields of the record identified by the designated reference may be accessed by the shorter form:

'@' '.' field ' '

Example:

```
WITH WORKER : @.AGE := 25; @.LEAVE(3) := 2; ENDWITH;
```

There is a standard identifier NIL which indicates an undefined reference value and is often used to mark the ends of linked lists. To traverse a linked list defined by a REF field in some record class, there is an analogue of the familiar for statement having the form:

'LINK' reference '=' reference 'BY' field ':' block 'ENDLINK' ';'

The iteration is discontinued at the first NIL value encountered (which may be the first).

Example:

```
@(FORMAN(6).NEXT) := NIL; SUM := 0;
LINK P = WORKER BY NEXT : SUM := SUM + @(P.AGE) ; ENDLINK;
```

A record class is actually an array of records so if the programmer desires, he may use it as a simple array while avoiding any dynamic allocations vis a vis the record class. Access to the records must still be through variables declared REF TO class, but these variables can be treated as integers, which they actually are.

Example:

```
"ASSUMING NO RECORDS CURRENTLY ACTIVE FROM PERS"
LAST := NIL ;
FOR WORKER = 1 TO 50 :
   @(WORKER.NEXT) := LAST ; LAST := WORKER ;
ENDFOR;
TOP := 50;
LINK P = TOP BY NEXT :
   WITH P : @.NAME(1) := '#';@.AGE := 20; ENDWITH;
ENDLINK;
```

Scalar Types, Subtypes, and Case Statements

The programmer may introduce a new finite ordered primitive type (called a scalar) by naming it and supplying a list of the unique identifiers which denote the constant values of the new type. The values of the new type may be arranged in a hierarchy of groups or named subtypes. The definition of a new type takes the form:

'TYPE' ident '=' list-of-subtypes ';'

where list-of-subtypes has the form:

[empty | '('{subtype}$^{\geq 1}_{,}$ ')']

and subtype is

[ident | ident '=' list-of-subtypes | char-const]

The possibility of an empty list-of-subtypes is restricted to the CHAR scalar type and char-const is also so restricted. The definition of CHAR will be discussed later in this section.

Example:

```
TYPE AUTO =
   (GEN_MOTORS = (CHEVY,PONTIAC,CADDIE),
   FORD = (MUSTANG,MERCURY=(MONTEREY,COUGAR)),
   FIAT = (COUPE,S128,S131) );
```

This declaration specifies that a value of type AUTO will be a value of one of the subtypes GEN_MOTORS, FORD or FIAT. The values of subtype GEN_MOTORS are the three constants CHEVY, PONTIAC and CADDIE. FORD consists of MUSTANG and a subtype MERCURY, which itself consists of two constants MONTEREY and COUGAR. finally, the subtype FIAT has three constant values as indicated. In subsequent use, these scalar constants must be preceded and followed by a blank!

Scalar variables are declared in a fashion similar to normal FORTRAN declarations:

scalar {Fident {array-bounds}^{0,1}}$^{\geq 1}_{,}$ ';'

Example:

```
AUTO FAMILY(2) , MINE ;
```

The scalar case statement allows one from a group of blocks to be executed, the selection being determined by the current value of some scalar variable. The form of the statement is:

'CASE' scalar-var ':' scalar 'OF'

{{label}$_{,}^{\geq 1}$ ':' 'BEGIN' block 'END'}$^{\geq 1}$ {'ELSE' ':' 'BEGIN' block 'END'}^{0,1}

'ENDCASE' ';'

where scalar may be the name of any scalar type or subtype, and each label is a constant or subtype of that type. In the latter case, it is simply an abbreviation for the list of all constants included in the subtype.

Example:

```
CASE FAMILY(K) : FORD OF
   COUGAR, MUSTANG : BEGIN J := J+1; END
   ELSE : BEGIN J := J-1; END
ENDCASE;
CASE MINE : AUTO OF
   MERCURY,COUPE :
        BEGIN  ...  END
   S128 :
      BEGIN  ...  END
   GEN_MOTORS :
      BEGIN
      CASE MINE : GEN_MOTORS OF
         PONTIAC,CADDIE : BEGIN  ...  END
         ELSE : BEGIN  ...  END
      ENDCASE;
      END
```

```
    ELSE : "MUSTANG AND S131"
        BEGIN ... END
ENDCASE;
```

Each constant of the indicated scalar must occur exactly once as a label unless an ELSE block is present. In the latter case, ELSE collects all constants not explicitly listed. If the scalar-var is not within the range of values of scalar, an error has occurred which will be diagnosed at run-time if the runcheck option is enabled for case statements.

In the first example above, the only valid labels are those constants in subtype FORD, that is, MUSTANG, MONTEREY and COUGAR. As a consequence, the ELSE is identical to MONTEREY. The order of occurrence of labels is completely irrelevant except for ELSE which, if present, must come last.

In the SKOL language, the character data-type is not built-in as a language-defined primitive but is recognized as a special case since most constants have the special form of a single character symbol enclosed in apostrophes ('). The CHAR scalar type is declared explicitly by the programmer as a scalar type and can be hierarchically substructured like any other scalar. Most constants conform to the normal convention for characters, however.

To ease the burden for the programmer, there are some character subtypes built-in. The subtype ALPHABET consists of the capital letters 'A' through 'Z' and DIGIT means '0' through '9'. In addition, certain installation-dependent subtypes may be supplied; for example, RELATIONAL = ('<', '=', '>') or ARITHMETIC = ('+', '-', '*', '/').

A special facility is available to ask if a given scalar value is contained in a particular scalar subtype. The form of the expression is:

```
'IN_' scalar '(' scalar-expr ')'
```

Example:

```
IF IN_DIGIT(CH(K)) : ...
```

The following three functions are also included:

FIRST , LAST (scalar)

VALUE (digit-expr)

For example, FIRST (ALPHABET) = 'A', LAST (DIGIT) = '9' and VALUE ('3')= 3.

Example:

```
TYPE CHAR =
   (NAME_SYMBOL = (ALPHABET=,DIGIT=, '_'),
   DELIMETER=
      (ARITHMETIC=('+', '-', '*', '/')
      RELATIONAL=('<', '=',  '>'),
      LOGICAL=('¬', '&', '|'),
      PUNCTUATION=(',', ';', ':', ' ', '.', '?'),
      BRACKET=('(', ')'),
      QUOTE=('"', '''')  ),
   SPECIAL=('$', '@', '#', '%'),
   EOL  "NOTICE THAT IDENTS ARE OKAY FOR CHAR CONSTS"
   );
```

This flexibility of the character data-type allows the programmer to arrange the various character subtypes and special characters in an order that corresponds to their use in a particular application.

The declaration for the CHAR type must be followed by three constant definitions:

```
CONSTANT BITS_PER_BYTE = ?,
         BITS_PER_WORD = ?,
         SHORT_BYTE    = ?;
```

where the value of SHORT_BYTE should be

2 **(BITS_PER_BYTE - 1)

The user must also supply the auxiliary subprograms RDSTR9, WTSTR9, INIT99, INCV99, IRPL99, IRPL98 and IDEL99 which are used to implement the character facilities (see Appendix D and sample programs in Appendix E).

Special functions $INCHAR and $OUTCHAR are provided to map external characters to their internal integers and vice versa; for example, with the above declared CHAR and I containing character 'B' read under A1 format, we get $INCHAR(I) = 2 and $OUTCHAR('_') output under an A1 format is '_'.
Each user subprogram which uses $OUTCHAR or the C format in an OUTPUT command must contain the specification:

'CHAR_COMMON' ';'

and before any character manipulation is performed, the following initializing command must be performed:

'CHAR_SETUP' ';'

Character Strings, Contexts and String Modification

In addition to fixed-length character arrays like CHAR CARD(81), it is possible to have varying length character strings with a fixed maximum length called the size. They are declared in the form:

'STRING' {Fident '(' +Iconst ')'}$^{\geq 1}_{,}$ ';'

Example:

STRING NAME(30), WORD(10);

Built-in functions SIZE (string) and LENGTH (string) are available to obtain the size and current length of any string. Actually, the latter is an integer variable which can be changed by assignment, but is intended to be im-

plicitly reset by string updating statements. Prior to use, the string should be initialized to the empty string by the command:

DELETE string;

which is described in the following.

To designate substrings of a string, there is a notation for string-context whose form is one of:

string '(' index {'...' index}^{0,1} ')'

or

string '(' index '...' '|' length ')'

or

string '(' length '|' '...' index ')'

where index means a valid integer index into the string and length is a positive integer.

The first form denotes the substring consisting of all indices between the two limits inclusive. If there is only one, then the limits are equal. The second form denotes a substring of the indicated length which begins with the indicated index. The third form denotes a substring of the indicated length which ends with the indicated index.

Example:

NAME (2 ... K+2)

WORD (4| ... LENGTH(WORD))

NAME (1 ... |5)

WORD (5)

Notice the second example which denotes the last four characters of the current value of the string WORD.

The notation introduced above for substrings can be used to denote any empty position in or at the ends of a string if the proper meaning is attached

to substring denotations involving a length of 0. If we redefine the notation (index ... | length) to mean the subsequence (possibly empty) beginning just before index and having the given length, then the notation

```
WORD (2 ... | 0)
```

denotes the position before the second character of WORD.

Because of the symmetry of our notation, the extremes of WORD can be described by the following two denotations

```
WORD (1 ... | 0)
```

and

```
WORD (0 | ... LENGTH (WORD))
```

The reason for wanting to denote empty substrings within a string is so that a single all-powerful replacement command can be indicated by a string context and a replacing string expression.

The general string replacement statement takes the form:

```
'REPLACE' [string-context | string] 'BY'
   ['NULL' | char-expr | string-context] ';'
```

Example:

```
REPLACE WORD (2 ... 4) BY 'Z' ;
REPLACE WORD (1 ... | 0) BY NAME (3 | ... K);
REPLACE NAME BY WORD (2);
```

Arbitrary substring "deletions" can be accomplished by replacement using NULL and "insertions" using an empty (length = 0) string context. SKOL contains the following statement forms for this:

```
'DELETE' [string | string-context] ';'
'INSERT' [char-expr | string-context]
   ['BEFORE' | 'AFTER'] string '(' index ')' ';'
```

Example:

```
DELETE WORD ; DELETE NAME (6 ... LENGTH (NAME));
INSERT NAME (1 ... | 3) BEFORE WORD (2);
INSERT CH AFTER WORD ( LENGTH (WORD));
```

For reasons of efficiency, there is a concatenation statement which is implemented separately from the general replacement command. The form of the statement is:

'CATENATE' string-expr 'ONTO' string ';'

where string-expr is:

$\{[\text{char-expr} \mid \text{string-const} \mid \text{string} \mid \text{string-context}]\}^{\geq 1}_{\text{'\&'}}$

and '&' indicates concatenation.

Example:

```
CATENATE '@''B'  & CH & NAME (1 ... | 2) ONTO WORD;
CATENATE NAME (2 ... 4) ONTO WORD;
```

Because the conversion between external character format and internal integers is not defined by FORTRAN but rather by the programmer's type declaration, special facilities are required for input and output of character string data. These are of the form:

['READ' | 'WRITE'] 'STRING' {'(' file ')'}^{0,1}
string {'(' index '...' index ')'}^{0,1} ';'

The default index range is 1 ... LENGTH (string) for WRITE and 1 ... SIZE (string) for READ.

Example:

```
STRING CARD (81);
READSTRING (MYFILE) CARD (2 ... 81) ; LENGTH(CARD) := 81;
CARD (1) := '1' ;    "FOR PAGE-EJECT"
WRITESTRING CARD; "DEFAULT RANGE = 1 ... LENGTH(CARD)"
```

Notice that READSTRING does not set the LENGTH; the standard files for text input and output are the defaults and the initial character is used for control on output. In the example above, the first 80 characters of the next record of the file named MYFILE are converted to internal format and stored into the string CARD at positions 2 through 81. This character sequence is then listed after a page eject on the standard print file.

It is quite easy for the programmer to implement a MOVE statement which has the form:

```
'MOVE' string-context 'TO' string-context ';'
```

and causes a substring of one string to replace a string-context of a second string while being deleted from the first string.

The following macro-definition will implement such a MOVE statement:

```
DEFINE ';MOVE # TO #;' =
   ';REPLACE #2 BY #1 ; DELETE #1 ;' ;
```

The meaning of the # within a define statement is explained in the section on "Constants, Definitions and Text Substitution".

There is a special string assignment statement of the form:

```
'#' string ':=' string-expr ';'
```

Example:

```
#NAME := 'JONES' & BLANK & 'JOHN' ;
#NOTHING := ''   "SAME AS DELETE"
```

Routines, Coroutines, Processes and Recursion

Simple parameterless routines can be defined and executed at different places within a major program segment (the subprograms of FORTRAN). The routine definitions are placed after the RETURN statement for that segment. The form of the routine definition is simply:

'ROUTINE' ident ':' block 'ENDROUTINE' ';'

To invoke execution of such a routine requires:

'EXECUTE' routine ';'

Example:

```
EXECUTE IN_CARD;
ROUTINE IN_CARD : ... ENDROUTINE;
```

It is also possible to declare a process to consist of several cooperating coroutines which resume one another or suspend the entire process. The main program (i.e., body of the segment) controls resumption of the suspended process and also decides which coroutine will be invoked first. The process declaration has the form:

'PROCESS' ident '=' '(' $\{\text{ident}\}_{1,}^{\geq 1}$ ')' ';'

where the list of idents refer to coroutines to be defined later. To initialize a process so that each constituent coroutine is asleep at its beginning, and so that the initial resumption of the process invokes a particular coroutine requires:

'START' process 'AT' coroutine ';'

The main program resumes the process by:

'RESUME' process ';'

and any of the constituent coroutines suspend the process in favor of the main program by:

'SUSPEND' process ';'

Within a coroutine, its own execution may be postponed in favor of another coroutine by:

'RESUME' coroutine 'FROM' coroutine ';'

The coroutines are defined (like routines) after the RETURN from the program segment and the form is:

'COROUTINE' ident ':' block 'ENDCOROUTINE' ';'

Example:

```
PROCESS LIVE = (PRODUCE, CONSUME);
START LIVE AT PRODUCE;
RESUME LIVE;
.
.
.
COROUTINE PRODUCE : ... RESUME CONSUME FROM PRODUCE; ... ENDCOROUTINE;
.
.
.
COROUTINE CONSUME :
... RESUME PRODUCE FROM CONSUME; ... SUSPEND LIVE ; ...
ENDCOROUTINE;
```

Each RESUME process in the main program sends control back to the place where the last SUSPEND process was executed unless a START statement has more recently been executed. In the latter case, control passes to the beginning of the coroutine named in the START statement. Because of this protocol, it is convenient to view the group of coroutines (i.e., the process) as a "semi-coroutine" of the main program; there is a master/slave relationship between the main program and the coroutine process, but each subsequent resumption of the slave process retains the context at termination of its previous period of activity. Our use of the word "semi-coroutine" is similar to but not quite the same as found in Dahl and Hoare [4].

If execution of a coroutine reaches the end of the block defining its body, a terminal error message is generated and the program aborts.

Some routines may have integer "value" parameters, local integer variables, and may freely invoke themselves recursively. Such routines must be predeclared in a specification of the form:

'RECUR' '(' +Iconst ')' ':' {ident {'('{'*'}$_{1,}^{\geq 1}$ ')'}^{0,1}}$_{1,}^{\geq 1}$ ';'

Example:

RECUR (100) : TREE (*), WHAT (*,*), P;

This example introduces three potentially recursive routines, TREE having one parameter, WHAT having two parameters, and P without parameters. A stack of maximum size 100 will be used to implement the recursive executions of these routines.

The subsequent routine-definitions for such recursive routines have the form:

'ROUTINE' ident{'('{ident}$_{1,}^{\geq 1}$ ')'}^{0,1} {'LOCAL' '('{ident}$_{1,}^{\geq 1}$ ')'}^{0,1} ':'
block 'ENDROUTINE' ';'

Example:

ROUTINE TREE (TOP) LOCAL (LSON,RSON):

...ENDROUTINE;

In this example, TREE has one parameter TOP and two local variables LSON and RSON whose values will remain intact over recursive calls, etc.

Recursive routines are invoked by execute statements of the form:

'EXECUTE' routine {'('{expr}$_{1,}^{\geq 1}$ ')'}^{0,1} ';'

Example:

EXECUTE WHAT (2-J, CH);

The expressions are calculated and assigned to the formal parameters at entry to the routine body. This mode of parameter communication is commonly refered to as "call by value".

The use of recursive routines is subject to one rather annoying restriction. If a FOR statement or DO statement in a recursive routine contains potentially recursive calls with different increments or final values for the iteration phrase, then unpredictable and usually incorrect behavior will re-

sult. Safety is provided by the use of LOOP ... ENDLOOP, making the control, increment, and limit variables LOCAL to the routine.

Appendix E contains a SKOL program which closely resembles a PASCAL program described in Wirth's recent book [14] to illustrate the use of recursive routines in conjunction with recursive data.

To consistently integrate the recursive capability with the situation terminations provided in the situation case statement, the following extension is available. A single-situation UNTIL statement may have the form:

'UNTIL' {'GLOBAL'}^{0,1} ident ':' block 'ENDUNTIL' ';'

and the presence of the word GLOBAL will cause all situation terminations with the indicated name to reset the recursion stack to its status at entry to the UNTIL statement.

Major subroutines (inherited from FORTRAN) are invoked by:

'CALL' subroutine {'(' {arg}$_{1,\ ,}^{\geq 1}$ ')'}^{0,1} ';'

where arg is defined in Appendix A.

Macro Procedures, Keyword Parameters and Defaults

The programmer may define macro procedures with formal parameters some or all of which have specified default actual parameters; these macro procedures are invoked by a calling sequence in which the correspondence between formal parameters ("keywords") and actual parameters is explicit and non-positional. Unspecified formals are given the defaults associated with them in the macro definition; if no default was specified, then an error message ensues.

A macro procedure definition has the form:

'MACRO' ident '('{ident {'=' Xexpr}^{0,1}}$_{1,\ ,}^{\geq 1}$ ')' '=' '''' text '''' ';'

where text is a piece of program text. Xexpr is explained below.

Example:

```
MACRO ORDER (REL=<, X,Y) =
   '; IF NOT (X REL Y) : SWAP ( X , Y ); ENDIF; ' ;
```

A macro procedure invocation has the form:

macro '(' {keyword '=' Xexpr}$_{,}^{\geq 1}$ ')' ';'

where each keyword is one of the formal parameters in the definition of macro.

Example:

```
ORDER (X=A(2) , Y = B(K));
ORDER (REL = >=, Y = T, X= P(J));
```

These two statements are translated into the following program text:

```
IF NOT (A(2) < B(K)):
   SWAP (A(2), B(K)); ENDIF;
IF NOT (P(J) >= T) :
   SWAP (P(J), T); ENDIF;
```

Notice that formal parameters can represent relations, operators, statements and procedure names as well as variables and expressions. An Xexpr is an extended expression which includes these.

General Formatted Input and Output

SKOL provides input and output statements to and from text files, employing a syntax in which the data format associated with a variable is textually adjacent to the variable rather than being in a separate list. All control format items occur within the sequence of data items at the appropriate positions.

In the case of output, each variable may be printed according to an explicitly specified format, an implicit format appropriate to the type of vari-

able or an implicit format preceded by the name of the variable and followed by a semicolon.

The input statement takes the form:

'INPUT' '('{['':' control | variable ':' data-format]}$^{\geq 1}_{,}$ ')' ';'

where control is

[{'/'}$^{\geq 1}$ | {+Iconst}^{0,1} 'X']

and data-format is a valid FORTRAN data format item.

