The Modula-2+ User's Manual

DRAFT

18 April 1986

Table of Contents

Introduction

This is a programmer's reference manual, not a language definition, not a compiler-writer's manual, not a programmer's tutorial. It's meant to help you write programs in Modula-2+.

I'm making no attempt to avoid circularity. I do not expect to have any linear readers, any readers who go from page 1 to page n. Therefore I feel free to use words and phrases on page 1 that don't get defined till ''later'' in the manuscript.

*

This document doesn't (by any means) contain everything you need to know about programming in Modula-2+ at SRC.

Lyle is starting to work on putting together the Public Interfaces Manual. The arbitrary dividing line between this manual and the Public Interfaces Manual is IMPORT. If you have to import an object to use it, the central documentation for that object is not supposed to be here, even if the object is actually known to the compiler -- even if the interface from which you import it is a dummy interface, like `System`. This manual and the Public Interfaces Manual can and in the future will refer to each other freely. I talk about `System.Word` and `Text.T` whenever I need to, but I don't take responsibility for documenting them here.

In this manual, ''supplied'' means not requiring import.

*

To print the manpage for the compiler, type:

```
man mp | enscript -Papn
```

where ''`n`'' in ''`-Papn`'' is your floor number.

To print the document for Loupe, the Modula-2+ symbolic debugger, type:

```
printdoc loupe
```

There's no good overview yet of procedures for releasing public interfaces and tools, but Sheng-Yang and Dave Redell have started working on one.

*

I sometimes slip and say `M.def` for ''definition module `M`'' and `M.mod` for ''implementation module `M`'' or ''main module `M`.'' But it's the module name, not the file name, that counts. In fact the module name and the file name can be completely different.

*

Typography: This document is not well typeset. Until somebody invents a way of doing good typesetting automatically, good typesetting in advance of

good words is a waste of time. After a while the words may or may not get good enough to warrant spending time on the typography.

*

Acknowledgements:

It would be folly to contemplate a project of this kind without Scribe to supply paginated table of contents, automatic maintenance of cross-reference, automatic updating of index.

Cast of thousands ... This is a joint effort by all the m2+ programmers at SRC. Paul and Violetta. JDD. Greg. Butler. Roy. Lyle. Jim. JRE. TomR. Karen. Sheng-Yang. Especially Mark.

And me? Here's a passage from *Mr. Scobie's Riddle*, by Elizabeth Jolley. The speaker is the dotty writer Miss Hailey, who wants to use in her next book an event from Mr. Scobie's life:

> "Ah," she put her hands up to her head. "The endless rewriting, the linking of events, the approach to drama, the dramatic moment and the resolution. But," she paused as if at a dramatic moment in her own life, "it is so rewarding, the rewriting, the choosing of *le mot juste, la petite phrase*. You're absooty sure you don't mind?"
>
> Mr. Scobie replied, in spite of some uneasiness, that he did not mind.

1. Tokens and Comments

A "token" is a keyword, an identifier, a literal number, a string, or an operator or delimiter. [[Why is a comment not a token too?]]

Blanks, tabs, and newlines can't occur within tokens except in quoted literal strings. There you can use all three, but you have to escape newlines. Elsewhere, blanks, tabs, and newlines are ignored except to separate two consecutive tokens.

1.1. Keywords

ABS	EXCEPTION	LOOP	REF
AND	EXCL	LOOPHOLE	REFANY
ARRAY	EXIT	MAX	REPEAT
ASSERT	EXPORT	MIN	RETURN
BEGIN	@EXTERNAL	MOD	SAFE
BITS	FALSE	MODULE	SET
BITSET	FINALLY	NARROW	THEN
BOOLEAN	FIRST	NEW	TO
BY	FLOAT	NIL	TRUE
@C	FOR	@NOCOUNT	TRUNC
CAP	FROM	NOT	TRY
CARDINAL	HALT	NUMBER	TYPE
CASE	HIGH	ODD	TYPECASE
CHAR	IF	OF	UNSIGNED
CHR		OR	UNTIL
CONST	IMPLEMENTS	ORD	VAL
DEC	IMPORT	@PASCAL	VAR
DEFINITION	IN	PASSING	WHILE
DISPOSE	INC	POINTER	WITH
DIV	INCL	PROCEDURE	
DO	INTEGER	QUALIFIED	
ELSE	LAST	RAISE	
ELSIF	LOCK	RAISES	
END	LONGFLOAT	REAL	
EXCEPT	LONGREAL	RECORD	

and IMPLEMENTATION, which is so long that it throws the chart off if you put it in its proper order. Historical information about the origin of these identifiers appears in Appendix V, page 149.

1.2. Identifiers

"Identifiers" are sequences of letters and digits. The first character of an identifier must be a letter. Examples:

```
x  scan  Modula  Apply  GetSymbol  firstLetter
```

Case is significant -- `jam` and `Jam` and `JAM` and `jAM` are different identifiers. Pragmatic note about choosing identifiers at page 106.

1.3. Literal Numbers

There are no negative literal numbers. Literal numbers represent nonnegative integers or real numbers.

To construct a literal decimal integer, you use the digits "`0`" through "`9`".

To construct a literal octal integer, you use the digits "`0`" through "`7`" and follow them with the letter "`B`".

To construct a literal hex integer, you use the digits "`0`" through "`f`" (lower-case only for the alpha digits) and follow them with the letter "`H`". To differentiate literal hex numbers from identifiers, no hex number is allowed to start with an alpha digit; you have to use a dummy zero if your hex number would have started with "`a`" through "`f`". Examples:

```
1980  3764B  7bcH 177B 3fH 0ffH
```

To construct a literal (decimal) real, you must always use a decimal point and at least one digit to the left of the decimal point. A real number may contain a decimal scale factor: the letter "`E`", pronounced either "times ten to the power of" or (more often) "E". Examples:

```
12.3   45.67E-8   0.8
```

1.4. Strings

Strings are sequences of characters enclosed in double quotes.

If you want to write a string on several lines, you have to terminate each line but the last with a "`\`"; neither the "`\`" nor the newline will be part of the resulting string. If you want the resulting string to include a newline, you have to use the escape sequence "`\n`".

If you want to use a "`\`" or a "`"`" in a string, you must escape it with a "`\`":

```
"\\"
"\""
```

Here are all of the escape sequences available to you in strings:

`\n`	newline (linefeed)
`\t`	tab
`\r`	carriage return
`\f`	form feed
`\b`	backspace
`\\`	backslash
`\"`	double quote

A "`\`" followed by up to three octal digits specifies the character whose ASCII code is that octal value (e.g. `\014` for form feed, `\033` for escape). The `Char` interface contains constant definitions for the ASCII control characters (`Char.NP` for `\014`, `Char.ESC` for `\033`, and so on).

Examples:

```
"a"
"Don't Worry!"
"codeword \"Barbarossa\""
```

Pragmatic note about very long strings in Section 18.4, page 107.

1.4.1. Null Termination

To descend suddenly from the level of the language to the level of representation:

When you assign (`:=`) a string to a fixed-length array of CHAR, one null byte will get tacked on to the end if there's room. Suppose you have a five-byte buffer:

```
buf = ARRAY [0..4] OF CHAR
```

Then a four-character string stored in the buffer gets a null byte tacked on:

```
buf := "1234";
```

`buf[4] = "\000"`. But if you store a string that fills up the buffer,

```
buf := "12345";
```

no null byte. `buf[4] = "5"`. (There is no such thing as assigning to an open array, so don't worry about that one.)

When you pass a string as an actual to an open array of CHAR formal, no null byte gets tacked on. [[What about if I pass a string as an actual to a fixed-size array of CHAR formal?]]

When you assign a string to a `Text.T` or pass a string as an actual to a `Text.T` formal, there is a null byte at the end of the representation. But the `Text.Length` of the text is the number of characters in the source string, not counting the tacked-on null.

End of descent.

1.5. Operators and Delimiters

The special characters and character pairs are:

```
+       =       ,       ..
-       #       ;       :
*       <       (       )
/       >       [       ]
:=      <=      {       }
.       >=      ^       |
```

1.6. Comments

Comments can be inserted between any two tokens in a program. They are arbitrary character sequences opened by "(*" and closed by "*)". Comments can be nested:

```
(* Like (* this *) *)
```

and can extend over more than one line.

2. Names and Name Scope

The supplied identifiers and keywords of the language are reserved words in Modula-2+; they are available everywhere and at all times, and you may not redefine them. See Section 1.1, page 3, for the full list.

2.1. Name Scope

A scope is a section of program text in which a certain set of names is defined: Each name is bound to an object. Wirth designed the syntax of Modula so that scopes nest: that is, if you have two different scopes, either one contains the other or the two are disjoint. The meaning of a name at any given point is always determined by the smallest enclosing scope in which the name is defined.

A good way to think about scope is to picture a name table attached to each block -- a block is a module or a procedure. A name gets looked up in the name table of the block in which it occurs; if there's no entry for that name in the block's own name table, the name gets looked up in the name table of the block enclosing that block. Etc. The meaning of any name is unambiguous; usually it is also clear.

A good way to think about procedure declarations is that the enclosing block includes the name of the procedure and the procedure's own scope starts right after its name. In other words, you could actually (ugh) name a procedure and one of its formal parameters the same name:

```
PROCEDURE P(P: INTEGER);
```

A procedure's formal parameters are entered in the name table for the procedure, not the enclosing block. Their scope extends to the end of the procedure.

An implementation module automatically gets all the declared and imported names from its definition module.

2.2. Qualified Names

Qualified names look like this:

```
record.field
```

or this:

```
Module.Procedure
```

In other words, interpret the right-hand name in the context of the object named by the left-hand name.

Records and modules produce qualified names:

1. Suppose you declare a variable of a record type:

```
TYPE
  R = RECORD
    f1: INTEGER;
    f2: BOOLEAN;
  END;
VAR
  r: R;
```

Now you can use `r.f1`.

You can also interpret the name of a record field in the context of a dereferenced ref or pointer to the record. If, for instance, those declarations had looked like this:

```
TYPE
  R = RECORD
    f1: INTEGER;
    f2: BOOLEAN;
  END;
  P = REF R;
VAR
  r: R;
  p: P;
```

you would be able to talk about `p^.f1`.

2. Using the form

```
IMPORT M;
```

is quite comparable to defining a variable of a record type. All the names declared in the definition module M are now available to you as long as you qualify them with the prefix M and the dot:

```
M.T
M.Proc
```

etc.

2.3. Declaration

One way to get a name into the name table of a block is to declare it. When you declare a name, the scope of the definition extends from the declaration to the end of the block in which the declaration occurs; you can think of a declaration as adding a name to the name table for the block in which it occurs.

There are five kinds of declaration: constant, type, variable, exception, and procedure declarations. In cheat-sheet notation:

```
decl      = CONST      *( id                    '= const          '; )
          | TYPE       *( id                 ?('= type | '= REF)  '; )
          | VAR        *( ids ':   type       ?('= qi)            '; )
          | EXCEPTION  *( id ?( '( type ') ) ?('= qi)            '; )
          | procH                            ?('= qi)            ';
```

Each of these kinds of declaration has its own special flavor.

For constants, see Section 3.2, page 16.
For types, see Section 5.2, page 21.
For variables, see Section 6, page 31.
For exceptions, see Section 10.11.1, page 50.
For procedures, see Section 12.1, page 63.

Examples:

```
CONST
  A = 613;
  B = {1, 3, 7};
TYPE
  Index = [0..15];
  Handle = REF Object;
  Object = RECORD
    key: INTEGER;
    left, right: Handle;
  END;
VAR
  var: Text.T;
  x, y, z: INTEGER;
EXCEPTION
  HandleError(Handle);
  FatalError;
PROCEDURE Halt();
```

Usually it's an error to declare the same name twice in one block. The only cases where it's not are vacuous:

```
CONST
  A = 613;
  A = 613;
TYPE
  Index = CARDINAL;
  Index = CARDINAL;
END test.
```

It is OK to redefine a name from an enclosing block as long as you haven't yet used it in the current block.

The effect of declaring an enumeration type is to add to the name table for the block not only the name of the type but also the names of all the constants. I.e., after you do:

```
TYPE
  Color = (Red, Green, Blue);
```

the names `Red`, `Green`, and `Blue` are available for the rest of the block (as well as the name `Color`). They are not attached to the name `Color` (as for instance the names of the fields of a record are attached to the name of the record). So you cannot do:

```
TYPE
  Color = (Red, Green, Blue);
  Feeling = (Sad, Sorry, Blue);
```

That's seen as an attempt to redefine a constant and produces a fatal error.

The effect of declaring a record type is to define a set of field names that are meaningful in the context of a variable of that type. I.e. after you do:

```
TYPE
  R = RECORD
    f1: INTEGER;
    f2: BOOLEAN;
  END;
```

you can then do:

```
VAR
  r: R;
```

and thereafter use the names `r.f1` and `r.f2`. (See Section 2.2, page 7.)

2.4. Forward Reference

Generally it's an error to use a name before you've declared it, but there are two exceptions:

- You can use the name of a type in constructing a new ref or pointer type to the type and then declare the type.
- You can use a procedure and then declare it.

Here's a trivial example of using a type in defining a ref type before you've defined the original type:

```
TYPE
  ThingRef = REF Thing;
  Thing = RECORD
    key, entry: Text.T;
  END;
```

There's not much reason to do that. But you do often want to do something like this:

```
TYPE
  IntList = REF IntListRec;
  IntListRec = RECORD
    first: INTEGER;
    rest: IntList;
  END;
```

The ability to make forward type references is specific to REF and POINTER TO type constructors. So, for instance, the following is not allowed:

```
TYPE
  FileArray = ARRAY [0..9] OF FileNumber;
  FileNumber = CARDINAL;
```

Forward references to procedures are always allowed. You can call a procedure, assign it as a procedure variable, or pass it as a procedure parameter before declaring it. You can arrange the procedures of a module in any order you wish.

2.5. Import

Another way to get a name besides declaring it is to import it from some other module where somebody else has declared it for you. There are two forms of import:

```
IMPORT M;
```

and:

```
FROM M IMPORT X;
```

`M` in either case is the definition module `M` from somewhere out in the great world. (The manpage for the compiler describes the search rules used to find it.)

When you use the form:

```
IMPORT M;
```

you show that you're planning to use names from module `M`. Only the name `M` is inserted in the current scope; to use any of the names declared in `M`, you must qualify them:

```
M.PrintIt
M.Color
```

You get access only to names declared at the top-level scope of `M`, not names declared inside procedures or inside the module body.

When you use the form:

```
FROM M IMPORT X;
```

only the name `X` from module `M` is inserted in the current scope; the name `M` is not inserted. Whatever `X` meant in the context of the definition module `M` it now means in your current block. You refer to `X` simply as `X` (`M.X` won't work.) You must list explicitly every name you wish to use unqualified:

```
FROM Stdio IMPORT stdin, stdout, stderr;
```

Style note about qualified names in Section 19.4, page 111.

The scope of the constants of an enumeration type is the same as the scope of the type. That is, if you do:

```
FROM M IMPORT E;
```

and `M` declared `E` as an enumeration type that consisted of `A`, `B`, and `C`:

```
E = (A, B, C);
```

then `A` and `B` and `C` get dragged in along with `E`, and you can say `A` just as well as `E`.

When you import the name of a record type, you can then use that name as context for talking about any of its fields. For instance, the definition of `TimeZone` in the `Time` interface says:

```
TYPE
  TimeZone = RECORD
    minutesWest: INTEGER;
    dstAlgorithm: INTEGER;
  END;
```

If you do:

```
FROM Time IMPORT TimeZone;
VAR
  X: TimeZone;
```

you can then talk about `X.minutesWest` with no further ado.

2.6. Declaration versus Import

When you import somebody else's module, you get only the types and procedures it declares, not the ones it imports. When somebody else imports your module, she gets only the types and procedures you have declared, not the ones you imported.

So when you are writing a definition module, if you want to pass a type or a procedure on to people who import you, you must use the form:

```
IMPORT M;
TYPE
  X = M.X;
```

Or:

```
IMPORT M;
PROCEDURE X = M.X;
```

For more about pass-through procedures in definition modules, see Section 13.4, page 72.

2.7. WITH Statements

WITH statements are useful for filling up records. Suppose for example that I have:

```
TYPE
R = RECORD
  key: INTEGER;
  left,right: REFANY;
END;
VAR
  r: R;
```

Then the usual way of talking about the fields of r is to call them `r.key`, `r.left`, and `r.right`. In a WITH statement, however, I can talk about them unqualified:

```
WITH r DO
  key := 0;
  left := NIL;
  right := NIL;
END
```

Wirth says that the WITH statement:

> specifies a record variable and a statement sequence. In these statements the qualification of field identifiers may be omitted, if they are to refer to the variable specified in the with clause. If the designator denotes a component of a structured variable, the selector is evaluated once (before the statement sequence). The with statement opens a new scope.

The point is that bit about "the selector is evaluated once." Continuing the example above:

```
VAR
  a: ARRAY [1..10] OF R;
WITH a[i]^ DO
  Wr.PrintF( Stdio.stdout, "%d\n", key );
  INC(i);
  Wr.PrintF( Stdio.stdout, "%d\n", key );
```

The second occurrence of `key` has the same value as the first; the `i` got evaluated at the top of the WITH and no amount of incrementing or decrementing thereafter makes the slightest bit of difference.

New restriction on use of the WITH statement: To patch around a compiler bug, the compiler will give an error if a ref-containing (RC) VAR parameter (or element of same, or field of same) is given as a WITH expression. It's not clear when this will be fixed, if ever.

For example:

```
PROCEDURE P(
    VAR r: RecordType;
    VAR a: ArrayOfRecordType);
  BEGIN
    WITH r DO ss; END;     (* not allowed if RecordType is RC *)
    WITH r.f DO ss; END;   (* not allowed if field f is RC *)
    WITH a[x] DO ss; END;  (* not allowed if a's element type is RC *)

    WITH <designator>^<etcetera> DO ss; END; (* always allowed *)
  END P;
```

See the style note on WITH statements, Section 19.5, page 112.

2.8. Nested Modules

Any, all, or none of the above may apply to nested modules. See if you like Section 13.7, page 73. No promises intended or implied.

3. Constants and Constant Expressions

For the "types" of constants, see Section 15.3, page 82. For type-checking involving constants, see Chapter 15, page 81.

3.1. Constant Expressions

Some constructions in Modula require constant expressions, so you need to know how to construct one. The class of constant expressions is a restricted set of all expressions, restricted by the requirement that the compiler be able to evaluate the expression at compile time.

The simplest kinds of constant expressions are constants:

- Literal numbers are constants. See Section 1.3, page 4.
- Strings are constants. See Section 1.4, page 4.
- Elements of enumeration types are constants. (CHAR and BOOLEAN are supplied enumeration types; their elements are constants.)
- NIL is a constant.
- Literal sets are constants. You write a literal set by writing the name of the set type followed by "{" followed by a list of set elements followed by "}". If the set type is the supplied type BITSET, you can omit its name. You can specify a consecutive range of set elements by writing the first element of the range, followed by ".." (pronounced "dot-dot"), followed by the last element of the range. Examples:

```
Char.Set {"0" .. "9"};
{0 .. System.BitsPerWord - 1};
```

You can apply these operators to constant expressions and still come up with constant expressions:

the relations
  = # < > <= >= IN
the unary operators
  + - NOT
the add (binary infix) operators
  + - OR
the mul (binary infix) operators
  * / DIV MOD REM AND

You can apply these supplied procedures to constant expressions and still come up with constant expressions:

NUMBER (with fixed-size arrays)
HIGH (with fixed-size arrays)
FIRST
LAST
MIN
MAX
`System.Size` (with fixed-size arrays)
`System.TSize`
`System.ByteSize` (with fixed-size arrays)
`System.TByteSize`

At the moment, you're not allowed to use any other procedures in making a constant expression.

Future possibilities: There are some other supplied procedures that are candidates for use in constant expressions:

FLOAT, LONGFLOAT, and TRUNC
ODD
ORD, CHR, and VAL
CAP

Violetta says: "If you have a strong desire to have any of these made usable in constant expressions, you should speak up now."

Here are some consequences of the rules we've just enunciated:

- The only procedure-, ref-, or pointer-valued constant expression is NIL.
- There are no array-valued or record-valued constant expressions.

3.2. Declaring Constants

A CONSTANT declaration defines a constant:

```
CONST
  HashTableEntries = 200;
  Limit = 2*N -1;
  Digits  = Char.Set {"0" .. "9"};
  AllBits = {0 .. System.BitsPerWord - 1};
  Message = "Hello, world.\n"
```

The thing on the right-hand side of a constant declaration must itself be either a constant or a constant expression; see immediately below for constant expressions.

There's a style note about declaring constants in Section 19.7, page 112.

Declaring an enumeration type also declares the constants that are elements of the type:

```
TYPE
    Meals = (Breakfast, Lunch, Dinner);
```

4. Supplied Types

The point of a type is to define the structure of variables of that type, to limit the values that variables of that type can assume, and to limit the operations that can be applied to variables of that type.

The language supplies some types. (All other types are constructed with type-constructors, see Chapter 5, page 21.)

Complete list of the supplied types:

INTEGER
CARDINAL
REAL
LONGREAL
UNSIGNED
BITSET
BOOLEAN
CHAR
REFANY

4.1. Supplied Numeric Types

INTEGER	For us, FIRST(INTEGER) = -2^{31} = -2147,483,648 and LAST(INTEGER) = $2^{31} - 1$ = 2,147,483,647.
CARDINAL	equivalent to the subrange type `[0 .. LAST(INTEGER)]`.
UNSIGNED	the integers from 0 to $2^{32}-1$.
REAL	real numbers. See Section 16.1, page 89, for an explanation of how real-number types are implemented.
LONGREAL	double-precision real numbers. See Section 16.1, page 89, for an explanation of how real-number types are implemented.

4.2. Other Supplied Types

BITSET

```
= SET OF [0 .. System.BitsPerWord - 1]
```

When you're writing a constant bitset, you're allowed to omit the type name BITSET before the curly bracket.

```
{3,7,9}
```

is equivalent to:

```
BITSET {3,7,9}
```

BOOLEAN — the truth values TRUE and FALSE. BOOLEAN is an enumeration type declared as

```
TYPE
  BOOLEAN = (FALSE, TRUE);
```

So

```
ORD( FIRST(BOOLEAN) ) = 0
ORD( LAST(BOOLEAN) ) = 1
```

CHAR — the character set provided by the implementation (which Wirth calls the "used computer system"; I like the phrase) -- 256 of them for us. 0 to 127 are the ASCII character set. CHAR behaves like an enumeration type. The Char interface contains constant definitions for all of the ASCII control characters.

REFANY — is a place-holder type, for holding values of any ref type. REFANY is not itself a ref type. Any time we mean "ref type or REFANY," we have to say so explicitly.

If you have

```
TYPE
  RI = REF INTEGER;
  RC = REF CHAR;
VAR
  ri: RI;
  rc: RC;
  ra: REFANY;
```

then you can say

```
ra := ri;
ra := rc;
```

but you can't just say

```
ri := ra;
```

You have to say something like

```
ri := NARROW(ra, RI);
```

or

```
TYPECASE ra OF
| RI(ri): ...
END;
```

See Section 10.4, page 45.

4.3. Kinds of Supplied Types

Here are the kinds of the supplied types. Every time we add a type constructor we'll explain where the types it constructs fit in this classification:

	numeric	non-numeric
ordinal	INTEGER CARDINAL UNSIGNED	BOOLEAN CHAR
non-ordinal	REAL LONGREAL	BITSET REFANY

5. Constructing Types

5.1. The Type Constructors

To get types other than the supplied types, you use type constructors. In cheat-sheet notation:

```
  '[ const '.. const ']
| '( ids ')
| ARRAY **type ', OF type
| RECORD fields END
| SET OF type
| REF type
| POINTER TO type
| PROCEDURE tFormals ?(': ti) ?(RAISES exSet)
| BITS const FOR type
```

Opaque types don't show up here; see Section 5.2 (immediately below) and Section 13.3, page 70. BITS FOR is in the chapter on representation, in Section 16.2, page 91.

