Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Ed Satterthwaite	Location	Palo Alto
Subject	Mesa 4.0 Compiler Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>COMPILER40.BRAVO

This memo describes changes to the Mesa language and compiler that have been made since the last release (October 17, 1977). As usual, the list of compiler-related change requests closed by Mesa 4.0 will appear separately as part of the Software Release Description.

The language accepted by the Mesa 4.0 compiler has several significant extensions and a few minor changes. It features a process mechanism, enhanced arithmetic capabilities, long and base-relative pointers, and more general block structure.

Because of changes in symbol table and BCD formats, all existing Mesa programs must be recompiled. There are minor incompatibilities with Mesa 3.0 at the source level in the areas of signed/unsigned arithmetic and the scope of OPEN in an iterative statement. These incompatibilities should have negligible impact on existing programs. The syntax and semantics of declaring (but not calling) machine-coded procedures have changed substantially.

Page and section numbers in this update not otherwise qualified refer to the *Mesa Language Manual, Version 3.0.* The BNF descriptions of new or revised syntax follow the conventions introduced in that manual. For phrase classes used but not redefined here, see its Appendix D. Revisions of phrase class definitions are cumulative; except as noted, the appearance of "..." as an alternative indicates that an existing definition is being augmented. A definition without "..." supersedes any definition of the same phrase class in the manual.

Arithmetic

Mesa 4.0 supports double-precision integer arithmetic (type LONG INTEGER) and provides some help with floating-point computations (type REAL). In conjunction with these changes, the rules governing combination of signed and unsigned values have been more carefully defined (see the Appendix to this memo).

Syntax

```
PredefinedType ::=   INTEGER | CARDINAL | LONG INTEGER | REAL |
                     BOOLEAN | CHARACTER | STRING | UNSPECIFIED | WORD

Primary   ::=   ... | identifier [ Expression ] | LONG [ Expression ]
```

Signed and Unsigned Arithmetic

The rules governing the use of signed and unsigned representations in single-precision arithmetic have been reformulated. In previous versions of Mesa, conditions under which an operation was considered to overflow were not well defined. As a consequence, options such as overflow detection and reliable range checking were precluded. Mesa 4.0 does *not* offer these options, but it does remedy the defects in the language definition.

The precise rules governing signed/unsigned arithmetic are somewhat lengthy. They appear in an appendix to this memo with some background information explaining the motivation and philosophy. In their effect on the acceptance or rejection of source text, the new rules differ little from those in previous versions of Mesa; the main change is that CARDINAL - CARDINAL is now assumed to produce a result with unsigned (instead of unknown) representation (see Section 2.5.1, pages 10-12). Thus the immediate practical effect of the new rules is minor; however, programmers should read the appendix carefully so that their code will work correctly even when it becomes possible to request overflow and range checks.

> The effects of the new rules with respect to subtraction are worth emphasizing. If both operands have valid signed representations, the result is an INTEGER. If both have only unsigned representations, the result is a CARDINAL and is considered to overflow if the first operand is less than the second.

```
i: INTEGER;   m, n: CARDINAL;   s, t: [0..10);
i ← m-n;                  -- should be used only if it is known that m >= n
i ← IF m >= n THEN m-n ELSE -(n-m);      -- should be used otherwise
IF m-n > 0 ...            -- comparison (and subtraction) are unsigned
IF m > n ...              -- a better and safer test
IF s-t < 0 ...            -- comparison (and subtraction) are signed
```

Range Assertions

The new rules mentioned above assume that there are implicit conversion functions mapping CARDINAL to INTEGER and vice-versa. In both directions, the "conversion" amounts to an assertion that the argument is an element of INTEGER ∩ CARDINAL. The programmer can make such a *range assertion* explicit. If *S* is an identifier of a subrange type and *e* is an expression with compatible type *T*, the form *S*[*e*] has the same value as *e* and is additionally an assertion that *e* IN [FIRST[*S*∩*T*] .. LAST[*S*∩*T*]] is TRUE.

> Note that this is not equivalent to LOOPHOLE[*e*, *S*] but is an assertion about the range of a value that already has an appropriate type.

In Mesa 4.0, *such assertions must be verified by the programmer*. There is *not* an option to generate code that checks these assertions, whether implicit or explicit. An assertion can be used to control the assumed representation of a subexpression; otherwise, it is currently treated as a comment by the compiler.

> *Examples*
>
> INTEGER[*n*], *IndexType*[*i-j*]

Long Integers

Mesa 4.0 supports double-precision integers. There is a new predeclared type LONG INTEGER, values of which occupy two words (32 bits) of storage and range over $[-2^{31} .. 2^{31})$. There is no special denotation for LONG INTEGER constants. The type of any decimal or octal constant in $[2^{16} .. 2^{31})$ is LONG INTEGER; smaller constants are converted as required by context. The arithmetic operators +, -, *, /, MOD, MIN, MAX, (unary) - and ABS have double-precision extensions that perform the mapping

$$(\text{LONG INTEGER})^n \rightarrow \text{LONG INTEGER};$$

furthermore, LONG INTEGERs are ordered, and the relational operators =, #, <, <=, >, >= and IN have extensions that perform the mapping

$$(\text{LONG INTEGER})^n \rightarrow \text{BOOLEAN}.$$

Some fine points:

All LONG INTEGERs have a signed representation; the Mesa 4.0 language does not provide LONG CARDINAL.

Addition, subtraction, and comparison of LONG INTEGERs is fast; multiplication and division are done by software and are relatively slow.

In Mesa 4.0, it is not possible to declare a type that is a subrange of LONG INTEGER.

Mesa provides an automatic coercion from any single-precision numeric type (INTEGER, CARDINAL, etc.) to LONG INTEGER. This coercion is called *widening* and is discussed in more detail below. It is applied when necessary to match inherent and target types (e.g., in assignments). Also, if any operand of an arithmetic or relational operator is a LONG INTEGER, the double-precision operation is used. In most cases, widening of any shorter operands is automatic. Thus single- and double-precision quantities can be mixed freely within expressions to yield double-precision results.

The form LONG[*e*] explicitly forces the widening of any expression *e* with a single-precision numeric type. There are no automatic conversions from LONG INTEGER to any single-precision type (but see the *Mesa 4.0 System Documentation* for some standard procedures).

Widening of a single-precision constant is done at compile-time. Currently, no other arithmetic or relational operations on LONG INTEGERs are performed at compile-time, even if all operands are constant.

Widening of a single-precision expression is substantially more efficient if that expression has an unsigned representation.