Example:

```
INPUT (:5X, J : I3, A(J) :F10.5, :/, :X, WHAT : L1);
```

The general output statement has the form:

'OUTPUT' '(' {[':' carriage-control | ':' control | output-item]}$^{\geq 1}_{,}$ ')' ';'

where carriage-control is:

['$PAGE' | '$SKIP' | '$SKIP2' | '$OVER']

and output-item is:

[variable {':' {[data-format | 'C']}^{0,1}}^{0,1} | output-text]

When variable is followed simply by : then the general type-dependent format is used, and when variable stands alone, then its name is printed before the data. Output-text is printed as is. The carriage-controls should come in first position or after a '/' control item. In first position, the colon may be omitted before a carriage-control. The format specification :C indicates conversion of internal character (CHAR) representation to external text.

Example:

```
OUTPUT($PAGE,:3X,J:,A(J),:5X,CH:C,'$');
```

This statement will print on the first line of a new page (assuming J=42, A(J)=142.36, and CH='P')

```
      42 A(J) = 142.36;     P$
```

The input and output statements read from and write to text files $INPUT and $OUTPUT respectively. These file identifiers are associated with the standard text input and output files unless the programmer requests otherwise by redefining them in the program text (see section on text substitution).

The general format used in the OUTPUT statement is governed by the current definition of $GENFORM which is FORTRAN format G12.5, but can be redefined by the programmer.

The user should especially note that the carriage-control character will be set to blank if no explicit carriage-control has been specified!

If a particular sequence of data-format and control items is used in several places in a program, then this "format" may be named and defined by a FORMAT definition of the form:

'FORMAT' ident '=' '(' format-list ')' ';'

where format-list is:

$$\{[\text{control} \mid \text{data-format} \mid \text{output-text} \mid +\text{Iconst '('} \text{ format-list } \text{')'}]\}_{,}^{\geq 1}$$

These formats can then be used in read and write statements of the form:

$$['READ' \mid 'WRITE'] \text{ '('} \text{ file ','} \text{ format } \text{')'} \{\text{variable}\}_{,}^{\geq 0} \text{ ';'}$$

where format is the name of a previously defined format sequence.

Example:

```
FORMAT PERSON = (1X, 2I5, F10.2);
READ ($MYFILE , PERSON) @(WORKER.AGE),J,@.PAY ;
FORMAT PRETTY = ('1', 1X, I3, ' ) ', F10.5,/,' ');
WRITE ($OUTPUT , PRETTY) K, B(K);
```

Augmentation Statements

Because of the ubiquitous requirement to update the values of variables by incrementing or decrementing their current values, the SKOL language provides an augmentation statement of the form:

['INCR' | 'DECR'] variable {'BY' expr}^{0,1} ';'

The BY phrase defaults to BY 1 if omitted, and INCR, DECR mean, respectively, "add to", "subtract from".

Run-time Error Checks and Variable Traces

The SKOL/FORTRAN precompiler will insert run-time consistency checks into the target FORTRAN code for certain statement forms, but the decision of when and where to insert checks is under very precise user control. For this purpose, there is a runcheck option statement of the form:

'RUNCHECK' '(' $\left\{\{'-'\}^{0,1}\ ['ALL'|'FOR'|'CASE'|'UNTIL'|'TRACE']\right\}_{,}^{\geq 1}$ ')' ';'

A minus means suppress checks for the designated class of statements and the absence of minus means insert checks until told otherwise. The ALL means all the others. When TRACE is enabled, major control flow is traced by printout. This means that CALL, EXECUTE, RESUME and SUSPEND generate output messages, and for each repetition of a LOOP...WHILE...ENDLOOP,REPEAT...ENDREPEAT,FOR...ENDFOR or LINK...ENDLINK an output message is generated.

It is quite easy to suppress run-checks for short time-critical loops while leaving them on for less critical portions of a large program.

A variable tracing statement is available of the form:

'TRACE' {variable}$_{,}^{\geq 1}$ ';'

and causes subsequent assignments of new values to variable to be accompanied by a "dump" of the variable name and the new value. The section on "Constants,

Definitions, and Text Substitution" describes how this is accomplished.

Example:

```
RUNCHECK (ALL,-FOR);
TRACE A(K),@.VAL,P;
```

These two statements cause run-time checks to be inserted everywhere but in FOR statements, and tracing of all assignments of the form A(K) := expr, @.VAL := expr or P := expr. Implicit assignments generated by the implementation of iterations and the NEW statement are also traced.

Error Diagnostics

Every effort has been made to diagnose errors in the non-FORTRAN features of SKOL at the time of the SKOL to FORTRAN precompilation. The degree to which this has been achieved is somewhat surprising for precompilers, not to mention macro-implemented precompilers. In particular, the compile-time checks on the use of records, references and the CASE statement approach what could be accomplished by a very good compiler. Appendix F contains a sample of SKOL diagnostics with explanations. Appendix H explains messages which diagnose serious control syntax errors.

Run-time error messages are issued for zero increments in FOR statements, final value not exact in iteration phrases of FOR statements, terminations of UNTIL blocks without encountering a situation statement, expressions out-of-range in CASE statements, space exhausted in record classes, illegal termination of coroutines, attempts to read or write illegal characters, character string modifications which exceed the maximum size allowed, collision in character mapping (usually a duplicate occurrence of the same character constant in the CHAR type specification), stack underflow and overflow in invocations and returns for recursive routines.

Restrictions and Extensions Dependent on FORTRAN

Whenever Fident appears in the syntax of SKOL, it means any valid identifier (symbolic name) in the FORTRAN language dialect being used. However, if portability is desired, each Fident should be of form:

letter {[letter|digit]}$^{0...5}$

where a...b means any integer between a and b inclusive.

The name of a record class must be an Fident4 which must be a valid FORTRAN identifier after the appending of two digits. For standard FORTRAN, this means:

letter {[letter|digit]}$^{0...3}$

Other names indicated as simply ident should conform to:

[letter|special-symbol] {[letter|special-symbol|digit]}$^{\geq 0}$

where special-symbol is any non-alphanumeric character which is not used as a meaningful symbol of FORTRAN or SKOL. The simplest rule is to avoid everything in the 48-character FORTRAN set, ':', ';', '@', '"', '|', '&' and '¬' (but '$' is okay).

A few FORTRANs (including the standard) do not allow an integer subscript expression to be itself a subscripted array element (e.g., A(P(K)) is illegal). When such a restriction is in effect, it implies the following restriction in SKOL:

Reference variables may not be arrays

For example, REF TO class: S(5); would be illegal. Note that @(@(P.NEXT).AGE) must be expressed by use of a temporary reference variable Q, as follows:

```
Q := @(P.NEXT);
... @(Q.AGE) ...
```

Some dialects of FORTRAN (notably IBM FORTRAN IV) allow the programmer to indicate what should be done in case an input (i.e., READ) statement encoun-

ters an end-of-file or the attempted read results in an error. In this case, SKOL is extended so that the new syntax of READ statement (not including string version) is:

'READ' '(' file ',' format {',' 'END' '=' situation}^{0,1}
{',' 'ERR' '=' situation}^{0,1} ')' {variable}$^{\geq 0}_{,}$ ';'

Other non-standard features available in a local dialect of FORTRAN may be used with a certain loss of portability. Such features might include multiple ENTRYs to a subprogram, direct-access I/O, etc. Warning: the direct-access I/O available in IBM FORTRAN IV may not be used in SKOL because that extension uses the single-quote character as a separator. Such statements can be used only if the SKOL precompiler is turned off temporarily (see Appendix C).

The optional data portion of a variable specification is not strictly standard FORTRAN so on some compilers the programmer will be forced to use the separate FORTRAN DATA statement.

Items of the form

variable ':' data-format

may be replaced by

iterated-data ':' +Iconst '(' format-list ')'

in READ, WRITE, INPUT and OUTPUT commands where iterated-data has the form:

'(' 'FOR' simpleIvar '=' simpleIexpr 'TO' simpleIexpr ':'
{[iterated-data|variable]}$^{\geq 1}_{,}$ ')'

Example:

```
INPUT ((FOR I=1 TO 3 : (FOR J=1 TO 3 : I,J,A(I,J))) : 3(3 (2I2,F10.5,5X),//));
```

Coroutine resume and suspend are not legal within DO iteration statements for some FORTRAN compilers (e.g., WATFIV) which don't implement the extended DO. The remedy is to simply change such a DO to the more general FOR statement provided by SKOL.

Matrix Operations: An Example of Language Extension

To give a general feeling for the kind of language extension that can be implemented with a modest amount of effort by someone reasonably familiar with the use of the MORTRAN2 macro-translator [12], we shall describe the implementation of some simple operations on matrices of REAL values. First, the syntax of the new features:

Matrices are declared in the form:

'MATRIX' {Fident '(' +Iconst ',' +Iconst ')'}$^{\geq 1}_{,}$ ';'

and are modified by simple MATRIX assignments of the form:

'SET' matrix ':=' Mexpr ';'

where Mexpr is one of:

'(' Rexpr ')'

or

{'$.'}^{0,1} matrix {['+'|'*'] {'$.'}^{0,1} matrix}^{0,1}

The notation $.MAT shall indicate the transpose of matrix MAT and the parenthesized real expression will be assigned to all positions of the matrix.

Example:

```
MATRIX A(5,10), B(10,10), C(5,10), D(10,5);

SET A := (0); SET B := (1.0); SET C := (.5-X**2);

SET A := A+C; SET D := $.A;

SET C := A*B;

SET B := $.C*C;
```

We make no pretense that these facilities are completely satisfying for all possible applications. Furthermore, to simplify the implementing macros, we have suppressed all error checking (see [12] for examples of that). The following macros will provide the above matrix facilities. Note that text of the form %' ... '=' ... ' is a macro-definition which may appear anywhere in program text.

```
DEFINE ';MATRIX #;' = '";REAL " $%%2#1,;' ;
DEFINE '$%%2#(#,#),' =
   '%''$#1.1'' = '' #2 ''
    %''$#1.2'' = '' #3 ''
    %''$$.#1.1'' = '' #3 ''
    %''$$.#1.2'' = '' #2 ''
    #1 ( #2 , #3 ), $%%2' ;
DEFINE ',$%%2;' = ';' ;
DEFINE ';SET #:=#;' =
   '";"@LG@LSO" DO "@LCOO I@LCO2 "=" 1,$#1.2
    ";DO" @LCOO I@LCO1 "=" 1,$#1.1;$%%1#1 := #2;
    @LCOO CONTINUE ;'   ;
DEFINE '$%%1#:=#;' =
   ' #1 (I@LCO1,I@LCO2) := #2 (I@LCO1,I@LCO2);@LUO';
DEFINE '$%%1#:=(#);' =
   ' #1 (I@LCO1,I@LCO2) := ( #2 );@LUO';
DEFINE '$%%1#:=#+#;' =
   ' #1 (I@LCO1,I@LCO2) :=  #2 (I@LCO1,I@LCO2)
    + #3 (I@LCO1,I@LCO2);@LUO';
DEFINE '$%%1#:=#*#;' =
   'R@LCO4 := 0.0 ";DO"@LCO5 I@LCO3 "=" 1,$#2.2;
    @LCO5 R@LCO4 := R@LCO4+ #2 (I@LCO1,I@LCO3)
    * #3 (I@LCO3,I@LCO2); #1 (I@LCO1,I@LCO2) :=
    R@LCO4;@LUO';
DEFINE '$.#(#,#)' = ' #1 ( #3 , #2 )' ;
```

The replacement parts of these macro-definitions can be understood after a moderate amount of study of [12]. the full list of definitions is included here to indicate how little extra machinery is necessary to implement some use-

ful features. A similar example in [12] checks for errors and issues warnings.

Other Uses of DEFINE

The small program INPOST, in Appendix E, contains some simple but very useful applications of the DEFINE text-substitution facility of SKOL inherited from MORTRAN. The program converts simple expressions with infix operators to the postfix notation using a temporary stack for operators and parentheses.

The algorithm used to implement INPOST requires commands NEXTCH to obtain the next input character, STACK to push the current input character onto the stack, UNSTACK to move the top element from the stack to the end of the partially completed postfix string, and POP to discard the topmost element of the stack. An expression TOP is needed to inspect the topmost stack element.

A character string OPSTK(20) is used to implement the stack and strings CARD(80), LINE(120) for input and output. The above commands are then imple- by the following DEFINEs:

```
DEFINE ';NEXTCH;' = ';INCR IC; CH := CARD(IC);' ;
DEFINE ';STACK;' = ';CATENATE CH ONTO OPSTK; NEXTCH;' ;
DEFINE ' TOP ' = 'OPSTK( LENGTH(OPSTK))' ;
DEFINE ';POP;' = ';DECR LENGTH(OPSTK);' ;
DEFINE ';UNSTACK;' = ';CATENATE TOP ONTO LINE;POP;' ;
```

and then the program is described in terms of these composite commands. Note that STACK employs the command NEXTCH and UNSTACK uses both TOP and POP. The clarity of the program would probably be enhanced by an additional command defined by:

```
DEFINE ';PASS_THRU;' = ';CATENATE CH ONTO LINE;NEXTCH;' ;
```

Program INPOST also uses the DEFINE facility to allow debug output to be incorporated into the program which will be erased or left intact, depending on a single DEFINE located at the top of the program. The keyword DEBUG is appended as prefix to any command which should only be generated when DEBUG mode is active and then the DEBUG mode is activated or deactivated by:

```
DEFINE ';DEBUG ' = '; '  ;  "ON"
```

or

```
DEFINE ';DEBUG #;' = ';' ;  "OFF"
```

The first DEFINE causes removal of all DEBUG prefixes and the second causes removal of all DEBUG-prefixed statements.

WARNINGS

FORTRAN demands that the symbolic names of variables, functions and subroutines be restricted to no more than 6 letters or digits. In order to use longer names (possibly with embedded special characters) in SKOL programs, one should simply use the desired name (carefully delimited by blanks!!) in all specifications and uses of the variable, function or subroutine, and then insert a DEFINE before the first occurrence of the desired name; the definition should call for replacement of the blank-embedded name by a blank-embedded valid FORTRAN symbolic name.

Example:

```
DEFINE ' $A_LONG_NAME ' = ' REAL01 '  ;
CONSTANT ARRAY_MAX_SIZE = 120 ;

REAL $A_LONG_NAME ( ARRAY_MAX_SIZE ),X;

X := $A_LONG_NAME (J)+2.5 ;
FOR I = 1 TO ARRAY_MAX_SIZE -1 :  ...  ENDFOR;
```

Notice that the constant ARRAY_MAX_SIZE does not require a DEFINE (FORTRAN will only see 120) but all occurrences must be blank-embedded just like the variables, functions and subroutines.

The need to embed certain names in blanks is an embarassment caused by the lack of a lexical scanner in the MORTRAN translator. Given this annoyance, the best solution is probably to systematically embed all names in blanks. It is difficult to know exactly when the blanks are needed in the source program since occasionally the SKOL precompiler will put them in. For example,

```
FOR I = 1 TO ARRAY_MAX_SIZE:
```

will work but

```
FOR I = 1 TO ARRAY_MAX_SIZE+1:
```

will not work!

A run-time error message indicating a "collision in character mapping" means either a duplicate occurrence of the same character constant in the user-defined CHAR data-type or else that the character conversion scheme implemented by the SKOL subprograms in Appendix D does not work for the given machine/character code/FORTRAN combination being used.

Users are reminded to end all programs and all files included by %Ud cards by a terminator control card with %% in the first two columns.

All SKOL statements are terminated by a semicolon and the programmer is strongly advised to carefully make sure that no semicolons are missing. The diagnostic capabilities of the SKOL precompiler are quite good so long as the program isn't missing statement terminators. Especially beware omission of the terminating semicolon for a DEFINE statement!

The one place where semicolons are <u>not</u> wanted but might be mistakenly placed is after the END terminating a CASE group in a scalar or situation CASE statement.

A Macro and FUNCTION for String Equality

It is very convenient in some applications to be able to test for equality between two variable-length strings with a notation like

EQUAL (STR1, STR2)

which can be embedded in logical expressions.

To this purpose we propose the following macro DEFINE and supporting LOCIGAL FUNCTION:

```
DEFINE' EQUAL(#,#)' = 'EQU999( #1 ,
   SIZE(#1), LENGTH(#1), #2 , SIZE(#2), LENGTH(#2))' ;
FUNCTION EQU999(STR1,S1,L1,STR2,S2,L2)LOGICAL:
   "TEST EQUALITY OF TWO STRINGS"
   INTEGER S1,S2,L1,L2,I;
   CHAR STR1(S1), STR2(S2);

   UNTIL ALL_SAME OR MISMATCH:
      IF L1¬= L2 : MISMATCH; ENDIF;
      FOR I = 1 TO L1 :
         IF STR1(I)¬ = STR2(I) : MISMATCH; ENDIF;
      ENDFOR;
      ALL_SAME;
   THENCASE:
      ALL_SAME : BEGIN EQU999 := TRUE ; END
      MISMATCH : BEGIN EQU999 := FALSE ; END
   ENDUNTIL;
   RETURN;
ENDFUNCTION;
```

Each subprogram in which EQUAL is used must have the specification:

```
LOGICAL EQU999 ;
```

REFERENCES

[1] N. Wirth, "The Programming Language PASCAL," Acta Informatica, 1, 35-63 (1971).

[2] K. Jensen and N. Wirth, "PASCAL: User Manual and Report," Vol. 18 of Lecture Notes in Computer Science, edited by G. Goos and J. Hartmains, Springer-Verlag, Berlin-Heidelberg-New York, 1974.

[3] M.E. Conway, "Design of a Separable Transition-Diagram Compiler," CACM Vol. 6, No. 7, (1963), 396-408.

[4] O.J. Dahl and C.A.R. Hoare, "Hierarchical Program Structures" in Structured Programming by Dahl, Dijkstra and Hoare, Academic Press, New York, 1972.

[5] W.T. Hardgrave, "Positional versus Keyword Parameter Communication in Programming Languages," SIGPLAN Notices, Vol. 11, No. 5, May 1976, pp 52-58.

[6] D.E. Knuth, "Structured Programming with GOTO Statements," Computing Surveys, Vol. 6, No. 5, (1974), 261-301.

[7] C.T. Zahn, "A Control Statement for Natural Top-Down Structured Programming," in Programming Symposium: Proceedings, Colloque sur la Programmation, edited by B. Robinet, Springer-Verlag, Berlin (1974), 170-180.

[8] D.E. Knuth and C.T. Zahn, "Ill-chosen use of Event," CACM, Vol. 18, No. 6, (June 1975), 360.

[9] A.J. Cook and L.J. Shustek, "MORTRAN2, A Macro-based Structured FORTRAN Extension," Conference Digest of Spring 1975 IEEE COMPCON.

[10] A.J. Cook, "Experience with Extensible, Portable FORTRAN Extensions," SIGPLAN Notices (summer 1976).

[11] A.J. Cook and L.J. Shustek, "A User's Guide to MORTRAN2," Computation Group Technical Memo No. 165, available from Computation Research Group (Bin 88), SLAC, Stanford, Ca. 94305, U.S.A.

[12] C.T. Zahn, "A User Manual for the MORTRAN2 Macro-translator," Computation Group Technical Memo No. 167 (see [11]).

[13] C.A.R. Hoare, "Notes on Data Structuring," in Structured Programming (see [4]).

[14] N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[15] P. Henderson and R. Snowdon, "An Experiment in Structured Programming," BIT (European), Vol. 12, 1972, pp 38-53.