5.2. Declaring Types

Each occurrence of a type constructor produces a new type. For example:

```
TYPE
  TypeA = ARRAY [0 .. 3] OF INTEGER;
  TypeB = ARRAY [0 .. 3] OF INTEGER;
VAR
  instanceA: TypeA;
  instanceB: TypeB;
BEGIN
  instanceB := instanceA;     (* illegal *)
```

The assignment is illegal because TypeA and TypeB are different types. One feels, "But they're the same." But they're not.

So you have to give a name to any constructed type that you want to use more than once.

Normally, you define a name for a type with a declaration of the form:

```
TYPE
  name = type;
```

That thing on the right-hand side of the equals sign can be a previously defined type or a new type created by using a type constructor.

In a DEFINITION module you can also declare a name for a new type with an opaque word type declaration:

```
TYPE
  name;
```

or an opaque ref type declaration:

```
TYPE
  name = REF;
```

For more about opaque types, see Section 13.3, page 70.

5.3. Subranges

You can make a new type that is a subrange of an ordinal type by specifying the lowest and the highest value in the subrange:

```
TYPE
  Index   = [0 .. N-1];
  Cap     = ["A" .. "Z"];
  Workday = [Monday .. Friday];
  Showdog = [FIRST(Dog) .. Poodle];
```

The lower bound must not be greater than the upper bound. Both bounds must be constant expressions.

The resulting type is an ordinal type, numeric if the basetype is numeric, non-numeric if the basetype is non-numeric:

	numeric	non-numeric
ordinal	INTEGER	BOOLEAN
	CARDINAL	CHAR
	UNSIGNED	non-numeric subrange
	numeric subrange	
non-ordinal	REAL	BITSET
	LONGREAL	REFANY

In a SAFE module, the compiler checks to be sure that any value assigned to a variable or passed to a value parameter of a subrange type lies within the bounds.

Empty subranges are illegal.

Style note in Section 19.9, page 113.

5.4. Enumerations

An enumeration is a list of identifiers. These identifiers are used as constants. They are numbered from left to right, starting with 0. (See ORD, Section 11, page 55.)

The resulting type is a non-numeric ordinal type.

Examples of constructing enumeration types:

```
TYPE
  Color = (Red, Blue, Yellow);
  Suit  = (Club, Diamond, Heart, Spade);
```

Enumerations behave like thinly disguised numeric types. For instance:

```
TYPE
  Color = (Red, Blue, Yellow);
VAR
  c,d: Color;
BEGIN
  c := Blue;
  d := Yellow;
  IF c = d THEN Wr.PrintF( Stdio.stdout, "Same color.\n" );
  IF c < d THEN Wr.PrintF( Stdio.stdout, "Earlier color.\n" );
```

Their features -- special scoping rules, ORD, and VAL -- keep the disguise from getting in your way. Subranges don't need such features because there's no disguise.

For scope rules, see Section 2.3, page 9. For ORD and VAL, see Section 11, page 55.

Empty enumerations are illegal.

5.5. Fixed-Size Arrays

A fixed-size array consists of a fixed number of components that are all of the same type. The declaration of the array type specifies the types of the components and the indices. Each index type must be an ordinal type.

The resulting type is a non-numeric non-ordinal type.

Examples of constructing fixed-size array types:

```
TYPE
  ChildOfTheWeek = ARRAY Week OF Child;
  QueenForADay = ARRAY Weekday OF Housewife;
  SRCPhoneList = ARRAY Person, [0 .. 3] OF CHAR;
```

A declaration of the form:

```
ARRAY T1, T2, ... , Tn OF T
```

with n index types `T1` ... `Tn` is an abbreviation for the construction

```
ARRAY T1 OF ARRAY T2 OF ... ARRAY Tn OF T
```

In other words, an n-dimensional matrix.

To talk about an element of an array, you subscript the name of the array: You write the name of the array followed by "`[`" followed by the index of the element followed by "`]`"; i.e. to talk about the *ith* element of array `a`, write:

```
a[i]
```

For subscripting to type-check, the index expression must be assignable to a variable of the index type; see Section 15.6, page 86.

To talk about the fourth element of `offset`, an instance of array type

`Offset` defined as ARRAY `8..12` OF INTEGER, you say `offset[11]`, not `offset[4]`.

An expression of the form `a[e1, e2, ... , en]` stands for `a[e1][e2]...[en]`.

You can assign to an array as a whole as well as to the individual elements.

5.6. Open Arrays

An open array also consists of components that are all of the same type, but the number of components is not fixed at compile time. The declaration of an open array type specifies the type of the components. The type of the index is always CARDINAL. The lower bound is always 0; the upper bound is determined at runtime.

The resulting type is a non-numeric non-ordinal type.

An open array of `System.Byte` or `System.Word` is special. First let's talk about ordinary open array types.

When you want to write a procedure that operates on arrays of arbitrary lengths -- for example, a procedure to compute the average of an array of numbers, or to sort an array of records, or to form the dot product of two arrays of numbers -- you should declare the formal parameter to be an open array.

The actual parameter you pass to an open array formal can be either a fixed-size array or an open array; its element type must be the same as the element type of the formal parameter. (See Section 15.7, page 87.) The procedure can use the supplied procedures HIGH and NUMBER to determine the size of the array actually passed to it.

If you are calling a procedure with an open array formal and you are passing it an array that is one-dimensional with an index type starting at 0, then you and the procedure will agree on how the array is indexed. Otherwise, it's up to you to see things from the point of view of the procedure you've called.

For instance:

```
PROCEDURE Swap(a: VAR ARRAY OF INTEGER; i: INTEGER; j: INTEGER);
(* Swap a[i] and a[j] *)
  VAR t: INTEGER;
  BEGIN
    t := a[i]; a[i] := a[j]; a[j] := t;
  END Swap;

VAR
  a: ARRAY [1..10] OF INTEGER;
```

With those declarations, what happens if you call `Swap(a,1,2)`? What happens is that the second and third elements of `a` get swapped, not the first and second. The lower bound of an open array parameter is always `0`.

If the element type of an open array formal is `System.Word`, then the corresponding actual parameter may have any type but its size has to be

divisible by 32. If the element type of a formal parameter is `System.Byte`, then the corresponding actual parameter may have any type but its size has to be divisible by 8.

You can construct a ref type whose referent is an open array. The size of the open array is determined by the number passed as the second parameter to NEW when the open array variable is created.

That's the end of the story. You can't use an open array type like a fixed-size array type -- you can't declare a pointer to an open array type, you can't embed open arrays in arrays or records, you can't return an open array as the result of a procedure. So in practice there are only two ways that you ever get an open array variable: Either you declare an open array parameter to a procedure, or you dereference a value that is a ref to an open array.

There are just three things you can do with an open array variable:

1. subscript it,
2. pass it to one of five supplied procedures HIGH, NUMBER, `System.Adr`, `System.Size`, and `System.ByteSize`,
3. or pass it to a procedure whose corresponding formal parameter is an open array.

You cannot use an unsubscripted open array variable on either side of an assignment statement.

```
TYPE
  Open = ARRAY [0..2] OF INTEGER;
VAR
  open: Open;
PROCEDURE P(a: ARRAY OF INTEGER);
  BEGIN
    a[0] := open[0];     (* dandy *)
    a    := open;        (* illegal *)
  END P;
```

You can use an open array type constructor in a procedure formal; all other types in procedure formals must have names (identifiers). Therefore this procedure heading is OK:

```
PROCEDURE Sum(a: ARRAY OF INTEGER): INTEGER;
```

while these similar-looking headings are not OK:

```
PROCEDURE Sum10(a: ARRAY [0..9] OF INTEGER): INTEGER;
PROCEDURE CountChars(s: SET OF CHAR): INTEGER;
```

5.7. Records

A record consists of a fixed number of fields of possibly different types. The declaration of a record type specifies a type and an identifier for each field.

The resulting type is a non-numeric non-ordinal type.

Examples of constructing record types:

```
TYPE
  Date = RECORD
    day: [1 .. 31];
    month: [1 .. 12];
    year: [0 .. 2000];
  END;
  Employee = RECORD
    name, first: ARRAY [0 .. 9] OF CHAR;
    age: [0 .. 99];
    salary: REAL;
  END;
```

You use a field-selection dot to talk about the fields of a record instance. For example (continuing the examples above), if you have:

```
VAR
  inauguration: Date;
  president: Employee;
```

then you talk about the fields of those records this way:

```
inauguration.day   (* the day field of the inauguration record *)
president.age      (* the age field of the president record *)
```

(For talking about record fields inside a WITH statement, see Section 2.7, page 13.)

A record type may have one or more variant sections. The field at the head of a variant section is called the "tag field." It must be of an ordinal type. The case labels are constants of the type indicated by the tag field.

```
TYPE
  EmployeeStatus = (Faculty, Staff);
  UniversityEmployee = RECORD
    name: Text.T;
    birthday: INTEGER;
    CASE status: EmployeeStatus OF
    | Faculty:
        almaMater: Text.T;
        tenureTrack: BOOLEAN;
    | Staff:
        preferredParking: ARRAY [0 .. 9] OF INTEGER;
    END;
  END;
```

In other words, if a record variable of type `UniversityEmployee` has the value `Faculty` in the `status` field, the record will consist of five fields (`name`, `birthday`, `status`, `almaMater`, and `tenureTrack`); if it has the value `Staff` in the `status` field, it will consist of four fields (`name`, `birthday`, `status`, and `preferredParking`).

In the simplest way of using a variant record, you initialize its tag field shortly after creating it and never touch the tag again. Changing the tag field leaves the values of the variant fields undefined.

It's also possible to write a variant section without a tag field -- but with utmost care. If you have a tag field, the compiler and runtime system can check it for you. If you don't have a tag field, you have to supply the checking yourself. Tagless variant records are usually for low-level programming. There are some real-life examples in Section 16.5, page 94.

Tagless ref-containing variant records are not supported.

5.8. Sets

You make a set type because you want to be able to talk about any subset of some ordinal type.

For instance, suppose you have the enumeration type `TerminalMode` declared as:

```
TYPE
  TerminalMode = (ReverseVideo, AutoWrap, AutoRepeat);
```

Because any combination of these modes makes sense, you would represent the state of a terminal with a variable of type `TerminalModeSet`:

```
TYPE
  TerminalModeSet = SET OF TerminalMode;
```

So a variable of type `TerminalModeSet` can assume any of the following values:

```
TerminalModeSet {}
TerminalModeSet {ReverseVideo}
TerminalModeSet {AutoWrap}
TerminalModeSet {AutoRepeat}
TerminalModeSet {ReverseVideo, AutoWrap}
TerminalModeSet {ReverseVideo, AutoRepeat}
TerminalModeSet {AutoWrap, AutoRepeat}
TerminalModeSet {ReverseVideo, AutoWrap, AutoRepeat}
```

The resulting type is a non-numeric non-ordinal type.

Modula-2+ allows sets that require more than one word to represent.

```
SET OF [0..1023]
```

is perfectly reasonable. See Section 16.7, page 96, for implementation restrictions.

5.9. Refs

A ref is the address of a variable in garbage-collected storage. You declare a ref type as follows:

```
TYPE
  R = REF INTEGER;
```

or:

```
TYPE
  NodeRep = RECORD
    next, prev: Node;
  END;
  Node = REF NodeRep;
```

You can construct a ref type that refers to variables of any type. The resulting type is a non-numeric non-ordinal type.

Variables referenced by refs are created only with the supplied procedure NEW. If the first parameter to NEW is a variable with a ref type, then a variable of the referent type is created and a reference to it is assigned to the

ref variable. You can assign the supplied constant NIL to any variable with a ref type.

(Many people think of variables as named slots for storing values; by "variable" we mean simply a slot, whether named or allocated by NEW.)

To talk about the variable referenced by a ref, you use the dereferencing operator "^", as in:

```
node^.next := NIL;
```

5.10. Pointers

A pointer is the address of a variable. You declare a pointer type as follows:

```
TYPE
  KeyboardRecord = RECORD
    keys: KeySet;
    cursor: Point;
  END;
  PtrKeyboardRecord = POINTER TO KeyboardRecord;
```

You can construct a pointer type that points to variables of any type. The resulting type is a non-numeric non-ordinal type.

You generate a pointer value by calling NEW or by calling `System.Adr`. If the first parameter to NEW is a variable with a pointer type, then a variable of the referent type is created and its address is assigned to the pointer variable. You can assign the supplied constant NIL to any variable with a pointer type.

If you created a pointer value by calling NEW you can release the storage with DISPOSE. If you call DISPOSE twice on the same value you can expect catastrophe. You can also expect catastrophe if you call DISPOSE on a pointer you created with `System.Adr`.

To talk about the variable addressed by a pointer, you use the dereferencing operator "^", as in:

```
allKeysUp := ptrKeyboardState^.keys = KeySet{};
```

5.11. Procedure Types

It's useful to be able to pass procedures as actual parameters to other procedures and to embed procedures in data structures. So you want to be able to declare procedure-valued variables, and in order to declare procedure-valued variables, you need procedure types. For instance:

```
TYPE
  FleaAction = PROCEDURE;
  DogAction  = PROCEDURE(VAR Breed := Cur);
```

The resulting type is a non-numeric non-ordinal type.

The syntax for the header of a procedure type constructor is very like the header for a procedure declaration. In cheat-sheet notation:

```
type     = PROCEDURE    tFormals ?(': ti) ?(RAISES exSet)
procH    = PROCEDURE id  formals ?(': ti) ?(RAISES exSet)

formals  = '( **(?VAR ids ': ?(ARRAY OF) ti ?default) '; ')
tFormals = '( **(?VAR        ?(ARRAY OF) ti ?default) '; ')
```

You can assign procedure `P` to a variable `t` of procedure type `T` only if they match; see Section 15, page 81. And there are some additional restrictions:

1. `P` must not be declared local to another procedure.
2. `P` must not be known to the compiler. (Being known to the compiler is different from what we've been calling being "supplied," i.e. not requiring import.) `[[` And how does an earnest seeker after truth discover what procedures the compiler knows about? `]]`

The constant NIL is compatible with procedure types, so procedure variables can be initialized to NIL, and you can return NIL from a procedure whose result-type is a procedure type.

5.12. Kinds of Types

Here are the kinds of all the types, supplied and constructed:

	numeric	non-numeric
ordinal	INTEGER	BOOLEAN
	CARDINAL	CHAR
	UNSIGNED	enumeration
	numeric subrange	non-numeric subrange
non-ordinal	REAL	BITSET
	LONGREAL	REFANY
		fixed-size array
		open array
		record
		set
		ref
		pointer
		procedure

6. Variables

A variable declaration associates an identifier with a type. The type defines the structure of the variable, limits the values that the variables can assume, and limits the operations that can be applied to the variable.

For example:

```
VAR
  k: INTEGER;
  c: Wr.Consumer;
  a: ARRAY Index OF CARDINAL;
```

Variables whose identifiers appear in the same list all get the same type. That is to say, this:

```
VAR
  i: CARDINAL;
  j: CARDINAL;
```

is the same as this:

```
VAR
  i,j: CARDINAL;
```

The other way of creating a variable is by calling NEW. (Many people think of variables as named slots for storing values; by "variable" we mean simply a slot, whether named or not.)

For rules about identifiers, see Section 1.2, page 4.

Variables of any ref type or of type REFANY are always initialized to NIL. The conservative assumption to make about all other variables is that their initial value is undefined. For cases where performance is crucial and you want to investigate the exact dimensions of that conservative assumption, see Section 16.3, page 93.

7. Expressions

An expression specifies a computation that produces a value. In constructing an expression you can call procedures and use operators.

Parentheses in expressions group operands with their operator.

The only really tricky thing about expressions is type-checking; see Chapter 15, especially Section 15.5, page 83.

7.1. Operands

If an operand in an expression is not itself an expression, then it is something simpler:

- a literal number, a string, a literal set, or NIL (see pages 4, 4, 15)
- the name of a constant, an exception, a variable, or a procedure with any number of syntactically acceptable dots, uparrows, and square brackets

If the name is the name of a variable, its value is the current value of the variable. [[Is this a surprise?]]

If the name is the name of a procedure or a procedure-valued variable and is followed by a (possibly empty) parameter list, the name represents a call on the procedure with those parameters and the value is the value returned by the call. If the name is the name of a procedure but is not followed by a parameter list, then the value is the procedure itself, not a call on the procedure.

7.2. Operators

7.2.1. Operator Precedence

There are four classes of operators. From highest to lowest in precedence:

```
NOT
*  /  DIV  MOD  AND
+  -  OR
=  #  <  <=  >  >=  IN
```

Sequences of operators of the same precedence are executed from left to right.

The operators are overloaded. That is to say, the same operator symbol may stand for several different operations; the actual operation is identified by the types of the operands.

7.2.2. Arithmetic Operators

Addition is adding.
-- Gertrude Stein, "Arthur a Grammar"

symbol	operation
+	addition
-	subtraction
*	multiplication
/	real division
DIV	integer division
MOD	integer modulus

When used as prefix unary operators,

```
+
```

means identity and

```
-
```

means sign inversion. [[More to come about what sign inversion means with various kinds of operands.]]

The operators DIV and MOD are defined by the following rules:

- `x DIV y` is equal to the quotient `x/y` truncated towards zero.
- `x MOD y` is equal to `x - (x DIV y) * y` -- this is the remainder of the division `x DIV y`; in a short while this is going to be the definition for something called REM and MOD is going to be redefined as the mathematical mod function.

When an overflow occurs in an arithmetic operation, it is ignored. [[What happens on floating-point overflow? What happens on floating-point underflow? What happens on divide by zero?]]

7.2.3. Logical Operators

symbol	operation
AND	logical conjunction
OR	logical disjunction
NOT	negation

The expression

```
p OR q
```

means

```
IF p THEN TRUE ELSE q END;
```

```
p AND q
```

means

```
IF p THEN q ELSE FALSE END;
```

Non-Pascal programmers will be surprised (and Pascal programmers once again disgusted) by the precendence of the operators AND and OR in the expression syntax. You'd like to write:

```
IF a < b AND c < d THEN ... END;
IF a < b OR  c < d THEN ... END;
```

but instead you must write:

```
IF (a < b) AND (c < d) THEN ... END;
IF (a < b) OR (c < d) THEN ... END;
```

[[Paul McJones:]]

Another result of operator precedence and the ordering FALSE < TRUE: If you want to compute

p implies q

(often indicated symbolically as "p => q"), you can write in Modula-2:

```
p <= q
```

7.2.4. Set Operators

symbol	operation
+	set union
-	set difference
*	set intersection
/	symmetric set difference

```
x IN (s1 + s2)    iff  (x IN s1) OR (x IN s2)
x IN (s1 - s2)    iff  (x IN s1) AND NOT (x IN s2)
x IN (s1 * s2)    iff  (x IN s1) AND (x IN s2)
x IN (s1 / s2)    iff  (x IN s1) # (x IN s2)
```

You might expect the operator NOT to apply to sets, but it doesn't.

7.2.5. Relations and Set Relations

symbol	relation
=	equal
#	not equal
<	less
<=	less or equal
>	greater
>=	greater or equal

Many of the relations also apply to sets, where they have different meanings. Where s and t are sets and e is an element:

```
s = t   means that s and t have the same elements
s # t   means that s and t do not have the same elements
s <= t  means that every element of s is an element of t
s >= t  means that every element of t is an element of s
e IN s  means that e is an element of s
```

The set relations

```
<=      >=
```

mean improper inclusion; that is to say, wherever `s = t` is true, it is also true that `s >= t` and `s <= t`.

7.3. Applicability Charts

See the type-checking chapter, 81, to understand this classification. For operands of these basetypes, these operators apply. (Of course, you also need to look at the type-checking chapter to see whether they type-check.)

Binary Operators

INTEGER	*		DIV	MOD	AND	+	-	OR	=	#	<	<=	>	>=	IN
CARDINAL	*		DIV	MOD	AND	+	-	OR	=	#	<	<=	>	>=	IN
UNSIGNED	*		DIV	MOD	AND	+	-	OR	=	#	<	<=	>	>=	IN
numeric subrange	*		DIV	MOD	AND	+	-	OR	=	#	<	<=	>	>=	IN
System.Address	*		DIV	MOD	AND	+	-	OR	=	#	<	<=	>	>=	IN
BOOLEAN					AND			OR	=	#	<	<=	>	>=	IN
CHAR									=	#	<	<=	>	>=	IN
enumeration									=	#	<	<=	>	>=	IN
non-numeric subrange									=	#	<	<=	>	>=	IN
REAL	*	/				+	-		=	#	<	<=	>	>=	
LONGREAL	*	/				+	-		=	#	<	<=	>	>=	
FLOATING-POINT	*	/				+	-		=	#	<	<=	>	>=	
BITSET	*	/				+	-		=	#		<=		>=	IN
REFANY									=	#					
fixed-size array															
open array															
record															
set	*	/				+	-		=	#		<=		>=	IN
ref									=	#					
pointer									=	#					
procedure									=	#					
STRING									=	#					
VOID									=	#					
System.Word															
System.Byte															

Unary Operators

INTEGER		+	-
CARDINAL		+	-
UNSIGNED		+	-
numeric subrange		+	-
System.Address		+	-
BOOLEAN	NOT		
REAL		+	-
LONGREAL		+	-
FLOATING-POINT		+	-

8. The Syntax of Statements

8.1. What Is a Statement?

There are several kinds of statements:

assignments
control structures
WITH statements
procedure calls

Syntactically these are all the same kind of thing: statements.

Assignment, procedure call, RETURN, and EXIT are elementary (simple, atomic) statements. The other statements (Wirth sometimes calls them "structured statements") are composed of parts that are themselves statements.

Statements in a statement sequence are separated by semicolons. Since the empty statement is allowed, you can (and the format note in Section 20, page 113, will tell you you should) end the last statement in a statement sequence with a semicolon.

For assignments, see Chapter 9, page 41.
For control structures, see Chapter 10, page 43.
For WITH statements, see Section 2.7, page 13.
For procedure calls, see Chapter 12.2, page 64.

8.2. A Statement Is Not an Expression

I'm not sure that this is the right place to say this, but you'd like to be able to use statements as expressions, and you can't. For instance, it feels sensible to say:

```
x := IF a THEN b ELSE c END;
```

and you can't. You have to say:

```
IF a THEN x := b ELSE x := c END;
```

which doesn't seem as tight and nice.

9. Assignments

You set the value of a variable by doing an assignment. The assignment operator is written

```
:=
```

and pronounced "gets." After an assignment is executed, the variable on the left-hand side has the value obtained by evaluating the expression on the right-hand-side. The old value of the variable is gone.

The simplest kind of assignment looks like this:

```
y := 0;
```

Any expression can serve as the right-hand side of an assignment statement, just so long as the type of the expression is assignable to the type of the variable. (See Section 15.6, page 86, for type-checking rules.) For instance:

```
i := k;
p := i = j;
z := Math.Log(x + y);
s := Char.Set {"A" .. "Z", "a" .. "z"};
b := ((i + j) * (l + m)) = 0;
```

The left-hand side can be a variable name with any number of dots, uparrows, and square brackets:

```
array[i] := (i + j) * (i - j);
table^.key := i;
word[i+1].char := "A";
```

The variable you get to after applying all those operators can have any type except open array.