Examples

```
i: INTEGER;
ii: LONG INTEGER;
c2: LONG INTEGER = 2;              -- a compile-time constant
c4: LONG INTEGER = c2*c2;          -- not a compile-time constant
ii ← 0; ii ← ii+1; ii ← i; ii ← (ii+i)/c2;      -- all valid
ii ← LONG[0]; ii ← (ii+LONG[i])/c2;             -- also valid (and explicit)
i ← ii; ii ← LONG[c4];                          -- invalid
```

Reals

A standard representation for floating-point values has not yet been chosen. Mesa 4.0 nevertheless provides some help with floating-point computation. It allows declaration and assignment of REAL values; furthermore, REAL expressions constructed using the standard infix operators (except MOD) are converted to sequences of procedure calls by the compiler.

A REAL value is assumed to occupy two words (32 bits) of storage. Beyond this, no assumptions are made about the representation of REALS. Users of real arithmetic must provide and install an appropriate set of procedures for performing the arithmetic operations (see the *Mesa 4.0 System Documentation* also). The procedures must be assignable to variables declared as follows:

```
FADD, FSUB, FMUL, FDIV: PROCEDURE [REAL, REAL] RETURNS [REAL];

FCOMP: PROCEDURE [REAL, REAL] RETURNS [INTEGER];
  -- returns a value that is: 0 if equal, negative if the first is less, positive otherwise

FLOAT: PROCEDURE [LONG INTEGER] RETURNS [REAL];
```

This scheme has the following consequences:

All other arithmetic operations (ABS, MIN, etc.) are fabricated from these primitives.

The source language provides no denotation for real constants, since the compiler does not know the internal format expected by the user-supplied procedures. As discussed below, values of type INTEGER or LONG INTEGER are automatically converted to type REAL at run-time; thus integer constants can appear in real expressions but will be reconverted each time the expression is evaluated.

Of course, implementers of floating-point packages are free to provide their own procedures for constructing REAL values from, e.g., octal constants, but a REAL "constant" currently cannot be a compile-time constant and cannot appear in a DEFINITIONS module (unless it is defined using a LOOPHOLE).

Examples

```
Two: REAL = 2;        -- means Two: REAL = FLOAT[2];
Half: REAL = 1/Two;   -- means Half: REAL = FLOAT[1]/Two;
Bug: REAL = 1/2;      -- means Bug: REAL = FLOAT[0];   (integer division)
```

Implicit Conversions

Conversions from INTEGER or CARDINAL to LONG INTEGER and from LONG INTEGER to REAL are called *widening*. Widening is automatic in the following situations:

An expression will be widened from its inherent type to match its target type (see Section 3.5, pages 37-39). This occurs in assignments and assignment-like contexts (such as record construction or extraction).

The types of the operands of an arithmetic operator will be balanced by widening until all match the type of the widest operand (but not further, even if the target type is wider).

In Mesa 4.0, automatic widening is not completely implemented in the following situations:

Operands of MIN and MAX will be widened to match the target type if one is well defined and otherwise to match the type of the first operand, but there is no general balancing.

> The endpoints in the right operand of IN will be widened to match the type of the left operand, but there is no general balancing.
>
> Expressions appearing in the arms of conditionals will be widened as required by the target type, but there is no general balancing when the target type is ill-defined.
>
> The expressions selecting the arms of a **SelectExpr** or **SelectStmt** will be widened to match the type of the selector, but the selector itself is never widened.

The following examples illustrate widening.

```
i, j: INTEGER;   ii: LONG INTEGER;   x: REAL;
ii ← i;   x ← i;   x ← ii;   x ← IF i < j THEN i ELSE ii
i + ii, ii + 1            -- added as LONG INTEGERS (for any target type)
i + x, x + 1, ii + x      -- added as REALS
x > i*j + ii   -- multiplied as INTEGERS, added as LONG INTEGERS, compared as REALS
```

The following are currently considered errors.

```
ii IN [i .. x)
(IF i < j THEN i ELSE ii) < x          -- ill-defined target for IfExpr
SELECT i FROM x => ...; > ii => ...; ENDCASE
```

In cases in which automatic widening is not implemented or does not give the desired result, the operator LONG or user-supplied procedure *FLOAT* can be used.

```
m, n: CARDINAL;   ii: LONG INTEGER;
ii ← m + n                    -- added as CARDINALS (overflow lost)
ii ← LONG[m + n]              -- ditto
ii ← LONG[m] + LONG[n]        -- added as LONG INTEGERS (overflow captured)
```

> A fine point: There are system-provided procedures for performing certain multiplication and division operations in which the operands and results do not all have the same precision. These procedures provide less expensive equivalents of, e.g., LONG[*m*]*LONG[*n*]. See the *Mesa 4.0 System Documentation.*

Long Pointers and Array Descriptors

Mesa 4.0 implements both long pointers and array descriptors with long pointers as base components. These pointers provide access to the entire virtual memory of the Dstar. For compatibility, long pointers are also supported on the Alto, but they do not provide any additional addressing capability.

Syntax

```
TypeConstructor  ::=  ... | LongTC
LongTC  ::=  LONG TypeSpecification
ArrayDescriptorTC ::=  DESCRIPTOR FOR TypeSpecification |
                       DESCRIPTOR FOR PackingOption ARRAY OF TypeSpecification
```

The type constructor LONG can be applied to INTEGER (discussed in the preceding section), any pointer type, or any array descriptor type. An attempt to lengthen any other type is an error.

The type constructor DESCRIPTOR FOR can be applied to any array type, including one designated by a type identifier. (This corrects an oversight in previous versions of Mesa). In addition, specification of an **IndexType** for the described array type can be omitted if its constructor follows immediately. In this case, a subrange of CARDINAL with zero origin and indefinite upper bound is assumed for the index type.

Long Pointers

A long pointer value occupies two words (32 bits) of storage. Long pointers are typically created by lengthening (short) pointers as described below. In particular, NIL is automatically lengthened to provide a null long pointer when required by context. The standard operations on pointers (dereferencing, assignment, testing equality, comparison if ORDERED, etc.) all extend to long pointers

> On the Dstar, NIL is lengthened by prefixing a word of zeros and thus has an MDS-independent representation. All other pointers are lengthened by adding the MDS base. Every pointer generated in this way is represented by an 8 bit field of zeros followed by a 24 bit virtual address. Long pointers with certain other formats can be created using LOOPHOLE and will be correctly dereferenced by the hardware. There is no normalization prior to operations on pointers, however, and such pointers will give anomolous results in, e.g., comparisons.
>
> On the Alto, pointers are lengthened by prefixing a word of zeros. In all dereferencing operations, that prefix is discarded (without a check for zero) and the remaining word is interpreted as the actual address.

Both automatic widening and explicit widening (using the operator LONG) are provided for pointer types as well as for numeric types. Widening an expression of type POINTER TO *T* produces a value of type LONG POINTER TO *T*, i.e., only the length attribute is changed by the widening. The rules and restrictions governing widening in Mesa 4.0 that are discussed in the preceding section apply equally to pointers.