APPENDIX A

FORMAL SYNTAX OF SKOL

```
Aexpr = { sign }^0,1  {{{ Aprimary }^≥1_'**' }^≥1_['*' | '/'] }^≥1_['+' | '-']

Aprimary = [ Aconst | '(' Aexpr ')' | Avar | Afunction '(' { arg }^≥1_, ')' ]

action = [ initialization | assignment | invocation | interruption | selection |
           repetition | input-output | augmentation | allocation | focussing ]

allocation = [ 'NEW' | 'FREE'] reference ';'

arg = [ expr | array-name | function | subroutine ]

array-bounds = '(' { +Iconst }^≥1_, ')'

assignment =
      [ variable ':=' expr ';' | '#' string ':=' string-expr ';' |
        'REPLACE' [ string-context | string ] 'BY'
           ['NULL' | char-expr | string-context ] ';' |
        'DELETE' [ string | string-context ] ';' |
        'INSERT' [ char-expr | string-context ]
           [ 'BEFORE' | 'AFTER' ] string '(' index ')' ';' ]

augmentation =
      [ 'CATENATE' string-expr 'ONTO' string ';' |
           [ 'INCR' | 'DECR' ] variable { 'BY' expr }^0,1 ';' ]
```

```
block = { command }^≥0

Cconst = '(' {{ sign }^0,1 Rconst }^2_,  ')'

carriage-control =
   [ '$PAGE' | '$SKIP' | '$SKIP2' | '$OVER' ]

char-const = '''' [ non-'-char | '''''' ] ''''

char-expr = [ char-const | char-var ]

command =
   [ action | definition | format-declaration | pragmat ]

common-declaration = 'COMMON' '/' Fident '/'
   { Fident { array-bounds }^0,1 }^≥1_,  ';'

control = [ { '/' }^≥1 | { +Iconst }^0,1 'X' ]

data = '/' { [ { '-' }^0,1 Aconst | Lconst | char-const | 'NIL' ] }^≥1_,  '/'

data-format =
   [ { +Iconst }^0,1 [ 'I' | 'L' | 'A' ] +Iconst |
     { sign }^0,1 {Iconst 'P' }^0,1 { +Iconst }^0,1
        [ 'F' | 'G' | 'E' | 'D' ] +Iconst '.' Iconst ]
```

```
definition =
   [ 'CONSTANT' { ident '=' value }1,≥1 ';' |
     'DEFINE' '''' pattern '''' '='
        '''' replacement '''' ';' |
     'MACRO' ident '(' { ident { '=' Xexpr }0,1 }1,≥1 ')'
        '=' '''' text '''' ';' ]

exponent = 'E' { sign }0,1 { digit }1,2

expr = [ Lexpr | Aexpr | char-expr | 'NIL' ]

Fident = letter { [ letter | digit ] }0...5

Fident4 = letter { [ letter | digit ] }0...3

Ftype = [ 'REAL' | 'INTEGER' | 'LOGICAL' | 'COMPLEX' ]

field-group = [ 'REF' | Ftype | 'CHAR' ] ':'
  { ident { array-bounds }0,1 }1,≥1

focussing = 'WITH' reference ':' block 'ENDWITH' ';'

format-declaration = 'FORMAT' ident '=' '(' format-list ')' ';'

format-list = { [ control | data-format | output-text |
   +Iconst '(' format-list ')' ] }1,≥1
```

```
Iconst = { digit }^≥1

ident = [ letter | special-symbol ]
   { [ letter | special-symbol | digit ] }^≥0

index = [ +Iexpr | scalar-expr ]

initialization =
    [ 'CHAR_SETUP' ';' |
       'MAKEAVAIL' class ';' |
       'START' process 'AT' coroutine ';' ]

input-output =
    [ 'INPUT' '(' { [ ':' control [ variable ':' data-format ] }^≥1_, ')' ';' |
      'OUTPUT' '(' { [ carriage-control | ':' control | output-item ] }^≥1_, ')' ';'
    [ 'READ' | 'WRITE' ] '(' file ',' format ')' { variable }^≥0_, ';' |
    [ 'ENDFILE' | 'REWIND' | 'BACKSPACE' ] file ';' |
    [ 'READ' | 'WRITE' ] 'STRING' { '(' file ')' }^0,1
       string { '(' index '...' index ')' }^0,1 ';' ]

interruption =
    [ 'STOP' ';' | 'PAUSE' ';' | 'RETURN' ';' |
      'SUSPEND' process ';' | situation ';' ]
```

```
invocation =
    [ 'CALL' subroutine { '(' { arg }1,≥1 ')' }0,1 ';' |
      'EXECUTE' routine { '(' { Iexpr }1,≥1 ')' }0,1 ';' |
      'RESUME' process ';' |
      'RESUME' coroutine 'FROM' coroutine ';' |
       macro '(' { keyword '=' Xexpr }1,≥1 ')' ';' ]

Lconst = [ 'TRUE' | 'FALSE' ]

Lexpr = {{ 'NOT' }0,1 Lprimary }['OR'|'AND']≥1

Lprimary = [ Lconst | '(' Lexpr ')' | Lvar |
    Lfunction '(' { arg }1,≥1 ')' | { Aexpr }relop2 |
    'IN_' scalar '(' scalar-var ')' ]

label = [ scalar-const | scalar ]

length = non-negative-Iexpr

list-of-subtypes = [ empty | '(' { subtype }1,≥1 ')' ]

output-item = [ output-text |
    variable { ':' { [ data-format | 'C' ] }0,1 }0,1 ]

output-text = '''' { [ non-'-symbol | '''''' ] }≥1 ''''

parameters = '(' { Fident }1,≥1 ')'
```

```
pragmat =
    [ 'TRACE' { variable }(≥1, ',') ';' |
      'RUNCHECK' '(' {{ '-' }(0,1) [ 'ALL' | 'FOR' |
         'CASE' | 'UNTIL' | 'TRACE' ] }(≥1, ',') ')' ';' ]

process-declaration =
    'PROCESS' ident '=' '(' { ident }(≥1, ',') ')' ';'

program = { program-segment }(≥1) terminator-line

program-segment =
    [ 'MAIN' ':' segment-body 'ENDMAIN' ';' |
      'SUBROUTINE' Fident { parameters }(0,1) ':'
          segment-body 'ENDSUBROUTINE' ';' |
      'FUNCTION' Fident parameters Ftype ':'
          segment-body 'ENDFUNCTION' ';' |
      'BLOCKDATA' ':' { specification }(≥1) 'ENDBLOCKDATA' ';' ]

Rconst =
    [ { digit }(≥1) '.' { digit }(≥0) { exponent }(0,1) |
      { digit}(≥1) exponent | '.' { digit }(≥1) { exponent }(0,1) ]

record-class-declaration =
    'RECORD' 'CLASS' '(' +Iconst ')' 'OF' Fident4 ':'
       { [ 'REF' | Ftype | 'CHAR' ] ':'
            { ident { array-bounds }(0,1) }(≥1, ',') ';'
       }(≥1) 'ENDRECORD' ';'
```

```
recursion-declaration = 'RECUR' '(' +Iconst ')' ';'
    { ident { '(' { '*' }≥1 , ')' }0,1 }≥1 , ';'

relop = [ '=' | '¬=' | '<' | '>' | '<=' | '>=' ]

repetition =
    [ 'REPEAT' { Iexpr 'TIMES' }0,1 ':' block 'ENDREPEAT' ';' |
      'LOOP' ':' block 'WHILE' Lexpr ':' block 'ENDLOOP' ';' |
      'FOR' Ivar '=' Iexpr 'BY' Iexpr ':' block 'ENDFOR' ';' |
      'FOR' Ivar '=' Iexpr { 'BY' Iexpr }0,1
          'TO' Iexpr ':' block 'ENDFOR' ';' |
      'LINK' reference '=' reference 'BY' field ':'
          block 'ENDLINK' ';' |
      'DO' simpleIvar '=' simpleIexpr 'TO' simpleIexpr ':'
          block 'ENDDO' ';' ]

routine-definition =
    [ 'ROUTINE' ident { '(' { ident }≥1 , ')' }0,1
       { 'LOCAL' '(' { ident }≥1 , ')' }0,1 ':'
       block 'ENDROUTINE' ';' |
      'COROUTINE' ident ':' block 'ENDCOROUTINE' ';' ]

scalar-type-definition =
    'TYPE' ident '=' list-of-subtypes ';'

segment-body = { specification }≥0 { statement-function }≥0
    { command }≥1 { routine-definition }≥0
```

```
selection =
    [ 'IF' { Lexpr ':' block }(≥1, 'ELSEIF'
          { 'ELSE' ':' block }(0,1) 'ENDIF' ';' |
      'UNTIL' { ident }(≥2, 'OR' ':' block 'THENCASE' ':'
          {{ situation }(≥1, ',' ':' 'BEGIN' block 'END' }(≥1 'ENDUNTIL' ';' |
    'UNTIL' { 'GLOBAL' }(0,1) ident ':' block 'ENDUNTIL' ';' |
    'CASE' scalar-var ':' scalar 'OF'
          {{ label }(≥1, ',' ':' 'BEGIN' block 'END' }(≥1
          { 'ELSE' ':' 'BEGIN' block 'END' }(0,1) 'ENDCASE' ';' ]

sign = [ '+' | '-' ]

simpleIexpr = [ +Iconst | simpleIvar ]

special-symbol =
    [ '$' | symbol-not-in-FORTRAN-set-and-not-:;@"|&¬ ]

specification =
    [ definition | pragmat | format-declaration |
      scalar-type-definition | record-class-declaration |
      variable-declaration | process-declaration | recursion-declaration |
      common-declaration | 'CHAR_COMMON' ';' |
      'EXTERNAL' { Fident }(≥1, ',' ';' ]

statement-function =
    Fident '(' { Fident }(≥1, ',' ')' '=' Aexpr ';'
```

```
string-const = '''' { [ non-'-char | '''''' ] }^≥0 ''''

string-context = string '('
     [ index { '...' index }^0,1 | index '...' '|' length |
        length '|' '...' index ] ')'

string-expr =
     { [ char-expr | string-const | string | string-context ] }^≥1_'&'

subscripts = '(' { Iexpr }^≥1_',' ')'

subtype = [ ident | char-const | ident '=' list-of-subtypes ]

terminator-line = line-with-%%-in-first-two-columns

text = { [ non-'@#-symbol | '''''' | '@@' | '##' ] }^≥0

value = [ Aconst | '(' '-' Aconst ')' | Lconst | char-const | 'NIL' ]

variable = [ Fident { subscripts }^0,1 |
     '@' '(' reference '.' field ')' | '@' '.' field ' ' ]

variable-declaration =
     [ 'STRING' { Fident '(' +Iconst ')' }^≥1_',' ';' |
        [ Ftype | scalar | 'REF' 'TO' class ':' ]
           { Fident { array-bounds }^0,1 { data }^0,1 }^≥1_',' ';' ]

Xexpr = text-without-semicolons
```

APPENDIX B

SYNTAX FLOW GRAPHS FOR SKOL

The following pages contain syntax flow graphs for a large part of the SKOL language. Those syntactic categories whose definition seemed to be readily understandable from the linear notation of Appendix A have been omitted.

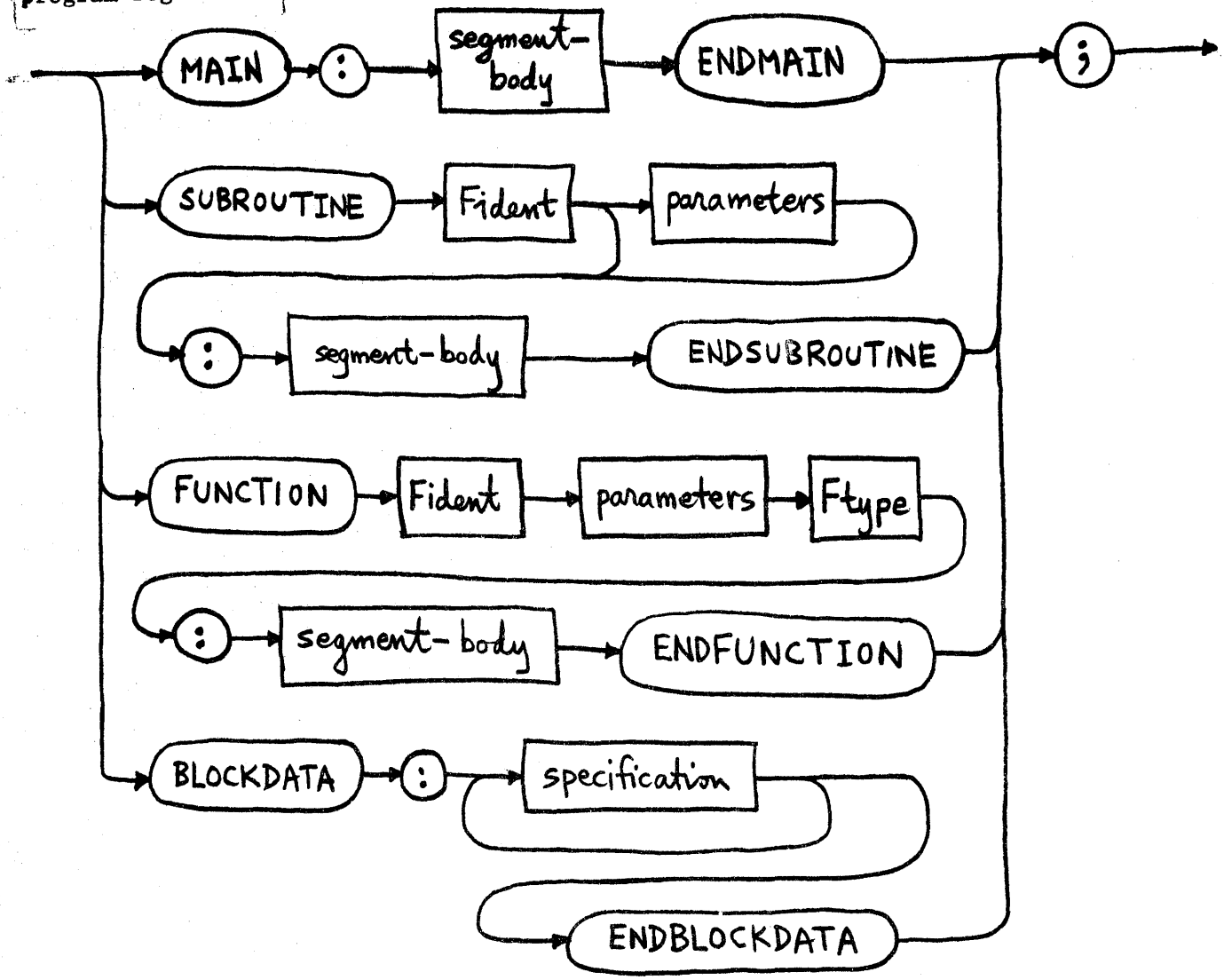
Notes:

1) Scalar, class, reference, field, subroutine, routine, process, coroutine, macro, keyword, situation, string, file, integer and format must be identifiers for objects of the type described by the word.

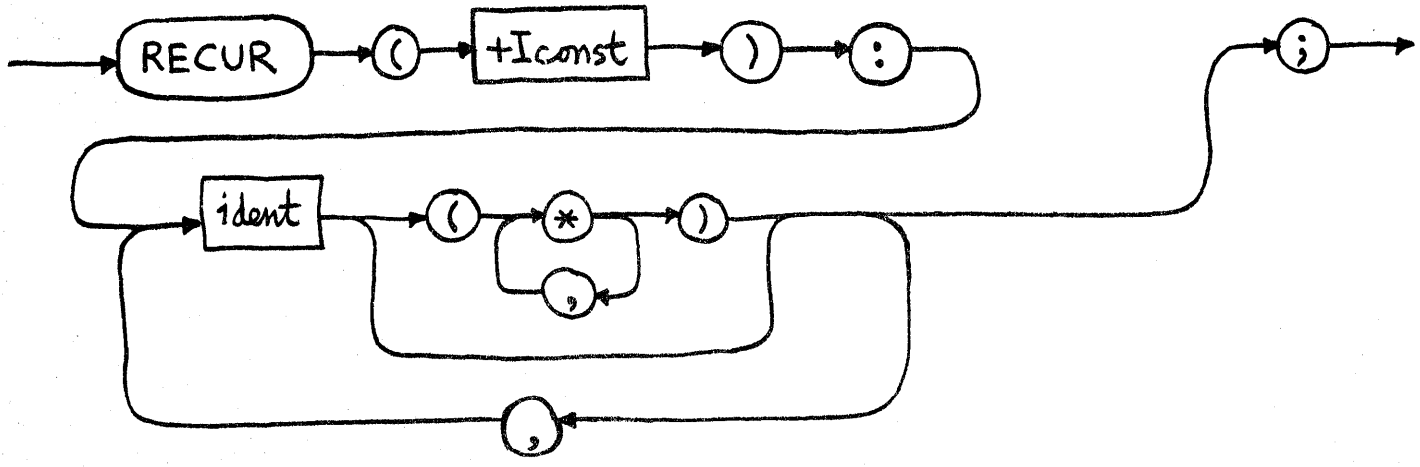
2) Capital letters, I, R, L, C, A are codes for integer, real, logical, complex, arithmetic where the latter includes integer, real and complex. The character '+' means strictly positive.

3) ?var, ?expr, ?function, ?const mean, respectively, variable, expression, function, constant of type ?, where ? is one of the built-in or programmer declared types.