You can't (sigh) put a procedure call or anything that looks like a procedure call (like LOOPHOLE or NARROW) on the left-hand side of an assignment statement.

Syntactically, assignments are statements; see Section 8, page 39.

[[What does it *mean* to assign something of one type to a variable of another type?]]

Weird facts about getting extra tacked-on nulls in assigning strings on page 5.

10. Control Structures

[[Many more examples needed. Contributions welcome.]]

10.1. RETURN

Syntax in cheat-sheet notation:

```
s = ...
  | RETURN ?e
```

Executing a RETURN statement terminates the current activation of the procedure in which it occurs.

If the procedure returns a result, the RETURN statement is the only way to terminate the procedure and specify the result:

```
PROCEDURE RadiansFromDegrees(
    degrees: INTEGER)
    : REAL;
  BEGIN
    RETURN(2. * 3.14159 * FLOAT(degrees) / 360.);
  END RadiansFromDegrees;
```

If the procedure doesn't return a result, it can terminate normally by executing a RETURN statement or by falling off the bottom.

```
PROCEDURE Bump(self: T);
  BEGIN
    IF self^.closed THEN RETURN; END;
    INC(self^.count);
  END Bump;
```

(Procedures can also terminate by raising exceptions.)

For more about procedures, see Section 12.1, page 63. [[Style query in section 20.8, page 117.]]

10.2. IF

Syntax in cheat-sheet notation:

```
s = ...
  | IF ++(e THEN ?ss) ELSIF ?(ELSE ?ss) END
```

The expression following an IF or an ELSIF is of type BOOLEAN. The expressions following the IF and any ELSIFs are evaluated in order till one yields the value TRUE. Then its associated statement sequence is executed.

If there's an ELSE clause and if all the Booleans following the IF and ELSIFs evaluate to FALSE, the ELSE statement sequence gets executed.

For example:

```
IF (ch >= "A") AND (ch <= "Z") THEN
  ReadID();
ELSIF (ch >= "0") AND (ch <= "9") THEN
  ReadNumber();
ELSIF (ch = "\"") OR (ch = "'") THEN
  ReadString();
ELSE
  Ignore();
END;
```

If the Boolean expression after an IF or an ELSIF is a constant expression, the compiler produces code only if the expression evaluates to TRUE. For example:

```
IF FALSE THEN
  Wr.PrintF(Stdio.stderr, "Error in framus\n");
END;
```

will produce no code whatsoever. [[John Ellis says: "This belongs in the section on performance, not here." Am I the only person at SRC stupid enough to write a constant expression that evaluates to FALSE and try to use it in an IF statement? It's OK to say I am; I'm not proud.]]

10.3. CASE

Syntax in cheat-sheet notation:

```
s        = ...
         | CASE e ?OF '| ++(elements ': ?ss) '| ?(ELSE ?ss) END
elements = ++(const | const '.. const) ',
```

A CASE statement decides what statement sequence to execute according to the value of an expression. The case labels are constant expressions or ranges of constant expressions.

The basetype of the expression must be an ordinal type. The types of the constant expressions in the case labels must be compatible with that type (see Section 15.5.1, page 83). No value may occur more than once in the case labels.

At runtime, the expression is evaluated, its value is matched against the case labels, and the statement sequence whose label contains the value is executed. If the value of the expression doesn't occur in any of the labels, the ELSE statement sequence is selected; if there is no ELSE, you get a fatal error (so it's fairly common for ELSE to have an empty statement sequence).

Straightforward example:

```
CASE v OF
| "a", "e", "i", "o", "u":
    Wr.PrintF(Stdio.stdout,
      "Always a Vowel: %c\n", v);
| "y":
    Wr.PrintF(Stdio.stdout,
      "Sometimes a Vowel: %c\n", v);
| "w":
    Wr.PrintF(Stdio.stdout, "Welsh Vowel: %c\n",
      v);
| "l" .. "n", "r":
    Wr.PrintF(Stdio.stdout, "Liquid: %c\n", v);
ELSE
  Wr.PrintF(Stdio.stdout,
    "Not a Lower-Case Vowel or Liquid: %c\n",
    v);
END;
```

Pragmatic note in Section 18.5, page 107.

10.4. TYPECASE

Syntax in cheat-sheet notation:

```
s     = ...
      | TYPECASE e ?OF '| ++(( ti parId | ++ti ', ) ': ?ss) '| ?(ELSE ?ss) END
parId = '( id ')
```

A TYPECASE statement decides what statement sequence to execute according to the runtime type of the value of an expression whose static type is REFANY.

The type of the expression has to be REFANY and the case labels have to be the names of ref types. [[No compile-time check about repeated ref types in labels because of opaque ref types, right?]]

At runtime, the expression is evaluated and the value is of some ref type. The compiler knew it only as a REFANY. Now your program wants to know the actual (ref) type. [[John Ellis says, "The mention of 'compiler' here isn't very good form for a discussion of a language." Please consider the compiler to be a personification, introduced to make the linguistic relationships more dramatic.]]

If the value of the expression is NIL, the first case arm is selected. Otherwise the ref type of the value is matched against the case labels. The first statement sequence whose case label contains the ref type is executed. If the type doesn't occur in any of the labels, the ELSE statement sequence is selected; if there's no ELSE, you get a fatal error.

So far so good. If you want to use the value of the expression in one of the arms, there's one additional piece of magic. If an arm has only one type in its label, if the type is followed by a parenthesized variable of that type, and if that arm is selected, then the value of the original REFANY expression gets assigned to that variable before the statement sequence gets executed.

For instance:

```
PROCEDURE Print(r: REFANY);
  VAR
    rb: Ref.Boolean;
    ri: Ref.Integer;
    rt: Text.T;
  BEGIN
    TYPECASE r OF
    | Ref.Boolean (rb):
        IF (rb^ = TRUE) THEN
          Wr.PrintF(Stdio.stdout,
            "Boolean: TRUE\n");
        ELSE
          Wr.PrintF(Stdio.stdout,
            "Boolean: FALSE\n");
        END;
    | Ref.Integer (ri):
        Wr.PrintF(Stdio.stdout, "Integer: %d\n",
          ri^);
    | Text.T (rt):
        Wr.PrintF(Stdio.stdout, "Text: %t\n", rt);
    ELSE
      (* do nothing *)
    END;
  END Print;
```

The supplied procedure NARROW is a shorthand for uses of TYPECASE that have only one case arm. See page 57.

10.5. LOOP and EXIT

Syntax in cheat-sheet notation:

```
s = ...
  | LOOP ?ss END
  | EXIT
```

The statement sequence between LOOP and END is executed repeatedly. One way to stop looping is to execute an EXIT statement:

```
LOOP
  Wr.PrintF(Stdio.stdout, "%t",
    NARROW(l^.first, Text.T));
  l := l^.tail;
  IF l = NIL THEN EXIT; END;
  Wr.PrintF(Stdio.stdout, ", ");
END;
```

If the LOOP occurs within a procedure, another way to stop looping is to execute a RETURN.

The third way is to raise an exception that's not handled within the LOOP.

Executing an EXIT terminates the smallest enclosing LOOP. So if you have a LOOP within a LOOP, you can't terminate the outer LOOP using EXIT. (In case it ever crossed your mind, you can't use EXIT to terminate WHILE, REPEAT, and FOR loops.)

When you need to, you can exit from multiple LOOPs by declaring a local exception and then using TRY EXCEPT; see Section 10.11, page 50.

10.6. FOR

Syntax in cheat-sheet notation:

```
s = ...
  | FOR id ':= e TO e ?(BY const) DO ?ss END
```

The statement sequence in a FOR statement gets executed repeatedly while a progression of values is assigned to its "control variable," the variable whose name appears immediately after the FOR:

```
FOR i := 0 TO HIGH(a) DO a[i] := 0; END;
```

In this example the control variable is `i` and the progression of values assigned to it is `0, 1, 2, 3, ..., HIGH(a)`.

The control variable must have an ordinal type. It can't be a component of a structured variable (i.e. can't contain any square brackets or uparrows and can contain only qualification dots, not record-field dots); it cannot be imported; it can be a parameter, but it can't be a VAR parameter. The value of the control variable after the end of the FOR-loop is undefined.

The from-value (there's no explicit FROM) and the TO-value can be arbitrary expressions as long as their types are the control variable can be assigned to the from-value and compared to the TO-value. The expressions are evaluated once before the FOR-loop is entered (so assignment statements in the loop body can't affect the progression of values assigned to the control variable). If the from-value is greater than the TO-value in an incrementing FOR, or smaller than the TO-value in a decrementing FOR, then the body of the loop is not executed at all.

There's a more general form of FOR-loop that allows the difference between successive values of the control variable to be something other than 1, including negative differences (which let you iterate downwards instead of upwards). For instance:

```
FOR i := HIGH(a) - 1 TO 0 BY -1 DO
  a[i + 1] := a[i];
END;
```

The BY-value has to be a nonzero constant expression of basetype INTEGER.

A FOR-loop of the form:

```
FOR i := low TO high BY step DO ss END
```

can be considered as an abbreviation for

```
IF step > 0 THEN
  i := low;
  WHILE i <= high DO ss; INC(i, step); END;
ELSIF step < 0 THEN
  i := high;
  WHILE i >= low DO ss; DEC(i, step); END;
END;
```

except that

1. `low` and `high` are evaluated only once and in no particular order (since `step` is a constant, it's not evaluated at runtime at all);
2. the value of `i` after exiting the loop is undefined, whether the exit comes from falling out or from an exception
3. the last INC or DEC may be omitted if it would cause overflow (but it's not guaranteed to be)

The comparison used to determine when to stop looping depends on the sign of the BY-value. If it's positive, the loop body is executed only for values of the control variable that are less than or equal to the TO-value. If it's negative, the loop body is executed only for values of the control variable that are greater than or equal to the TO-value. In other words, you're never allowed to get beyond the TO-value.

Do not change the value of the control variable within the body of a FOR-loop; the behavior of a loop whose control variable has been tampered with is undefined.

10.7. WHILE

Syntax in cheat-sheet notation:

```
s = ...
  | WHILE e DO ?ss END
```

The expression after the WHILE in a WHILE statement has to be of type BOOLEAN. The expression gets evaluated before each execution of the statement sequence, and the repetition stops as soon as it yields the value FALSE. So the statement sequence may never be executed.

Example:

```
WHILE timeToSleep DO
  Thread.Wait(lock, somethingChanged);
END;
```

10.8. REPEAT

Syntax in cheat-sheet notation:

```
s = ...
  | REPEAT ?ss UNTIL e
```

The expression after the UNTIL in a REPEAT statement has to be of type BOOLEAN. The expression gets evaluated after each execution of the statement sequence, and the repetition stops as soon as it yields the value TRUE. So the statement sequence is executed at least once.

Example:

```
REPEAT
  k := i MOD j;
  i := j;
  j := k;
UNTIL j = 0;
```

10.9. TRY FINALLY

Syntax in cheat-sheet notation:

```
s = ...
  | TRY ?ss FINALLY ?ss END
```

The statement sequence following the TRY keyword in a TRY FINALLY statement is called the body, and the statement sequence following the FINALLY keyword is called the cleanup. First the body executes and then the cleanup. But the point of using TRY FINALLY is that once the body starts executing the cleanup will always be executed no matter how the body terminates. There are four ways for the body to terminate:

- falling off the bottom,
- executing an EXIT in a LOOP containing the TRY FINALLY,
- executing a RETURN in a procedure containing the TRY FINALLY,
- and raising an exception whose handler is outside the TRY FINALLY.

For instance:

```
PROCEDURE Copy(from: Rd.T; to: Wr.T);
  BEGIN
    TRY
      WHILE NOT Rd.EOF(from) DO
        Wr.PutChar(to, Rd.GetChar(from));
      END;
    FINALLY
      Rd.Close(from);
      Wr.Close(to);
    END;
  END Copy;
```

The reader and writer always get closed no matter what happens inside the WHILE-loop in the body of the TRY FINALLY.

10.10. LOCK

Syntax in cheat-sheet notation:

```
s = ...
  | LOCK var DO ?ss END
```

In concurrent programming you often need to execute a sequence of statements while holding a monitor lock (`Thread.Mutex`). You do so with the LOCK statement:

```
LOCK self^.lock DO
  IF NOT self^.closed THEN
    INC(self^.count);
  END;
END;
```

This is roughly equivalent to but better than the following TRY FINALLY:

```
TRY
  Thread.Acquire(self^.lock);
  IF NOT self^.closed THEN
    INC(self^.count);
  END;
FINALLY
  Thread.Release(self^.lock);
END;
```

The LOCK version is better because it saves away the address of the monitor lock and therefore works even if you accidentally clobber the variable `self`. (It's also better because you can't forget to say `Thread.Release`.) This might go into the pragmatics section: If you want help from the concurrency powers-that-be, don't use `Thread.Acquire` and `Thread.Release`; use LOCK.

10.11. Exceptions

You declare an exception in order to have it handled in a TRY EXCEPT statement. You call the supplied procedure RAISE to raise an exception. You write a handler in a TRY EXCEPT statement to handle an exception.

10.11.1. Declaring an Exception

You declare an exception by specifying a name and the type of at most one result. For example:

```
EXCEPTION Overflow;
EXCEPTION InvalidCharacter (CHAR);
```

The result type can be anything but an open array type.

Exception names obey normal scope rules; they behave like the names of constants.

Supplied exceptions: There are none.

10.11.2. TRY EXCEPT and RAISE

Syntax in cheat-sheet notation:

```
s        = ...
         | TRY ?ss EXCEPT handlers END
handlers = ?'|  ++(++qi ', ?parId ': ?ss) '| ?(ELSE ?parId ?ss)
         | ELSE ?parId ?ss
```

Exceptions are used in two ways:

- One is to pass control between procedures based on their dynamic relationship; that is, exceptions allow a procedure to pass control upward to the procedure that called it, or to the procedure that called that procedure, etc. An important special case of this use is to get control to the debugger, which behaves for this purpose like the originating procedure.
- The second is to perform a forward go-to within a single procedure. This use is also just a special case of the first, but it has a static meaning within the individual procedure: You can look at the source just for that procedure and understand what's going on.

*

Let's look first at the dynamic case:

At the time when you call RAISE, each procedure activation on the stack has a well-defined notion of its position in the procedure's code. (This position is often called the program counter.) The position of the most recent procedure on the stack is the point at which it called RAISE. The position of the previous procedure is the point at which it called the most recent procedure; the position of the procedure before that is the point at which it called that procedure; and so on.

RAISE examines each procedure activation in turn backwards, starting with the procedure that called it, looking for a handler for the exception being raised and tidying up as necessary while it's looking. In full detail:

- Within each procedure activation, RAISE looks for TRY FINALLY bodies and TRY EXCEPT bodies that contain the current position, starting with the smallest enclosing body.
    - When RAISE finds a TRY FINALLY, it executes the cleanup. If the cleanup raises another exception, the first exception is completely forgotten and the whole process begins again from the new RAISE.
    - When RAISE finds a TRY EXCEPT, it looks for a matching handler, i.e. a handler whose label includes the name of an exception that is the same as the exception being raised.

- When RAISE reaches a procedure boundary without finding a matching handler, it executes essentially the same cleanup action that takes place during a procedure return, freeing the storage allocated by the procedure activation. Then it starts looking at the procedure activation that called the one just freed.

Once RAISE finds a matching handler, RAISE passes control to the statement sequence of the handler. If the handler's statement sequence terminates normally, control passes to the next statement after the END of the TRY EXCEPT.

Some handlers have a parenthesized variable name between the label and the statement sequence. An exception can be declared to return a value or not; see Section 10.11.1, page 50. If the declaration does include a return value, the RAISE must include as a second argument an expression whose type is assignable to the type of the return value, and the handler may (or may not) decide to make use of the return value. If the handler does plan to use the return value, it includes a parenthesized variable name between the label and the statement sequence. Before passing control to the statement sequence, RAISE assigns the exception's return value to the parenthesized variable. So that's why the type of the exception's return value must be assignable (see Section 15.6, page 86) to the type of the variable.

Strange restriction: The parenthesized variable cannot be a ref-containing global variable.

There's nothing unusual about the execution environment of handlers and cleanups. The only thing that's unusual is that they got invoked by RAISE.

Stylized example:

```
TRY
  s1;
  TRY s2; s3; FINALLY s4; END;
  s5;
EXCEPT
| e: s6;
END;
```

If no exceptions are raised, the sequence of execution will be:

```
s1; s2; s3; s4; s5
```

If exception e is raised in s1, the sequence will be:

```
s1 (partially); s6
```

If exception e is raised in s2, the sequence will be:

```
s1; s2 (partially); s4; s6.
```

If RAISE finds no matching handler anywhere in the stack, it calls the debugger. In this important special case, RAISE preserves the execution context and refrains from doing any cleanup, so as to give the debugger access to the context in which the exceception was raised. The original execution context is also preserved if RAISE finds a matching handler whose statement sequence consists solely of a single call to RAISE.

If you happen to be writing a debugger or some program that plays a similar

role, you will want to know about ELSE-handler, which uses the keyword ELSE instead of a label and catches all exceptions. If there's a parenthesized variable in an ELSE-handler, it should have type `System.FailArg`.

*

So much for the dynamic case. You can also use exceptions to perform forward go-tos within a single procedure. Using RAISE may seem clumsy for this purpose, but it can be the best way to achieve your aim:

```
EXCEPTION ExitForLoop;

...

  TRY
    FOR i := 0 TO HIGH(a) DO
      IF a[i] < 0 THEN RAISE(ExitForLoop); END;
    END;
  EXCEPT
  | ExitForLoop:
      Wr.PrintF(Stdio.stdout,
        "Found negative.\n");
  END;
```

You can also use RAISE to exit nested LOOPs.

10.12. TRY PASSING

Syntax in cheat-sheet notation:

```
s = TRY ?ss PASSING exSet ?'; END
```

TRY PASSING is a shorthand for exception handling:

```
TRY ss; PASSING {ex1, ex2}; END;
```

means

```
VAR failArg: System.FailArg;

...

  TRY
    ss;
  EXCEPT
  | ex1: RAISE(ex1);
  | ex2: RAISE(ex2);
  ELSE (failArg) RAISE(System.Fail, failArg);
  END;
```

This is a way of doing at the block level what a RAISES clause does at the procedure level (see page 64).

11. Supplied Procedures

Here's a classification of the supplied procedures:

```
Numbers
    ABS DEC INC MAX MIN ODD
Conversions
    CHR FLOAT LONGFLOAT LOOPHOLE NARROW ORD TRUNC VAL
Sets
    EXCL INCL
Storage Allocation
    NEW DISPOSE
Control
    ASSERT HALT RAISE
Type Deconstructors
    FIRST LAST HIGH NUMBER
```

We're going to try to present these supplied procedures as if they were in a definition module. But many of the functions can't be expressed in ordinary Modula-2+. Liberties are taken as necessary when the language is inadequate to describe a procedure, e.g., for LOOPHOLE, which takes a TYPE parameter; or NEW, which applies to parameters of any REF type; or ABS, which returns a value of the same type as its parameter.

[[More work needed on the descriptions to fit in the new stuff on type-checking.]]

```
(*****************************************************************)
(*                                                               *)
(* Notation:                                                     *)
(*    <>          angle brackets enclose informal descriptions   *)
(*    := empty    means the parameter can be omitted             *)
(*                                                               *)
(*****************************************************************)
```

11.1. Numbers

```
PROCEDURE ABS(x: INTEGER): INTEGER;
PROCEDURE ABS(x: REAL): REAL;
PROCEDURE ABS(x: LONGREAL): LONGREAL;
  (* Returns the absolute value of x. *)

PROCEDURE DEC(VAR x: T; n: CARDINAL := 1);[1]
  BEGIN
    IF BaseType(x) IN {INTEGER, UNSIGNED} THEN
      x := x - n;
    ELSIF BaseType(x) = <an enumerated type> THEN
      x := VAL(T, ORD(x) - n);
    ELSE
      ERROR
    END;
  END DEC;

PROCEDURE INC(VAR x: T; n: CARDINAL := 1);
  BEGIN
    IF BaseType(x) IN {INTEGER, UNSIGNED} THEN
      x := x + n;
    ELSIF BaseType(x) = <an enumerated type> THEN
      x := VAL(T, ORD(x) + n);
    ELSE
      ERROR
    END;
  END INC;

PROCEDURE MAX(a, b: Numeric): Numeric;
  (* Returns the larger of two values.  a and b must be compatible;
     see Section 15.5.1, page 83. *)

PROCEDURE MIN(a, b: Numeric): Numeric;
  (* Returns the smaller of two values.  a and b must be compatible;
     see Section 15.5.1, page 83. *)

PROCEDURE ODD(x: INTEGER): BOOLEAN;
PROCEDURE ODD(x: UNSIGNED): BOOLEAN;
  (* ((x MOD 2) # 0)
     Returns TRUE if x is odd, FALSE if x is even. *)
```

[1]Bug in DEC and INC: The size in bits of the parameter, as obtained by calling System.Size(x), must be 8, 16, or 32. Underflow checking is done incompletely, if at all. The parameter must be byte-aligned; if it isn't, the result is undefined.

11.2. Conversions

```
PROCEDURE CHR(x: INTEGER): CHAR;
PROCEDURE CHR(x: UNSIGNED): CHAR;
PROCEDURE CHR(x: System.Word): CHAR;
PROCEDURE CHR(x: System.Byte): CHAR;
  (* Converts a number into a character with that representation.
     CHR is a shorthand for VAL:
        CHR(x) = VAL(CHAR,x)
     No range checking is done; the result of doing (for instance)
     CHR(899) is undefined. *)

PROCEDURE FLOAT(x: Numeric): REAL;
  (* Converts a number into a REAL.

     If the basetype of the parameter is INTEGER, FLOAT returns
     the REAL that most closely approximates that integer --
     some precision may be lost.  If the basetype is LONGREAL,
     FLOAT simply truncates the extra digits of precision
     (rather than rounding).  If the basetype is REAL, FLOAT
     just returns the parameter value. *)

PROCEDURE LONGFLOAT(x: Numeric): LONGREAL;
  (* Converts a number into a LONGREAL.

     If the basetype of the parameter is INTEGER, LONGFLOAT
     returns the REAL that represents that integer -- no
     precision is lost.  If the basetype is REAL, LONGFLOAT
     simply expands it without changing the number it
     represents.  If the basetype is LONGREAL, LONGFLOAT just
     returns the parameter value. *)

PROCEDURE LOOPHOLE(e: T; X: TYPE): x;
  (* Converts a value from one type to another without changing
     the representation of the value.  Not even the number of
     bits in the representation can change, i.e. System.TSize(T)
     must equal System.TSize(X).

     [[  Type-checking information needed.  ]] LOOPHOLE takes an
     expression of any type [[  ?  ]] as its first parameter and
     the name of a type [[  ?  ]] as its second parameter.  It
     returns the value of type X that has the same bit pattern
     as the parameter e.  The size of e (i.e., the number of
     bits required to represent a value of its type) must be
     identical to the size of a value of type x.  For more
     information, see Section 16.6, page 96. *)

PROCEDURE NARROW(r: REFANY; t: TYPE): t;
  (* Converts a value from type REFANY to a ref type.

     NARROW is a convenient shorthand for TYPECASE (Section
     10.4, page 45).  For example:
        x := NARROW(ref, T);
     is equivalent to:
        TYPECASE ref OF
        | T(x): (* empty statement sequence *)
        | ELSE RAISE(System.NarrowFault, ref);
        END;

     If the first parameter to NARROW is NIL, NARROW returns
     NIL. *)
```

```
PROCEDURE ORD(element: T): CARDINAL;
  (* Converts an element of an enumeration to its position in the
     enumeration order (so that you can do some kind of
     arithmetic on the ordinal value and then convert back at
     some later point using VAL). The ordinal value of an
     enumeration element is determined by the original
     declaration and doesn't change through subsequent
     application of constructions like subrange or BITS FOR.
     All enumerations start from 0. *)

PROCEDURE TRUNC(r: RealOrLongreal): INTEGER;
  (* Converts a REAL or LONGREAL to the integer obtained by
     truncating towards 0. If the truncated value cannot be
     represented as an INTEGER, the result will be some
     undefined value with no error indication. *)

PROCEDURE VAL(T: Enumeration; e: S): T;
  (* Converts an enumeration type and a position into the element
     with that position in the enumeration type. If T is a
     subrange, VAL returns the element with position S in the
     original enumeration type, not in the subrange. *)
```

11.3. Sets

```
PROCEDURE EXCL(VAR s: S; e: E);[2]
  (* s := s - S {e};

     Excludes element e in set s. The basetype of S must be a
     set type with element type E. *)

PROCEDURE INCL(VAR s: S; e: E);
  (* s := s + S {e};

     Includes element e in set s. The basetype of S must be a
     set type with element type E. *)
```

[2]Bug in INCL and EXCL: The set parameter must be byte-aligned; if it isn't, the result is undefined.