The operator @ applied to a variable of type *T* produces a pointer of type LONG POINTER TO *T* if the access path to that variable itself involves a long pointer (other than the implicitly accessed MDS pointer) and of type POINTER TO *T* otherwise.

Limited pointer arithmetic continues to be supported in Mesa 4.0, but programmers are encouraged to use BASE and RELATIVE pointers (described in the next section) if the purpose of the arithmetic is simple relocation. If either operand in a pointer addition or subtraction is long, all operands are widened and the result is long.

> *Examples*
>
> ```
> R: TYPE = RECORD [f: T, ...];
> p, q: POINTER TO R;
> pp, qq: LONG POINTER TO R;
> pT: POINTER TO T;
> ppT: LONG POINTER TO T;
> ```
>
> The following are valid.

```
pp ← qq;  pp ← NIL;  pp ← p
pp = qq, pp = NIL, pp = q          -- long comparisons
pT ← @p.f;  ppT ← @pp.f
ppT ← @p.f                         -- pointer lengthened
pp+ii, pp+i, p+ii, pp-qq, pp-q     -- long results
```

The following are not valid.

```
pp = ppT                           -- type clash
p ← pp; pT ← @pp.f                 -- no automatic shortening
```

Long Array Descriptors

In a long array descriptor, the BASE component is a long pointer and the descriptor occupies three words (48 bits) of storage. All the standard operations on array descriptors (indexing, assignment, testing equality, LENGTH, etc.) extend to long array descriptors. The type of BASE[*desc*] is long if the type of *desc* is long.

Array descriptors are widened, either automatically or explicitly, according to the usual rules and restrictions. Long array descriptors are created by applying DESCRIPTOR[] to an array that is only accessible through a long pointer (other than the MDS pointer), by applying DESCRIPTOR[,,] to operands the first of which is long, or by widening a (short) array descriptor.

Examples

```
d: DESCRIPTOR FOR ARRAY OF T;
dd: LONG DESCRIPTOR FOR ARRAY OF T;
i, n: CARDINAL;
pp: LONG POINTER TO ARRAY [0..0) OF T;
x: T;
dd ← DESCRIPTOR[pp, 10, T];  dd ← d
x ← dd[i]
pp ← BASE[dd];  n ← LENGTH[dd]
```

Base and Relative Pointers

Mesa 4.0 deals more satisfactorily with base-relative pointers, i.e., pointers that must be *relocated* by adding some base value before they are dereferenced. Such pointers are useful for reducing the number of bits stored when objects can be identified by small offsets, and for dealing with collections of interlinked data items that are subject to relocation as entire aggregates.

Syntax

```
PointerTC        ::=  Ordered BaseOption POINTER OptionalInterval PointerTail
BaseOption       ::=  empty | BASE
TypeConstructor  ::=  ... | RelativeTC
```

RelativeTC ::= **TypeIdentifier** RELATIVE **TypeSpecification**

In a **PointerTC**, a nonempty **OptionalInterval** declares a subrange of a pointer type, the values of which are restricted to the indicated interval (and can potentially be stored in smaller fields). Normally, such a subrange type should be used only in constructing a relative pointer type as described below, since its values cannot span an MDS.

The **BaseOption** BASE indicates that pointer values of that type can be used to relocate relative pointers. Such values behave as ordinary pointers in all other respects with one exception: subscript brackets never force implicit dereferencing (see below). The attribute BASE is ignored in determining the assignability of pointer types.

A **RelativeTC** constructs a *relative pointer* or *relative array descriptor* type. The **TypeIdentifier** must evaluate to some (possibly long) pointer type which is the type of the base, and the **TypeSpecification** must evaluate to a (possibly long) pointer or array descriptor type.

Note that the form

LONG **TypeIdentifier** RELATIVE **TypeSpecification**

is always in error, since LONG cannot be applied to a relative type. The type designated by the **TypeSpecification** can be lengthened (to give a relative long pointer) using the form

TypeIdentifier RELATIVE LONG **TypeSpecification** .

Relative Pointers

In the following discussion, assume the declarations

BaseType: TYPE = BASE POINTER TO ...;
FullType: TYPE = POINTER TO ...;
RelativeType: TYPE = *BaseType* RELATIVE *FullType*;
base: BaseType;
offset: RelativeType;
p: FullType.

If *FullType* is some pointer, long pointer, or pointer subrange type, *RelativeType* is declared to be a relative pointer type. Values with type *RelativeType* are pointers that must be relocated, by adding some value of type *BaseType*, before they can be dereferenced. Also, relative pointers are never widened automatically. With respect to other operations (assignment, testing equality, comparison if *FullType* is ORDERED, etc.), relative pointers behave like pointers of type *FullType.* In particular, the amount of storage required to store such a pointer is determined by *FullType.* Note, however, that *RelativeType* and *FullType* are distinct types, incompatible with respect to, e.g., assignment and comparison.

Relocation of a relative pointer is specified by using subscript-like notation in which the type of the "array" is *BaseType* and that of the "index" is *RelativeType,* i.e., the absolute pointer is denoted by an expression with the form

base[*offset*]

This expression has the type *FullType* and the value LOOPHOLE[*base*]+*offset.* Note that *base*[*offset*] is not a variable; typical variable designators are *base*[*offset*]↑ or *base*[*offset*].*field.* (In addition, the usual rules for implicit dereferencing apply in, e.g., an **OpenItem**). Relocation prior to dereferencing is mandatory; *offset*↑, *offset.field,* etc. are errors.

Some fine points:

> The type of *base*[*offset*] is more precisely defined as follows: if *FullType* is a subrange pointer type, the subrange is discarded to obtain some type *T*; otherwise, *T* is *FullType*. If *FullType* is not a long pointer type but *BaseType* is, then the final type is LONG *T*; otherwise, it is *T*. In other words, the resulting type is long if either the base type or the relative type is.
>
> The declaration of a relative pointer does not associate a particular base value with that pointer, only a basing type. Thus some care is necessary if multiple base values are in use. Note that the final type of the relocated pointer is largely independent of the type of the base pointer; the relative pointer determines the type. Sometimes this observation can be used to help distinguish different classes of base values without producing relocated pointers with incompatible types.
>
> The base type must have the attribute BASE. Conversely, the attribute BASE always takes precedence in the interpretation of brackets following a pointer expression. Consider the following declarations:
>
> *p:* POINTER TO ARRAY *IndexType* OF ...;
> *q:* BASE POINTER TO ARRAY *IndexType* OF ... .
>
> The expression *p*[*e*] will cause implicit dereferencing of *p* and is equivalent to *p*↑[*e*]. On the other hand, *q*[*e*] is taken to specify relocation of a pointer, even if the type of *e* is *IndexType* and not an appropriate relative pointer type. In such cases, the array must (and always can) be accessed by adding sufficient qualification, e.g., *q*↑[*e*]; nevertheless, users should exercise caution in using pointers to arrays as base pointers.