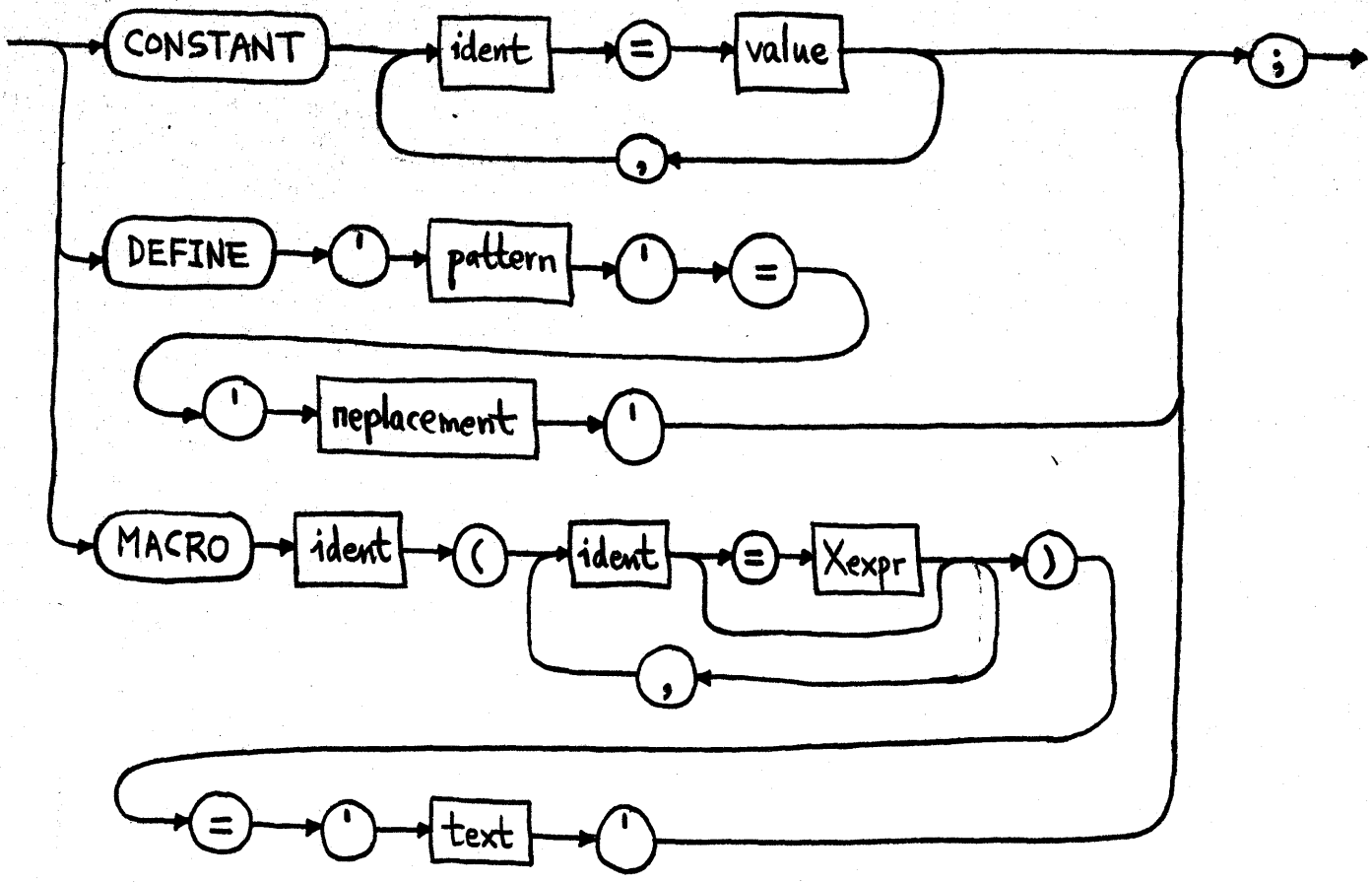
program-segment



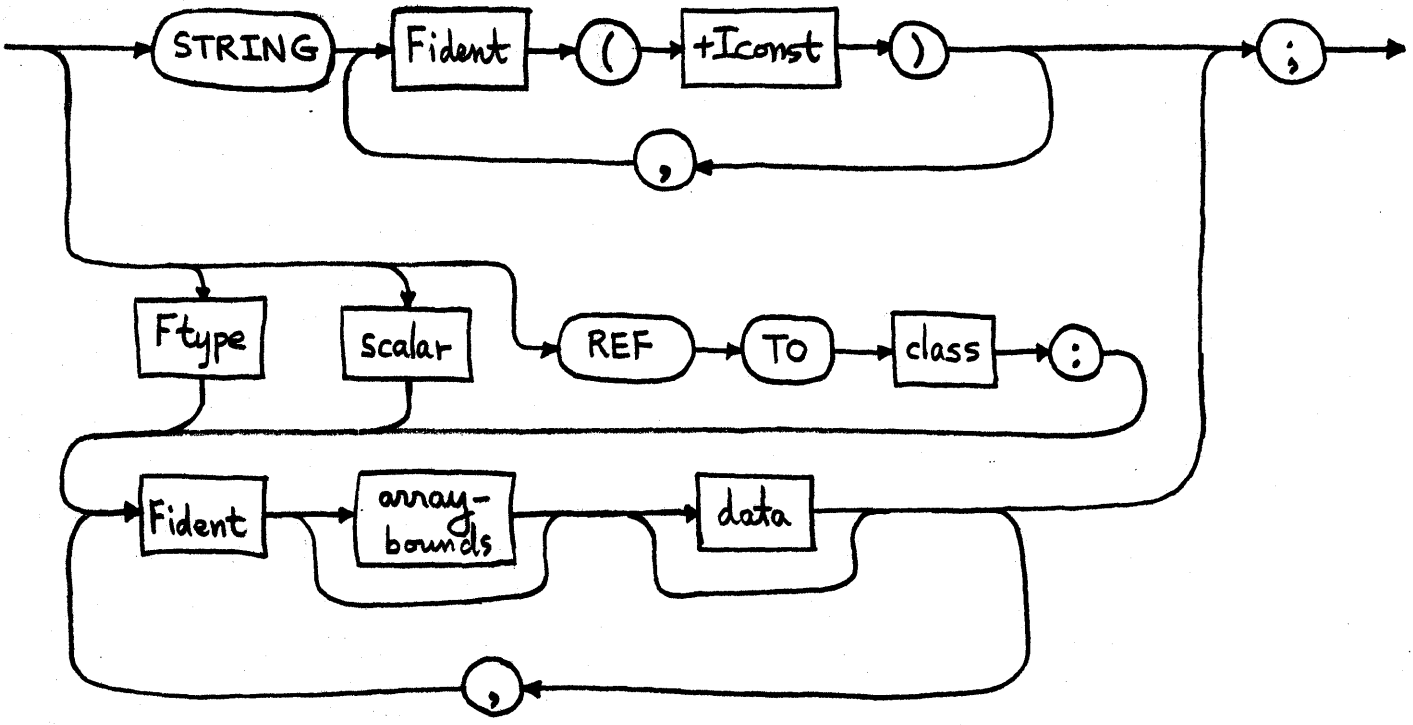
recursion-declaration



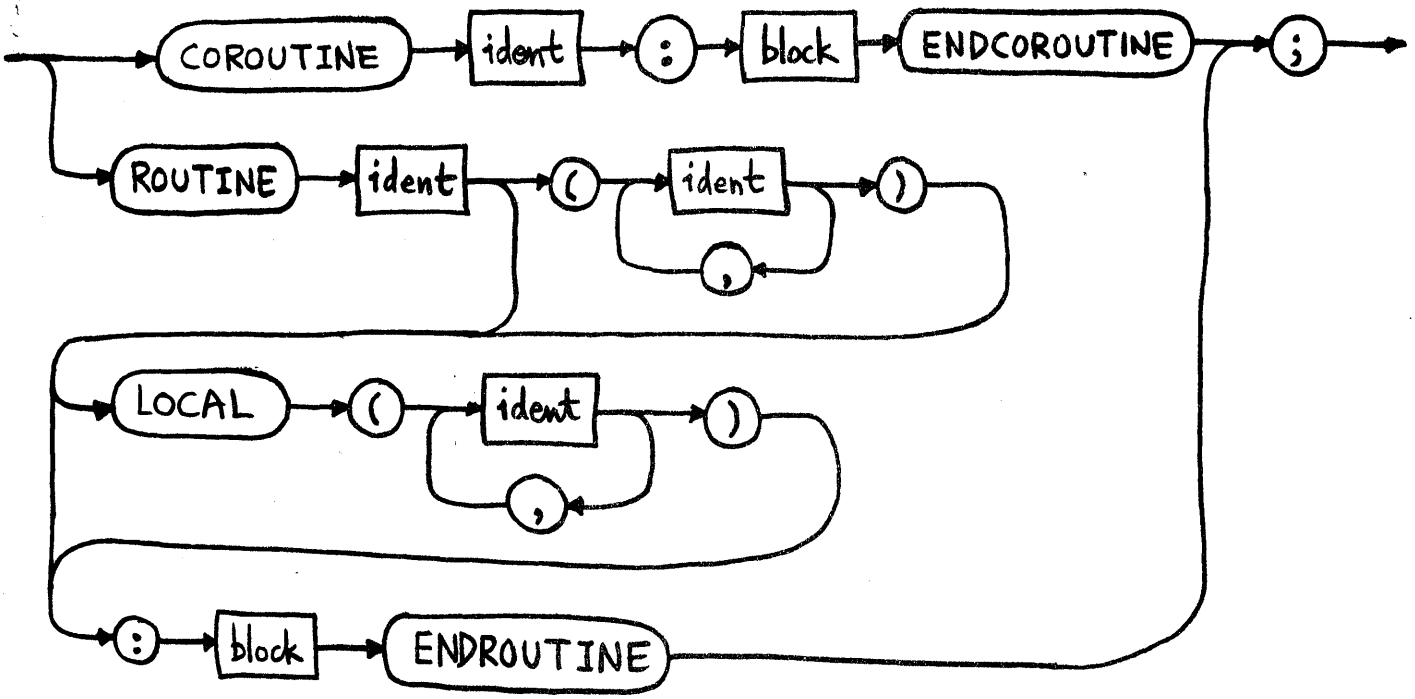
definition



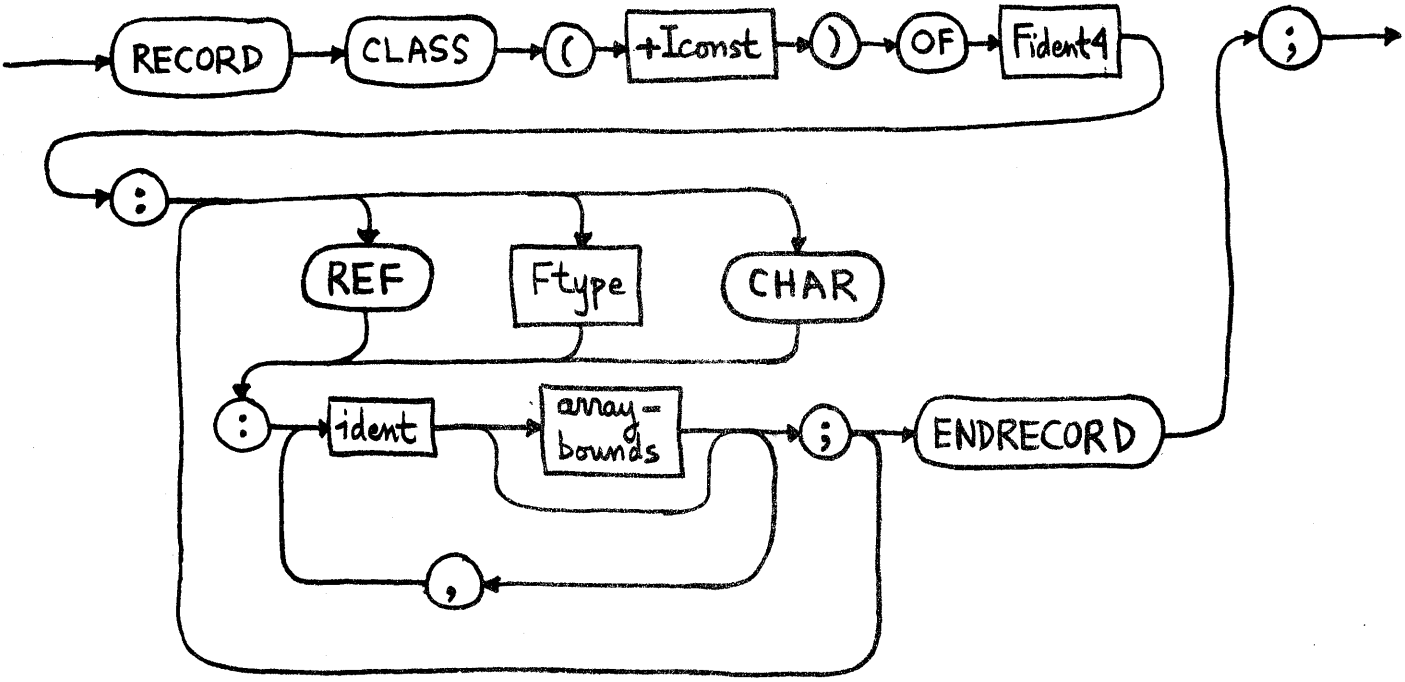
variable-declaration



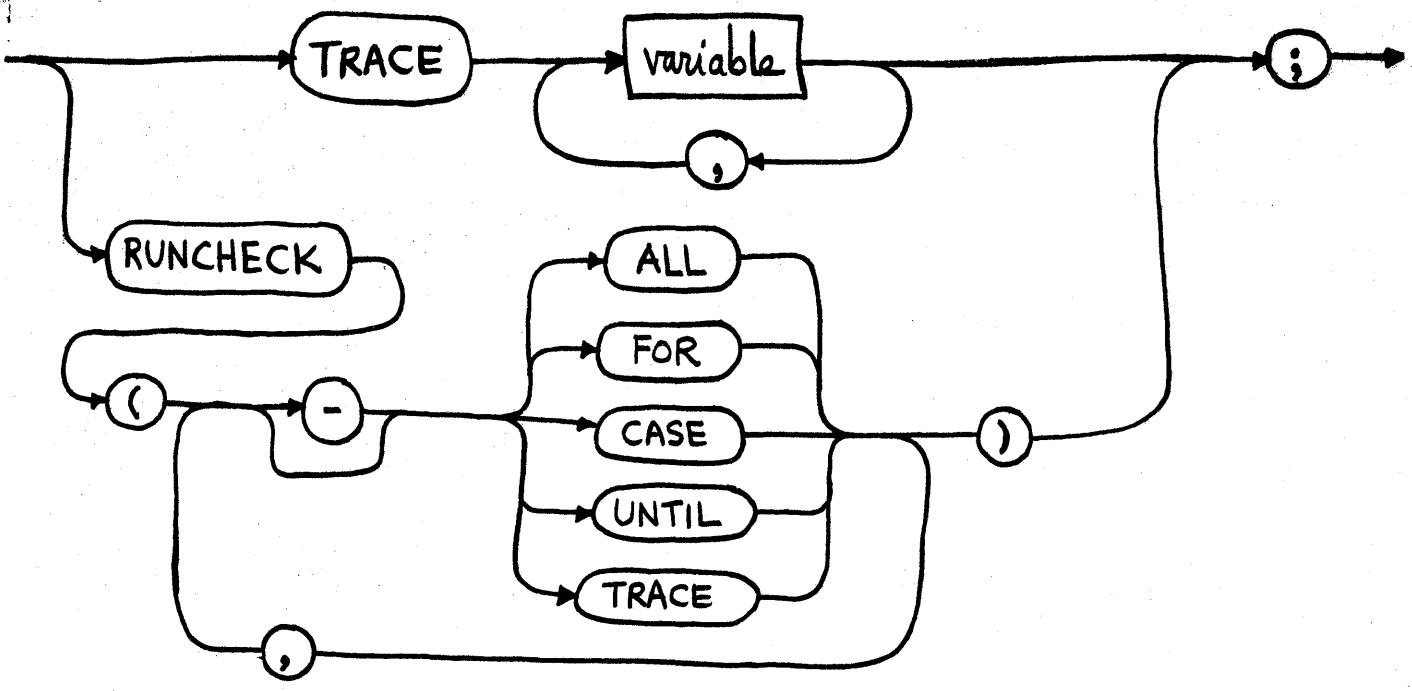
routine-definition



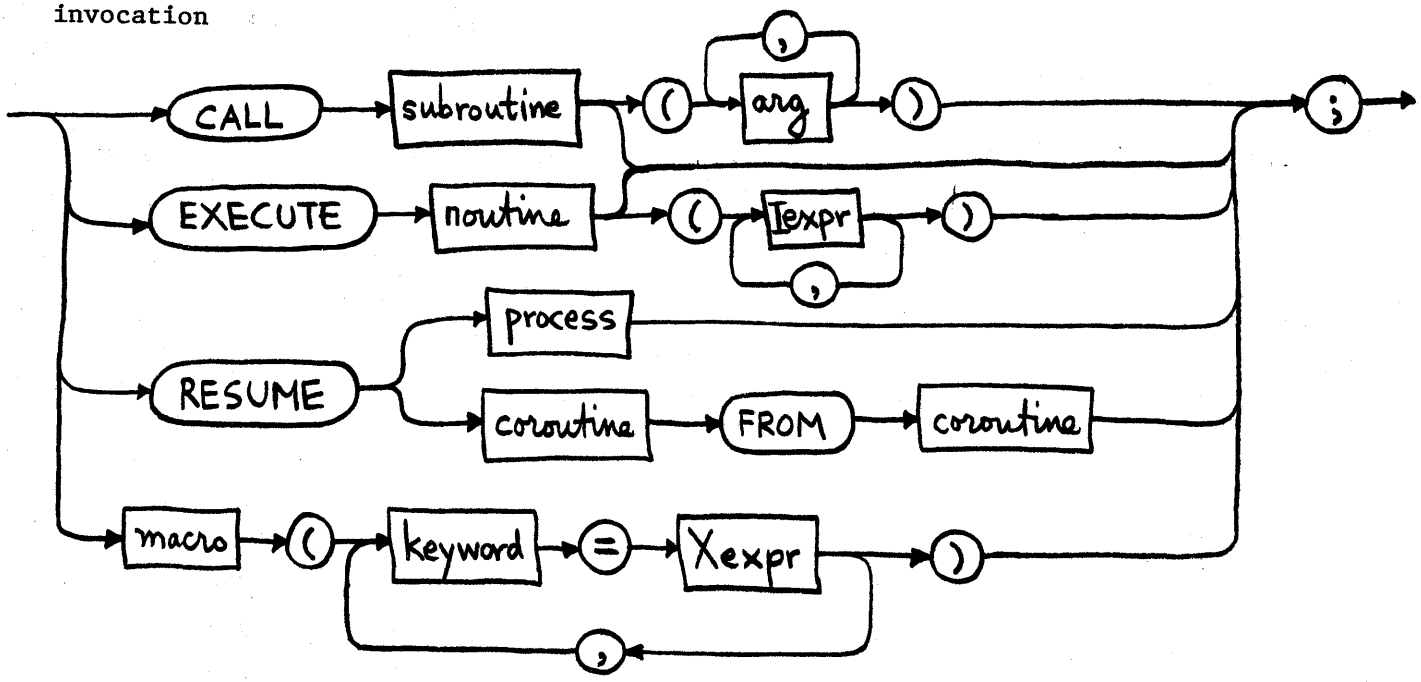
record-class-declaration



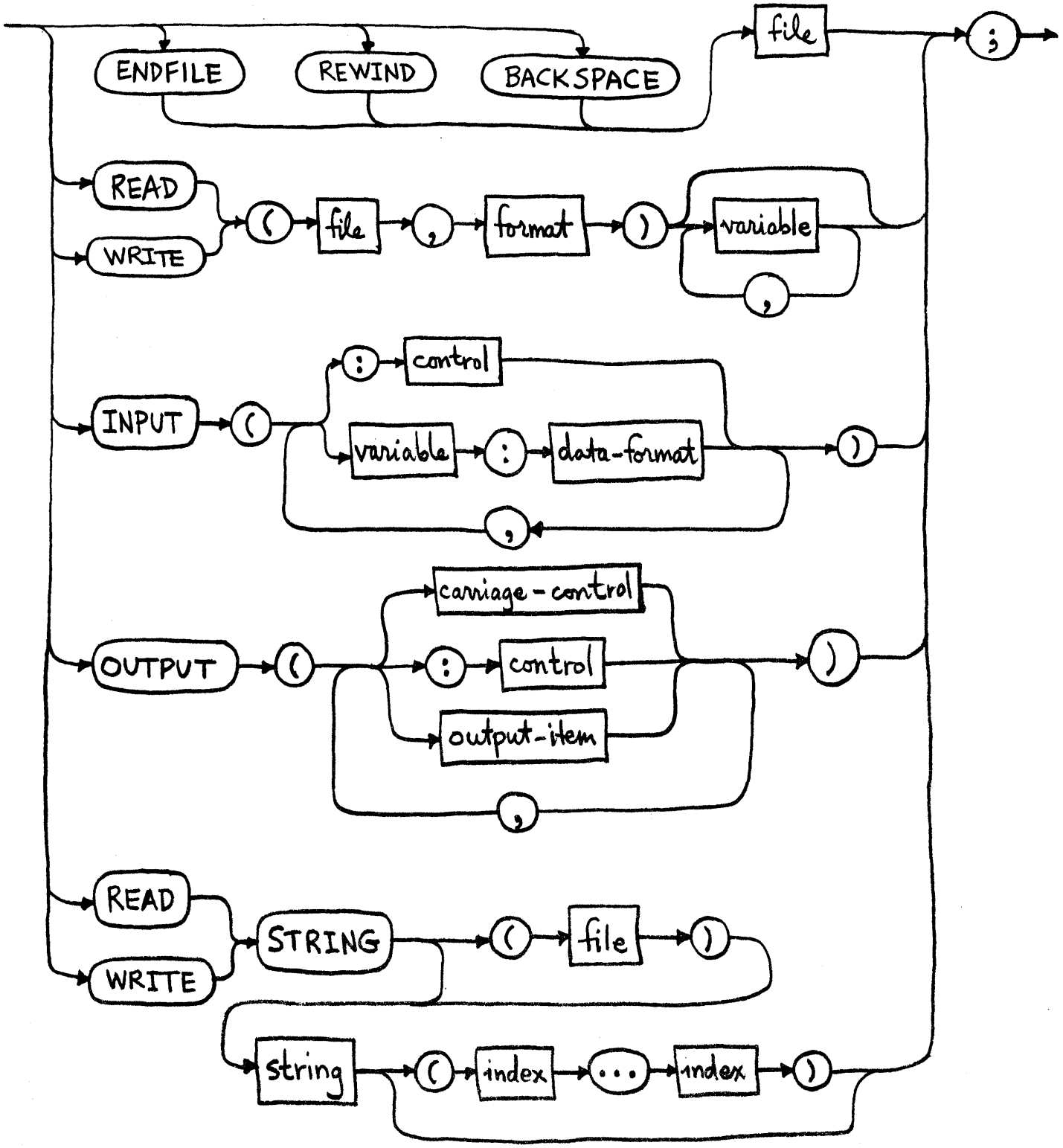