11.4. Storage Allocation

```
(* The NEW procedure has three forms:

   (1) PROCEDURE NEW(VAR v: RefOrPointer);
   (2) PROCEDURE NEW(VAR a: RefOpenArray; nElements: CARDINAL);
   (3) PROCEDURE NEW(VAR v: RefVariantRecord; tag: Tag);

  NEW allocates storage at runtime for a new variable to which a
  ref or a pointer will refer. *)

PROCEDURE NEW(VAR v: RefOrPointer);
  (* (1) Use the basic form whenever the type of v is a ref except
         when it's a ref to an open array or an immutable variant
         record with only one tag field.

         Use the basic form also whenever the type of v is a
         pointer.  You can use NEW of a pointer to allocate
         storage from a private storage allocator that you've
         built for reasons of your own; see Section
         18.6, page 107. *)

PROCEDURE NEW(VAR a: RefOpenArray; nElements: CARDINAL);
  (* (2) When the first parameter to NEW is a ref to an open array,
         the second parameter gives the number of elements. *)

PROCEDURE NEW(VAR v: RefVariantRecord; tag: Tag);
  (* (3) When the first parameter to NEW is a ref to a variant
         record, you can supply a tag value as the second
         parameter.  This value is assigned to the first
         top-level tag field of the record.  If the value of a
         field of a record is itself a variant record, there is
         no way to set its tag with NEW; and there's no way to
         set anything beyond the first top-level tag with NEW.

         NEW allocates enough space to hold the largest variant
         no matter what tag you supply. *)

PROCEDURE DISPOSE(VAR v: T);
  (* If you used NEW with a pointer, you use DISPOSE to free
     the storage allocated by NEW.

     If T is a ref, DISPOSE is a no-op. *)
```

11.5. Control

```
PROCEDURE ASSERT(condition: BOOLEAN; message: ARRAY OF CHAR := empty);
  (* If the condition is FALSE, ASSERT aborts the program, dumping
     core and printing the message, if there is one.  ASSERT
     generates code even if the condition is a constant
     expression that evaluates to TRUE. *)

PROCEDURE HALT(n: CARDINAL := 0);
  (* Terminates an Ultrix process or a Topaz address space and
     never returns.  Returning from or falling off the bottom of
     the main program causes HALT(0).

     By convention, n = 0 means that the program completed
     normally, n # 0 means that it failed in some way.  Most
     programs use n = 1 to indicate failure.

     HALT first "cleans up", which basically means flushing
     stdout and stderr (see CleanUp.def).  It then calls an exit
     routine.  Under Ultrix this is the C library routine exit;
     under Topaz, it's OS.Exit. *)

PROCEDURE RAISE(e: EXCEPTION; arg: T);
PROCEDURE RAISE(e: EXCEPTION);
  (* You call RAISE to raise an exception.  It never returns.
     If the exception was declared with a result of type T, you
     have to supply the second parameter (passable as a T);
     otherwise you're not allowed to use a second parameter.

     See Section 10.11.2, page 51, for a detailed description of the
     semantics of RAISE. *)
```

11.6. Type Deconstructors

```
PROCEDURE FIRST(T: TYPE): t;
  (* Returns the first (smallest) value of a type.

     After peeling off any layers of BITS type declarations (but
     not subrange), T must be an enumeration or a subrange type
     or INTEGER, UNSIGNED, REAL, or LONGREAL.  [[  If T was a
     BITS FOR, is the type of the result the BITS FOR type or
     the basetype?  ]]  [[  What about System.Address?  ]]

     For an enumeration type, the result is the first (leftmost)
     enumeration constant in the list.  For a subrange type, the
     result is the smallest value in the subrange.

     For UNSIGNED, the result is 0.  For INTEGER, the result is
     -2**31.  For REAL, FIRST returns the smallest number
     representable as a REAL, which has value approximately
     -1.701412E38.  For LONGREAL, FIRST returns the smallest
     number representable as a LONGREAL, which has value
     approximately -1.701411834604692E38. *)

PROCEDURE LAST(t: TYPE): t;
  (* Returns the last (largest) value of a type.

     After peeling off any layers of BITS type declarations, t
     must be an enumeration or a subrange type or INTEGER,
     UNSIGNED, REAL, or LONGREAL.

     For an enumerated type, the result is the last (rightmost)
     enumeration constant in the list.  For a subrange type, the
     result is the largest value in the subrange.

     For UNSIGNED, the result is 2**32 - 1.  For INTEGER, the
     result is 2**31 - 1.  For REAL, FIRST returns the largest
     number representable as a REAL, which has value
     approximately 1.701412E38.  For LONGREAL, FIRST returns the
     largest number representable as a LONGREAL, which has value
     approximately 1.701411834604692E38. *)

PROCEDURE HIGH(a: T): CONSTANT;
  (* Returns the largest index of array a.  The basetype of a
     must be an array or open array.  The result is a constant
     of the index type of the array. *)

PROCEDURE NUMBER(a: T): CARDINAL;
  (* The basetype of a must be an array or open array type.
     @NOCOUNT open arrays are not allowed.

     NUMBER returns the number of elements of a as its result.
     This will be a constant if the basetype of a is an array. *)
```

12. Procedures

12.1. Procedure Declarations

Remember that the keyword PROCEDURE is used both for procedures and for procedure types. In cheat-sheet notation:

```
type     = ...
         | PROCEDURE tFormals   ?( ': ti ) ?(RAISES exSet)
procH    = PROCEDURE id formals ?( ': ti ) ?(RAISES exSet)
```

In this section we're going to be talking about procedures, not procedure types.

Some idealized procedure headings:

```
PROCEDURE P();
PROCEDURE P(param: T; param: T; param, param, param: T);
PROCEDURE P(param: T := Constant);
PROCEDURE P(VAR param: T);
PROCEDURE P(param: T): T;
PROCEDURE P(): RAISES {Exception, Exception};
```

The heading of a procedure declaration specifies the procedure's name and its "formal" parameters; its result, if any; and the limits, if any, on the exceptions it can raise. The procedure body contains declarations and statements. The procedure's name is repeated at the end of the procedure body.

For instance:

```
PROCEDURE Fill(
    VAR a: Rd.T;
    b: INTEGER := 65)
    : Wr.T
  RAISES {Rd.ScanFailed};
  BEGIN
    ss;
  END Fill;
```

If the procedure heading specifies a result, the body of the procedure must contain one or more RETURN statements; you'll get either a compile-time or a runtime error if you omit a necessary RETURN statement. The expression you return must be assignable to the type of the result you declared; see Section 15.6, page 86.

A formal parameter name stands for the actual parameter specified when the procedure gets called. There are two kinds of formal parameters, "VAR parameters" and "value parameters." VAR parameters are indicated by the symbol VAR, value parameters by the absence of the symbol VAR. For example:

```
PROCEDURE Search(
    VAR (*in*) str: ARRAY OF CHAR;
    text: Text.T)
    : INTEGER;
```

`str` in this example is a VAR parameter, and `text` is a value parameter.

VAR parameters do call by reference, not call by value return. [[I don't know the right way to express that thought.]] There are several reasons why you might want to make a parameter be a VAR parameter: because you want to avoid the expense of copying a large value, because you want to pass a value out by means of the variable, or (least likely) because you're planning to share information with another thread by means of the variable. Otherwise you would write a value parameter.

Anything you declare within the procedure body is local to the procedure; see Section 2.3, page 8.

Procedure declarations can be nested.

Procedures can be recursive or mutually recursive; no special declaration is required.

You can restrict what exceptions a procedure can raise to a specified set by attaching a RAISES clause to its declaration. The set is the union of the set you specify (it's OK to specify an empty set) and System.Fail. At execution time, if your procedure raises an exception that you didn't mention in your RAISES clause, the exception is converted to System.Fail. The result passed by System.Fail identifies the original exception (and its result, if any), so that no information is actually lost.

The RAISES clause is part of the type of a procedure, so it affects type-checking; see page 84.

12.2. Procedure Calls

There are two kinds of procedure call:

- procedure call as a statement, for a procedure that returns no value;
- and procedure call as part of an expression, for a procedure that does return a value.

The correspondence between the actual parameters, the ones that you give to a procedure in calling it, and their matching formals, the ones that appeared in the declaration, depends entirely on ordering -- the first actual matches the first formal, the second actual matches the second formal, etc.

See Section 15.7, page 87, for what kinds of actuals you can pass to what kinds of formals.

When you call a procedure, the actual parameter you supply to a VAR formal has to be a variable. The identity of the variable is determined at the time of the call and does not change during the call. If you use the BITS FOR constructor to control the layout of variables in memory, you may not be able to pass those variables by VAR; see Section 16.2, page 92.

The actual parameter you supply to a value formal can be any (passable) expression. The expression is evaluated at the time of the call and is then assigned to the formal. The name of the formal behaves like a local variable inside the procedure.

Example of procedure calls:

```
ch := Rd.GetChar(Stdio.stdin);
Wr.PutChar(Stdio.stdout, ch);
```

12.3. Default Parameters

You can specify constant values as defaults for formal value parameters in both procedure headings and procedure types.

A formal parameter without a default cannot appear after a formal with a default.

Matching between actuals and formals works as usual, but if there are fewer actuals than formals and the remaining formals have defaults then the default values get supplied. That is to say, if you have four formals, all with defaults, and two actuals, then the first actual matches the first formal, the second actual matches the second formal, and the third and fourth formal default. There's no way to get the default for the third formal and supply an actual for the fourth formal.

Supplying default parameters allows you to make upward-compatible changes that satisfy new clients, while requiring nothing more of your old clients than recompilation. You can also use default parameters to highlight the normal-case use of a procedure. And you can sometimes use default parameters to get the effect of having a variable number of parameters.

Thus you can declare a procedure such as:

```
PROCEDURE Equal(
    t1, t2: Text.T;
    ignoreCase: BOOLEAN := FALSE)
    : BOOLEAN;
```

or a procedure type such as:

```
TYPE
  CompareProc =
    PROCEDURE(
      Text.T,
      Text.T,
      REFANY := NIL)
      : Text.Comparison;
```

Examples of calling such procedures:

```
VAR
  compareProc: CompareProc;
BEGIN
  ASSERT(Text.Equal(t, "abcd"));
  ASSERT(compareProc(t, "abcd") = Text.Eq);
```

Defaulting is a very limited mechanism. You can't default arrays and records (since there aren't any constants available) and you can default procedures, refs, and pointers only to NIL.

If you specify defaults in an interface you have to specify the same ones in its implementation. Similarly, if a procedure type specifies defaults, any procedure assigned to a variable of that type must specify the same defaults.

12.4. Notes

```
Date: Tue, 7 Jan 86 00:27:55 pst
From: luca (Luca Cardelli)
To: mcvl, rovner, violetta
Subject: More unspecified semantics

  I couldn't find any discussion on passing and returning
procedures from other procedures, either in Wirth's book or in
mcvl's manual.  Did I miss it?

The modula2+ manual says that "The result of a function
procedure can be of any type, including the structured types,
except open array." This is just too good to be true. My guesses
are:

1) procedures cannot be returned from procedures (ever? can they be
   returned if declared at the top level? or if declared at a level
   lower than the level of the procedure which returns them?)

2) procedures can be passed to other procedures only if the former
   are declared at the top level (would it be hard to relax this
   to allow passing procedures which are declared at a lower level
   than the level of the procedure they are given to?)

Note that all the above options respect the stack discipline, hence
they could be implemented without having real closures.

Oh! I just noticed: there is this line which says that "Variables of a
procedure type T may assume as their value a procedure P ... [which]
must not be declared local to another procedure".  I assume this
refers to "local variables", not to "formal parameters", otherwise
it would even forbid passing procedures to procedures.

        Luca

"Heavens! Why would you want to do THAT??"
```

From: rovner (Paul Rovner)
Date: 7 Jan 1986 1021-PST (Tuesday)
To: luca (Luca Cardelli)
Cc: mcvl, rovner, violetta
Subject: Re: More unspecified semantics
In-Reply-To: Your message of Tue, 7 Jan 86 00:27:55 pst.

Luca,

Good questions.

"top-level" procedures can be passed as args, returned as results, assigned to variables, etc. Our reference manual should say this.

Nested procedures cannot be treated as values, only called by name in an enclosing scope (see the last paragraph on page 82 of Wirth, ed. 3; you found it, apparently, and quoted it in your last paragraph, which I do not understand).

Yes, it would be hard to "allow passing procedures which are declared at a lower level than the level of the procedure they are given to." Come see me for details if you want ... example: it would be necessary (and difficult) in the callee to distinguish nested from top-level actuals.

Paul

13. Modules

There are four kinds of modules: definition, implementation, main, and nested. We don't often use nested modules, so we'll pretend they don't exist and then explain their special properties in a separate section, Section 13.7, page 73.

13.1. Definition Modules

The point of a definition module is to make the objects it declares available to clients, other modules that import it. Definition modules are also called interfaces. To find out about import, see Section 2.5, page 11.

A definition module contains constant, type, variable, and exception declarations and procedure headings. It does not include complete procedure declarations or executable statements.

It is an error for a definition module to import itself. It is also an error for two or more definition modules to form an import cycle (A imports B, B imports A).

13.2. Implementation Modules

An implementation module contains local objects and statements hidden from the clients of its definition module.

Most definition modules have matching implementation modules. Your definition module can get away without an implementation module only if you declare no opaque ref types (page 71) and no procedures or variables other than pass-throughs (page 72).

An implementation module automatically gets all the declared and imported names from its definition module. Opaque types have to be made concrete and procedure headings need to be repeated and filled out with procedure bodies, but other types, variables, constants, and exceptions and all imports are available with no further ado.

An implementation module is not actually obliged to implement everything (or for that matter anything) in its matching definition module.

13.3. Opaque Types

In a definition module, you can declare a type without giving the type's complete definition. Such a type is called "opaque." You declare an opaque type to hide the details of the complete definition, called the "concrete type," from the importers of the definition module.

There are two sorts of opaque types, opaque word types and opaque ref types. You declare an opaque word type by saying:

```
TYPE
  T;
```

The importer knows that the concrete type is a one-word non-ref type. ("Word" means a word of storage.) You declare an opaque ref type by saying:

```
TYPE
  T = REF;
```

The importer knows that the concrete type is a ref type (but not the type of the referent).

These declarations provide the importer with enough information so that she can do an assignment to a variable of the opaque type. But she can't use any of the other operations normally provided for user-defined types like:

```
=     +     ^
```

[[Not true, you can do "=" on opaque ref. Crumbs.]]

She gets access to the concrete type by calling procedures implemented in modules that define the concrete type.

She can declare variables of the opaque type and she can create variables of the opaque type with NEW. In the case of opaque ref types, she cannot create a variable of the (unknown) referent type because she doesn't know the size.

13.3.1. Opaque Word Types

If T is an opaque word type declared in definition module D, then the module implementing D declares the concrete type T using a standard type declaration. For instance:

```
TYPE
  T = INTEGER;
```

The concrete definition of an opaque word type is usually a pointer or a discrete-number type. In any case, the concrete definition of T must not be a ref type or REFANY and must satisfy:

```
System.TByteSize(T) = System.TByteSize(System.Word)
```

13.3.2. Opaque Ref Types

If T is an opaque ref type declared in definition module D, then the module implementing D declares the concrete type T using a standard type declaration. For instance:

```
TYPE
  T = REF INTEGER;
```

The concrete definition of an opaque ref type must be a ref type. It may not be REFANY.

Here's an example of the common case, in which the concrete type is defined in the implementation module:

```
SAFE DEFINITION MODULE Dog;
IMPORT Text;
TYPE
  T = REF; (* opaque *)

PROCEDURE Wash(name: Text.T): T;
END Dog.
```

```
SAFE IMPLEMENTATION MODULE Dog;
TYPE
  T = REF DogRec; (* concrete *)
  DogRec = RECORD k: [0 .. 9]; f: Text.T; END;

PROCEDURE Wash(name: Text.T): T;
  (* code that implements Wash *)
  BEGIN ss; END Wash;

BEGIN ss; END Dog.
```

But a feature of opaque ref types (not of opaque word types) allows any module that imports D to make a concrete definition for T, as long as all the modules making concrete definitions agree about them. (In fact, the implementation of D doesn't have to include a concrete definition for T if it doesn't want to.) Since every application of a type constructor (including REF) produces a unique type, the way for two implementation modules to make the same concrete definition is to import the type from the same interface.

You'll get a error at runtime during module initialization if two modules make different concrete definitions for the same opaque ref type.

13.4. Pass-Throughs

In a definition module you can specify "pass-through" values for declared procedures, exceptions, and variables. For instance:

```
PROCEDURE InitMutex(VAR s: Mutex)
  = ThreadsPort.InitializeMutex;

EXCEPTION Alerted = ThreadsPort.Alerted;

VAR
  n: CARDINAL = Params.numParameters;
```

This kind of declaration equates the identifier on the left of the equals sign with the one on the right.

The type-checking for pass-throughs is:

- For the PROCEDURE declaration, the right-hand side must be the name of a procedure and the types of the two sides must not just match (page 84) but have the same names for corresponding parameters.
- For the EXCEPTION declaration, the right-hand side must be the name of an EXCEPTION. Either both sides have no parameter or both sides have a parameter and the parameter types are the same.
- For the VAR declaration, the right-hand side must be the name of a variable and the types of the variables on both sides must be the same.

The pass-throughs make it possible for you to implement a single interface with multiple modules:

```
SAFE DEFINITION MODULE Umbrella;
IMPORT Part1, Part2, Text;

TYPE
  T = Part1.T;

PROCEDURE Create(name: Text.T): T = Part1.Create;

PROCEDURE Print(t: T) = Part2.Output;
END Umbrella.
```

Unfortunately, your client is compilation-dependent on the interfaces to those multiple definition modules, `Part1` and `Part2`, which have more of the flavor of implementation than most public interfaces and are fairly likely to change.

13.5. IMPLEMENTS

Sometimes you want to export more than one interface from one implementation module, for instance because most of your clients use only a small fraction of the types and procedures you're making available. Providing a smaller interface to those clients allows them to reduce their compilation dependencies.

If you're writing an implementation module that uses IMPLEMENTS, you get the names from all the definitions modules you're implementing in the order in which you name them:

```
SAFE MODULE M IMPLEMENTS A, B, C;
```

It's up to you to avoid name conflicts. Vacuous redefinitions are OK (page 9).

If you're using the IMPLEMENTS feature, implementation module M no longer automatically implements definition module M. If in implementation module M you want to implement definition module M, you have to name it explicitly:

```
SAFE MODULE M IMPLEMENTS M, N;
```

As usual, only one implementation module is allowed to implement a given definition module.

13.6. Main Modules

A main module is essentially an implementation module with no definition module. All you can do with a main module is run it.

[[This seems like the place to talk about linking ... also maybe the imc?]]

13.7. Nested Modules

Nested modules are a technique for structuring large implementation or main modules without introducing separate interfaces.

A nested module can import things only from its containing module. In other words, if a nested module wants to use Rd, its containing module has to import Rd; the nested module has no way of reaching outside its containing module to other modules.

The EXPORTS list in a nested module header makes names from within the nested module available to its containing module. If EXPORTS is followed by QUALIFIED, these names must be qualified in the containing module with the name of the nested module.

If a procedure is declared in an interface, its implementation can't be in a nested module; it has to be at the outer level of its implementation module.

13.8. Initialization

The statement sequence before the END of a module is its body. Every main module contains a body; its body is its reason for being. The typical implementation module contains a body too; the purpose of an implementation module's body is to initialize module variables.

A module expects its body to execute before its procedures are called from outside. There is an invisible Boolean variable, "`initialized`," that's associated with each implementation module. Suppose you have an implementation module D; the compiler generates code for the body of D that looks something like this:

```
PROCEDURE _init();
  BEGIN
    IF initialized THEN RETURN; END;
    initialized := TRUE;
    FOREACH Module IN <imports> DO
      Module._init();
    END;
    <explicitly programmed statements of body, if any>
  END _init;
```

Now suppose you take a group of modules that use this scheme for initialization and you link them together and run them. If they have no import cycles and if within each module no threads are forked until the first explicit statement of that module body is executed, then the body of each module will execute before any of its procedures are called from outside. [[Hard to follow; needs another pass.]]

If a module uses a constant from another module or declares a variable to have a type that's imported from another module, that alone will not trigger initialization of the other module unless the type of the variable is an opaque ref type.

An import cycle is a chain of imports of the form implementation module `A` imports definition module `B` and implementation module `B` imports definition module `A` (with as many steps in between as you need to disguise this effect). Guarding against import cycles is your responsibility.

A cyclic import structure may result in unpleasant surprises at runtime. For example, suppose module `A` contains procedure `P`, which relies on the initialization of module `A` to execute correctly; module `A` imports module `B`, and `B` calls `A.P` as part of `B`'s initialization. When `A`'s initialization procedure is called, the first thing it does is to call the initialization procedure of `B`. That calls `A.P`. `A.P` executes incorrectly because `A` has not yet been completely initialized.

If multiple threads of control exist during initialization, other dangers arise. A scenario similar to the one above can occur even if imports are not cyclic. In addition, it would be possible for a module body to execute twice if one thread begins executing the body at almost the same time as another does, and before either has had a chance to set "`initialized`" TRUE.

14. Safety

14.1. Living in Harmony with the Garbage Collector

From: rovner (Paul Rovner)
Date: 13 Mar 1986 1749-PST (Thursday)
To: src (Modula-2+ programmers)
Subject: Guidelines for programming with garbage collection

Ignore this (long) message if you never write code that either uses or rubs shoulders with REFs. Remember that Wr.T's, Rd.T's and Text.T's are REFs.

Thanks to Tom Rodeheffer for suggestions on the presentation.

The collector is almost ready to be released (this is NOT a release message). This means that our consciousness needs to be raised anew about "storage safety" ... what it means, why you care, and how you can continue appearing to live cleanly even with the Garbage Collector watching your every move.

A program is "storage safe" if it does nothing that could break the collector. This is a good property for a program that lives with other programs in an address space managed by the collector to have.

A broken collector is not satisfied to sit there quietly. Misery loves company. The collector makes it a point when broken to go around and break as many other programs as it can find. Sometimes including yours. And usually in mysterious and wondrous ways to ensure that suitable penance is paid by its master (me) to figure out what happened.

Storage safety is much less of an issue when the collector is turned off, as it has been until very recently in Taos. Programs that do things that would break the collector often run just fine in its absence. You have often found it easy, no doubt, to forget when defining modules to declare them SAFE.