Mesa 4.0 supplies no mechanisms for constructing relative pointers. It is expected that such values will be created by user-supplied allocators that pass their results through a LOOPHOLE or from pointer arithmetic involving LOOPHOLEs.

> *Examples*
>
> *p*↑ ← *base*[*offset*]↑
>
> *p* ← *base*[*offset*]   -- valid pointer assignment (but often unwise)
>
> The following are invalid.
>
> *p* ← *offset*;   *p*↑ ← *offset*↑
>
> *p*[*offset*]   -- *p* has incorrect type

Relative Array Descriptors

Relative array descriptor types are entirely analogous to relative pointer types; indeed, values of such types can be viewed as array descriptors in which the base components are relative pointers. Note the following:

> In the constructor of a relative array descriptor type, the **TypeSpecification** must evaluate to a (possibly long) array descriptor type.
>
> In the notation introduced above, a reference to an element of the described array has the form
>
> *base*[*offset*][*i*]

where *i* is the index of the element.

Relative array descriptors are constructed using the DESCRIPTOR operator. If *p* is *B* RELATIVE pointer, the form DESCRIPTOR[*p*, *n*, *T*] produces a value with type *B* RELATIVE DESCRIPTOR FOR ARRAY OF *T*. Also, the operators BASE and LENGTH can be applied to a *B* RELATIVE array descriptor; the former produces a *B* RELATIVE pointer.

Block Structure

The previous concepts of procedure body and compound statement have been merged. A *block* can appear anywhere a statement is acceptable and can introduce new identifiers with scope smaller than an entire procedure (or module) body. In addition, catch phrases and exit labels can now appear at the outermost level of a procedure body.

The syntax for declaring procedures with bodies expressed in machine code has also been revised (in anticipation of more general inline procedures). The corresponding semantics are machine dependent and are not specified here.

Syntax

```
ModuleBody  ::=  Block

ProcedureBody  ::=  Block

Statement  ::=  ... | Block | ...         -- replaces CompoundStmt

Block ::=   BEGIN
            OpenClause
            EnableClause
            DeclarationSeries
            StatementSeries
            ExitsClause
            END

EnableClause ::=     empty |
                     ENABLE CatchItem ; |
                     ENABLE BEGIN CatchSeries END ; |
                     ENABLE BEGIN CatchSeries ; END ;

MachineCode   ::=  MACHINE CODE BEGIN InstructionSeries END

InstructionSeries ::=   empty | ByteList |
                        ByteList ; InstructionSeries

ByteList  ::=  Expression | ByteList , Expression
```

In addition, the phrase classes **Body**, **CompoundStmt** and **MachineCodeTC** are deleted.

During the execution of a Mesa program, frames are allocated at the procedure and module level only. Any storage required by variables declared in an internal **Block** (one used as a **Statement**) is allocated in the frame of the smallest enclosing procedure or module. When such internal blocks are disjoint, the areas of the frame used for their variables overlay one another.

The scopes of identifiers introduced in the various components of a block are summarized by the following diagram, where indentation is used to show the scope of each phrase:

```
BEGIN
OpenClause
    EnableClause
        DeclarationSeries
            StatementSeries
  ExitsClause
END
```

Note that any newly declared identifiers are visible only in the **DeclarationSeries** and **StatementSeries** of the block. Any exit labels are visible within the **EnableClause** (as well as the more deeply indented constructs); on the other hand, any catch phrase in the **EnableClause** is not enabled within the **ExitsClause.** If the **Block** is used as a module or procedure body, the parameters and results are visible throughout the **Block.** Thus it is possible to open records designated by parameters or to assign return values within an **ExitsClause** (but the assigned values cannot involve internally declared variables).

A CONTINUE statement appearing in the **EnableClause** of a **Block** causes exit from that block. A similarly placed RETRY statement causes reexecution of the block. In the latter case, any initializing values in the **DeclarationSeries** are recomputed.

Note that an optional semicolon can now terminate a **CatchSeries** in an **EnableClause.**

Nested Block Structure

With the introduction of blocks, procedure bodies can appear where they were syntactically prohibited in previous versions of Mesa. Special rules apply to the inheritance of scope when a procedure body is declared within the **DeclarationSeries** or (with nesting) within the **StatementSeries** of a **Block.** Within the inner procedure body:

Identifiers made visible by the **OpenClause** remain visible (unless redeclared).

Catch phrases in the **EnableClause** are not inherited and not enabled.

Identifiers declared in the **DeclarationSeries** remain visible (unless redeclared).

Jumps to labels in the **ExitsClause** are prohibited.

Assume the following skeletal declaration:

```
Outer: PROCEDURE [...] =
  BEGIN
  ENABLE s => Handler[];
  ...
  Inner: PROCEDURE [...] = BEGIN ... END;
  ...
  EXITS
    Label => ...
  END
```

If the signal *s* is raised in an instance of *Inner*, *Handler* is not invoked there. *Handler* will, of course, be invoked eventually if *s* propagates to the enclosing instance of *Outer*. (This noninheritance rule prevents double execution of handlers in such situations.) In Mesa 4.0, the statement GO TO *Label* is considered an error within the body of *Inner*.

Iterative Statements

For consistency with blocks, the scope rules for iterative statements have been revised slightly. In addition, a new statement form that terminates one iteration of the loop body and initiates the next has been added.

Syntax

```
Statement   ::=    ... | LoopCloseStmt

LoopStmt ::= LoopControl
                 DO
                 OpenClause
                 EnableClause
                 StatementSeries
                 LoopExitsClause
                 ENDLOOP

LoopCloseStmt   ::=   LOOP
```

The scopes of identifiers introduced in the various components of a loop are summarized by the following diagram (cf. **Blocks**):

```
LoopControl
  DO
  OpenClause
      EnableClause
        StatementSeries
    LoopExitsClause
  ENDLOOP
```

In previous versions of Mesa, the scope of the **OpenClause** excluded the **LoopExitsClause.** As in the case of blocks, any exit labels are visible within the **EnableClause**, and any catch phrase in the **EnableClause** is not enabled within the **ExitsClause.**

The statement LOOP can appear only within the body of an iterative statement. Executing it terminates the current iteration of the smallest enclosing **LoopStmt**, after which the **LoopControl** is updated/reevaluated and, if appropriate, the next iteration is started. Thus the construct

```
DO ... LOOP ... ENDLOOP
```

is an abbreviation for

```
DO
  BEGIN
  ... GO TO Skip ...
  EXITS   Skip => NULL;
  END
ENDLOOP .
```

Included Identifier Lists

In Mesa 4.0, an item in the DIRECTORY clause can explicitly list the identifiers eligible for inclusion from a designated module. Such *included identifier lists* serve as compiler-checked (but programmer-maintained) lists of intermodular connections and dependencies.