pragmat



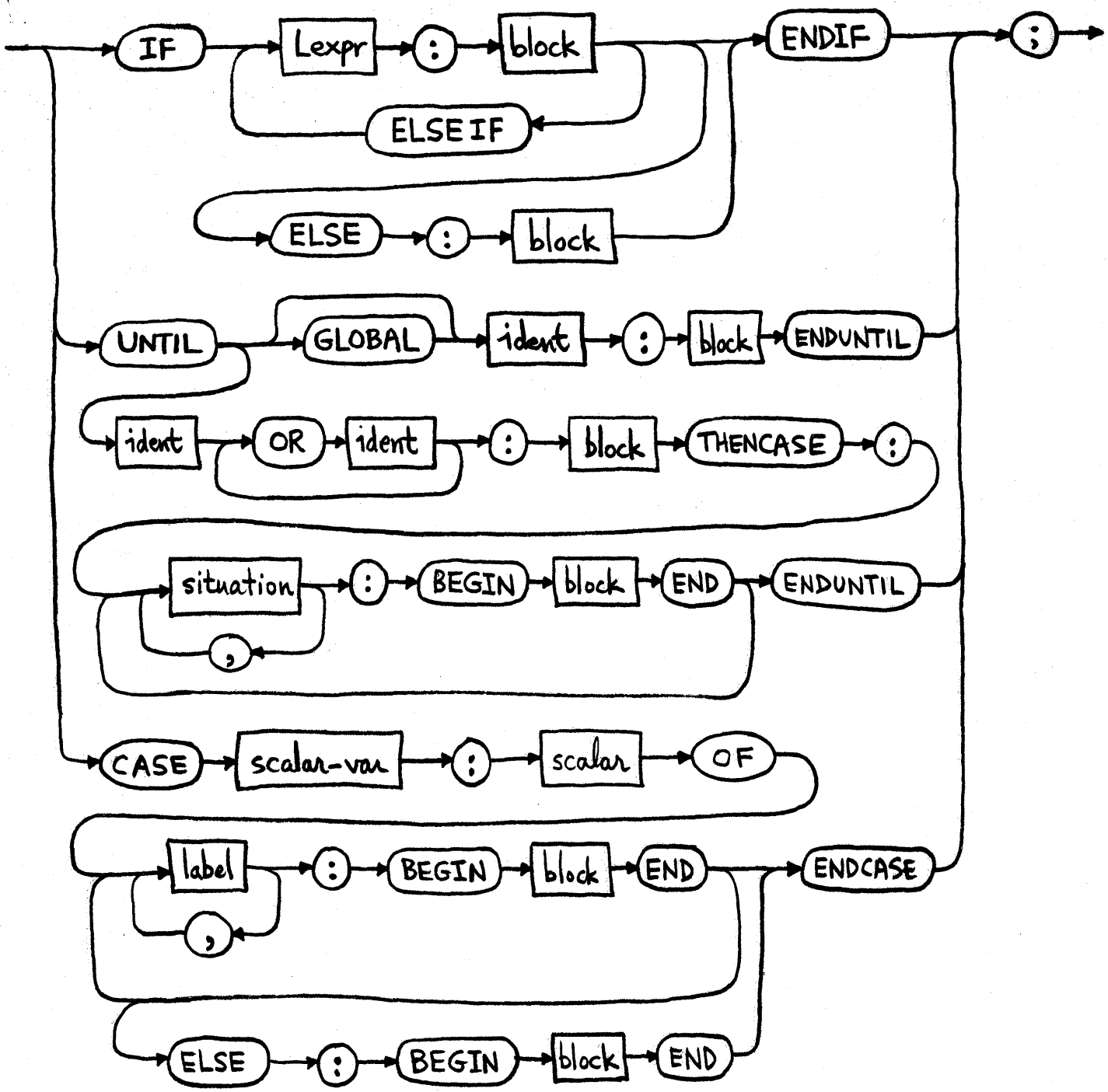
invocation



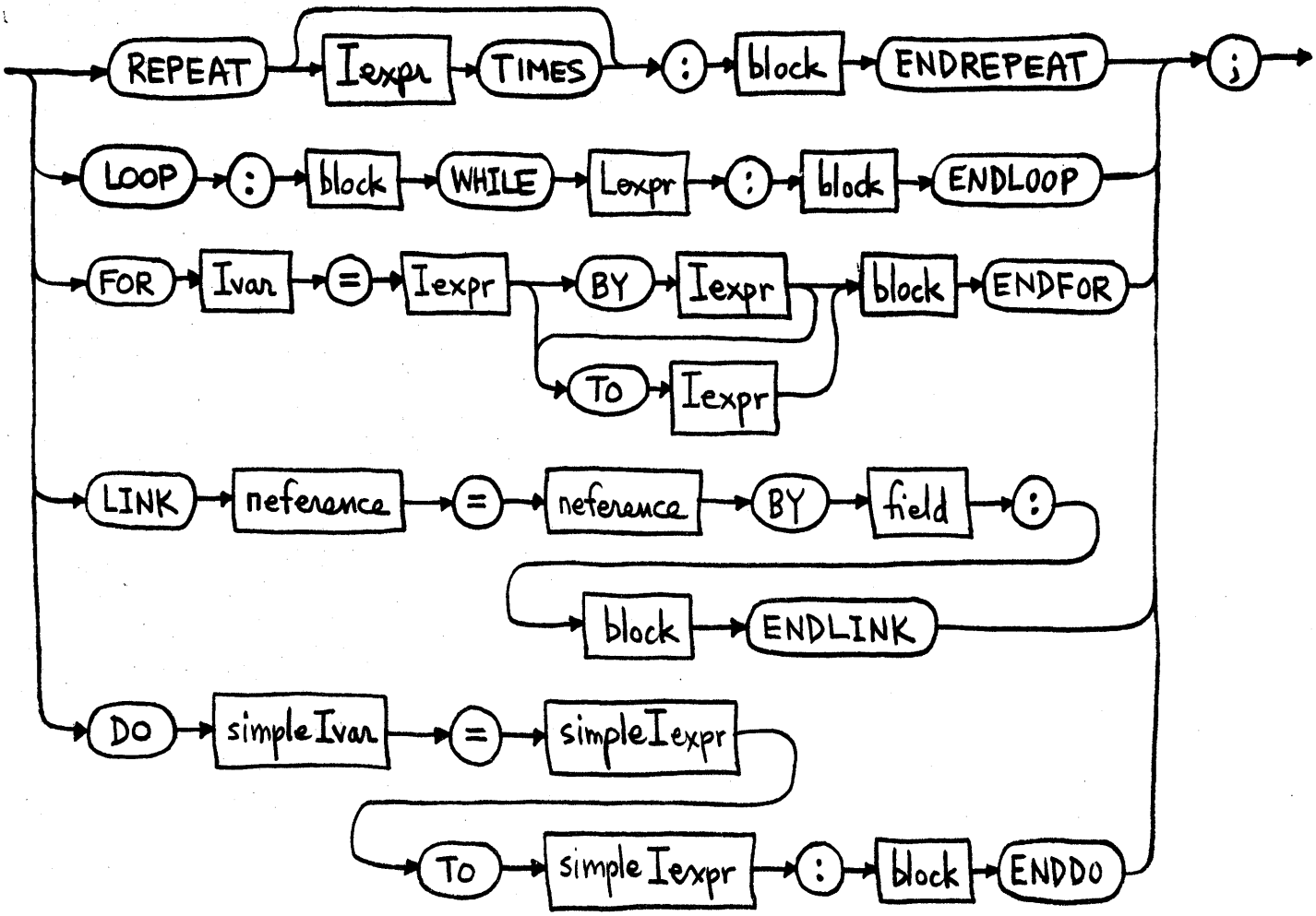
input-output



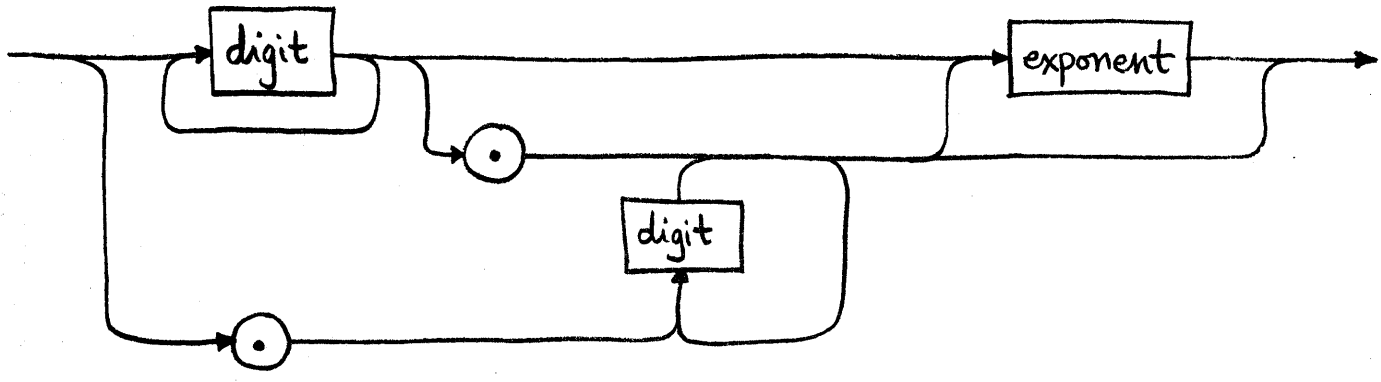
selection


IF
Lexpr
:
block
ELSE IF
ELSE
:
block
ENDIF
;
UNTIL
GLOBAL
ident
:
block
ENDUNTIL
ident
OR
ident
:
block
THENCASE
:
situation
:
BEGIN
block
END
ENDUNTIL
,
CASE
scalar-var
:
scalar
OF
label
:
BEGIN
block
END
ENDCASE
,
ELSE
:
BEGIN
block
END