The SAFE declaration enables compile-time checks that point out potential safety violations in your code. The action of a module declared SAFE is extremely unlikely to break the collector. The more SAFE modules, the more reliable our systems will be. Warm feelings accompany compiler acceptance of SAFE modules.

But not all modules can be declared SAFE. Ones that provide low-level facilities or have critical performance requirements often require the use of unsafe features. This does not mean that such code will break the collector, only that the compiler is not smart enough to "prove" that it won't. Generally, there should be few unsafe modules, and the reasons for their existence should be both good and well-understood.

Exhortation

Accordingly, I hereby exhort you not only to cease forgetting to declare your modules SAFE, but also to work as necessary to

understand clearly why a module that cannot be declared SAFE must remain in such a sinful state. I would be happy (anxious, even) to help in such matters. And if language or compiler changes are shown to be necessary as we proceed, we will make them.

Generally, we should be willing to pay up to (say) a 15% performance penalty for new code to be declared SAFE, and we should be willing to cause modest disruption as necessary to make interface changes that enable client modules to be declared SAFE.

Technical stuff

Current known safety holes

(All but the last item below will be removed from this list eventually)

(*) Thread stack overflow is not detected.

(*) Bogus procedure descriptors can be created and called, even in SAFE modules.

(*) There is inadequate checking for the use of unsafe procedures imported from the System module by SAFE modules, e.g., System.Copy.

(*) Changing a variable upon which an active VAR parameter or WITH head is based can break the collector. Examples:

Passing a VAR actual parameter that addresses a field of a collectible object, then while the callee is active deleting the last accessible REF to the object (e.g. in a concurrent thread, or via the callee's use of an alias) and triggering a collection that reclaims the object, then storing through the (dangling) VAR parameter in the callee. Smash.

Deleting the last REF to an object by the action of (e.g.) the body of a WITH statement when a field of the object is referenced by the WITH head. If a collection is triggered and reclaims the object before code in the WITH body stores through the address computed for the WITH head, such a store will smash memory.

(*) There are many ways to break the collector by making concurrent access to REFs in counted storage outside the protection of a Mutex when at least one of the threads is making changes. Don't do it without seeing me first.

If your programs are all declared SAFE, you can stop reading here.

Unfortunately, a deep understanding of all that follows requires knowledge of some messy implementation details. Think of this as the price for not using SAFE.

Terminology: "RC" means "REF-containing." "Counted storage" includes only global variables and storage allocated by NEW for a REF variable.

Assignment of an RC value is "reference-counted" only if the left-hand side names a global variable, or a dereferenced REF, or a VAR parameter that addresses the low segment (bit 30 = 0). Notice that I did not say "... if the left-hand side addresses counted storage" because the left-hand side may name a

dereferenced POINTER that happens to address counted storage; such an assignment will not be counted.

A reference-counted assignment causes the count in the header of the object referenced by the old contents of the target cell to be decremented, and the count in the header of the object referenced by the new contents to be incremented. NIL is not counted.

As it turns out, the code doing the reference-counted assignment does not itself change the object headers, instead, it drops an inc-this/dec-that entry into a queue for the collector to process when it gets around to it.

The list below itemizes "unsafe" language features, with notes for each explaining why it is unsafe. It is OK to use these features IF you understand fully what the dangers are in each specific case AND you have arranged to avoid them somehow. Most of you should not use these features; if you must, please speak with me before you do.

Storing a value through a POINTER.
  This will smash memory if the pointer is bogus.
  RC values stored through a pointer are not counted and will not be traced by the (not yet implemented) trace and sweep collector.

Passing a dereferenced pointer as a VAR parameter.
  Assignment in the callee will smash memory if the pointer is bogus. RC assignment in the callee may or may not be counted. If it is counted, and a bogus old RC value resides in the target, decrementing it will smash memory. The SAFE callee might read a bogus RC value thru the ptr. Bogus RC values are bad because memory will get smashed by count maintenance or store-thru's.

Reading an RC value through a POINTER.
  Might pick up a bogus RC value, e.g., thru a bogus pointer.

Loopholing a value to an RC value.
  Might get a bogus RC value.

Changing the tag of a variant record which (record) has RC arms.
  Might pick up a bogus RC value, either to use or to decrement.
  Might lose track of an RC value, leaving a dangling reference.

Running without NIL or subscript-bounds checking.
  Might smash memory. Might pick up a bogus RC value.

Calling a procedure imported from an unsafe interface.
  The callee might break the collector.

Passing an RC variable as a VAR parameter of a different type, e.g., open array of System.Word.
  Might lose track of an RC value. Callee might write a bogus value, to be picked up later by the caller.

Additional notes for programmers who live on the edge

All REFs in RC local variables are initially NIL.

If you want to guarantee that an object will not be collected when there are no references to it in counted storage that are accessible to the program, arrange to leave a 32-bit pattern identical to the REF value in a quad-byte aligned cell on the active stack of some "alive" thread.

```
BEWARE of passing a dereferenced pointer to an RC value as a VAR
parameter. Come see me if you have code or plan to have code
that does this.


Come see me if you have questions.

Paul
```

14.2. Notes

A nested module is automatically considered safe if the module it is nested in is SAFE.

```
Imports in a SAFE DEFINITION module must now really be safe
(i.e. no variable or procedure imports from unsafe modules),
even if the corresponding IMPLEMENTATION module is not safe.

If a SAFE IMPLEMENTATION module has a corresponding unsafe
DEFINITION module, the imports in the DEFINITION module must be
safe because it is a part of a SAFE module.

Another way of thinking about this is as follows. Let

         defIsSafe := (definition is safe)
         implIsSafe := (implementation is safe)

SAFE checking is turned on while processing an IMPLEMENTATION
module if implIsSafe is TRUE. Safe checking is turned on while
processing a DEFINITION module if either implIsSafe or defIsSafe
is TRUE.
```

It is no longer an error to declare a ref-containing variant record type with no variant tag, e.g.

```
RECORD
  CASE BOOLEAN OF
  | TRUE:  ref: REFANY;
  | FALSE: adr: System.Address;
  END;
END;
```

in SAFE module. However, accessing the fields of such a variant is an error and trying to assign wholesale to any structure containing such a record is unSAFE.

The compiler ensures that an implementation marked SAFE is actually safe by enforcing the following rules:

- Array bounds checking is enabled in SAFE modules.
- NIL checking (detecting attempts to dereference a REF whose value is NIL) is enabled in SAFE modules.
- VAR and PROCEDURE imports to SAFE modules must come from SAFE interfaces. [[PMcJ: Except System? To be fixed?]]
- A SAFE module may not dereference a POINTER to an RC structure.
- A SAFE module may not perform any assignment through a POINTER or pass a POINTER dereference as a VAR parameter.

- A SAFE module may not apply LOOPHOLE or a type-transfer function to obtain an RC type.
- The types `System.Address` and `System.Word` are not compatible with ref types. [[Roy says that this contradicts System.def.]]
- A variant record with RC arms has its variant tags set at the time it is allocated (via extra parameters to NEW). A variant with RC arms must have a tag field. The tag field of a variant with RC arms cannot be subsequently assigned to or passed as a VAR parameter.
- Variant tag checking for RC fields is always enabled in safe modules. It is an error to access a field in a variant arm not corresponding to the value of the tag field.
- An assignment statement whose left-hand side contains a record with RC variants is not allowed unless the record, or the structure it is a component of, is a local variable or a value parameter. [[There must be some less tortured way to say this. I have a record with ref-containing variant fields. If I want my module to be a SAFE module, I'm not allowed to assign to that field, that record, or anything containing that record unless the record or the thing containing the record is a local variable or a value parameter. Needs lots more work.]]
- An RC structure may be passed as actual to a VAR formal parameter only if the type of the RC structure and the parameter are *equal*.
- RC local variables and procedure variables are set to NIL upon entry to a procedure. Global RC variables are set to NIL by the linker.

[[Two different sections on safety now jammed together here but not yet combined. This marks the seam.]]

We define a safe program as one that will not violate the invariants of the storage allocator, i.e., it will not cause memory to be smashed. Operationally, this means that a safe program will not alter storage that has not been properly allocated (e.g., by accessing off the end of an array or storing through an invalid pointer). A safe program can still get the wrong answer, but it will not cause an independent program sharing the same address space to do so.

Implementations [[everything, not just implementations, yes?]] should be safe whenever possible. It is best to consider unsafe constructs as akin to type system breaches.

Static checking for safety is enabled in an implementation or program module via use of the keyword SAFE. For example,

```
SAFE IMPLEMENTATION MODULE Thread;
SAFE MODULE ThreadClient;
```

Definition modules can also be marked SAFE, e.g.,

```
SAFE DEFINITION MODULE Thread;
```

This indicates that the corresponding implementation module is safe, hence it is safe to import procedures and variables from the interface. If the corresponding implementation module begins with the keyword SAFE then it is guaranteed safe by the compiler; otherwise, it is safe on the assumption that the implementor knows what she is doing. If a memory-smashing bug occurs, implementation modules not checked by the compiler are the prime candidates for scrutiny.

15. Type-Checking

The idea of this chapter is to put most things about type-checking in one spot so that you have a fighting chance of seeing patterns in the information.

15.1. Same Type

Two expressions have the same type only if they have the same supplied type or if they have a constructed type resulting from the same occurrence of a type constructor. Two different occurrences of the same type constructor produce two different types. Refer to Section 5.2, page 21, if you'd like to worry about this point some more.

Obviously we need to say something about renaming:

```
TYPE
  Dog = ARRAY [0..99] OF INTEGER;
  Canine = Dog;
```

`Canine` is the same type (not just the same basetype) as `Dog`.

15.2. Basetype

Throughout this chapter we're going to use the idea of basetype, the type arrived at by stripping off layers of subrange and BITS FOR. Look at this example:

```
                                A  B  C  M  N  X  Y  Z
TYPE

    A = (Red,Blue,Yellow);      s  s  d  d  d  d  d
    B = [Red .. Blue];             s  d  d  d  d  d
    C = [Blue .. Yellow];             d  d  d  d  d

    M = SET OF A;                        s  d  s  d
    N = BITS 32 FOR M;                      d  s  d

    X = SET OF A;                              d  d
    Y = BITS 32 FOR M;                            d

s = same basetype
d = different basetype
```

A and B have the same basetype because B is merely a subrange of A. B and C have the same basetype because they are both subranges of the same type. A and M do not have the same basetype.

M and N have the same basetype because N is merely a BITS FOR of M. M and X do not have the same basetype. N and Y have the same basetype because both are BITS FORs of the same type.

Since CARDINAL is a subrange of INTEGER, the basetype of a CARDINAL

variable is INTEGER, and CARDINAL itself never appears in the typechecking rules. Non-negative subranges of INTEGER are treated specially in several places whether declared as CARDINALs or not.

15.3. Types for Constants

It makes life simpler to think of constants as having types, just the way variables do.

We're going to consider a constant number without a decimal point in the range

```
FIRST(INTEGER) .. LAST(INTEGER)
```

to be a subrange of INTEGER. For instance, we'll consider the constant 5 to be the subrange of INTEGER

```
[5..5]
```

Similarly, we're going to consider a constant number without a decimal point in the range

```
LAST(INTEGER) + 1 .. LAST(UNSIGNED)
```

to be a subrange of UNSIGNED. For instance, we'll consider the constant 3000000000 to be the subrange of UNSIGNED

```
[3000000000..3000000000]
```

For constant numbers with decimal points we actually need to invent a new "type," which we'll call FLOATING-POINT.

For strings we'll invent the new type STRING, and we'll sometimes need to distinguish between one-character and multi-character strings.
Multi-character strings can have zero, one, or more characters.
One-character strings can have only one character.

And for NIL we need a type that we'll call VOID.

Enumeration constants have the enumeration type. So for instance TRUE and FALSE have type BOOLEAN.

The type of a user-declared constant is the type of the constant expression on the right-hand side of the declaration.

(These constant types are what we need to understand type-checking and have nothing to do with the way the compiler actually implements constants.)

15.4. Kinds of Types

Here are the kinds of all the types, supplied and constructed, plus all the "types" we've supplied for the constants, plus System.Address,

System.Word, and System.Byte:[3]

	numeric	non-numeric
ordinal	INTEGER	BOOLEAN
	CARDINAL	CHAR
	UNSIGNED	enumeration
	numeric subrange	non-numeric subrange
	System.Address	
non-ordinal	REAL	BITSET
	LONGREAL	REFANY
	FLOATING-POINT	fixed-size array
		open array
		record
		set
		ref
		pointer
		procedure
		STRING
		VOID
		System.Word
		System.Byte

15.5. Type-Checking Expressions

At the bottom of an expression there are variables and constants. Expressions have types that depend on the types of their operands. For example `x + y` has a type, which depends on the types of `x` and `y`. Variables are declared to have types. And now you see the good of giving those fake "types" to constants too, so that when an expression mixes constants with variables we can talk about the types of the operands. This section is about the various operators, the rules for what types they accept, and the types of the results they return.

15.5.1. Compatibility

OK, now we've going to gather a whole lot of stuff together so that we can use the notion of compatibility over and over. Compatibility is symmetric -- if `x` is compatible with `y`, then `y` is compatible with `x`.

If two expressions are of the same type, they're compatible.

If two expressions are of the same basetype, they're compatible.

And there are a few other compatible types:

---

[3]The `System` types are covered in the soon forthcoming Public Interfaces Manual. I have been badgered into saying that `System.Address` is signed. Any other information you glean about `System.Address` in this manual is a byproduct of my trying to explain something else, like type-checking.

- Non-negative subranges of INTEGER are compatible with UNSIGNED. (Therefore CARDINAL, which is the entire non-negative subrange of INTEGER, is compatible with UNSIGNED.
- System.Address is compatible with INTEGER, UNSIGNED, and all pointer (but not ref) types.
- The fake FLOATING-POINT constant type is compatible with both REAL and LONGREAL.
- A one-character STRING constant is compatible both with CHAR and fixed-size arrays of CHAR. A multi-character STRING constant is compatible with fixed-size arrays (not open arrays) of CHAR.
- REFANY is compatible with all ref types.
- Procedure types are compatible if they "match": They must have the same number of formal parameters. If there are parameters, corresponding parameters must either be of the same type or be open arrays of the same element type; they need not have the same names. If there are defaults, corresponding parameters must have the same default values. If there are results, the result types must be the same. If there are RAISES clauses, the same exceptions must be named; they need not be named in the same order.
- The VOID type, whose only representative is NIL, is compatible with System.Address, any pointer type, any ref type, REFANY, and any procedure type.

Laid out longways, the things that are compatible even though they don't have the same basetype are:

```
CHAR                                    one-character STRING
fixed-size array of CHAR                STRING
FLOATING-POINT                          REAL, LONGREAL
INTEGER:
  any INTEGER                           System.Address
  non-negative subrange of INTEGER      UNSIGNED
LONGREAL                                FLOATING-POINT
pointer                                 System.Address, VOID
procedure                               matching procedure, VOID
REAL                                    FLOATING-POINT
ref                                     REFANY, VOID
REFANY                                  ref, VOID
STRING:
  one-character STRING                  CHAR, fixed-size array of CHAR
  multi-character STRING                fixed-size array of CHAR
System.Address                          INTEGER, UNSIGNED, pointer, VOID
UNSIGNED                                non-negative subrange of INTEGER,
                                           System.Address
VOID                                    pointer, ref, REFANY, procedure,
                                           System.Address
```

15.5.2. Relations and Set Relations

The operators in the expressions:

```
x = y
x # y
```

both apply unless either x or y is

an array
a record
a System.Word
a System.Byte

and type-check if they are compatible.[4]

The operators in the expressions:

```
x < y
x <= y
x > y
x >= y
```

all apply if x and y are either ordinal or numeric expressions and type-check if they are compatible.

The operators in the expressions:

```
s = t
s # t
s <= t
s >= t
```

all apply if s and t are sets and type-check if they have the same basetype.

```
e IN s
```

applies if s is a set and e is an ordinal expression, and it type-checks if e is compatible with the element type of s.

The operators are overloaded -- they mean different things for different kinds of operands. See Chapter 7, page 33, for semantics. See also Section 16.4, page 93, for the semantics of UNSIGNED.

They all return BOOLEANs.

15.5.3. Arithmetic and Set Operations

[4]At the moment the compiler won't let you use = and # on two strings or on two NILs; this is a bug. And the compiler will let you use = and # on procedure variables of two different (albeit matching) types; that's a design error.

The operators in the expressions:

```
x + y
x - y
x * y
x DIV y
x MOD y
```

all apply if x and y are ordinal numeric types and type-check if they are compatible.[5]

The rules for what they return are as follows:

- If the basetype of either operand is System.Address, the result is a System.Address.
- Else if the basetype of either operand is an UNSIGNED, the result is an UNSIGNED.
- Otherwise the basetype of both operands is an INTEGER, and the result is an INTEGER.

The operators in the expressions:

```
x + y
x - y
x * y
x / y
```

all apply if x and y are non-ordinal numeric types and type-check if they are compatible.

The rules for what they return are as follows:

- If the basetype of either operand is a REAL, the result is a REAL.
- If the basetype of either operand is a LONGREAL, the result is a LONGREAL.
- Otherwise the operands are both FLOATING-POINT, and the result is a FLOATING-POINT.

The operators in the expressions:

```
x + y
x - y
x * y
x / y
```

all apply if x and y are sets and type-check if they have the same basetype.

The rule for what they return is:

- The result is the same basetype as the operands.

Again, the operators are overloaded -- they mean different things for different kinds of operands. See Chapter 7, page 33, for semantics. See also Section 16.4, page 93, for the semantics of UNSIGNED. (Strings are also "compatible" with variables of type Text.T by means of procedures declared

---

[5]At the moment, the compiler won't let you apply * to System.Address, but that's a bug.

in the Text interface.)

15.6. Type-Checking Assignments

Assignable is nearly like compatible, with the following exceptions:

- REFANY and all ref types are compatible, but you can't assign a REFANY to a ref. (You can assign a ref to a REFANY.)
- INTEGER and UNSIGNED are not compatible, but you can assign an INTEGER to an UNSIGNED and vice versa.
- You can assign a string to a `Text.T`.
- The left-hand side can't be a constant.

Laid out longways, the things that typecheck for assignment even though they don't have the same basetype are:

left-hand side	right-hand side
CHAR	one-character STRING
fixed-size array of CHAR	one-character STRING
fixed-size array of CHAR	multi-character STRING
INTEGER	System.Address
INTEGER	UNSIGNED
LONGREAL	FLOATING-POINT
pointer	System.Address
pointer	VOID
procedure	matching procedure
procedure	VOID
REAL	FLOATING-POINT
ref	VOID
REFANY	any ref
REFANY	VOID
System.Address	any pointer
System.Address	INTEGER
System.Address	UNSIGNED
System.Address	VOID
Text.T	STRING
UNSIGNED	INTEGER
UNSIGNED	System.Address

For the semantics of assignment see Chapter 9, page 41. See Section 16.4, page 93, for the semantics of UNSIGNED. See Section 1.4, page 4, for the semantics of strings. See Section 5.11, page 28, for special restrictions on assigning procedure variables.

15.7. Type-Checking Procedure Call

Passable by value is very different from passable by VAR.

Let's first do passable by value. Here are the things you can pass by value even if the actual and the formal have different basetypes. First you get everything from assignable, then a bunch of additional stuff:

```
actual                                  formal
-------------------------------------------------------------
FLOATING-POINT                          LONGREAL
FLOATING-POINT                          REAL
INTEGER                                 System.Address
INTEGER                                 UNSIGNED
pointer                                 System.Address
procedure                               matching procedure
ref                                     REFANY
STRING                                  fixed-size array of CHAR
STRING                                  Text.T
one-character STRING                    CHAR
System.Address                          INTEGER
System.Address                          UNSIGNED
System.Address                          pointer
UNSIGNED                                INTEGER
UNSIGNED                                System.Address
VOID                                    pointer, ref, REFANY, procedure,
                                            System.Address
-------------------------------------------------------------
open array of T                         open array of T
fixed-size array of T                   open array of T
STRING                                  open array of CHAR
anything = 8 bits                       System.Byte
anything = 32 bits                      System.Word
anything divisible by 8                 array of System.Byte
anything divisible by 32                array of System.Word
```

And now let's do passable by VAR. Here are the only things you can pass by VAR if the actual and the formal have different types -- not different basetypes, different types:

```
actual                                  formal
-------------------------------------------------------------
open array of T                         open array of T
fixed-size array of T                   open array of T
System.Address                          pointer
pointer                                 System.Address
anything = 8 bits                       System.Byte
anything = 32 bits                      System.Word
anything divisible by 8                 array of System.Byte
anything divisible by 32                array of System.Word
```

15.7.1. Notes

Supplied Procedures: Be sure everything works out OK with passable.

16. Representation Issues

Reasons to be interested in representation:

- The innocent reason: You'd like to understand roughly how much storage your program is going to require.
- The reason of the Tree of the Knowledge of Good and Evil: You have to deal with some real thing that isn't under your control, like a device register or a packet format defined by some protocol.
- The squeezing-it-out reason: You have something you're going to optimize to the hilt. You therefore design a representation that will allow the most important operations to go as fast as possible. This representation won't always be the one the compiler would choose by default; when it's not, you have to tell the compiler what to do instead.

16.1. Data Representation

The VAX supports a number of data types but it's fundamentally a 32-bit machine.

Addresses are 4 bytes (32 bits) and therefore Modula-2+ pointers, refs, REFANYs, and System.Addresses are all 4 bytes. A procedure is an address; so it's represented as 4 bytes. The discrete-number types are all 4 bytes too: INTEGER, CARDINAL, UNSIGNED, and System.Word. A 4-byte quantity on the VAX is called a Longword; a 2-byte quantity is called a Word. So a Modula-2+ System.Word is (alas) a VAX Longword.

Not everything is 4 bytes. BOOLEANs, CHARs, and System.Bytes are 1 byte. (Only the low-order bit of a BOOLEAN is actually significant.)

Negative discrete numbers are represented using 2's complement.

REAL uses the VAX single-precision floating-point format ("F_floating"), which is 4 bytes. LONGREAL uses one of the two VAX double-precision floating point formats ("D_floating"), which is 8 bytes. The absolute value of a nonzero REAL value R is in the approximate range

.29 * 10**-38 < |R| < 1.7 * 10**38

The precision is approximately 7 decimal digits. The LONGREAL values have approximately the same range as REAL values but have approximately 16 decimal digits of precision.

A VAX address is a 4-byte number that names a byte. If what you want to name is a multi-byte quantity, you give the smallest address. If the quantity

is a discrete number, then the byte with the smallest address is the least significant byte of the number (Little-Endian). [1] The constant NIL is represented as address 0.

```
                    Most Significant ... Least Significant
Longwords             1|                                 0|
Bytes                 4|      3|       2|       1|       0|
Bits                 32|     24|      16|       8|       0|
```

A set is represented as a bit vector indexed by the element type of the set. Bit `i` of the vector is 1 if `i` is a member of the set, 0 if `i` is not. In our compiler, the size of the bit vector is unconstrained, even though the Silver Book would limit them to 32 bits on the VAX.

Each data type has an alignment, and what the alignment means is that any variable of that type is stored at an address that is a multiple of the alignment. So for instance if the alignment of a type is 2, then a variable of that type could be stored starting at address 0 or address 2 or address 4 or address 6 ... The only normal alignments are 1, 2, and 4.

The alignments of the supplied types are all equal to their sizes in bytes (System.TByteSize). So for instance the alignment of an INTEGER is 4, which means that INTEGER variables always appear on VAX Longword boundaries.

The alignment of enumerations and subranges is always 4 too. The alignment of a set is always the smallest legal alignment that is as big as or bigger than the domain of the set -- the maximum number of elements the set could contain.