Syntax

```
IncludeList  ::=  IncludeItem | IncludeList , IncludeItem

IncludeItem ::=        identifier : FROM FileName |
                       identifier : FROM FileName USING [ IdList ]
```

If the USING clause is absent, the item's **identifier** has all the properties and uses described in Sections 7.2.1 and 7.2.2. The only effect of a USING clause is to enumerate (and potentially restrict) the set of identifiers made accessible to the including module. Use of the **identifier,** either within an OPEN clause or for explicit qualification, makes visible only those identifiers in the **IdList.**

Some fine points.

Only identifiers declared in the **DeclarationSeries** that is part of the **ModuleBody** of the included module are mentioned in the **IdList**; in particular, neither the included module's own identifier nor identifiers of record fields, enumeration constants, etc. appear in this list.

Each identifier appearing in the **IdList** must be defined in the module designated by the **IncludeItem.**

A warning is generated for each identifier appearing in the **IdList** but not used explicitly in the including module. Identifiers used only implicitly (to describe attributes of explicitly included identifiers) should not be listed.

The **IdList** restricts the set of identifiers available for inclusion from a module. It does not restrict export into an included interface. The identifier of an exported item should not appear in the list unless the intention is to reference a different item with the same name through an imported instance of the interface.

The following example assumes the declaration of *SimpleDefs* appearing on page 92.

```
DIRECTORY
  SimpleDefs: FROM "simpledefs" USING [Range, PairPtr];
Example: PROGRAM =
  BEGIN
  First: PROCEDURE [p: SimpleDefs.PairPtr] RETURNS [SimpleDefs.Range] =
    BEGIN
    RETURN [IF p = NIL THEN 0 ELSE p.first]
    END;
  END.
```

Note that *Pair* does not appear in the included identifier list (because it is only referenced implicitly, through the definition of *PairPtr*), nor does *first* (because it is declared in a record, not in the body of *SimpleDefs* itself). Any reference to *SimpleDefs.limit* would be an error in this example.

Processes

Mesa 4.0 supports a process mechanism in which processes are created by forking to procedures and are synchronized by entry to monitors. Most of the information about the semantics and intended usage of Mesa processes appears in the *Mesa 4.0 Process Update* (henceforth cited as *Process*). The *Mesa 4.0 Change Summary* contains a complete example, and additional examples appear in the *Process* document. This section summarizes the syntax and deals with a few linguistic details.

Syntax

```
PredefinedType  ::=  ... | MONITORLOCK | CONDITION

ProgramTC ::=        ... |
                     MONITOR ParameterList ReturnsClause LocksClause

LocksClause ::=      empty |
                     LOCKS Expression |
                     LOCKS Expression USING identifier : TypeSpecification

TypeConstructor  ::=  ... | ProcessTC

ProcessTC  ::=  PROCESS ReturnsClause

Declaration ::=      IdList : Access EntryOption TypeSpecification Initialization ; |
                     IdList : Access TYPE = Access TypeSpecification ;

EntryOption  ::=  empty | ENTRY | INTERNAL

RecordTC  ::=  MonitoredOption MachineDependent RECORD [ VariantFieldList ]

MonitoredOption  ::=  empty | MONITORED

Statement  ::=  ... | WaitStmt | NotifyStmt | JoinCall

Expression  ::=  ... | ForkCall | JoinCall

WaitStmt  ::=  WAIT Variable OptCatchPhrase

NotifyStmt  ::=  NOTIFY Variable | BROADCAST Variable

ForkCall  ::=  FORK Call

JoinCall  ::=  JOIN Call
```

Forking and Joining

Processes are created and destroyed by FORK and JOIN operations. If procedure P has type PROCEDURE T RETURNS T', then the expression FORK P[...] produces a *process handle h* with type PROCESS RETURNS T'. JOIN requires a process handle as its operand. The form JOIN h produces an argument record of type T' (or stands as a statement if the **ReturnsClause** is empty). As type mappings,

FORK: PROCEDURE T RETURNS $T' \times T \rightarrow$ PROCESS RETURNS T'

JOIN: PROCESS RETURNS $T' \rightarrow T'$.

Some fine points:

A catch phrase can be attached to a FORK or JOIN (by specifying it in the **Call**).

Unlike an ordinary procedure call, a FORK returns a value with some process type (not a record type), and that value cannot be discarded by writing an empty extractor.

Monitored Modules

A **ProgramTC** containing MONITOR can be used only in a **ModuleHead** to specify the type of a program module. The **LocksClause** provides additional information about the program body and is not part of the module's type. If a monitor is to be exported, the correct type for the interface item in the DEFINITIONS module is obtained by replacing MONITOR by PROGRAM and deleting the **LocksClause**.

Synchronization of processes is based upon variables with the system-defined types MONITORLOCK and CONDITION. A distinguished MONITORLOCK with the identifier *LOCK* is implicitly declared in the global frame of any MONITOR with an empty **LocksClause.** If the **MonitoredOption** MONITORED appears in the definition of a record type, each record of that type similarly contains an implicitly declared and distinguished MONITORLOCK with identifier *LOCK*. Lock and condition variables can also be declared explicitly, but any MONITORLOCK so declared is not distinguished, even if its identifier is *LOCK* (see below).

When a variable with type MONITORLOCK or CONDITION is a component of a (local or global) frame, it is initialized automatically when the frame is created. In all other cases, a system procedure must be called to establish appropriate initial values (see *Process*, Section A.6).

Entry Procedures

The **EntryOption** ENTRY can appear only in a declaration within a monitor; when it does, the **TypeSpecification** must evaluate to a procedure type and the initialization must specify a procedure body (**Block**). Note that ENTRY does not imply PUBLIC, but PUBLIC ENTRY is a permissible (and common) combination.

Entry into a monitor through an ENTRY procedure is protected by a monitor lock. The identity of that lock is determined by the declaration of the monitor. If the **LocksClause** is empty, entry is controlled by the distinguished variable *LOCK*. Otherwise, the **LocksClause** must designate a variable with type MONITORLOCK, a record containing a distinguished lock field, or a pointer that can be dereferenced (perhaps several times) to yield one of the preceding. There are two cases (see *Process*, Section A.4.2):

> If the USING clause is absent, the monitor is a multi-module one. The lock is located by evaluating the LOCKS expression in the context of the monitor's main body; i.e., the monitor's parameters, imports, and global variables are visible, as are any identifiers made accessible by a global OPEN. Evaluation occurs upon entry to, and again upon exit from, the ENTRY procedure (and for any internal WAITs). The location of the designated lock can thus be affected by assignments within the procedure to variables in the LOCKS expression. To avoid disaster, it is essential that each reevaluation yield a designator of the same MONITORLOCK.
>
> If the USING clause is present, the monitor is an object monitor. The lock is located as above with one exception: any occurrence of the identifier declared in the USING clause is bound to that argument of the ENTRY procedure having the same identifier and a compatible type. If there is no such parameter, the ENTRY is in error. The same care is necessary with respect to reevaluation; to emphasize this, the distinguished argument is treated as a read-only value within the body of the ENTRY procedure.