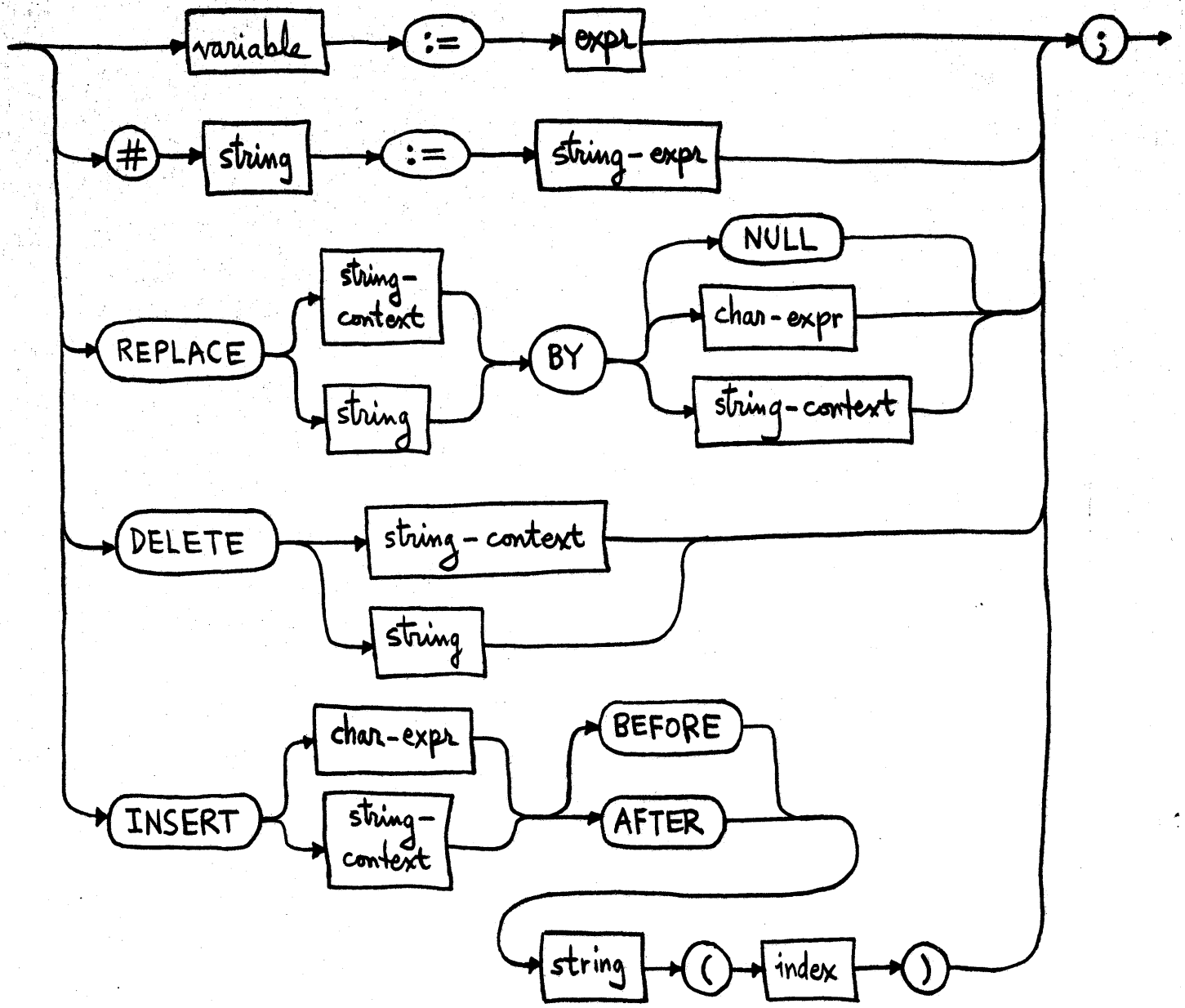
repetition



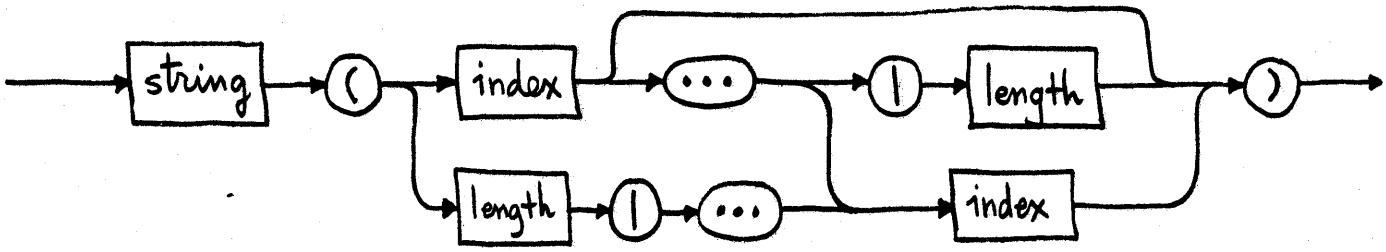
Rconst



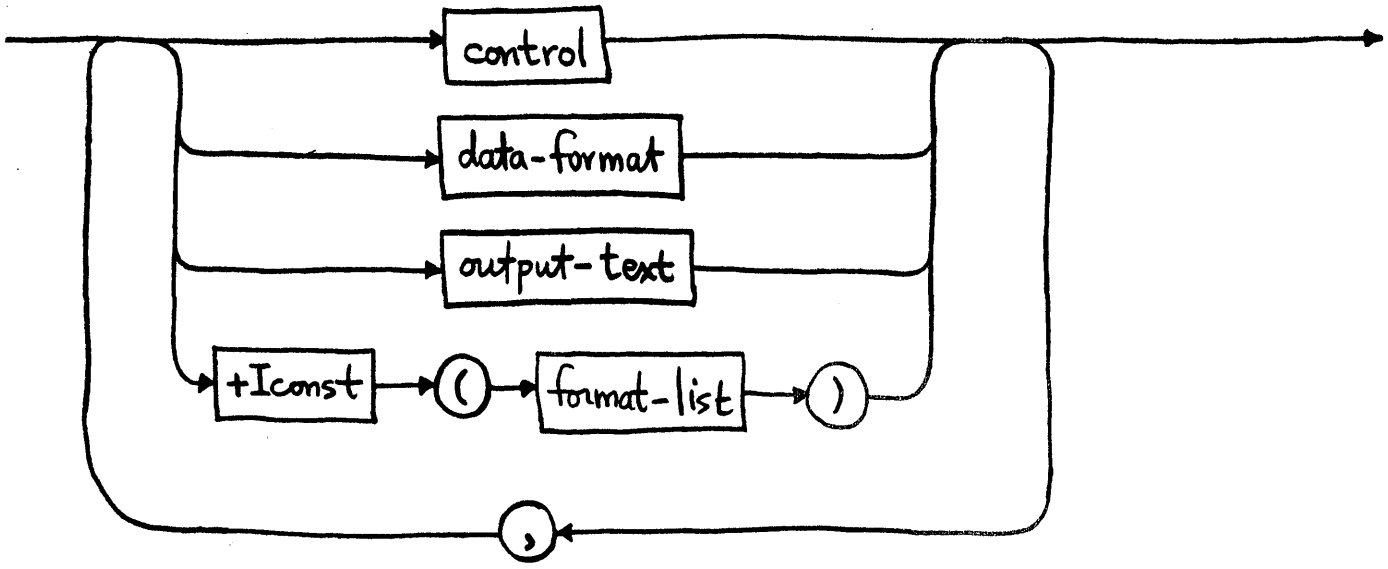
assignment



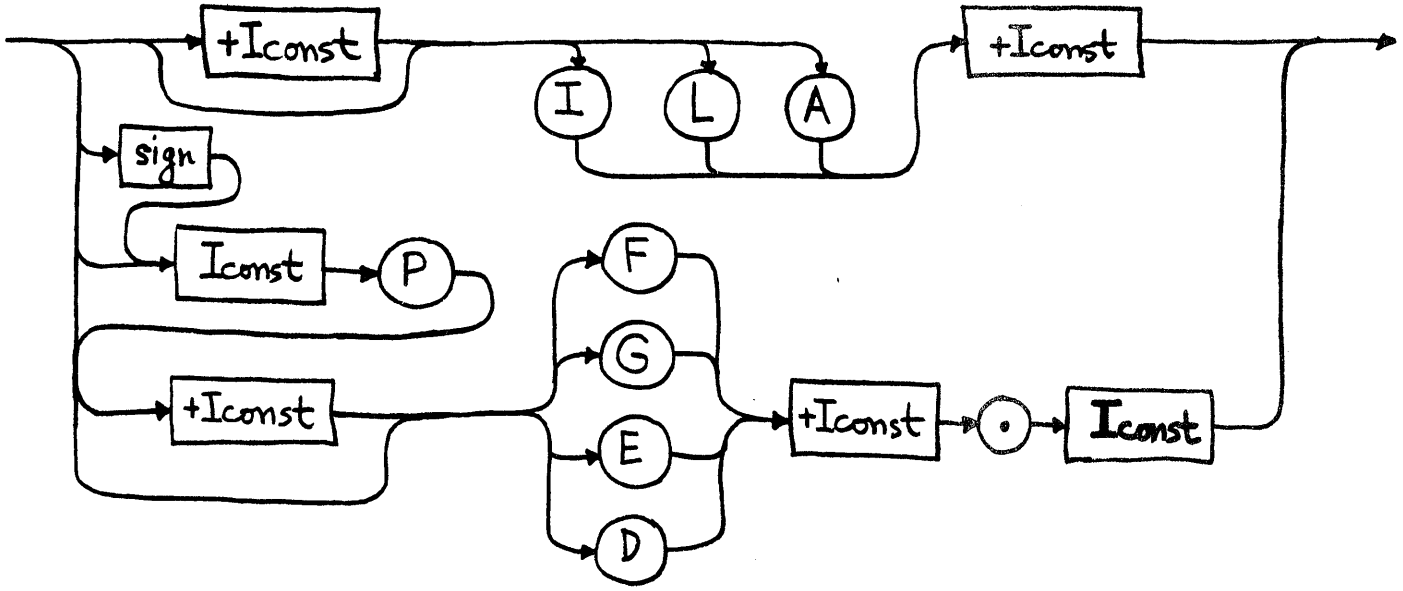
string-context



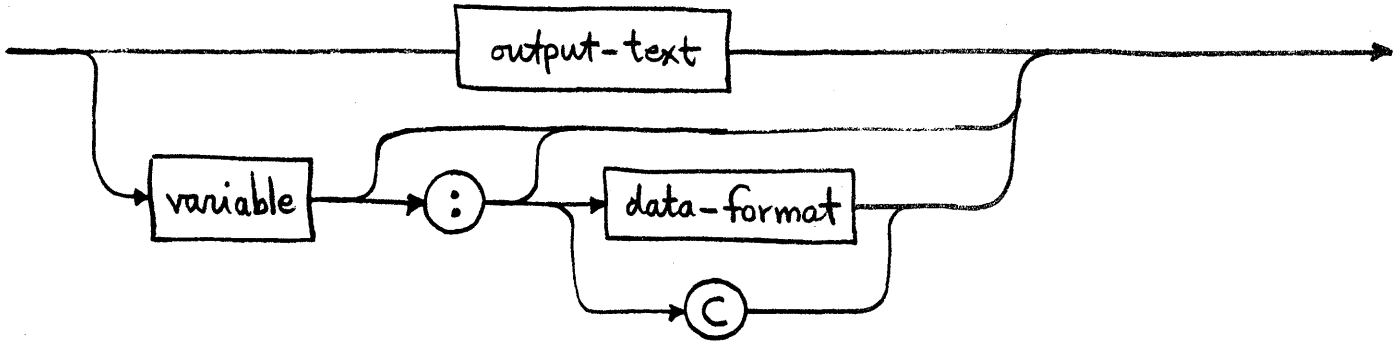
format-list



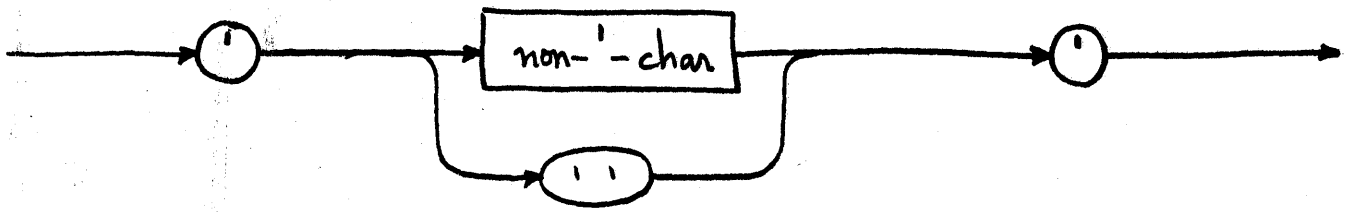
data-format



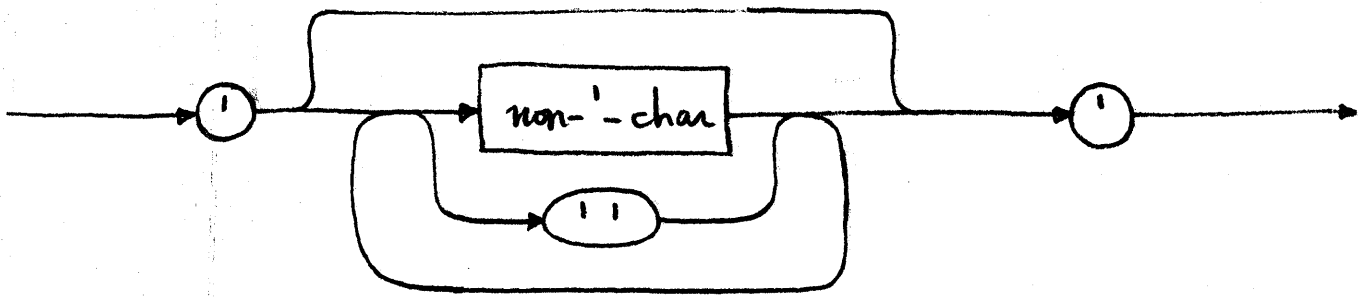
output-item



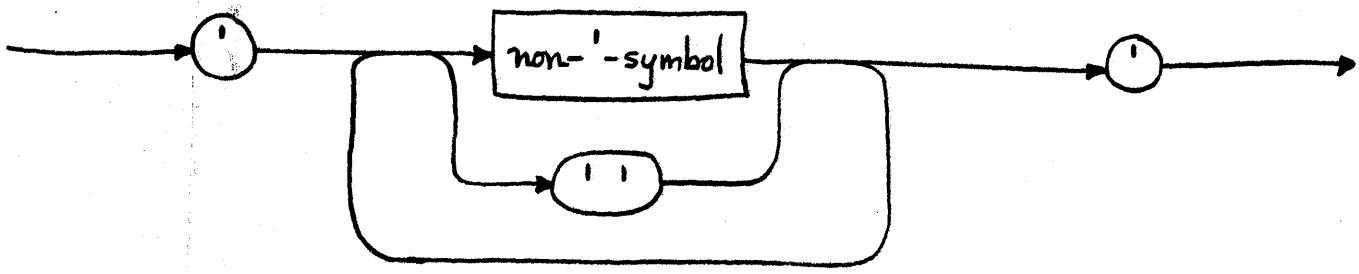
char-const



string-const



output-text



APPENDIX C

CONTROL COMMANDS FOR THE SKOL TRANSLATOR

The translator can be controlled by lines with % in column 1, according to the following command formats in which no embedded blanks are allowed:

Command	Action
%%	Resume reading input characters from the previous file used for input. If current file was the first, then no more characters are read.
%Ui or %Uij	Start reading input from device given by decimal digit i or digits ij.
%F	Assume subsequent text is FORTRAN.
%M	Resume reading SKOL input.
%Cij	Read subsequent input from columns 1 through n where n is given by the two decimal digits ij.
%A0	SKOL source text is not output as FORTRAN comments.
%A1	SKOL source text is output as FORTRAN comments.
%A2	Each line of SKOL text is listed on two FORTRAN comment cards using columns 41-80.
%Q0	SKOL comments must be between pairs of ".
%Q1	SKOL comments are terminated by an end-of-line unless closed by " on the same line.
%L	Begin listing lines of SKOL text.
%N	Stop listing SKOL text.
%E	Start new page in SKOL listing.

Default modes are %U1, %M, %C80, %A0, %Q0, %N

APPENDIX D

CHARACTER STRING UTILITY PROGRAMS

The following seven subprograms are required for character and string manipulations. They will usually be installed on a file so that their inclusion into the user program requires only one control card. The declaration for the CHAR data type must precede the inclusion of character utilities and both must lie outside other program segments (see examples in Appendix E).

```
"ROUTINES FOR BASIC CHARACTER SETUP AND INPUT/OUTPUT"
"    ALSO STRING REPLACEMENT                                  "
"THESE ROUTINES SHOULD WORK IF:                               "
"    1--- MACHINE INTEGER ARITHMETIC IS 1-COMPLEMENT,         "
"         2-COMPLEMENT, OR SIGN-MAGNITUDE AND IS              "
"         FULL WORDLENGTH.                                    "
"    2--- SIGN-BIT IS LEFTMOST IN WORD.                       "
"    3--- A SINGLE CHARACTER READ INTO AN INTEGER UNDER"
"         A1 FORMAT IS LEFT ADJUSTED IN THE WORD WITH         "
"         BLANK FILL.                                         "
"    4--- THE BIT REPRESENTATION FOR CHARACTER ZERO IN        "
"         YOUR MACHINE'S CODE IS NOT ALL ZEROES.              "
"                                                             "
"ON CDC 6000 OR 7000 SERIES USE THE R1 FORMAT OF              "
"    EXTENDED FORTRAN AND REIMPLEMENT THE CONVERSION          "
"    PORTIONS OF RDSTR9,WTSTR9,INIT99 AND INCV99.             "
"                                                             "
"PDP/11 ALSO REQUIRES SOME WORK.                              "
;
RUNCHECK(-ALL);
"THESE 3 CONSTANTS ARE MACHINE DEPENDENT"
"    SHORT BYTE = 2**(BITS PER BYTE - 1) "
CONSTANT SHORT BYTE=128,BITS PER WORD=32,BITS_PER_BYTE=8;
SUBROUTINE RDSTR9(NDEV,ARRAY,SIZ,N1,N2):
   "READ ARRAY(N1...N2) FROM DEVICE NDEV AND CONVERT TO "
   "    INTERNAL CODES FOR CHAR.  STRING LENGTH IS NOT SET."
   INTEGER NDEV,N1,N2,SIZ,ARRAY(SIZ);
   CHAR COMMON; EQUIVALENCE(C(1),OUTCH9(1)),(T(1,1),INCH9(1,1));
   INTEGER C( LAST(CHAR) ),T(2, SHORT BYTE );
   INTEGER I,J,W,WSTAR,INDTAB,BUFFER(200),NCHAR;
   FORMAT STR999=(200A1);

   "READ RECORD OF CHARS INTO SUBARRAY"
   READ(NDEV,STR999) (FOR I=N1 TO N2:ARRAY(I));
   DO I=N1 TO N2:
      W:=ARRAY(I);
      WSTAR:=IABS(W)/BSHIFT+1;
      IF W < 0:
         INDTAB:=1;
      ELSE:
         INDTAB:=2;
      ENDIF;
      ARRAY(I):=T(INDTAB,WSTAR);
      IF ARRAY(I) = 0:
         CALL RUNERR( $CONVERT );
      ENDIF;
   ENDDO;
   RETURN;
ENDSUBROUTINE;
```

```
SUBROUTINE WTSTR9(NDEV,ARRAY,SIZ,N1,N2):
   "WRITE STRING ARRAY(N1...N2) ONTO DEVICE NDEV "
   "   AFTER APPROPRIATE CONVERSION FROM INTERNAL "
   "   CHAR CODES TO A1 FORMAT.                   "
   INTEGER NDEV,N1,N2,SIZ,ARRAY(SIZ);
   CHAR COMMON; EQUIVALENCE(C(1),OUTCH9(1)),(T(1,1),INCH9(1,1));
   INTEGER C( LAST(CHAR) ),T(2, SHORT BYTE );
   INTEGER I,J,W,WSTAR,INDTAB,BUFFER(200),NCHAR;
   FORMAT STR999=(200A1);
   NCHAR:=N2-N1+1;
   IF NCHAR < 1: RETURN; ENDIF;
   DO I=N1 TO N2:
      IF IN CHAR(ARRAY(I)):
         IF C(ARRAY(I)) ¬= 0 :
            BUFFER(I-N1+1):=C(ARRAY(I));
         ELSE:
            CALL RUNERR( $CONVERT );
         ENDIF;
      ELSE:
         CALL RUNERR( $CONVERT );
      ENDIF;
   ENDDO;
   WRITE(NDEV,STR999) (FOR I=1 TO NCHAR :BUFFER(I));
   RETURN;
ENDSUBROUTINE;

SUBROUTINE INIT99:
   "INITIALIZE CHARACTER CONVERSION TABLES"
   CHAR COMMON; EQUIVALENCE(C(1),OUTCH9(1)),(T(1,1),INCH9(1,1));
   INTEGER C( LAST(CHAR) ),T(2, SHORT BYTE );
   INTEGER I,J,W,WSTAR,INDTAB,BUFFER(200),NCHAR;
   BSHIFT:=2**( BITS_PER_WORD - BITS_PER_BYTE );
   DO I=1 TO 2:
      DO J=1 TO SHORT_BYTE:
         T(I,J):=0;
      ENDDO;
   ENDDO;

   DO I=1 TO LAST(CHAR):
      IF C(I) ¬= 0 :
         IF C(I) < 0:
            INDTAB:=1;
         ELSE:
            INDTAB:=2;
         ENDIF;
         WSTAR:=IABS(C(I))/BSHIFT+1;
         IF T(INDTAB,WSTAR) ¬= 0:
            CALL RUNERR( $CHARMAP );
         ENDIF;
         T(INDTAB,WSTAR):=I;
      ENDIF;
   ENDDO;
ENDSUBROUTINE;

FUNCTION INCV99(CH) INTEGER:
   "COMPUTE INTERNAL CODE FOR CHAR CH READ BY A1 FORMAT"
   CHAR COMMON;
   INTEGER CH,CHSTAR,IND;
   CHSTAR:=IABS(CH)/BSHIFT +1;
   IF CH < 0:
      IND:=1;
   ELSE:
      IND:=2;
   ENDIF;
   INCV99:=INCH9(IND,CHSTAR);
   IF INCV99 = 0:
      CALL RUNERR( $CONVERT );
   ENDIF;
ENDFUNCTION;
```

```
"STRING UPDATING ROUTINES"
FUNCTION IRPL99(S,SIZ,LEN,I1,I2,T,K1,K2) INTEGER:
   "REPLACE S(I1...I2) BY T(K1...K2) AND RETURN "
   "    THE NEW CURRENT LENGTH OF S, "
   "    WHILE CHECKING THAT S DOES NOT OVERFLOW "
   "    ITS MAXIMUM SIZE, SIZ.                  "
   INTEGER SIZ,S(SIZ),T(K2),SHIFT;
   SHIFT:=(K2-K1)-(I2-I1);
   IF SHIFT < 0:
      FOR I=I2+1 TO LEN:
         S(I+SHIFT):=S(I);
      ENDFOR;
   ELSEIF SHIFT > 0:
      IF LEN+SHIFT > SIZ:
         CALL RUNERR( $REPL );
      ENDIF;
      FOR I=LEN BY -1 TO I2+1:
         S(I+SHIFT):=S(I);
      ENDFOR;
   ENDIF;
   FOR K=K1 TO K2:
      S(I1+(K-K1)):=T(K);
   ENDFOR;
   IRPL99 := LEN + SHIFT ;
   RETURN;
ENDFUNCTION;

FUNCTION IRPL98(S,SIZ,LEN,I1,I2,CH) INTEGER:
   "REPLACE S(I1...I2) BY THE SINGLE ELEMENT CH, "
   "    OTHERWISE LIKE IRPL99                    "
   INTEGER SIZ,S(SIZ),SHIFT,CH;
   SHIFT:=I1-I2;
   IF SHIFT < 0:
      FOR I=I2+1 TO LEN:
         S(I+SHIFT):=S(I);
      ENDFOR;
   ELSEIF SHIFT > 0:
      IF LEN+SHIFT > SIZ:
         CALL RUNERR( $REPL );
      ENDIF;
      FOR I=LEN BY -1 TO I2+1:
         S(I+SHIFT):=S(I);
      ENDFOR;
   ENDIF;
   S(I1):=CH;
   IRPL98 := LEN + SHIFT ;
   RETURN;
ENDFUNCTION;

FUNCTION IDEL99(S,SIZ,LEN,I1,I2) INTEGER:
   "DELETE S(I1...I2) AND RETURN THE NEW CURRENT "
   "    LENGTH OF S.                             "
   INTEGER SIZ,S(SIZ),SHIFT;
   SHIFT:= -(I2-I1+1);
   FOR I=I2+1 TO LEN:
      S(I+SHIFT):=S(I);
   ENDFOR;
   IDEL99 := LEN + SHIFT ;
   RETURN;
ENDFUNCTION;
   RUNCHECK(ALL);

%L
%%
```

APPENDIX E

SAMPLE PROGRAMS IN SKOL

This appendix contains four SKOL programs which are briefly described as follows:

1. Generate the first 100 prime numbers.
2. Build and print a perfectly balanced binary tree.
3. Read and print listing of a stream of telegrams (see reference [15]).
4. Convert simple expressions from infix to postfix operator notation (see pp 73-75 of reference [2]).

```
%L
%AØ
" SKOL PROGRAM TO GENERATE PRIMES "
RUNCHECK(-ALL);
MAIN:
   CONSTANT NPRIME=1ØØ;
   INTEGER PRIME( NPRIME ),TRIAL,ITEST,NXTPRM,IPR;

   PRIME(1):=2; PRIME(2):=3;
   FOR IPR=3 TO NPRIME:
      UNTIL FOUND_NEXT_PRIME:
         FOR TRIAL=PRIME(IPR-1)+2 BY 2:
            UNTIL IS_PRIME OR IS_COMPOSITE:
               FOR ITEST=2 TO IPR-1:
                  IF MOD(TRIAL,PRIME(ITEST)) = Ø :
                     IS_COMPOSITE;
                  ELSEIF PRIME(ITEST)**2 > TRIAL :
                     IS_PRIME;
                  ENDIF;
               ENDFOR;
               IS_PRIME;
            THENCASE:
               IS_PRIME:
                  BEGIN NXTPRM:=TRIAL; FOUND_NEXT_PRIME; END
               IS_COMPOSITE:
                  BEGIN END
            ENDUNTIL;
         ENDFOR;
      ENDUNTIL;
      PRIME(IPR):=NXTPRM;
   ENDFOR;

   OUTPUT($PAGE,:5X,'THE ',IPR:I4,'-TH PRIME IS ',PRIME( NPRIME ):);
ENDMAIN;
%%
```

```
%L
   "PROGRAM TO BUILD PERFECTLY BALANCED BINARY TREE"
   "    SEE PROGRAM 4.3 ON PAGE 196 OF N. WIRTH'S NEW"
   "    BOOK ON ALGORITHMS + DATA STRUCTURES = PROGRAMS"
   TYPE CHAR=(' ','X','.');
%U3    "INCLUDE CHAR FACILITIES"
   MAIN:
   PROCESS NEXT_NUMBER=(GETNUM);
   RECORD CLASS(30) OF NODE:
      INTEGER:KEY; REF:LEFT,RIGHT;
   ENDRECORD;
   REF TO NODE:NEWNOD,ROOT,T;
   INTEGER NUM,ARRAY(20),I,DEPTH;
   STRING OUTLIN(50),BLANKS(5);
   CHAR_COMMON;
   RECUR(100):TREE(*),PRINT_TREE(*,*);

   RUNCHECK(-ALL);
   MAKEAVAIL NODE; CHAR_SETUP; START NEXT_NUMBER AT GETNUM;
   DELETE BLANKS; REPEAT 5 TIMES: CATENATE ' ' ONTO BLANKS; ENDREPEAT;
   RESUME NEXT_NUMBER;
   DEPTH:=0; K:=1;
   LOOP: WHILE NUM >= K: INCR DEPTH; K:=2*K; ENDLOOP;
   DECR DEPTH;
   "2**DEPTH < = NUM < 2**(DEPTH+1)"
   EXECUTE TREE(NUM);
   OUTPUT($PAGE,:20X,'INDENTED TREE',:/,:$SKIP2);
   EXECUTE PRINT_TREE(ROOT,0);
   OUTPUT($PAGE);
   RETURN;

   COROUTINE GETNUM:  "ALIAS FOR PROCESS NEXT_NUMBER"
      "DELIVERS NEXT INPUT VALUE IN GLOBAL NUM"
      REPEAT:
         INPUT((FOR I=1 TO 20: ARRAY(I)):20I4);
         FOR I=1 TO 20:
            NUM:=ARRAY(I); SUSPEND NEXT_NUMBER;
         ENDFOR;
      ENDREPEAT;
   ENDCOROUTINE;

   ROUTINE TREE(N) LOCAL(NL,NR,NEWNOD):
      "BUILD BALANCED N-NODE BINARY TREE"
      "RETURNS REF TO TOP NODE IN GLOBAL ROOT"
      IF N = 0:
         ROOT:= NIL ;
      ELSE:
         NL := N /2; NR := N - NL -1;
         RESUME NEXT_NUMBER;
         NEW NEWNOD ;
         WITH NEWNOD:
            @.KEY := NUM;
            EXECUTE TREE( NL );@.LEFT :=ROOT;
            EXECUTE TREE( NR );@.RIGHT :=ROOT;
         ENDWITH;
         ROOT:= NEWNOD ;
      ENDIF;
   ENDROUTINE;
```

```
   ROUTINE PRINT_TREE(T,H):
      "PRINT BINARY TREE T WITH INDENTATION H"
      IF T = NIL:
         IF H <= DEPTH:
            DELETE OUTLIN; CATENATE ' ' ONTO OUTLIN;
            REPEAT H TIMES:
               CATENATE BLANKS(1...5) ONTO OUTLIN;
            ENDREPEAT;
            CATENATE '.' ONTO OUTLIN;
            WRITESTRING OUTLIN;
         ENDIF;
      ELSE:
         WITH T:
            EXECUTE PRINT_TREE(@.LEFT , H +1);
            DELETE OUTLIN; CATENATE ' ' ONTO OUTLIN;
            REPEAT H TIMES:
               CATENATE BLANKS(1...5) ONTO OUTLIN;
            ENDREPEAT;
            CATENATE 'X' ONTO OUTLIN;
            WRITESTRING OUTLIN;
            OUTPUT($OVER,:60X,@.KEY :);
            EXECUTE PRINT_TREE(@.RIGHT , H +1);
         ENDWITH;
      ENDIF;
   ENDROUTINE;


   ENDMAIN;
%%
```

```
%L
TYPE CHAR=(ALPHABET=,DIGIT=,' ','*',',');
%U3
%A2

MAIN:
   "HENDERSON ET AL TELEGRAM PROBLEM, COROUTINE SOLUTION"
   RUNCHECK(-TRACE);
   CONSTANT OVERLENGTH_LIMIT=5,SPACE=' ';
   STRING TERMWD(4),NOCHRG(4),SPACES(3);
   CHAR CIL;
   STRING WORD(10), LINE(31),BUFFER( 80 );
   INTEGER CWC,I,K,IBP;
   LOGICAL OW,EQU999;
   CHAR COMMON;
   DEFINE ' EQUAL(#,#)'=  ' EQU999( #1 , SIZE(#1), LENGTH(#1),
      #2 , SIZE(#2), LENGTH(#2)) ' ;
   PROCESS HENDERZAHN=(NEXT_INPUT_LETTER,NEXT_INPUT_WORD,
      TELEGRAM_READER,OUTPUT_LISTING);

   CHAR SETUP;
   IF SIZE(WORD) > SIZE(LINE)-1:
      OUTPUT($PAGE,'WORDSIZE TOO BIG FOR LINE');
      RETURN;
   ENDIF;
   START HENDERZAHN AT OUTPUT_LISTING;
   RESUME HENDERZAHN;
   RETURN;

   COROUTINE OUTPUT_LISTING:
      #SPACES := ' ';
      OUTPUT($PAGE);
      UNTIL LAST_TELEGRAM:
         REPEAT: "EACH TELEGRAM"
            RESUME TELEGRAM_READER FROM OUTPUT_LISTING;
            IF LENGTH(WORD)=0: LAST_TELEGRAM; ENDIF;
            OUTPUT($SKIP2);
            UNTIL END_OF_TELEGRAM:
               REPEAT: "EACH OUTPUT LINE OF TELEGRAM"
                  #LINE:=' ';
                  LOOP:  "EACH TELEGRAM WORD"
                     CATENATE WORD ONTO LINE;
                     RESUME TELEGRAM_READER FROM OUTPUT_LISTING;
                     IF LENGTH(WORD)=0: END_OF_TELEGRAM; ENDIF;
                  WHILE LENGTH(LINE)+ LENGTH(SPACES)+ LENGTH(WORD)
                     <= SIZE(LINE):
                     CATENATE SPACES ONTO LINE;
                  ENDLOOP;
                  WRITESTRING LINE;
               ENDREPEAT;
            ENDUNTIL;
            WRITESTRING LINE;
            OUTPUT($SKIP,CWC:I3,' WORDS CHARGED');
            IF OW: OUTPUT('WORDLENGTH EXCEEDS ', OVERLENGTH_LIMIT :I3); ENDIF;
         ENDREPEAT;
      ENDUNTIL;
      OUTPUT($SKIP2,'*****',:5X,'ALL TELEGRAMS LISTED',:5X,'*****');
      SUSPEND HENDERZAHN;
   ENDCOROUTINE;
```

```
   COROUTINE TELEGRAM_READER:
      #TERMWD:='ZZZZ'; #NOCHRG:='STOP';
      REPEAT:  "EACH TELEGRAM"
         CWC:=Ø; OW:= FALSE ;
         UNTIL TELEGRAM TERMINATED:
            REPEAT:  "EACH WORD"
               RESUME NEXT INPUT WORD FROM TELEGRAM_READER;
               IF EQUAL(WORD,TERMWD):
                  DELETE WORD; TELEGRAM_TERMINATED;
               ENDIF;
               IF NOT( EQUAL(WORD,NOCHRG)): INCR CWC; ENDIF;
               IF LENGTH(WORD)> OVERLENGTH LIMIT : OW:= TRUE ; ENDIF;
               RESUME OUTPUT_LISTING FROM TELEGRAM_READER;
            ENDREPEAT;
         ENDUNTIL;
         RESUME OUTPUT_LISTING FROM TELEGRAM_READER;
      ENDREPEAT;
   ENDCOROUTINE;

   COROUTINE NEXT_INPUT_WORD:
      RESUME NEXT_INPUT_LETTER FROM NEXT_INPUT_WORD;
      REPEAT:
         EXECUTE SKIP_BLANKS;
         UNTIL END_OF_WORD:
            FOR LENGTH(WORD)=1 TO SIZE(WORD):
               WORD( LENGTH(WORD)):= CIL;
               RESUME NEXT_INPUT_LETTER FROM NEXT_INPUT_WORD;
               IF CIL= SPACE : END_OF_WORD; ENDIF;
            ENDFOR;
            WORD( LENGTH(WORD)) := '*';
            REPEAT:
               RESUME NEXT_INPUT_LETTER FROM NEXT_INPUT_WORD;
               IF CIL = SPACE : END_OF_WORD; ENDIF;
            ENDREPEAT;
         ENDUNTIL;
         RESUME TELEGRAM_READER FROM NEXT_INPUT_WORD;
      ENDREPEAT;
   ENDCOROUTINE;

   ROUTINE SKIP_BLANKS:
      UNTIL NON_SPACE:
         REPEAT:
            IF CIL ~= SPACE :
               NON_SPACE;
            ELSE:
               RESUME NEXT_INPUT_LETTER FROM NEXT_INPUT_WORD;
            ENDIF;
         ENDREPEAT;
      ENDUNTIL;
   ENDROUTINE;

   COROUTINE NEXT_INPUT_LETTER:
      REPEAT:
         READSTRING($INPUT)BUFFER;
         FOR IBP=1 TO SIZE(BUFFER):
            CIL:=BUFFER(IBP);
            RESUME NEXT_INPUT_WORD FROM NEXT_INPUT_LETTER;
         ENDFOR;
      ENDREPEAT;
   ENDCOROUTINE;

ENDMAIN;
```

```
FUNCTION EQU999(STR1,S1,L1,STR2,S2,L2) LOGICAL:
   "TEST EQUALITY OF TWO STRINGS"
   INTEGER S1,S2,L1,L2,I;
   CHAR STR1(S1),STR2(S2);

   UNTIL ALL_SAME OR MISMATCH:
      IF L1 ~= L2 : MISMATCH; ENDIF;
      FOR I=1 TO L1:
         IF STR1(I) ~= STR2(I): MISMATCH; ENDIF;
      ENDFOR;
      ALL_SAME;
   THENCASE:
      ALL_SAME: BEGIN EQU999 := TRUE ; END
      MISMATCH: BEGIN EQU999 := FALSE ; END
   ENDUNTIL;
   RETURN;
ENDFUNCTION;

%%
```

```
%L
   "SKOL PROGRAM TO CONVERT INFIX TO POSTFIX"
   TYPE CHAR=(ALPHABET=,' ','(',')',ADD=('+','-'),'*');
%U3   "INCLUDE CHAR UTILITIES"
MAIN:
   CHAR COMMON;  CHAR CH; STRING CARD(80),LINE(120);
   STRING OPSTK(20); INTEGER IC;
   DEFINE ';DEBUG #;'=  ' ;  ' ; RUNCHECK(-ALL);
   DEFINE ';NEXTCH;'=  ';INCR IC; CH:=CARD(IC);' ;
   DEFINE ';STACK;'=  ';CATENATE CH ONTO OPSTK; NEXTCH;
                      DEBUG OUTPUT(''___STACK '');  ' ;
   DEFINE ' TOP '=  ' OPSTK( LENGTH(OPSTK))  ' ;
   DEFINE ';POP;'=  ';DECR LENGTH(OPSTK);' ;
   DEFINE ';UNSTACK;'=  ';CATENATE TOP ONTO LINE;
                        DEBUG OUTPUT(''___UNSTACK '', TOP :C);POP;';

   CHAR SETUP;  DELETE LINE; CATENATE ' ' ONTO LINE;
   READSTRING CARD;  IC:=1;  CH:=CARD(IC);
   CATENATE CARD(1... SIZE(CARD)) ONTO LINE;
   WRITESTRING LINE; OUTPUT($SKIP2);
   DELETE LINE; CATENATE ' ' ONTO LINE;
   LOOP: WHILE CH = ' ' : NEXTCH; ENDLOOP;
   DELETE OPSTK;  CATENATE '(' ONTO OPSTK;
   UNTIL FINISHED:
      REPEAT:
         DEBUG OUTPUT(CH:C);
         CASE CH:CHAR OF
            ALPHABET:
               BEGIN
               CATENATE CH ONTO LINE; NEXTCH;
               DEBUG OUTPUT('___PASS THRU');
               END
            '(': BEGIN STACK; END
            ' ',')':
               BEGIN
               IF TOP = '(' :
                  POP;
                  IF CH = ' ': FINISHED; ENDIF;
                  NEXTCH;
               ELSE:
                  UNSTACK;
               ENDIF;
               END
            ADD:
               BEGIN
               IF TOP = '(': STACK; ELSE: UNSTACK; ENDIF;
               END
            '*':
               BEGIN
               IF TOP = '*': UNSTACK; ELSE: STACK; ENDIF;
               END
         ENDCASE;
      ENDREPEAT;
   ENDUNTIL;
   WRITESTRING LINE;

ENDMAIN;

%%
```

APPENDIX F

SAMPLE PRECOMPILER DIAGNOSTICS

The following listing was produced by the SKOL compiler and illustrates most of the diagnostic facilities. There are notes after the listing to explain some of the less obvious messages. Appendix H should be consulted to decode the control error diagnostics "UNCLOSED x FOUND AT y".

```
109.          Ø   MAIN;
110.          Ø       "TEST OF SKOL DIAGNOSTICS"
111.          Ø       TYPE CHAR=(ALPHABET=('A','B','C'),EOL,TAB,' ');
112.          Ø       CHAR CH;
113.          Ø       STRING STR1(2Ø),STR3,STR4(1Ø);
114.          Ø       CONSTANT PI=3.14,BAD,NDIM=2;
115.      ???   BAD CONSTANT DEFINITION BAD
116.          Ø       TYPE COLOR=(REDS=(RED,PINK),GREEN,BLUE,BLACK);
117.          Ø       COLOR HUE;
118.          Ø       SCAL1 SC1,SC2;  COLOR:C1,C2;
119.      ???   UNEXPECTED COLON
120.          Ø       RECORD CLASS(N) OF WHAT:
121.          Ø          COLOR:C1;CHAR:CH1;REF:LL,RL;
122.      ???   BAD TYPE COLOR IN FIELD LIST
123.          Ø       ENDRECORD;
124.          Ø       REF TO WHAT:P,Q(4);
125.          Ø       REF TO NONO: P2;
126.      ???   RECORD CLASS NONO UNDECLARED
127.          Ø       PROCESS XYZ=(AA,BB);
128.          Ø       RECUR(1ØØ):X1,X2(*);
129.          Ø
130.          Ø       MAKEAVAIL NONO;  START XYZ AT X1;  CHAR_SETUP;
131.      ???   RECORD CLASS NONO UNDECLARED
132.      ???   UNDECLARED COROUTINE X1
133.          Ø       DELETE STR1;
134.          Ø       REPLACE STR1(1...|Ø) BY 'A' ;
135.          Ø       MACRO MAC1(KEY1=Ø,KEY2)='ZZ:= KEY1 ;Z2( KEY2 ):= KEY1 +1' ;
136.          Ø       UNTIL S1 OR S2 OR S3:
137.          Ø          IF P: S4; ENDIF;
138.          Ø          DELETE SC1; CATENATE 'AB' ONTO STR2(1...3);
139.      ???   UNDECLARED STRING SC1
140.      ???   UNDECLARED STRING STR2(1...3)
141.      ???   UNDECLARED STRING STR2(1...3)
142.      ???   UNDECLARED STRING STR2(1...3)
143.      ???   UNDECLARED STRING STR2(1...3)
144.      ???   UNDECLARED STRING STR2(1...3)
145.      ???   UNDECLARED STRING STR2(1...3)
146.          Ø          MAC1(KEY2=3);
147.          Ø          MAC1(3.2,4);
148.      ???   BAD KEYWORD PARAMETER 3.2
149.      ???   BAD KEYWORD PARAMETER 4
150.      ???   OBLIGATORY PARAMETER KEY2 MISSING
151.          Ø          LOOP:
152.          Ø             I:=I+1;  READ ( $INPUT, PERSON )LIST;
153.      ???   UNDEFINED FORMAT PERSON
154.          Ø             IF A = O: REPEAT 7 TIMES:
155.          Ø          ENDLOOP;
156.          Ø          FOR I=1Ø BY -1 DO BEGIN END;  S2;
157.      ???   UNCLOSED D FOUND AT X
158.      ???   UNCLOSED Z FOUND AT X
159.      ???   MISSING WHILE
160.      ???   BAD SYNTAX AT---I=1Ø BY -1 DO BEGIN END
161.      ???   UNEXPECTED BEGIN
```

```
162.          Ø      THENCASE:
163.          Ø         S5: BEGIN WRITE STRING($MYFILE)NON_STRING; END
164.      ???   SITUATION LABEL S5 NOT DECLARED
165.      ???   BAD $
166.      ???   BAD SYNTAX
167.      ???   BAD
168.      ???   BAD SYNTAX
169.      ???   UNDECLARED STRING NON STRING
170.      ???   UNDECLARED STRING NON_STRING
171.          Ø         S1,S3:
172.          Ø            BEGIN J:= LENGTH(STR2);
173.      ???   UNDECLARED STRING STR2
174.          Ø            DO I=1 TO 1Ø A:=Ø; ENDDO;
175.      ???   := HAS HAD ITS : INTERPRETED AS COLON
176.          Ø            IF SIZE(STR2) = Ø:
177.      ???   UNDECLARED STRING STR2
178.          Ø               STR1(2):='?';
179.      ???   UNKNOWN CHARACTER |?|
180.          Ø            ELSE:
181.          Ø               DELETE STR2;
182.      ???   UNDECLARED STRING STR2
183.          Ø               INSERT '+' AFTER STR1( SIZE(STR1));
184.      ???   UNKNOWN CHARACTER |+|
185.          Ø               CATENATE EOL ONTO STR4; OUTPUT(EOL:C);
186.      ???   UNDECLARED STRING STR4
187.      ???   UNDECLARED STRING STR4
188.      ???   UNDECLARED STRING STR4
189.      ???   CHAR EOL CANNOT BE OUTPUT
190.          Ø               CASE I:SCAL2 OF
191.      ???   UNDECLARED SCALAR TYPE SCAL2
192.      ???   UNDECLARED SCALAR TYPE SCAL2
193.      ???   UNDECLARED SCALAR TYPE SCAL2
194.          Ø                  2:BEGIN END
195.          Ø                  ELSE:
196.          Ø                     BEGIN
197.          Ø                     CASE CH:ALPHABET OF
198.          Ø                        'A','D':BEGIN END
199.          Ø                     WHILE B <= N: A:=1;
200.      ???   CASE LABEL AND BEGIN MISSING
201.      ???   UNCLOSED C FOUND AT A
202.      ???   UNCLOSED B FOUND AT A
203.      ???   UNCLOSED C FOUND AT A
204.      ???   UNCLOSED W FOUND AT A
205.      ???   UNCLOSED Y FOUND AT A
206.      ???   UNCLOSED T FOUND AT A
207.      ???   MISSING LOOP
208.          Ø                     LINK P=Q(2) BY NEXT:
209.      ???   BAD FIELD NEXT
210.          Ø                     END
211.          Ø               ENDCASE;
212.      ???   UNCLOSED J FOUND AT B
213.      ???   MISSING BEGIN
214.          Ø            ENDIF;
215.          Ø            END
216.          Ø      ENDUNTIL;
217.      ???   MISSING IF
218.      ???   MISSING BEGIN
219.          Ø      UNTIL A OR B:
220.          Ø         EXECUTE X2(1,Ø.4);
221.      ???   MISSING UNTIL
222.      ???   WRONG NUMBER OF PARMS FOR X2
223.          Ø         FOR A(I) TO N: K:=2;
224.      ???   BAD FOR PHRASE
225.          Ø         DO 7 TIMES:
226.          Ø            FOR I=1Ø BY -1:
227.          Ø         ENDDO;
228.      ???   BAD SYNTAX AT---7 TIMES: FOR I=1Ø BY -1: ENDDO
229.      ???   UNEXPECTED COLON
230.          Ø      ENDUNTIL;
```

```
231.      ???   UNCLOSED H FOUND AT I
232.      ???   UNCLOSED U FOUND AT I
233.      ???   MISSING DO
234.          Ø
235.          Ø       ROUTINE D:
236.          Ø           REPEAT J:=Ø; B:=3; UNTIL P < 1;
237.      ???   MISSING UNTIL
238.      ???   ROUTINE D DECLARED BEFORE AN EXECUTE
239.      ???   BAD SYNTAX AT---J:=Ø
240.      ???   BAD SYNTAX AT---P < 1
241.          Ø           IF IN REDSPECTRUM(C1) THEN A:=Ø;
242.      ???   UNDECLARED SCALAR TYPE REDSPECTRUM
243.      ???   UNDECLARED SCALAR TYPE REDSPECTRUM
244.      ???   := HAS HAD ITS : INTERPRETED AS COLON
245.          Ø           OUTPUT X,XX;
246.          Ø           EXECUTE D;
247.      ???   ILLEGAL BACKWARD ROUTINE REF. D
248.          Ø           @(P.ZZ):=@(Q(2).CH1);
249.      ???   BAD FIELD ZZ
250.          Ø           WITH Q(K):
251.          Ø             OUTPUT(@.CH1 :C,@.XX );
252.      ???   BAD FIELD XX
253.          Ø           ENDWITH;
254.          Ø           OUTPUT(A:,Z;
255.      ???   BAD SYNTAX AT---A:,Z
256.      ???   UNEXPECTED COLON
257.          Ø           RESUME XYZ;
258.          Ø
259.          Ø       COROUTINE AA:
260.      ???   UNCLOSED Z FOUND AT G
261.      ???   UNCLOSED E FOUND AT G
262.          Ø           NEW P3; FREE Q(1);
263.      ???   REF P3 UNDECLARED
264.          Ø           N:=Ø; RESUME BB FROM AA;
265.          Ø           CASE HUE:COLOR OF REDS:BEGIN END ENDCASE;
266.          Ø       ENDCOROUTINE;
267.      ???   UNTREATED CASE VALUE
268.      ???   UNTREATED CASE VALUE
269.      ???   UNTREATED CASE VALUE
270.          Ø
271.          Ø       ROUTINE EFG:
272.      ???   ROUTINE EFG DECLARED BEFORE AN EXECUTE
273.          Ø           INCR K; @(SS3.LL):= NIL ;
274.      ???   UNDECLARED REF SS3
275.          Ø           SUSPEND XYZ;
276.          Ø       ENDROUTINE;
277.          Ø
278.          Ø       ROUTINE X1(P2) LOCAL(Z):
279.          Ø           RUNCHECK(-FOR,+CASE);
280.      ???   WRONG NUMBER OF PARMS FOR X1
281.          Ø           READSTRING CARD;
282.      ???   UNDECLARED STRING CARD
283.      ???   UNDECLARED STRING CARD
284.          Ø           IF IN DIGIT(CH):
285.      ???   UNDECLARED SCALAR TYPE DIGIT
286.      ???   UNDECLARED SCALAR TYPE DIGIT
287.          Ø             C:= VALUE(CH);
288.      ???   UNDECLARED SCALAR TYPE DIGIT
289.          Ø           ENDIF;
290.          Ø           EXECUTE X1;
291.          Ø           EXECUTE A;
292.          Ø           RESUME EFG;
293.      ???   UNDECLARED COROUTINE EFG
294.          Ø       ENDROUTINE;
295.          Ø
296.          Ø       ROUTINE D:
297.      ???   ROUTINE D DECLARED BEFORE AN EXECUTE
298.          Ø           Z := P2 ;
299.      ???   PARM OR LOCAL OUTSIDE SCOPE
300.      ???   PARM OR LOCAL OUTSIDE SCOPE
```

```
301.          Ø          MOVE STR1 TO STR2(1...3);
302.      ???   UNDECLARED STRING STR2
303.      ???   UNDECLARED STRING STR2
304.      ???   UNDECLARED STRING STR2
305.          Ø          REPLACE STR1(3|...|Ø) BY CH;
306.      ???   INCORRECT PLACE FOR STRING LENGTH
307.      ???   INCORRECT PLACE FOR STRING LENGTH
308.          Ø          DELETE STR3(4...);
309.      ???   UNDECLARED STRING STR3
310.      ???   UNDECLARED STRING STR3
311.      ???   UNDECLARED STRING STR3
312.          Ø          WRITESTRING OUTLIN;
313.      ???   UNDECLARED STRING OUTLIN
314.      ???   UNDECLARED STRING OUTLIN
315.          Ø       ENDROUTINE;
316.          Ø
317.          Ø    ENDMAIN;
318.               %%
319.               %L
320.          Ø       $BLOCKDATA$
321.          Ø      "END OF SKOL COMPILATION" $FINISH$
322.               %%
323.      ???   MISSING MAIN
324.      ???   ROUTINE OR COROUTINE BB NOT DEFINED
325.      ???   ROUTINE OR COROUTINE X2 NOT DEFINED
326.      ???   ROUTINE OR COROUTINE A NOT DEFINED
327.      ???   LABEL LEFT ON STACK
328.      ???   LABEL LEFT ON STACK
329.      ???   LABEL LEFT ON STACK
330.      ???   LABEL LEFT ON STACK
```