In laying out records, bytes of padding are used to make the fields fall where their alignments say they have to. The minimal amount of padding is used. The fields are laid out in the order in which they were declared. The alignment of the whole record is equal to the largest of the alignments of its fields.

For instance, I have a record of type `CiRecord`:

```
TYPE
  CiRecord = RECORD
    c: CHAR;
    i: INTEGER;
  END;
```

3 bytes of padding would be used in front of `i`. If the declaration had been:

```
TYPE
  IcRecord = RECORD
    i: INTEGER;
    c: CHAR;
  END;
```

no padding would be needed.

When the system lays out a variant section of a record, the tag field, if any, is laid out like an ordinary field. Each arm of a variant section is laid out like an ordinary record, but all of them start at the same location. The next field

after a variant section starts wherever it would have if the longest arm of the section had been there all by itself.

An array with `n` elements of type `T` is laid out just like a record with `n` fields, all of type `T`, except that if the elements required padding to fall on alignment boundaries then there's padding after the last element as well as all the others.

For instance, suppose we have an array of type `IcArray`:

```
TYPE
    IcArray = ARRAY [0..99] OF IcRecord;
```

(where `IcRecord` is the `IcRecord` from above). The number of bytes in this array is 800, not 797 or 500, since each record including the last is followed by 3 bytes of padding.

So: All variables of these types appear on byte boundaries. Only variables on byte boundaries can be passed by VAR in Modula-2+ -- that's for efficiency; things on byte boundaries can be referred to with VAX addresses.

Better yet, with these alignments most things end up on VAX Longword boundaries, and that permits the VAX to process them in the most efficient way it can.

16.2. BITS FOR

You can get finer-grained control over data representation by constructing types with the BITS FOR constructor. When you have finer control, you must think in terms of a bit-addressable memory even though the VAX is only byte-addressable. In this section we'll talk about alignment in terms of bits, not bytes.

The general form of this type constructor is:

```
BITS n FOR T
```

where `n` is a constant and `T` is a type. It creates a new type whose representation takes `n` bits and whose alignment is either 1 bit or 8 bits. The alignment is 1 bit if `n < 32` and 8 bits if `n >= 32`.

For instance, the declarations:

```
TYPE
  T = BITS 4 FOR [0..15];
  B = BITS 1 FOR BOOLEAN;
```

specify that each variable of type T occupies 4 bits and has an alignment of 1 bit; each variable of type B occupies 1 bit and has an alignment of 1 bit. Operations that apply to the subrange [0..15] also apply to type T; ones that apply to BOOLEAN also apply to type B.

`BITS n FOR T` is illegal for `n <= 0`. It's also illegal if `n` is smaller than the "natural" size of `T`. The natural size of a type in bits is `System.TSize` for most types -- all but subranges and enumerations. For those, the natural size is the minimum number of bits required to represent all the values.

You cannot say `BITS n FOR T` when `T` is:

```
ref
REFANY
open array
System.Word
System.Byte
```

Whenever a type-checking rule depends on the structure of the types, BITS FOR is transparent. See Section 15.2, page 81.

The usual reason for doing BITS FOR is either to pack fields into a record or to make a subrange or an enumeration take up less space (or both). The secondary reason is to pad out a field.

BITS FOR allows you to create variables that do not reside on byte boundaries and therefore cannot be addressed with VAX addresses. The practical consequence is that you cannot apply `System.Adr` to such variables or pass them by VAR.

The code generated for accessing unaligned variables is less efficient than the code for accessing aligned variables.

16.2.1. Data Representation Including BITS FOR

If you say `BITS n FOR T` and `n` is bigger than the natural size of `T`, the extra bits go on the "left," i.e. at higher bit addresses.

Usually the alignment of a record or an array is the largest alignment of its fields or elements; but if 1 bit is the largest alignment and the size of the whole record or array is 32 bits or more, then the alignment of the whole is 8, not 1.

If you're going to make an array of BITS FOR elements, there are some restrictions imposed on you:

1. If the sum of the element and the padding is less than 8 bits, you have to adjust the size of the element to make the sum a power of two (i.e., 1, 2, 4, or 8).
2. If the element (without padding) or any field of the element has a size between 9 and 16 bits, then you have to adjust the size of the element to make the padded-out length be a multiple of 16.
3. If the element (without padding) or any field of the element has a size between 17 and 32 bits, then you have to adjust the size of the element to make the padded-out length be a multiple of 32.

Danger: If you violate one of these, you get bad code, not a message from the compiler.

16.3. Variable Initialization

Variables of any ref type or of type REFANY are always initialized to NIL.

If you create a variable either by saying NEW of a ref (not pointer) type or by declaring a variable that's not contained within a procedure, then you can assume that the initial value of the variable is whatever value is represented by the appropriate number of zeros. Occasionally, when performance is crucial and that value happens to be the value that you want, you can avoid doing an assignment by using it.

You can't make any assumption about the initial value of a non-ref variable you declare within a procedure.

16.4. UNSIGNED

Values of type UNSIGNED are non-negative integers. You use the type UNSIGNED if you need to deal in integers larger than LAST(INTEGER).

```
LAST(INTEGER)   = 2^31-1 = 2,147,483,647
LAST(UNSIGNED)  = 2^32-1 = 4,294,967,295
```

You can declare types that are subranges of UNSIGNED. The FIRST value of such a type must be greater than or equal to 0, and the LAST value must be less than or equal to LAST(UNSIGNED).

The comparison operators:

```
<   <=   >   >=
```

have different implementations for INTEGER and UNSIGNED operands. The implementations have to be different since, for instance, `0FFFFFFFFH` is the representation of -1 in the type INTEGER and of LAST(UNSIGNED) in the type UNSIGNED, while `0H` is the representation of 0 in both types. The DIV and MOD operators are also different for the two types.

As described in the type-checking section, you can assign an UNSIGNED value to an INTEGER variable and an INTEGER value to an UNSIGNED variable; the assignment just copies the bits across with no checking. [[Surely we should either fix this or be prepared to hand back our Strongly Typed Language Merit Badge?]]

What is the result of -n when n has type UNSIGNED? Paul says:

> Compile-time error. Or 2's complement, maybe. We'll decide.

16.5. Tagless Variant Records

A tagless variant record allows a client program to view the same piece of storage in two or more different ways. The client has complete control over the choice of the view in use at any instant; when the client uses one of the field names to access part of a record, it gets the effect of loopholing that record into the particular variant containing the named field before

performing the access. Since the loophole is implicit, this language feature should be used with great care.

Here is an example from Hardware.def. A virtual address is always a virtual address, but sometimes we want to deal with it as a unit and sometimes we'd rather view it as a byte-page-region triple. We say:

```
TYPE
  VirtualAddress = RECORD
    CASE VARep OF
    | FALSE:
      addr: Address;
    | TRUE:
      byte: BITS LogBytesPerPage FOR PageBytes;
      page: BITS LogPagesPerRegion FOR VirtualPage;
      region: BITS 2 FOR Region;
    END;
  END;
```

If `va` is a variable of type `VirtualAddress`, then `va.addr` is the whole address, and `va.region` is the most significant two bits of the address.

Sometimes we use a tagless variant record to describe the fact that a particular piece of data has two (or more) completely different interpretations. We choose an interpretation based on some externally supplied information. For instance, the interpretation of part of one of the VAX internal processor registers depends upon whether or not the VAX is a MicroVAX-II:

```
CASE UVAXII OF
| TRUE:                              (* KA630 13.2.2 *)
  fill33A: Bits3;
  breakReceived: PackedBoolean;
  fill33B: Bits1;
  framingError: PackedBoolean;
  overrunError: PackedBoolean;
| FALSE:
  inputSource: Bits4;
  fill33C: Bits3;
END;
```

Once a program had determined that it was running on a MicroVAX-II, it might go on to access the overrunError field.

In the first example there was no way for the programmer to do the wrong thing, since both views of a virtual address are valid. In the second example, the programmer definitely can do the wrong thing by accessing the overrunError field (say) while running on a VAX that is not a MicroVAX-II. There is nothing that Modula-2+ can do to prevent this sort of programming error.

[[mrb: I'm still not sure I understand how to explain the general significance of this example. I did cut it down but I'm quite sure nothing essential was lost.]]

A final example: We have a server and a client who pass messages back and forth. For simplicity, there is a single type `T`, which can hold either a request or a reply; these share some storage but mostly have different structure. We want a request and a reply to be the same type so that we can share

operations on them (e.g., allocate one from a pool, or transmit one over a network, or whatever). It is unnecessary to say whether a given instance is a request or a reply since it is obvious from context.

The declaration is:

```
TYPE
  T = RECORD
    server: ServerType;
    seq: Seq;
    target: Target;
    CASE RequestOrReply OF
    | Request:
        request: RECORD
          CASE operation: Operation OF
          | ProbeServer:
          | ReadFromAddressSpace:
              readFromAddressSpace: RECORD
                space: AddressSpace;
                start: Address;
                count: CARDINAL;
              END;
          | DisconnectFromTarget:
          | ReadFromTarget:
              readFromTarget: RECORD start: Address; count: CARDINAL; END;
          END;
        END;
    | Reply:
        reply: RECORD
          result: Result;
          CASE operation: Operation OF
          | ProbeServer:
              probeServer: RECORD
                order: ByteOrder;
              END;
          | ReadFromAddressSpace:
              readFromAddressSpace: RECORD
                count: CARDINAL;
                data: ARRAY [0 .. MaxBlockSize - 1] OF System.Byte;
              END;
          | DisconnectFromTarget:
          | ReadFromTarget:
              readFromTarget: RECORD
                count: CARDINAL;
                data: ARRAY [0 .. MaxBlockSize - 1] OF System.Byte;
              END;
          END;
        END;
  END;
END;
```

16.6. LOOPHOLE

The supplied procedure LOOPHOLE provides a way to circumvent the Modula-2+ type system, while making the breach easy to notice. You should use it with care after you are familiar with the representation of the values of both types (T and x) on the VAX.

You can only loophole a value of one type into another type if the two types have the same size. LOOPHOLE never changes the representation of anything.

Typical uses of LOOPHOLE are conversion from INTEGER to UNSIGNED and

back and arithmetic on `System.Address`. [[And inside things like RPC runtime, pickles ... expand ...]]

It is illegal to specify an open array type as the target type of a loophole. The magic types `ARRAY OF System.Byte` and `ARRAY OF System.Word` in formal parameter lists are implicit loopholes that, unfortunately, don't show up at the point of call.

16.7. Implementation Restrictions

```
identifiers:
  Front end imposes length limit on a line of 5800 characters.
  Since an identifier cannot cross line boundaries, that's the
  limit.

string constants:
  no limit imposed by front end or new code generator, but see
  below.

set constants:
  set constants of user-defined set types, with a maximum of
  1025 elements, i.e. ranging from low-bound of subrange type on
  which set is based to INC(low-bound, 1024).

     e.g. TYPE SetType = SET OF [100..2000];
          CONST SetConst1 =  SetType{100,1124};   (* okay *)
          CONST SetConst2 =  SetType{100,1125};   (* not okay *)

  set constants of type BITSET with a maximum of 32 elements
  (limit is defined by the language: a BITSET type fits in the
  same space occupied by an integer)

     e.g.  CONST BitSetConst = {0..31};
```

Expect bad code rather than compiler complaints if you exceed these limits.

17. Mixing Modula-2+ with Other Languages

17.1. C, Pascal, and Unix

[[Need to winkle out references to Unix42 and replace with references to ... OS.def, right?]]

17.1.1. Modula-2+ Calling Sequence

[[These words (slated for rewriting anyway) are Mike Powell's.]]

The Modula-2+ calling sequence is compatible with the calling sequences of Powell's Modula-2, Unix C, and Berkeley Pascal. You can call procedures, pass parameters, and share variables with programs in those languages.

The major difference between the calling sequences relates to the passing of multiword value parameters. Both Modula-2+ and Pascal VAR parameters are passed by reference; since C doesn't have VAR parameters, the program must specify the address of the variable, which works out to be the same thing. All three languages pass single-word value parameters by pushing the values onto the stack. Double-precision floating point value parameters are passed by pushing two words onto the stack. (In Pascal, reals are double-precision; in C, both floats and doubles are passed as doubles; Modula-2+ reals are not passed as double-precision automatically; therefore, for communication between Modula-2+ and Pascal or C, use longreals.)

In Pascal, multiword value parameters (arrays, records, sets, etc.) are all pushed onto the stack. In C, although struct parameters are passed by pushing the value onto the stack, array parameters are passed by pushing the address onto the stack. In Modula-2+, multiword parameters (except longreals) are passed by reference (the called procedure makes a copy of the parameter). Pascal procedures called by Modula-2+ should thus declare multiword parameters to be VAR parameters. In the (unlikely) case that the Pascal procedure wishes to change a multiword parameter that Modula-2+ thinks is a value parameter, the Pascal procedure must make a copy of the parameter in a local variable. C procedures called by Modula-2+ should always expect pointers, as they usually do anyhow. A Modula-2+ procedure called by Pascal should be declared in Pascal to accept VAR parameters for multiword values.

A Modula-2+ open array parameter is passed as two parameters: a pointer to the start of the array followed by the number of elements in the array. Two special cases of interest are open arrays of Byte and Word. In these cases, the number of elements is the size of the variable, which may be of any type, in bytes and words (rounded up), respectively. For example, it would be possible to define the Unix read system call to have two parameters: the first,

a file number, and the second, an open array of bytes for the buffer.

For low-level access to C procedures, there is an "uncounted" open array type; a parameter of this type will be passed only the address of the array:

```
TYPE NoCountArray = ARRAY @NOCOUNT OF ElementType.
```

See Unix42.def and Unix42lib.def for examples of the use of this feature.

17.1.2. Pointers

The Modula-2+ runtime allocates storage from a heap using its own allocation and deallocation procedures. Generally, these are different from (and much better than) the Pascal NEW and DISPOSE procedures or the C malloc and free procedures. However, it is possible to generate and use pointers that are compatible with other languages.

Pointers may be allocated and deallocated using the Pascal (NEW/DISPOSE) or C (malloc/mfree) storage allocation procedures by declaring them as follows:

```
PascalPtr = POINTER @PASCAL TO Rec;
CPtr = POINTER @C TO Rec;
```

@PASCAL pointers have the Berkeley Pascal validity check applied to them; @C pointers are not checked.

17.1.3. Names

The external names of Modula-2+ procedures and statically allocated variables are generated by qualifying the local name with the names of the enclosing modules and procedures. A procedure called NextToken in a module Parse will be called _Parse_NextToken. The leading _ is appended to names by both the C and Pascal compilers, so C or Pascal would call this procedure Parse_NextToken. (The Berkeley Pascal compiler can easily be modified to accept _ in the middle of variable names by adding

```
|| c == '_'
```

to the identifier while loop in yylex.c.)

Each level of nesting inside a procedure or module adds qualifiers. E.g., the procedure SkipBlanks nested inside NextToken is called _Parse_NextToken_SkipBlanks. However, because of differences in the displays, it is unlikely that you can successfully call nested procedures between languages.

Although this scheme works well for programs that use modules, current Pascal and C programs expect to have global, unqualified names. To make a procedure or variable have a global name (from the point of view of C or Pascal programs), add the @EXTERNAL attribute before the name in the procedure or variable declaration. E.g.,

```
VAR @EXTERNAL errno : INTEGER;
PROCEDURE @EXTERNAL perror(msg : CString);
```

It is possible to create modules with Modula-2+ definitions that are

implemented in some other language. The Unix42 library module is an example. Notice that errno is specified @EXTERNAL above, because its name is unqualified.

Each module that exports VARs, or PROCEDUREs, or opaque ref types must have an initialization procedure. The initialization procedure for module X is called X__init (X, two underscores, init) and has no parameters. These procedures are called when the program starts up, and may be called more than once (e.g., if several modules import X). The Modula-2+ compiler generates code for its initialization procedures to execute the body only once, even though it is called more than once. If you implement an initialization procedure in C or Pascal, you will need to do the same thing.

The initialization procedure for a program module (one that is neither a definition module nor an implementation module) is the main program. It is always called main, no matter what the module is called, so Unix can execute it.

If you import a Modula-2+ procedure or variable to a C or Pascal or assembly language program (for example, if your main program is not a Modula-2+ program), you must ensure that the Modula-2+ runtime system gets started and then that the initialization procedures of the imported modules get invoked before procedures within are called. To initialize the Modula-2+ runtime system, invoke

```
runtime__init(argc,argv,envp)
```

where

```
int argc; char *argv[], *envp[];
```

are the arguments passed to the main program by Unix.

17.1.4. Interfacing with Unix

[[OUT OF DATE]]

The recommended interface to Unix is through the srclib module called Unix42. For the location of Unix42.def, do

```
man mp
```

17.2. How Procedure Call Is Implemented

This section describes how M2+ procedure call is implemented using VAX instructions. It is necessary to know this if you need to write assembly language procedures that either call or are called by M2+ procedures. This can be necessary in order to get the best possible performance for a frequently-executed procedure. This section assumes that you are familiar with the VAX, as you would need to be in order to write an assembly language program.

Someday it may become desirable to change the implementation of M2+ procedure call, based on our experience. The less code you have written that

takes advantage of the information in this section, the less work you'll be faced with when that happens. One prudent technique is to maintain an equivalent (but slower) M2+ version of each assembly language module that you write, both as documentation and as a fallback implementation that can be used in a pinch.

17.2.1. Calling M2+ from Assembly Language

The general scheme for calling an M2+ procedure Int.Proc with N parameters is

1. <code to save values of r0 - r5 as required>
2. <code to push the Nth actual parameter>
   ...
   <code to push the 1st actual parameter>
3. calls $ParmLongwords, _Int_Proc
4. <code to take the result, if any, from r0>

Explanations:

1. The called procedure is allowed to modify r0 - r5. Therefore the calling procedure must be prepared for the values in these registers to be different after the call.

2. The parameters are pushed in reverse order. The motivation for this convention in VAX Ultrix languages such as C is to support procedures that take a variable number of parameters. By pushing the required parameters last, they can be addressed with constant offsets from ap. You cannot write an M2+ procedure taking a variable number of parameters, but M2+ pushes the parameters in reverse order to make it easier to call C procedures from M2+.

   The meaning of "pushing a parameter" depends upon the type of the parameter. Three examples: If the formal parameter is an INTEGER passed by value, and the actual parameter is the constant 1, then

   ```
   pushl $1
   ```

   does it. If the formal parameter is an INTEGER passed by VAR, and the variable is the only local variable of the calling procedure, then

   ```
   pushal -4(fp)
   ```

   does it. If the formal parameter is a `VAR ARRAY OF CHAR` and the actual parameter is a 10-character array that is the only local variable of the calling procedure, then

   ```
   pushl $10
   pushab -4(fp)
   ```

   does it (first push the number of array elements, then push the address of the first element). You should check the code generated by the compiler (using the -k switch) for other parameter types.

3. The calls instruction transfers control to the procedure being called. ParmLongwords is at least N, but may be greater because some parameters require more than one longword of stack. For instance, open array parameters require two longwords.

   The identifier "_Int_Proc" should be "imported" at the beginning of the assembly code module using the directive

   ```
   .globl _Int_Proc
   ```

4. The called procedure pops all parameters off the stack, so the calling procedure should assume that sp has the value it had before it started pushing parameters.

   The result is either a one-longword value or a pointer to a larger value allocated on the stack. In the latter case, the calling program must use or copy the value before calling any more procedures.

17.2.2. Calling Assembly Code from M2+

To call assembly code from M2+, you declare each assembly code procedure in an interface, and make sure that each assembly-coded procedure matches its interface declaration. The following is the general scheme for implementing a procedure Int.Proc:

```
1.      .align 2
    _Int_Proc:
        .word <entry mask>
2.      moval -LocalBytes(sp),sp
3.      <code for body of procedure>
4.      ret
```

1. The ".align 2" directive word-aligns the entry mask; this is not required, but improves performance.

   The identifier "_Int_Proc" should be "exported" at the beginning of the assembly code module using the directive

   .globl _Int_Proc

   The ".word" directive creates the VAX entry mask. The entry mask specifies what registers are to be saved and what traps are to be enabled when the procedure is called; it is fully described in the section "procedure call instructions" of the VAX architecture handbook. For a procedure to be callable from M2+, it must save every register in r6 to r11 that it will modify. The entry mask must not enable integer or numeric overflow traps. Hence the entry mask may be

   .word 0x0fc0

   or any value obtainable by clearing bits in this word.
2. The moval instruction allocates space for LocalVarBytes bytes of local variables. LocalVarBytes should be a multiple of 4 (i.e. the stack should stay longword aligned). If the procedure uses no local variables then this instruction is clearly a no-op and should be omitted.

   At present, there is no way to extend a thread's stack if it runs out of room.

3. Actual parameters are addressed using positive offsets from ap. If each parameter takes a single longword then parameter i is located at sp+(4*i).

   It is generally most convenient to address local variables using negative negative offsets from fp rather than positive offsets from sp.
4. The ret instruction returns to the caller.

   If an interface is implemented entirely by assembly code (rather than by a mixture of assembly code and M2+ code), then the assembly code program must include an initialization procedure:

```
    .align 2
_Int__init:
    .word 0
    <code to initialize the module>
    ret
```

   The program must include this procedure even if there is no initialization work to do.

18. Performance and Other Pragmatic Issues

18.1. Notes

CASE statements are compiled in the straightforward way, since Wirth gives the compiler writer a license to do so. A jump table is used to dispatch the case index, hence the set of case selectors should be relatively dense. If they aren't, an IF-THEN-ELSIF structure is generally preferable. Obviously, this is a space-time tradeoff and thus depends on the individual situation. [[Note from Andy: " The new code generator is much smarter about CASE; it attempts to make this tradeoff itself.]]

Storage allocation through NEW(x), where x is a ref type, is intended to be inexpensive. For the implementation that we envision, NEW will require about 15-20 usec on a M68010. [[and on a VAX Firefly?]]

Some other timings for comparison purposes:

```
MODULE Timings;

PROCEDURE P;
    BEGIN
    END P;

PROCEDURE Q(i, j: INTEGER);
    BEGIN
    END Q;

PROCEDURE Test;
    VAR x, y: INTEGER;
    BEGIN
        P;              (* about 5 usec *)
        Q(x, y);        (* about 12 usec *)
    END Test;

END Timings.
```

From Violetta:

> More to come eventually, including: constructs that implicitly invoke out-of-line code, supplied procedures that expand in-line, cost of thread and synchronization operations.

18.2. The Optimizer

The Modula-2+ optimizer, invoked through the -O compiler switch, performs optimization on the tree structure used by the front end of the compiler. It attempts to perform the following program transformations to produce more efficient code:

- Common subexpressions (CSEs) that exceed a built-in, heuristic measure of complexity are evaluated only once. Lifetime information for such CSEs is computed and used to control

storage allocation, including assignment of registers.

- The assignment of local variables and CSEs to registers is done using a prioritized scheme, with priority based on a built-in, heuristic measure of access frequency.
- Evaluation of sufficiently expensive loop-invariant expressions is performed before entry to the loop.
- Assignment statements of the form

  <designator> := <designator> (+ | * | -) <expression>

  are transformed to a more efficient code sequence.
- Some tail-recursive procedure calls are detected and transformed to jumps. In more detail, the optimizer looks at the last statement in the procedure body. If it is a conditional (IF, CASE), it looks at the last statement in each arm, and so on recursively. Each such last statement is treated as follows:
  - If the procedure returns a value and the last statement is a RETURN statement and the RETURN expression is an acceptable recursive call, that call qualifies.
  - If the procedure does not return a value and the last statement is an acceptable recursive call, that call qualifies.