The following examples illustrate the selection of locks.

```
R: TYPE = MONITORED RECORD [...];
RR: TYPE = RECORD [..., specialLock: MONITORLOCK, ...];

M1: MONITOR =
  BEGIN
  -- LOCK: MONITORLOCK implicitly declared here
  P1: PUBLIC ENTRY PROCEDURE [...] =
    BEGIN  -- locks LOCK -- ... END;
  END.
```

```
M2a: MONITOR [p: POINTER TO POINTER TO R] LOCKS p =
  BEGIN
  P2: PUBLIC ENTRY PROCEDURE [...] =
    BEGIN  -- locks p↑↑.LOCK -- ... END;
  END.

M2b: MONITOR [p: POINTER TO POINTER TO RR] LOCKS p↑↑.specialLock =
  -- specification of the lock is mandatory here
  BEGIN
  P2: PUBLIC ENTRY PROCEDURE [...] =
    BEGIN  -- locks p↑↑.specialLock -- ... END;
  END.

M3: MONITOR LOCKS p USING p: POINTER TO R =
  BEGIN
  P3: PUBLIC ENTRY PROCEDURE [p: POINTER TO R, ...] =
    BEGIN  -- locks p.LOCK -- ... END;
  END.
```

Signals require special attention within the body of an ENTRY procedure. A signal raised with the monitor lock held will propagate without releasing the lock and possibly invoke arbitrary computations. For errors, this can be avoided by using the RETURN WITH ERROR construct described in the next section.

When an instance of an ENTRY procedure is to be destroyed because of a remote exit from a catch phrase (unwinding), the lock should also be released. In Mesa 4.0, it is the programmer's responsibility to determine if unwinding is possible and, if so, to provide a catch phrase for UNWIND that restores the monitor invariant. Code to actually release the monitor lock is automatically appended to the outermost enabled catch phrase for UNWIND in an ENTRY procedure. That catch phrase can have a NULL body if no other cleanup actions are required.

Internal Procedures

The **EntryOption** INTERNAL can appear only in a declaration within a monitor; when it does, the **TypeSpecification** must evaluate to a procedure type and the initialization must specify a procedure body (**Block**). Note that INTERNAL does not imply PRIVATE (if the default is PUBLIC), but PUBLIC INTERNAL is considered an improper combination of attributes (warning only).

A call of an INTERNAL procedure is permitted only within an ENTRY procedure or another INTERNAL procedure. Forking to an INTERNAL procedure is never allowed. An INTERNAL procedure can safely access monitored data and can perform WAIT, NOTIFY and BROADCAST operations. A WAIT operation implicitly references the monitor lock; thus an INTERNAL procedure of an object monitor that contains a WAIT must have a parameter designating the locked object as described above.

Some fine points:

> In Mesa 4.0, the attribute INTERNAL is associated with a procedure's body, not its type. Thus INTERNAL cannot be specified in a DEFINITIONS module, and checks on intermodular calls of internal procedures are not performed (except for the PUBLIC INTERNAL warning). Also, the attribute INTERNAL is lost when a procedure value is assigned to a variable or passed as an argument of a procedure. Such assignments should be done with caution.

Signals raised by INTERNAL procedures require special consideration. When the construct RETURN WITH ERROR is executed within an INTERNAL procedure, the monitor lock is *not* released prior to signal propagation.

Wait and Notify

Only ENTRY and INTERNAL procedures within a monitor can contain WAIT, NOTIFY and BROADCAST statements.

Error Returns

It is possible to delete a procedure instance before raising an error detected by that procedure. Within an ENTRY procedure of a monitor, the monitor lock is released before the error is raised. (Such procedures are expected to be the primary users of this facility.)

Syntax

```
ReturnStmt  ::=  ... | RETURN WITH ERROR Call
```

Consider the following skeletal code:

```
Failure: ERROR [...] = CODE;

Proc: ENTRY PROCEDURE [...] RETURNS [...] =
   BEGIN
   ENABLE UNWIND => ...;
   ..
   IF cond1 THEN ERROR Failure[...];
   IF cond2 THEN RETURN WITH ERROR Failure[...];
   ...
   END;
```

Execution of the construct ERROR *Failure*[...] raises a signal that propagates until some catch phrase specifies an exit. At that time, unwinding begins; the catch phrase for UNWIND in *Proc* is executed and then *Proc*'s frame is destroyed. Within an entry procedure such as *Proc*, the lock is held until the unwind (and thus through unpredictable computation performed by catch phrases).

Execution of the construct RETURN WITH ERROR *Failure*[...] releases the monitor lock and destroys the frame of *Proc* before propagation of the signal begins. Note that the argument list in this construct is determined by the declaration of *Failure* (not by *Proc*'s RETURNS clause). The catch phrase for UNWIND is not executed in this case. The signal *Failure* is actually raised by the system, after which *Failure* propagates as an ordinary error (beginning with *Proc*'s caller).

Multiword Constants

Record and array constructors in which all components are themselves constant define so-called *multiword constants*. Such constants are now constructed during compilation and can be encoded within Mesa symbol tables. This has the following consequences:

A declaration equating an identifier to a multiword constant (but not to a string literal) can appear in a DEFINITIONS module, and the constant value thereby becomes available to users of that module.

> Constant selection from such values (by field selection or by indexing with a constant subscript) is also done during compilation.

Furthermore, if an identifier is equated to a multiword constant in a program module, exactly one copy of that constant appears in the code, and its components can be read (using, e.g., a computed index) directly from the code segment. This allows table driven programming in which the tables are automatically swapped.

> A fine point: A packed array or an array of multiword elements is currently copied into a data area each time one of its elements is accessed.

The following declarations define multiword constants and can appear in a DEFINITIONS module.

```
Ident: RECORD [version: CARDINAL, id: CHARACTER, released: BOOLEAN] =
   [1, '#, FALSE];

Powers: ARRAY [1..4] OF CARDINAL = [2, 4, 8, 16];

Nonsense: CARDINAL = IF Ident.released THEN Ident.version ELSE Powers[2];
```

The following are not compile-time constants in Mesa 4.0.

```
"abc",    ("abc")[1].
```

Miscellaneous Language Changes

Local Strings

The body of a string literal is ordinarily placed in the global frame of the module in which the literal appears. Pointers to that body (the actual STRING values) can then be used freely with little danger that the body will move or be destroyed. Unfortunately, this scheme can consume substantial amounts of space in the (permanent and unmovable) global frame area.