Notes on Diagnostics:

Lines	Explanation
157-159	The IF statement and REPEAT statement on line 154 are not properly terminated and the WHILE phrase is missing from the LOOP: ... ENDLOOP; command on lines 151-155.
160-161	The FOR statement on line 156 is incorrectly formed.
164	The name S5 does not occur as a situation in the UNTIL phrase on line 136.
165-170	The names $MYFILE and NON_STRING have not been defined or declared.
175	There is a missing : after 10 on line 174.
179	'?' is not among the characters of CHAR defined on line 111.
189	Characters denoted by identifiers may not be output.
200-207	WHILE phrase on line 199 does not have a corresponding LOOP: in the previous lines.
209	P has been declared as a reference to record class WHAT on line 124, but NEXT is not among the fields declared for WHAT on lines 120-123.
212	LINK statement on line 208 not terminated.
213, 217 218, 221	Spurious messages caused by attempt to find LOOP corresponding to WHILE on line 199.
222	X2 is declared as recursive routine with one parameter on line 128, but invoked with two parameters on line 220.

231-233	FOR statement on line 226 not properly terminated; DO phrase on line 225 is incorrectly formed and is not recognized as matching the ENDDO on line 227.
237	Spurious message caused by foulup on lines 225-227.
244	THEN in line 241 should be colon.
249	ZZ is not a field of record class WHAT to which reference P refers.
252	@.XX is an abbreviation for @(Q(K).XX), Q is an array of references to WHAT, but XX is not a field in WHAT.
260-261	IF on line 241 and ROUTINE on line 235 not terminated before COROUTINE on line 259.
267-269	Three constant values of type COLOR (namely GREEN, BLUE, BLACK) defined in line 116 do not appear as case labels in line 265.
280	Recursive routine X1 declared without parameters in line 128, but defined with one in line 278.
299-300	Z and P2 are respectively a local variable and a parameter of recursive routine X1 on lines 278-294. The occurrences on line 298 are outside the valid scope.

APPENDIX G

FORTRAN EQUIVALENT OF TWO SKOL PROGRAMS

The following FORTRAN is the equivalent of the prime-generator SKOL program in Appendix E.

```
      INTEGER PRIME( 100 ),TRIAL,ITEST,NXTPRM,IPR
      PRIME(1)= 2
      PRIME(2)= 3
      I10263=  1
      I10264= 100
      I10265=  3
      IF((I10264-I10265)*I10263 .LT. 0) GOTO 10266
      IPR =  I10265
      GOTO 10262
10261  CONTINUE
      IF( IPR .EQ. I10264)GOTO 10266
      IPR =  IPR +I10263
10262  CONTINUE
      I10282=  2
      TRIAL =  PRIME(IPR-1)+2
      GOTO 10283
10281  CONTINUE
      TRIAL =  TRIAL +I10282
10283 CONTINUE
      I10313=  1
      I10314=  IPR-1
      I10315=  2
      IF((I10314-I10315)*I10313 .LT. 0) GOTO 10316
      ITEST =  I10315
      GOTO 10312
10311  CONTINUE
      IF( ITEST .EQ. I10314)GOTO 10316
      ITEST =  ITEST +I10313
10312  CONTINUE
      IF(( MOD(TRIAL,PRIME(ITEST)) .NE.  0 ))GOTO 10331
      GOTO 10300
      GOTO 10321

10331 CONTINUE
      IF (( PRIME(ITEST)**2 .LE.  TRIAL ))GOTO 10341
      GOTO 10290
10341 CONTINUE
10321 CONTINUE
      GOTO 10311
10316 CONTINUE
      GOTO 10290
10290 CONTINUE
      NXTPRM= TRIAL
      GOTO 10270
      GOTO 10351

10300 CONTINUE
      GOTO 10351
10351 CONTINUE
      GOTO 10281
10270 CONTINUE
      PRIME(IPR)= NXTPRM
      GOTO 10261
10266 CONTINUE
10360 FORMAT(  1H1 ,  5X ,      4HTHE  ,  I4 ,     13H-TH PRIME IS  , G1
     *2.5)
      WRITE( 6 ,10360) IPR , PRIME( 100 )
      RETURN
      END
```

The following is the FORTRAN for the SKOL program to build and print a balanced binary tree given in Appendix E.

```
      INTEGER NODE0
      INTEGER  NODE    1( 30 )
      INTEGER  NODE    2( 30 ), NODE    3( 30 )
      INTEGER    NEWNOD,   ROOT,   T
      INTEGER NUM,ARRAY(20),I,DEPTH
      INTEGER      OUTLIN ( 50 ),I10751,     BLANKS ( 5 ),I10761
      COMMON/ CHCODE / OUTCH9(     3) , INCH9(2, 128 ) , BSHIFT
      INTEGER OUTCH9,INCH9,BSHIFT
      INTEGER  S99999 ( 100 ), T99999 /1/, B99999 /0/, RC9999 /0/
      DO 10791 NODE0 = 1 ,  30
      NODE1(NODE0)= NODE0-1
10791 CONTINUE
      NODE0=  30
      CALL INIT99
      ASSIGN 10720TO J10720
      ASSIGN 10720TO J10690
      I10761= 0
      I10802= 5
      IF(I10802.LT.1)GOTO 10801
      DO10811 I10820=1,I10802
      I10761 =I10761 +1
      BLANKS (I10761)=     1
10811 CONTINUE
10801 CONTINUE
      ASSIGN 10830 TO I10690
      GOTO 10700
10830  CONTINUE
      DEPTH= 0
      K= 1
10841 CONTINUE
      IF(( NUM .LT.  K ))GOTO 10851
      DEPTH =  DEPTH +1
      K= 2*K
      GOTO 10841
10851 CONTINUE
      DEPTH =  DEPTH -1
      S99999 ( T99999 )=  B99999
      S99999 ( T99999 +1 )=     1
      T99999 =   T99999 +( 2 )
      S99999 ( T99999 )=  NUM
      T99999 =   T99999 +1
      B99999 =  T99999 -(    1 +1)
      GOTO 10770
10860  CONTINUE
10870 FORMAT(  1H1 ,  20X ,     13HINDENTED TREE ,  / ,  1H- )
      WRITE( 6 ,10870)
      S99999 ( T99999 )=  B99999
      S99999 ( T99999 +1 )=     2
      T99999 =   T99999 +( 2 )
      S99999 ( T99999 )=  ROOT
      T99999 =   T99999 +1
      S99999 ( T99999 )=  0
      T99999 =   T99999 +1
      B99999 =  T99999 -(    2 +1)
      GOTO 10780
10880  CONTINUE
10890 FORMAT(  1H1 )
      WRITE( 6 ,10890)
      RETURN
      RETURN
10720 CONTINUE
10901 CONTINUE
10910 FORMAT(  20I4 )
      READ( 5 ,10910) ( ARRAY(I) , I = 1 , 20 )
      I10923=  1
```

```
      I10924=  20
      I10925=  1
      IF((I10924-I10925)*I10923 .LT. 0) GOTO 10926
      I =  I10925
      GOTO 10922
10921  CONTINUE
      IF( I .EQ. I10924)GOTO 10926
      I =  I +I10923
10922  CONTINUE
      NUM= ARRAY(I)
      ASSIGN 10930 TO J10690
      GOTO 10710
10930  CONTINUE
      GOTO 10921
10926 CONTINUE
      GOTO 10901
      CALL RUNERR(    6)
      RETURN
10770 CONTINUE
      T99999 =   T99999 +(     4-    1 )
      IF((  S99999 ( B99999 +    1) .NE.  0 ))GOTO 10951
      ROOT= 0
      GOTO 10941
10951 CONTINUE
      S99999 ( B99999 +    2) =  S99999 ( B99999 +    1) /2
      S99999 ( B99999 +    3) =  S99999 ( B99999 +    1) -  S99999 ( B99
     *999 +    2) -1
      ASSIGN 10960 TO I10690
      GOTO 10700
10960  CONTINUE
      IF(( NODE0 .NE.  0 ))GOTO 10981
      CALL RUNERR(    5)
10981 CONTINUE
      S99999 ( B99999 +    4) = NODE0
      NODE0= NODE1(NODE0)
      NODE    1(  S99999 ( B99999 +    4) )=  NUM
      S99999 ( T99999 )=  B99999
      S99999 ( T99999 +1 )=     3
      T99999 =   T99999 +( 2 )
      S99999 ( T99999 )=  S99999 ( B99999 +    2)
      T99999 =   T99999 +1
      B99999 =  T99999 -(    1 +1)
      GOTO 10770
10990  CONTINUE
      NODE    2(  S99999 ( B99999 +    4) )= ROOT
      S99999 ( T99999 )=  B99999
      S99999 ( T99999 +1 )=     4
      T99999 =   T99999 +( 2 )
      S99999 ( T99999 )=  S99999 ( B99999 +    3)
      T99999 =   T99999 +1
      B99999 =  T99999 -(    1 +1)
      GOTO 10770
11000  CONTINUE
      NODE    3(  S99999 ( B99999 +    4) )= ROOT
      ROOT=  S99999 ( B99999 +    4)
10941 CONTINUE
      GOTO 99999
      RETURN
10780 CONTINUE
      IF((  S99999 ( B99999 +    1) .NE. 0 ))GOTO 11021
      IF((  S99999 ( B99999 +    2) .GT.  DEPTH ))GOTO 11041
      I10751= 0
      I10751 =I10751 +1
      OUTLIN (I10751)=     1
      I11052=  S99999 ( B99999 +    2)
```

```
      IF(I11052.LT.1)GOTO 11051
      DO11061 I11070=1,I11052
      I10751 =  IRPL99( OUTLIN ,  50 ,I10751, ((I10751 )-( 0 )+1),(I1075
     *1 ), BLANKS ,( 1 ),( 5 ))
11061 CONTINUE
11051 CONTINUE
      I10751 =I10751 +1
      OUTLIN (I10751)=     3
      CALL WTSTR9( 6 , OUTLIN ,  50 , 1 ,I10751 )
11041 CONTINUE
      GOTO 11011
11021 CONTINUE
      S99999 ( T99999 )=  B99999
      S99999 ( T99999 +1 )=     5
      T99999 =   T99999 +( 2 )
      S99999 ( T99999 )=  NODE    2(  S99999 ( B99999 +    1) )
      T99999 =   T99999 +1
      S99999 ( T99999 )=  S99999 ( B99999 +    2) +1
      T99999 =   T99999 +1
      B99999 =  T99999 -(    2 +1)
      GOTO 10780
11080  CONTINUE
      I10751= 0
      I10751 =I10751 +1
      OUTLIN (I10751)=     1
      I11092=  S99999 ( B99999 +    2)
      IF(I11092.LT.1)GOTO 11091
      DO11101 I11110=1,I11092
      I10751 =  IRPL99( OUTLIN ,  50 ,I10751, ((I10751 )-( 0 )+1),(I1075
     *1 ), BLANKS ,( 1 ),( 5 ))
11101 CONTINUE
11091 CONTINUE
      I10751 =I10751 +1
      OUTLIN (I10751)=     2
      CALL WTSTR9( 6 , OUTLIN ,  50 , 1 ,I10751 )
11120 FORMAT(  1H+ ,  60X ,G12.5)
      WRITE( 6 ,11120) NODE    1(  S99999 ( B99999 +    1) )
      S99999 ( T99999 )=  B99999
      S99999 ( T99999 +1 )=     6
      T99999 =   T99999 +( 2 )
      S99999 ( T99999 )=  NODE    3(  S99999 ( B99999 +    1) )
      T99999 =   T99999 +1
      S99999 ( T99999 )=  S99999 ( B99999 +    2) +1
      T99999 =   T99999 +1
      B99999 =  T99999 -(    2 +1)
      GOTO 10780
11130  CONTINUE
11011 CONTINUE
      GOTO 99999
10700 GOTO J10690,(10720,10930)
10710 GOTO I10690,(10830,10960)
10730 GOTO J10720, (10720)
99999 CONTINUE
      RC9999 =  S99999 ( B99999 )
      T99999 =  B99999 -1
      B99999 =  S99999 ( T99999 )
      GOTO(10860,10880,10990,11000,11080,11130), RC9999
      RETURN
      END
      BLOCKDATA
      COMMON/ CHCODE / OUTCH9(     3) , INCH9(2, 128 ) , BSHIFT
      INTEGER OUTCH9,INCH9,BSHIFT
      EQUIVALENCE (C(1),OUTCH9(1))
      INTEGER C(     3)
      DATA C(    1)/    1H /,C(    2)/    1HX/,C(    3)/    1H./
      END
```

APPENDIX H

EXPLANATION OF CONTROL ERROR DIAGNOSTICS

Errors in the use of SKOL control structures are diagnosed by messages of the form:

UNCLOSED x FOUND AT y

where x and y are single letter codes whose meanings are given below:

UNCLOSED	A	means	LOOP:
"	B	"	BEGIN (scalar case group)
"	C	"	CASE ... OF
"	D	"	REPEAT ... TIMES:
"	E	"	ROUTINE
"	F	"	ELSEIF
"	G	"	COROUTINE
"	H	"	FOR...BY...
"	I	"	DO
"	J	"	LINK
"	K	"	MAIN
"	L	"	SUBROUTINE
"	M	"	FUNCTION
"	Q	"	UNTIL (single situation)
"	R	"	WITH
"	S	"	FOR...TO...
"	T	"	THENCASE
"	U	"	UNTIL (multiple situation)
"	V	"	REPEAT:
"	W	"	ELSE
"	X	"	LOOP
"	Y	"	BEGIN (sit. case group)
"	Z	"	IF
"	9	"	BEGIN (unexpected)

FOUND AT	A	means	WHILE ...
"	B	"	END
"	C	"	ENDCASE
"	E	"	ROUTINE
"	F	"	ELSEIF
"	G	"	COROUTINE
"	I	"	ENDDO
"	J	"	ENDLINK
"	K	"	MAIN
"	L	"	SUBROUTINE
"	M	"	FUNCTION
"	N	"	ENDMAIN
"	O	"	ENDSUBROUTINE
"	P	"	ENDFUNCTION
"	R	"	ENDWITH
"	S	"	ENDFOR
"	T	"	THENCASE
"	U	"	ENDUNTIL
"	V	"	ENDREPEAT
"	W	"	ELSE
"	X	"	ENDLOOP
"	Z	"	ENDIF
"	2	"	ENDCOROUTINE
"	3	"	ENDROUTINE