A compiler wizard should be consulted before using the optimizer in the following situations:

1. The program LOOPHOLEs a pointer or a ref of one type into a pointer or ref of a different type, or uses System.Address for similar purposes, and then dereferences the result.
2. The program accesses shared data in a concurrent program outside a LOCK statement.

18.3. Choosing Identifiers

The supplied identifiers and keywords of the language use all upper-case letters, so you can avoid name conflicts if you never declare a new identifier that uses all upper-case.

The keywords that you're most likely to stumble over are the short ones, since the moment when you're looking for an abbreviation for Northward Oriented Termination is precisely the moment when you forget that NOT is already reserved. Here, then, is a list of the short keywords of the language:

ABS	ELSE	LOCK	REAL
AND	END	LOOP	REF
BITS	EXCL	MAX	SAFE
BY	EXIT	MIN	SET
@C	FOR	MOD	THEN
CAP	FROM	NEW	TO
CASE	HALT	NIL	TRUE
CHAR	HIGH	NOT	TRY
CHR	IF	ODD	TYPE
CONST	IN	OF	VAL
DEC	INC	OR	VAR
DIV	INCL	ORD	WITH
DO	LAST	PROC[6]	

18.4. Very Long Literal Strings

Weird, but every once in a while you need an enormous literal string. The longest that one can be is `[[` find out; might not be any problem any more `]]`; if you want one longer than that, use `Text.Cat` to build it up.

Of course, if it happens to be a `Wr.PrintF` format string, you can't build it up with `Text.Cat` ...

18.5. Dispatch Tables

CASE statements get you the most simple-minded kind of dispatch table you can imagine. This:

```
CASE e OF
| 1:
  Wr.PrintF(Stdio.stdout, "One\n");
| 1000:
  Wr.PrintF(Stdio.stdout, "Many\n");
END;
```

gives you a dispatch table with a thousand cells. Think hard about using an IF statement instead.

18.6. Private Allocation for Referents of Pointers

Normally, if the first parameter to form (1) of NEW is a pointer, `Storage.ALLOCATE` is called; similarly, if DISPOSE is given a pointer, it calls `Storage.DEALLOCATE`. But the user can specify other routines for non-collectible storage allocation and deallocation by placing them in a scope that is visible from the point where this form of NEW is called and naming them `ALLOCATE` and `DEALLOCATE`.

---

[6]Deprecated.

For example:

```
SAFE IMPLEMENTATION MODULE Mine;
FROM MyStorage IMPORT ALLOCATE, DEALLOCATE;
TYPE T = Type;
VAR v: POINTER TO T;
BEGIN
  NEW(v);       (* calls MyStorage.ALLOCATE(v,System.TSize(T));      *)
  DISPOSE(v);   (* calls MyStorage.DEALLOCATE(v,System.TSize(T)):   *)
END Mine.
```

18.7. RAISE for Flow Control

Under certain circumstances a RAISE statement is compiled to a go-to and consequently executes much faster. This optimization is performed only when all of the following conditions are true:

1. the exception raised doesn't have an argument
2. a non-pass-through handler for the exception, that is, a handler that doesn't simply raise the same exception, appears in the scope statically enclosing the RAISE statement
3. any TRY statements defining intervening scopes between the RAISE and the non-passthrough handler are of the TRY ... EXCEPT kind with no ELSE clauses.

Under these conditions, any pass-through handlers that appear in the intervening TRY ... EXCEPT statements are ignored, and the RAISE statement compiles to a go-to to the innermost statically enclosing real handler for the exception.

Notes:

A RAISE for an exception that has a handler in a statically enclosing scope executes considerably faster than the general RAISE because it compiles to a go-to. But if finalization is required because there is a TRY FINALLY statement enclosing the RAISE then this optimization is not used.

19. Programming Style

19.1. The .T Convention

In Modula-2+, it is common for an interface to define either one type or a small number of related types, plus the public procedures operating on this type or types. The name of the interface should be chosen to describe the type, or the "primary type" if the interface defines more than one. Specifically, interfaces should be named, wherever it makes sense to do so, for a single instance of the objects they manage. For example, we have the `Thread` and the `Text` interfaces. The primary type defined by an interface will simply be called `T` and should be referred to by its qualified name in importing programs, e.g. `Thread.T` and `Text.T`.

This naming convention is designed to lead to acceptably understandable and short names. It does not apply to interfaces that do not manage particular objects, e.g. the definition module `Pipeline`, which exports only procedures, or to interfaces where the name of the objects doesn't describe particularly well the function of the interface. For example, in a virtual memory interface, the primary type seems to be `PageRun`, but `PageRun` is not a good name for the interface, nor is `VM.T` a very descriptive alternative to `VM.PageRun`.

19.2. Don't Export Variables

There is no way to specify that a variable in a definition module is meant to be read but not written by clients. Furthermore, client access to such a variable is unsynchronized with actions of the implementation. For these reasons, definition modules should contain variables only when synchronization is not an issue; otherwise the variables should be declared in the corresponding implementation, and accessed by clients through procedures that read, change, or return their values.

19.3. Returning Multiple Values

VAR parameters are intended to enable procedures to return multiple values. They also can be used to reduce the overhead of passing large value parameters. However, since the language provides no syntax to distinguish these conceptually different applications of VAR parameters, a clarifying comment will help the reader to understand the procedure's semantics.

Examples:

```
PROCEDURE Sum(
    VAR a (*in*), b (*in*): Matrix;
    VAR result (*out*): Matrix);

PROCEDURE Transpose(VAR a (*inout*): Matrix);
```

Message from William Stoye:

> I've been reading your M2+ stuff, and there's one thing I thought I might mention, even though it's not very important, because other people might not bother... There appears to be a rival standard emerging for this, mainly from Andrew (whose RPC generator would like to know about purely `IN` or `OUT VAR` parameters, without having to look inside comments). Andrew uses `VAR` parameters with names like `INx`, `OUTy` instead, e.g.
>
> ```
> PROCEDURE Snapshot(
>     valueDesc: ValueDesc;
>     VAR OUTlimitSkulk: LimitSkulk;
>     VAR OUTmark: Mark;
>     VAR OUTlabelTSeq: LabelTSeq);
> ```
>
> It seems likely that this form will be in common use in the future and so it might be worth mentioning alongside the (*IN*) form.

19.4. Qualified Names

In an implementation module, imported names should generally be qualified. Qualification makes code explicit about its non-local dependencies, hence easier to read. For example, you would use `RealFns.Sqrt` and `RealFns.ArcTan` like this:

```
IMPORT RealFns;
x := RealFns.ArcTan(RealFns.Sqrt(3.0));
```

rather than

```
FROM RealFns IMPORT Sqrt, ArcTan;
x := ArcTan(Sqrt(3.0));
```

In a program dominated by `REAL`s, the unqualified form might be easier on the reader, since the function names are unlikely to be confused with functions from other interfaces. Similarly, heavily used procedures from the `Rd`, `Wr`, and `Stdio` interfaces might be used unqualified. If the program text would become greatly cluttered with a repeated interface name, use the unqualified form to eliminate the repeated name.

If you have any doubt, however, use the qualified name.

19.5. WITH Statements

The `WITH` statement makes record field names accessible without qualification and can therefore mask variables in enclosing scopes. Since the masking is implicit (the field names do not appear locally to remind the reader what is happening), subtle errors can occur if a significant amount of code appears inside a `WITH` statement.

We find it best to use the `WITH` statement only for replacing all or most of the

fields of a record. In essence, this facility should be treated as a Mesa-style record constructor [2] combined with an assignment statement.

19.6. Redeclaring

The implementation module M automatically gets all the declared and imported names from its definition module M. Some people redeclare and reimport all the names anyway to make implementation modules easier to read.

19.7. Declaring Constants

A constant declaration lets you define in a single place a constant value to be used several places in a program. Even if you're going to use the value only once, declaring it as a constant provides you with a way of highlighting the values that you've compiled into your program. If the number is part of the logic of the program, it's OK to use it as a number, but if it is in some sense a parameter compiled in as a constant rather than a variable for efficiency's sake, declare it as a constant and give it a good name.

For instance:

```
CONST
  HashTableEntries = 200;
TYPE
  HashTableIndex = [0 .. 2 * HashTableEntries - 1];
  HashTable = ARRAY HashTableIndex OF HashRecord;
```

19.8. ORD and VAL and LOOPHOLE

The supplied procedures ORD and VAL provide implementation-independent mappings between ordered types and non-negative integers. They should always be used in preference to LOOPHOLE. E.g.,

```
LOOPHOLE(x, INTEGER)     (* bad *)
INTEGER(x)               (* worse *)
ORD(x)                   (* good *)
```

ORD(x) and VAL(T,x) return x if x is of type UNSIGNED, INTEGER, or CARDINAL and T is the type of x. This fact has some rather surprising effects on the value of VAL(INTEGER, x) for values of x larger than LAST(INTEGER) but seems to be needed to avoid other problems.

19.9. Subranges

Whenever you can, you should use explicit subranges of INTEGER (rather than plain INTEGER) to make it clear what semantics you intend.

20. Formatting Conventions

20.1. Pretty-Printer

Given the existence of ppmp, much of this chapter is merely historical, for keeping the Greg and Mark honest. Use ppmp. It's not perfect yet, but it will improve if we all use it.

20.2. Spelling and Capitalization

Identifiers are written entirely in lower case except as follows:

The initial letter of each embedded word except the first is capitalized. For the purposes of this rule, an embedded acronym is a sequence of one-letter words. To avoid possible conflicts with keywords (see below), do not use identifiers of length greater than one that contain only capital letters.

Identifiers that name modules, procedures, exceptions, types, and constants start with an upper-case letter. All of these are compile-time constants. All variables, including procedure variables, start with a lower-case letter.

Thus, for example,

```
variableName            thisCPU
fieldName               status
ModuleName              Parser
ProcedureName           Insert
TypeName                VMCache
AnException             Overflow
ConstantName            WordSize
EnumerationElement      Offline
```

Note that the elements of an enumeration type are semantically similar to constants and are therefore capitalized.

Reserved words are written entirely in upper case. Other capitalizations of the same word are available to the programmer for other purposes (for example, `set` and `lock` are perfectly good variable names).

20.3. Punctuation

A space appears before and after a vertical bar and before and after equal signs in definitions and declarations. Spaces usually appear before and after binary operators (including assignment) when the statement containing them is long, but they may be omitted when the statement is short (e.g., `i:=i+1`). A newline sometimes appears in place of the space after these characters.

A space appears after colon, comma, and semicolon, but none before. A

newline often follows semicolon (and sometimes comma) instead.

Except as required by adjacent tokens, no spaces appear before or after left and right parentheses, left and right square brackets, left and right curly brackets, caret (up-arrow), dot-dot, and period. (A newline follows the period at the end of a module.) If the token on the left (or right) of these characters requires a space after (or before) it, one should appear. For example, consider the left parentheses in

```
PROCEDURE Positive(x: INTEGER): BOOLEAN;
```

and

```
TYPE Color = (Red, Green, Blue);
```

A space appears after left-comment and before right-comment.

A semicolon follows the last statement in a statement sequence and the last field in a field list; this makes insertions and deletions somewhat easier.

20.4. Indentation

Indenting is used to emphasize program structure. Each nesting level is the same width, either two spaces or four spaces wide. The following illustrates the recommended form of each Modula-2+ construct ('ss' represents a statement sequence):

```
CASE expr OF
| c1:
    ss
| c2:
    ss
| ELSE
    ss
END
```

```
CONST
    A = 613;
    B = {1, 3, 7};
```

```
IF bool THEN
    ss
ELSIF bool THEN
    ss
ELSE
    ss
END
```

```
FOR i := 1 TO 10 DO
    ss
END
```

```
LOCK d DO
    ss
END
```

```
LOOP
    ss
END
```

```
MODULE impl;
IMPORT I1, I2;
CONST C = 47;
VAR v: INTEGER;
PROCEDURE P;
    BEGIN
    END P;
BEGIN
    ss
END impl.
```

```
PROCEDURE P;
    VAR
        b: BOOLEAN;
        i: INTEGER;
    BEGIN
        ss
    END P
```

```
REPEAT
    ss
UNTIL bool
```

```
TRY
    ss
EXCEPT
| e1(v1):
    ss1
| e2(v2):
    ss2
| ELSE
    ssn
END
```

```
TRY
    ss
FINALLY
    ss
END
```

```
TRY
    ss
PASSING {e1,e2}
END
```

```
TYPE
    Index = [0..15];
    Object = RECORD
        f1: Type1;
        f2, f3: Type2;
    END;
    Handle = REF Object;
```

```
TYPECASE ra OF
| t1(v1):
    ss1
| t2(v2):
    ss2
| ELSE
    ssn
END
```

```
VAR
    var: Type1;
    x, y, z: Type2;
```

```
WHILE bool DO
    ss
END
```

```
WITH d DO
    ss
END
```

A statement sequence is indented under the construct that introduces it, which lines up vertically with its corresponding END. Note also the similarity of the case discriminations in CASE, TYPECASE, and TRY ... EXCEPT. Declarations are indented one level, including declarations of nested procedures. (Exception: the declarations in the outermost module are not indented; in a nested module, they would be.) If the declaration requires more than one line, its components are indented another level, with at most one type per line.

The forms above only apply to constructs that do not fit on a single line. For example, if the statement sequence following a THEN, ELSE, or case label is short (e.g., a single statement or a few short statements), it can appear on the same line with the tokens that introduce and terminate it. Similarly, if the statement sequence of a loop body is short (even non-existent), it can be

moved up to the line that introduces the loop, along with the trailing END. Thus,

```
IF bool THEN x := y; y := z END;

FOR i := 1 TO 10 DO a[i] := 0 END;

CASE x OF
| 1..3: y := 0;
| 4: y := -1;
| ELSE y := z;
END
```

are all acceptable forms. Put the whole construct on one line if it fits.

Long statements that require more than one line are broken where white space would normally appear, with the continuation lines indented two levels.

20.5. Comments

The text of a multiline comment begins on the same line as the opening left-comment. Subsequent lines are indented the same as the opening left-comment. The terminating right-comment appears on the last line of the comment.

```
(* A comment that fits entirely on one line by itself. *)

(* A long comment that does not fit on one line, and
the filler necessary to make it do so, and the filler
necessary to make it do so. *)
```

By convention, comments that refer to a group of items appear before the group. Comment boxes are often used to set off what is logically a section heading; they are constructed as follows:

```
(******************)
(*  Section Name  *)
(******************)
```

Comments associated with a single definition or declaration appear immediately after the definition or declaration (not before) and at the same level of indentation. In modules and in procedure implementations, they immediately follow the module (procedure) heading, which includes the import and export clauses. Comments in running code follow the normal flow of control.

When comment brackets are used to comment out a section of code, the terminating right-comment appears on a line by itself, lined up vertically with the opening left-comment.

20.6. Interfaces

A few additional guidelines make interfaces, which represent the most heavily read code, a bit more uniform from one to the next.

Procedures with more than two parameters of different types should have their parameters listed on separate lines; items should be grouped logically on each line. Each procedure should be immediately followed by a comment describing its operation (if it is not evident from the procedure's name). A blank line separates procedure declarations.

```
PROCEDURE ProcName(
  parm1: Type1;
  parm2, parm3: Type3;
  VAR (*in*) parm4: Type4)
  : ReturnType
  RAISES {E1, E2};
  (* Short, one-line description of the procedure.
  More information, if necessary, appears here, in
  multiline comment format. *)

PROCEDURE NextProc ...
```

A general comment describing the interface as a whole should appear immediately following the module header, before any definitions.

20.7. Don't Forget

() for procedure call with no arguments.

Need to add format for EXCEPTION, ARRAY.

Statements in a statement sequence are separated by semicolons. Since the empty statement is allowed, end the last statement in a statement sequence with a semicolon.

20.8. Formatting Question

I tend to write RETURN statements as if RETURN were a procedure call:

```
RETURN(3);
```

Of course, the other way is OK too, and perhaps less confusing to readers:

```
RETURN 3;
```

I feel uncomfortable that RETURN didn't get mentioned in this chapter before.

21. Compatibility with Ordinary Modula-2

21.1. Notes

Jorge says of the next paragraph "Needs motivation":

In Modula-2+ CARDINAL has been redefined to be the non-negative subrange of INTEGER. Modula-2's original CARDINAL type, which takes on the values [0..2^32-1], has been renamed UNSIGNED in Modula-2+. Use ppmp(1) to convert ordinary Modula-2 programs that make use of the ordinary meaning of CARDINAL.

Paul McJones says also we need to talk about:

- MIN, MAX applied to type not allowed.
- old-style opaques don't allow = or := (compare third edition page 169)
- export rules?

Somebody (not me) needs to sort out the differences between m2+ and third-edition Modula-2.

21.2. Syntax Extensions

Below are the categories in which we made extensions to Modula-2 as described in Appendix 1 of "Programming in Modula-2, Third Edition" by Niklaus Wirth [3]. The numbers at the left margin refer to the numbers in that Appendix (not to the numbers in our own rendering of Wirth's EBNF, Appendix VI, page 151).

21.2.1. Exceptions and Finalization

```
32+ ProcedureType = PROCEDURE [FormalTypeList]
    [RAISES raisees].

49  statement = [ ... |
52+   ... |
    TryStatement | ... ].

  TryStatement = TRY StatementSequence TryTail END.
  TryTail = FINALLY StatementSequence | PASSING raisees |
    PASSING raisees ";" |
    EXCEPT [HandlerArm {"|" HandlerArm}]
    [ELSE ["(" ident ")"] StatementSequence].
  HandlerArm = [QualidList ["(" ident ")"] ":"
    StatementSequence].
  raisees = "{" [QualidList] "}".
  QualidList = qualident {"," qualident}.

69+ ProcedureHeading = PROCEDURE ident [FormalParameters]
    [RAISES raisees].

71  declaration = ... |
74+     ... |
      EXCEPTION {IdentList ["(" qualident ")"] ";"}.

86  definition  = ... |
89+   ... |
      EXCEPTION {IdentList ["(" qualident ")"]
      ["=" qualident] ";"}.
```

21.2.2. Safety

```
15  type = ... |
16+   ... |
      REF ident | REF type | ... .

91+ CompilationUnit = [SAFE] DefinitionModule |
      [SAFE] [IMPLEMENTATION] ProgramModule.
```

21.2.3. Runtime Types

```
49  statement = [ ...
52+   ... |
      ... | TypecaseStatement | ... ].

  TypecaseStatement = TYPECASE expression OF tcase
    {"|" tcase} [ELSE StatementSequence] END.
  tcase = [QualidList ":" StatementSequence] |
    qualident ["(" ident ")"] ":" StatementSequence.
```

21.2.4. Opaque Types

```
86 definition = ... |
87+  TYPE {ident ["=" type | "=" REF] ";"} |
```

21.2.5. Open Arrays

```
21+ ArrayType = ARRAY [SimpleType {"," SimpleType}] OF type.
```

21.2.6. Concurrency

```
49 statement = [ ... |
52+  ... | LockStatement ].
  LockStatement = LOCK designator DO StatementSequence END.
```

21.2.7. Interface Extensions

```
86 definition = ... |
88+  VAR {ident ":" type ["=" qualident] ";"} |
89+  ProcedureHeading ["=" qualident] ";" |
    EXCEPTION
    {IdentList ["(" qualident ")"] ["=" qualident] ";"}.
```

21.2.8. Default Parameters

```
33+ FormalTypeList = "(" [[VAR] FType
34+  {"," [VAR] FType}] ")" [":" qualident].
  FType = FormalType | (qualident ":=" ConstExpression).

75 FormalParameters =
76+  "(" [ FPSect {";" FPSect}] ")" [":" qualident].
  FPSect = FPSection |
    (identlist ":" qualident ":=" ConstExpression).
```

21.2.9. Miscellaneous Extensions

```
7  [ [  Do we need to note our extension to strings?  ] ]

15 type = ...
16+  ... | BITS ConstExpression FOR type.

27 variant = [CaseLabelList ":" FieldListSequence].

61 case = [CaseLabelList ":" StatementSequence].
```

---

[7]Lines 27 and 61 were extensions with regard to second-edition Modula-2. Wirth has included these same extensions in third-edition Modula-2.

22. Notes

22.1. Compiling DEF modules

We used to say (somewhere):

DEFINITION modules, which serve as interfaces to IMPLEMENTATION modules, are not compiled in Modula-2+.

I don't want to say much about the current (temporary) way of getting what we hope someday to get more smoothly. Perhaps a pointer to the example directories ...

22.2. Operations on Mixed-Operand Expressions

Is a signed or unsigned comparison used when a CARDINAL variable is compared to an UNSIGNED constant? Signed.

Is a signed or unsigned comparison used when a `System.Address` variable is compared to an UNSIGNED constant? Signed.

Notes about System.Address:

If System.Address is mixed in arithmetic expressions with INTEGER, UNSIGNED, or a POINTER type, the result of the expression has type System.Address and signed arithmetic is used.

The following arithmetic operations are legal on (address-unsigned, address-integer, address-pointer) mixed operand pairs: +, -, DIV and MOD.

The code generator cannot handle multiplication with variable addresses. This check will be moved to the front end.

System.Address can be combined with INTEGER, UNSIGNED, and POINTER types using any of =, #, < , >, <=, >=. The type of the result is BOOLEAN. Signed comparison is used.

(The key to combining Address with other pointers, unsigned, and integers lies in the typechecking predicate compatible, which given address and one of the other types returns address as the result type).

MIN and MAX take 2 operands of the same type and so are not valid for mixed address-other type.

Binary operator chart (shows instructions used)

	int-int	uns-uns	adr-adr	adr-int	adr-uns	adr-ptr
=	jneq	jneq	jneq	jneq	jneq	jneq
<	jgeq	jgequ	jgeq	jgeq	jgeq	jgeq
>	jleq	jlequ	jleq	jleq	jleq	jleq
<=	jgtr	jgtru	jgtr	jgtr	jgtr	jgtr
>=	jlss	jlssu	jlss	jlss	jlss	jlss
#	jeql	jeql	jeql	jeql	jeql	jeql

+	addl3	addl3	movl op2,r1
			movl op1,r2
			addl2 r1,r2
				movl op2,r1
				movl op1,r2
				addl2 r1,r2
					movl op2,r1
					movl op1,r2
					addl2 r1,r2
						movl op2,r1
						movl op1,r2
						addl2 r1,r2

-	subl3	subl3	subl3	subl3	subl3	subl3
*	mull3	mull3	--	--	--	--
DIV	divl3	pushl pushl calls runtime__udiv	divl3	divl3	divl3	divl3
MOD	divl3 mull2 subl2	pushl pushl calls runtime__umod	divl3 mull2 subl2	divl3 mull2 subl2	divl3 mull2 subl2	divl3 mull2 subl2
MIN	jleq	jleq (?)	--	--	--	--
MAX	jgeq	jgeq (?)	--	--	--	--

```
(?) this is a bug, jlequ and jgequ should be used.
    predictably the results are wrong. Another bug to fix.
```

Appendix I
Syntax Cheat Sheet

> It is a great advantage for a system of
> philosophy to be substantially true.
> -- George Santayana

This cheat sheet covers only the recommended grammar of Modula-2+ as of 6 January 1986 plus IMPLEMENTS. For help in reading other people's programs (perhaps in order to translate them into the recommended grammar) see Appendix IV, page 143.

This is a cheat sheet, not a language definition. Where something is too ridiculously difficult to show, we don't show it. For instance, the tokens of the language include a literal \ character followed by a literal newline character; we don't try to show that. Nor do we expand *letter* out to `A | a | B | b` ... It's a cheat sheet, for people who like cheat sheets. If you don't like cheat sheets, you're not going to like this one.