If a string literal is followed by 'L (e.g., "abc"L), a copy of the string body is moved from the code to the local frame of the smallest enclosing procedure whenever an instance of that procedure is created. As a corollary, the space is freed and the string body disappears when the procedure returns. Thus it is important to insure that pointers to local string literals are not assigned to STRING variables with lifetimes longer than that of the procedure. Programmers should avoid using local string literals until performance tuning is necessary (except perhaps in calls of straightforward output procedures).

Character Arithmetic

The following arithmetic operations are now defined for values of type CHARACTER:

```
CHARACTER + INTEGER → CHARACTER
INTEGER + CHARACTER → CHARACTER
CHARACTER - INTEGER → CHARACTER
CHARACTER - CHARACTER → INTEGER.
```

Other arithmetic operations do not allow characters as operands, and values of type INTEGER and CHARACTER cannot be cross-assigned.

Examples

```
c: CHARACTER;
d: INTEGER ← c - '0;                -- consider a translation table instead
```

```
IF c IN ['a..'z] THEN c ← 'A + (c-'a)
```

Selections

More general expressions are allowed to label arms of selections when there is no initial relational operator.

```
Test ::= Expression | RelationTail            -- formerly Sum | RelationTail
```

Example

```
SELECT TRUE FROM
 i > 0, j > 0 => s1;              -- previously required (i > 0), (j > 0)
 p AND q      => s2;              -- previously required (p AND q)
 k > 0 OR q   => s3;
 ...
 ENDCASE      => sN
```

This is equivalent to (and perhaps more readable than)

```
IF i > 0 OR j > 0 THEN s1
ELSE IF p AND q THEN s2
ELSE IF k > 0 OR q THEN s3
...
ELSE sN  .
```

Discriminations

Previous versions of Mesa have required that all adjectives labeling an arm of a discrimination name identically structured variants; in Mesa 4.0, this restriction is lifted. If, however, the labels identify more than one variant structure, the record is not considered to be discriminated within that arm and only the common fields are visible (cf. ENDCASE).

Example

```
R: TYPE = RECORD [
  v: T,
  variant: SELECT tag:* FROM
    red, pink => [vRP: T],
    green => [vG: T],
    yellow => [vY: T],
    ENDCASE];

r: R;

WITH x: r SELECT FROM
 red, pink => ...;         -- x.v and x.vRP accessible
 green, yellow => ...;     -- only x.v accessible
 ENDCASE => ...;           -- only x.v accessible
```

Mesa 4.0 also allows computed or overlaid variant records to be compared without discrimination if all variants have the same length. As usual, caution is advised; two records interpreted as different variants can be represented by the same bit pattern when computed tags are used.

Compilation Options

The following compiler options have been added; they are controlled by switches in the usual way:

Switch	*Option Controlled*
alto	Generating code for an Alto or Dstar
run	Terminating compilation by running another program
sort	Sorting global variables and entry indices

The Alto/Dstar switch primarily affects the treatment of long pointers in the object code.

The run switch specifies running another program without returning to the executive. This switch is primarily intended for use in command files. The file name preceding the switch specifies the program to be run. The file is assumed to contain a program requiring standard (Bcpl) microcode if the file name's extension is ".RUN" and requiring Mesa microcode otherwise. The default extension is ".IMAGE". Prior to execution of the specified program, a new command file (COM.CM) is constructed containing the full file name plus any switches following the 'r. In the case of command-line input, the remainder of the command line is also appended.

The sorting switch has been added in anticipation of tools that will expedite updating a module in a configuration or subsystem when the new and old versions of the object code are sufficiently similar. When sorting is suppressed, the assignment of global frame offsets and entry indices depends only upon order of declaration in the source text; on the other hand, the generated code is likely to be somewhat less compact.

> Sorting of local variables is not suppressed. Unless a module uses global variables extensively, the object code expansion is unlikely to exceed ?%.

The defaults are to generate code for an Alto, to terminate by returning to the executive, and to sort global variables and entry points.

Internal Changes

The following internal changes are mentioned for completeness; see the *Mesa 4.0 System Update* for more information.

Main Body Procedure

The main body of a module is now executed in a separate local frame. Note however, that any storage required by blocks or local strings in the main body is still allocated in the global frame.

External Links

External links (for imported procedures, signals or frames) are now stored and indexed backwards from the global frame base or code base (as selected by a binding/loading option).

Alto/Mesa Microcode

Both the instruction set and the opcode numbers have changed substantially.

Frame Allocation

Instructions for allocating and freeing frames are now implemented in microcode; this greatly inceases the speed of any transfer involving a large argument record.

Distribution:
  Mesa Users
  Mesa Group

Appendix: Signed and Unsigned Arithmetic

Background and Overview

In any implementation of Mesa, the number of bits available for representing a value of a given type is fixed. Each numeric type of the language thus is restricted to some subrange of $\underline{Z}$, the set of integers as understood in mathematics. The following types, corresponding to the indicated subranges, are built into the language:

INTEGER	$[-2^{N-1} .. 2^{N-1})$	-- "signed integers"
CARDINAL	$[0 .. 2^{N})$	-- "unsigned integers"
LONG INTEGER	$[-2^{2N-1} .. 2^{2N-1})$	-- "double-precision integers"

Here N is the word length of the machine (N=16 for the Alto and Dstar). The programmer can also declare types that are themselves subranges of CARDINAL or INTEGER (but not LONG INTEGER), e.g., *T*: TYPE = [0..10).

Let v, x, and y be variables with numeric subrange types. In principle, execution of the assignment $v \leftarrow x \ominus y$ proceeds as follows:

The values of x and y are taken as elements of $\underline{Z}$.

Those values are combined using some function f that defines the operator $\ominus$ over $\underline{Z}$ and produces a result $f(x,y)$, also in $\underline{Z}$.

If the result is in the subrange of $\underline{Z}$ spanned by the type of v, $f(x,y)$ is assigned to v; otherwise a *range failure* occurs.

Unfortunately, the underlying hardware does not provide the function f but only a partial function f' over some subrange of $\underline{Z}$ with the property that f' agrees with f wherever both are defined; f' is said to *overflow* (or *underflow*) elsewhere. In fact, the hardware generally provides a family of partial functions related to f, one each for INTEGER, CARDINAL, and LONG INTEGER. The operator $\ominus$ thus is generic at the hardware level, and the compiler must choose the appropriate partial function for preserving the abstraction being used by the programmer (or for detecting its breakdown). The choice is made by considering an attribute of each operand called its *representation.*

If the type of any operand is LONG INTEGER, the rule is simple: all other operands are converted to LONG INTEGER and the result is computed in that domain. For INTEGERs (with *signed* representation), CARDINALs (with *unsigned* representation) and subrange types such as *T* (with both representations), the issues are more subtle. Some operators, such as the relationals, are clearly generic and were recognized as such in previous versions of Mesa. Many other operators produce the correct result modulo 2^N (i.e., the "right" bit pattern) no matter what representation is assumed; the representation affects only the definition of overflow.

Examples (N=16)

The bit patterns representing -1 and 177777B are identical, but (177777B > 1) is TRUE while (-1 > 1) is FALSE . Also, (-1 + 1) = 0 and there is no overflow, but (177777B + 1) cannot be represented as an unsigned number.

In a critique of Mesa [Wirth], Niklaus Wirth has argued strongly that the language should be defined so that the overflow condition can always be specified. Note that this is a necessary condition for implementing reliable range checking (also advocated by Wirth) but not a sufficient one. Mesa 4.0 *does not* provide options for overflow detection or range checking but *does* revise the language definition so that future versions can offer such options.

While we have found no rules for mixing signed and unsigned values that are entirely satisfactory, we believe that those presented in the following section are reasonably unobtrusive, compatible with existing code and relatively free of surprises.

Signed and Unsigned Numbers

This section discusses the rules now used by Mesa for choosing between signed and unsigned versions of operations on single-precision numbers. The new rules assume that there are conversion functions performing the following mappings:

CARDINAL → INTEGER

INTEGER → CARDINAL .

In both cases, the "conversion" amounts to an assertion that the argument is an element of INTEGER ∩ CARDINAL. The programmer can also make such a range assertion explicit as described in the main body of this memo. In Mesa 4.0, *such assertions must be verified by the programmer*. There is *not* an option to generate code that checks these assertions, whether implicit or explicit, or code that detects overflow in arithmetic operations.

For each of the operators +, -, *, /, MOD, MIN, and MAX, there are two single-precision operations, mapping as follows:

INTEGER^n → INTEGER	(signed arithmetic)
CARDINAL^n → CARDINAL	(unsigned arithmetic).

Similarly, there are two operations for each of the operators =, #, <, <=, >, >= and IN:

INTEGER^n → BOOLEAN	(signed comparisons)
CARDINAL^n → BOOLEAN	(unsigned comparisons).

There are no operations upon mixed representations in any case; thus all operands must be forced to have some common representation. The arithmetic operators also propagate that same representation to the result.

> A possible surprise is that CARDINAL is taken to be closed under subtraction; i.e., *m-n* is considered to overflow if *m* and *n* are CARDINALS and *m* < *n*.

For any arithmetic expression, the *inherent representations* of the operands and the *target representation* of the result are used to choose between the signed and unsigned operations (cf. the discussion of inherent and target types, Section 3.1, pages 37-39).

> The target type determines the target representation. The target type is derived from the type of the variable to which an expression is to be assigned, from a range assertion applied to a subexpression, etc. If all valid values of the target type are nonnegative, the target representation is *unsigned*; otherwise, it is *signed*. The arithmetic operators listed above propagate target representations unchanged to their operands, but the target representation of an operand of a relational operator is undefined. Thus each (sub)expression has at most one target representation.
>
> The inherent representation of a primary is determined by its type (if a variable, function call, etc.), by its value (if a compile-time constant), or explicitly (if a range assertion). Possible inherent representations are signed and unsigned; in addition,

compile-time constants in [0 .. 2^{N-1}) and primaries with types that are subranges of INTEGER ∩ CARDINAL are considered to have *both* inherent representations. Inherent representations of operands are propagated to results as described below.

The basic idea is that generic operations are disambiguated first by the inherent representations of their operands, next by the target representation, and finally by a default convention. If the operation cannot be disambiguated in any of these ways, the expression is considered to be in error. The exact rules follow:

> If the operands have exactly one common inherent representation, the operation defined for that representation is selected (and the target representation is ignored).
>
> If the operands have no common inherent representation but the target representation is well-defined, the operation yielding that representation is chosen, and each operand is "converted" to that representation (in the weak sense discussed above).
>
> If the operands have both inherent representations in common, then
>   if the target representation is well-defined it selects the operation;
>   otherwise the signed operation is chosen.
>
> If the operands have no representation in common and the target representation is ill-defined, the expression is in error.

In all cases, the inherent representation of the result is determined by the mapping performed by the selected operation.

The unary operators require special treatment. Unary minus converts its argument to a signed representation if necessary and produces a signed result. ABS is a null operation (with warning message) on an operand with an unsigned representation, and it yields an unsigned representation in any case. The target representation for the operand of LONG (or of an implied widening operation) is unsigned.

Examples

Assume the following declarations:

i, j: INTEGER; *m, n:* CARDINAL; *s, t:* [0..77777B]; *b:* BOOLEAN

The statements on each of the following lines are equivalent.

```
i ← m+n; i ← INTEGER[m+n]                    -- unsigned addition
i ← j+n; i ← n+j; i ← j+INTEGER[n]           -- signed addition
i ← s+t; i ← INTEGER[s]+INTEGER[t]           -- signed (overflow possible)
n ← s+t; n ← CARDINAL[s]+CARDINAL[t]         -- unsigned (overflow impossible)
s ← s-t; s ← CARDINAL[s]-CARDINAL[t]         -- unsigned (overflow possible)
b ← s-t > 0; b ← INTEGER[s]-INTEGER[t] > 0   -- signed (overflow impossible)
i ← -m;   i ← -INTEGER[m]
i ← m+n*(j+n);   i ← INTEGER[m] + (INTEGER[n]*(j+INTEGER[n]))
n ← m+n*(j+n);   n ← m + (n*(CARDINAL[j]+n))
```

```
i ← m+n*(s+n);   i ← INTEGER[m+(n*(CARDINAL[s]+n))]
b ← s IN [t-1 .. t+1];   b ← INTEGER[s] IN [INTEGER[t-1] .. INTEGER[t+1]]
FOR s IN [t-1 .. t+1] ...;   FOR s IN [CARDINAL[t-1] .. CARDINAL[t+1]] ...
```

The following statements are incorrect because of representational ambiguities.

```
b ← i > n;   b ← i+n IN [s .. j]
SELECT i FROM m => ...;  t => ...; ENDCASE
```

Both the following are legal and assign the same bit pattern to *i*, but the first overflows if *m<n*.

```
i ← m-n;   i ← IF m >= n THEN m-n ELSE -(n-m) .
```

Reference

Wirth, N. *On the peaceful coexistence of integers and cardinals*, Xerox PARC, 29 June 1977.