Tutorial on how to read the notation:

```
x | y     =   either x or y
?x        =   empty | x
+x        =   x | x x | x x x | ...
*x        =   ?(+x)
          =   empty | x | x x | x x x | ...
++x y     =   x *(y x)
          =   x | x y x | x y x y x | ...
**x y     =   ?(++x y)
          =   empty | x | x y x | x y x y x | ...
```

x and y stand for single grammar symbols or
things in parentheses.
A grammar symbol is either a word, with whitespace on either end,
or the characters from a ' up to the next whitespace.
Symbols in all caps or starting with ' are terminals.

There are five prefix operators:

```
?   +   *   ++   **
```

Prefix operators with ? or * generate the empty string; prefix operators with + never generate the empty string. Prefix operators with one character are unary; prefix operators with two characters are binary.

For instance:

```
?( '+ | '- ) term  =  term | '+ term | '- term
        dig +hex   =  dig hex | dig hex hex | dig hex hex hex | ...
            *dig   =  empty | dig | dig dig | dig dig dig | ...
         ++id ',   =  id | id ', id | id ', id ', id | ...
         **id ',   =  empty | id | id ', id | id ', id ', id | ...
```

Notation:

```
?x      =   empty | x
+x      =   x | x x | x x x | ...
*x      =   empty | x | x x | x x x | ...
++x y   =   x | x y x | x y x y x | ...
**x y   =   empty | x | x y x | x y x y x | ...
```

In the grammar, a non-terminal never generates an empty string.

Abbreviated non-terminals used widely in the grammar:

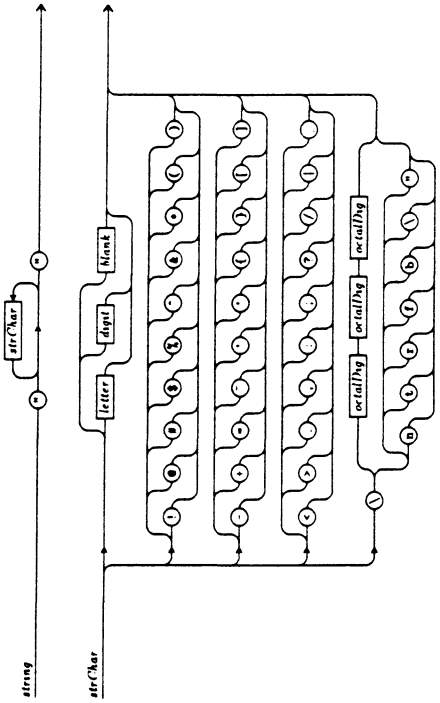
```
id      = identifier
ids     = identifiers
qi      = qualified identifier
ti      = type identifier
e       = expression
ss      = statement sequence
```

TOKENS

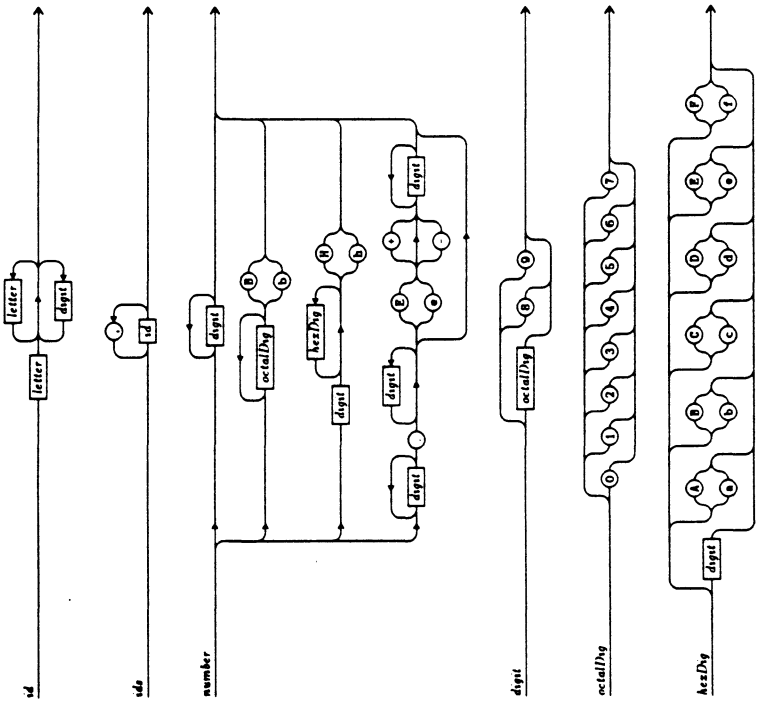
```
id          = letter *(letter | digit)
ids         = ++id ',
number      = +digit
            | +octalDig ( 'B | 'b )
            | digit *hexDig ( 'H | 'h )
            | +digit '. *digit ?(( 'E | 'e ) ?( '+ | '- ) +digit)
digit       = octalDig | '8 | '9
octalDig    = '0 | '1 | '2 | '3 | '4 | '5 | '6 | '7
hexDig      = digit
            | 'A | 'B | 'C | 'D | 'E | 'F
            | 'a | 'b | 'c | 'd | 'e | 'f
string      = '" *strChar '"
strChar     = letter | digit | blank
            | '!  | '@  | '#  | '$  | '%  | '^  | '&  | '*  | '(  | ')
            | '-  | '+  | '=  | '~  | '`  | ''  | '{  | '}  | '[  | ']
            | '<  | '>  | '.  | ',  | ':  | ';  | '?  | '/  | '|  | '_
            | '\n | '\t | '\r | '\f | '\b | '\\ | '\"
            | '\ octalDig octalDig octalDig
```

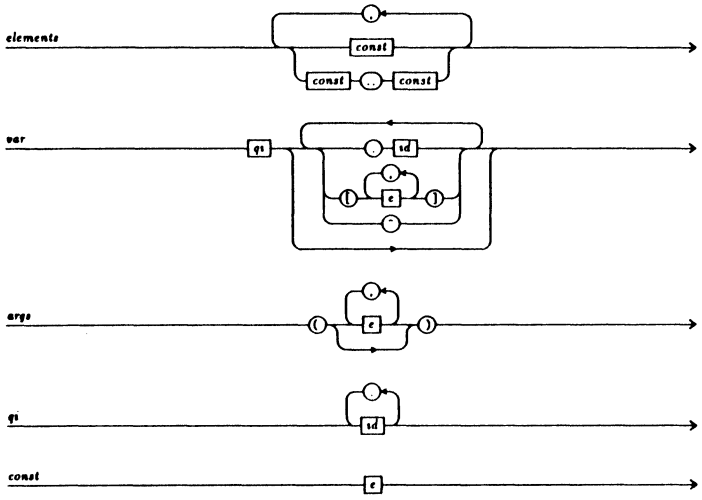
```
EXPRESSIONS  e        = sum
             | sum ('= | '# | '< | '<= | '> | '>= | IN) sum
             sum      = ?( '+ | '- ) ++term ('+ | '- | OR)
             term     = ++factor ('* | '/ | DIV | MOD | AND)
             factor   = number | string | var | var args | '( e ')
             | NOT factor | ?ti '{ ?elements '}
             elements = ++(const | const '.. const) ',
             var      = qi *( '. id | '[ ++e ', '] | '^ )
             args     = '( **e ', ')
             qi       = ++id '.
             const    = e

STATEMENTS   s  = var ':= e | var args | EXIT | RETURN ?e
                | IF              ++(e                    THEN ?ss) ELSIF ?(ELSE ?ss) END
                | CASE     e OF ?'| ++(elements               ': ?ss) '|    ?(ELSE ?ss) END
                | TYPECASE e OF ?'| ++(( ti parId | ++ti ', ) ': ?ss) '|    ?(ELSE ?ss) END
                | LOOP ?ss END
                | REPEAT ?ss UNTIL e
                | WHILE e                       DO ?ss END
                | FOR id ':= e TO e ?(BY const) DO ?ss END
                | WITH var                      DO ?ss END
                | LOCK var                      DO ?ss END
                | TRY ?ss ( EXCEPT handlers | FINALLY ?ss | PASSING exSet ?'; ) END
             ss = ++s '; ?';

             handlers = ?'|  ++(++qi ', ?parId ': ?ss) '| ?(ELSE ?parId ?ss)
                      | ELSE ?parId ?ss
             parId    = '( id ')

TYPES        type     = ti | '( ids ') | '[ const '.. const ']
                      | POINTER TO type | REF type
                      | PROCEDURE tFormals ?( ': ti ) ?(RAISES exSet)
                      | ARRAY **type ', OF type | SET OF type
                      | RECORD fields END
                      | BITS const FOR type
             ti       = qi

             fields   = ++( (ids ': type) | variants ) '; ?';
             variants = CASE ?( id ': ) ti OF ?'| **(elements ': fields) '| ?(ELSE fields) END

PROGRAMS     block    = *blockDecl ?(BEGIN ?ss) END id
             blockDecl= decl | procH '; block '; | moduleH ?export block ';
             decl     = CONST     *( id                  '= const          '; )
                      | TYPE      *( id              ?('= type | '= REF) '; )
                      | VAR       *( ids ':   type    ?('= qi)           '; )
                      | EXCEPTION *( id ?( '( type ') ) ?('= qi)         '; )
                      | procH                        ?('= qi)           ';

             procH    = PROCEDURE id formals ?( ': ti ) ?(RAISES exSet)
             formals  = '( **(?VAR ids ': ?(ARRAY OF) ti ?default) '; ')
             tFormals = '( **(?VAR       ?(ARRAY OF) ti ?default) '; ')
             default  = ':= const
             exSet    = '{ **qi ', '}
             moduleH  = MODULE id '; *import
             impsH    = MODULE id IMPLEMENTS ids '; *import
             import   = IMPORT ids '; | FROM id IMPORT ids ';
             export   = EXPORT ?QUALIFIED ids ';

             def      = ?SAFE DEFINITION     moduleH *decl END id '.
             impl     = ?SAFE IMPLEMENTATION moduleH block        '.
                      | ?SAFE                impsH   block        '.
             main     = ?SAFE                moduleH block        '.
```

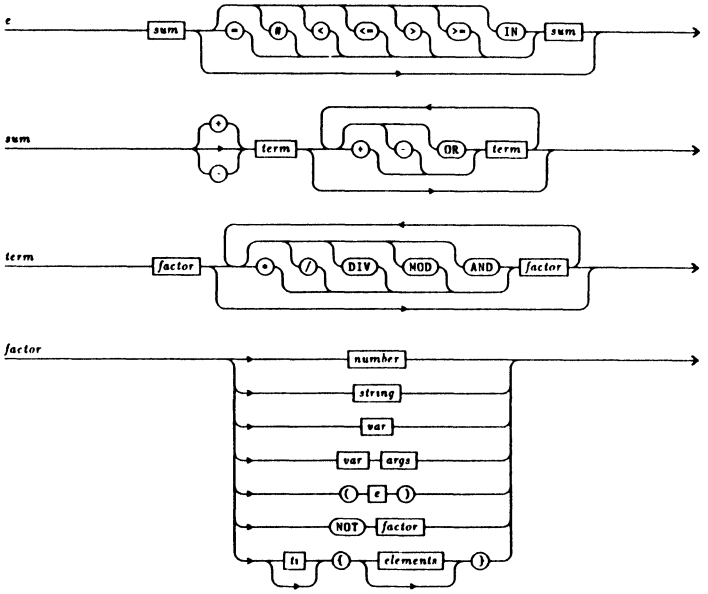
Appendix II
Railroad Diagrams

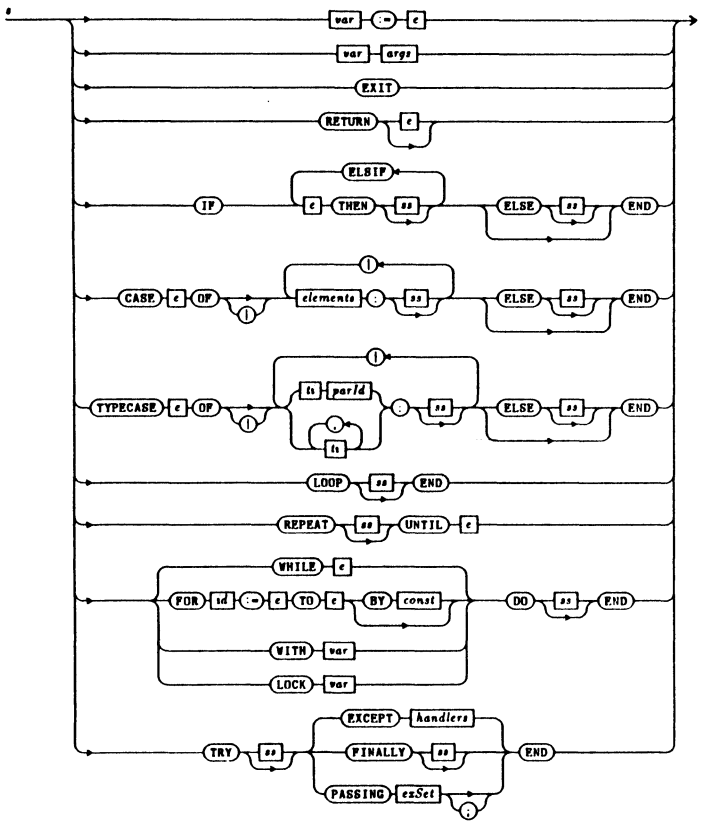


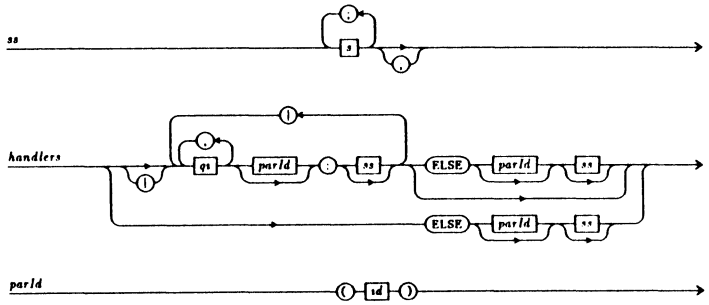
Tokens

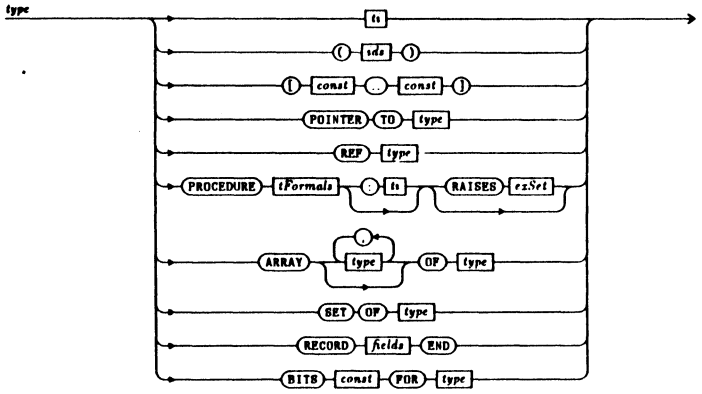


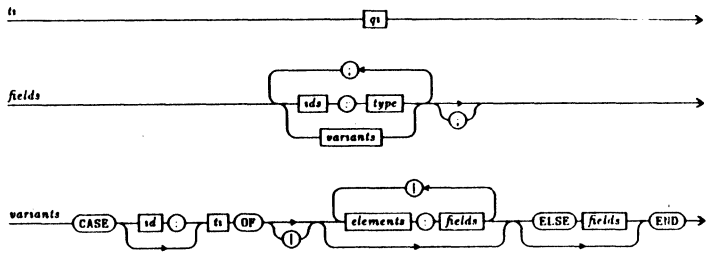

elements
const
const
.
const
var
qi
id
(
e
)
,
args
(
e
)
,
qi
id
,
const
e

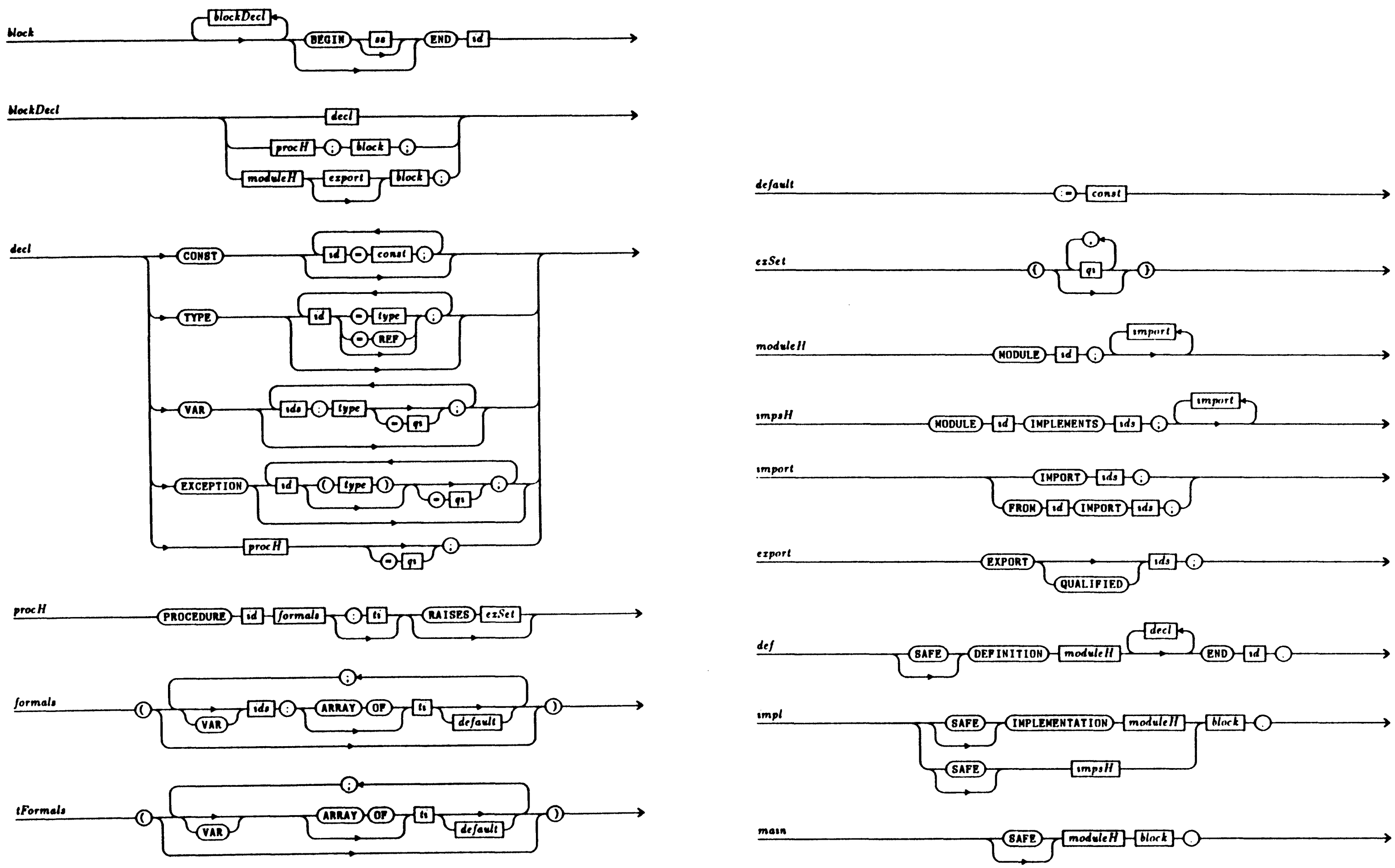

e
sum
=
#
<
<=
>
>=
IN
sum
sum
+
-
term
+
-
OR
term
term
factor
*
/
DIV
MOD
AND
factor
factor
number
string
var
var
args
(
e
)
NOT
factor
!i
(
elements
)


var
:=
e
var
args
EXIT
RETURN
e
ELSIF
IF
e
THEN
ss
ELSE
ss
END
CASE
e
OF
|
elements
:
ss
ELSE
ss
END
TYPECASE
e
OF
|
ts
parId
:
ss
,
ts
ELSE
ss
END
LOOP
ss
END
REPEAT
ss
UNTIL
e
WHILE
e
FOR
id
:=
e
TO
e
BY
const
DO
ss
END
WITH
var
LOCK
var
TRY
ss
EXCEPT
handlers
FINALLY
ss
PASSING
exSet
,
END


ss
;
s
,
handlers
|
qs
parId
:
ss
ELSE
parId
ss
ELSE
parId
ss
parId
(
id
)


type
ts
(sds)
[const .. const]
POINTER TO type
REF type
PROCEDURE tFormals : ts RAISES rzSet
ARRAY type , OF type
SET OF type
RECORD fields END
BITS const FOR type


ts
qs
fields
sds : type
variants
;
variants
CASE id : ts OF
elements : fields
ELSE fields
END
|


block
blockDecl
BEGIN
ss
END
id
blockDecl
decl
procH
;
block
;
moduleH
export
block
;
decl
CONST
id
=
const
;
TYPE
id
=
type
=
REF
;
VAR
ids
:
type
=
qs
;
EXCEPTION
id
(
type
)
=
qs
;
procH
=
qs
;
procH
PROCEDURE
id
formals
:
ts
RAISES
exSet
formals
(
;
VAR
ids
:
ARRAY
OF
ts
default
)
tFormals
(
;
VAR
ARRAY
OF
ts
default
)
default
:=
const
exSet
(
,
qs
)
moduleH
MODULE
id
;
import
impsH
MODULE
id
IMPLEMENTS
ids
;
import
import
IMPORT
ids
;
FROM
id
IMPORT
ids
;
export
EXPORT
QUALIFIED
ids
;
def
SAFE
DEFINITION
moduleH
decl
END
id
.
impl
SAFE
IMPLEMENTATION
moduleH
block
.
SAFE
impsH
main
SAFE
moduleH
block
.

Programs

pages 141 and 142
deliberately omitted

Appendix III
The Compiler's Error Messages

To be supplied by Paul and Violetta.

Appendix IV
Grot

A sunburnt bloody stockman stood,
And in a dismal, bloody mood
Apostrophized his bloody cuddy:
"This bloody moke's no bloody
good."

-- World War I soldiers' song

IV.1. Deprecated Features

[[Citation wanted so as to be able to credit the FORTRAN folks for this wonderful expression.]]

Why deprecate things?

1. Modula-2+ gives you several ways to say the same thing in ways that are syntactically different. Klaus put in some, and our extensions produced more. Programs will be easier to read in our community if we agree on one way to say these things. (It would be great to teach the pretty-printer to enforce these rules.)
2. Some Modula-2 constructs are really obsolete in our environment (e.g. PROCESS is dominated by `Thread.T`). In such cases it's not just a matter of syntax -- you should not use the obsolete constructs because they'll lead to an inferior program.
3. There are some uses of the language that are very difficult to prohibit (e.g. assignment to FOR loop index) yet are clearly bad practice.
4. There are bugs in today's compiler that allow programs that aren't really written in Modula-2+ to compile and execute. But you shouldn't depend upon these bugs; we'd like to see them fixed.

The following are classified not by the why, but the what (roughly): tokens, constants, types, statements, and naming.

Don't write the operator "<>" ; use "#" instead.

Don't write the operator "&" ; use AND instead.

Don't write the operator "~" ; use NOT instead.

Don't write a character literal consisting of an octal number followed by

"C". You can express any such literal as a single character string containing the "\" escape followed by three octal digits. Or, better yet, use the constants defined in the `Char` interface.

Don't write a string containing the "\" escape followed by fewer than three octal digits. The compiler accepts it, but this is a bug. Supply leading zeros so that the "\" escape is followed by three digits.

Don't write a string containing the "\" escape followed by something other than one of the designated escape sequences. The compiler accepts the "\" escape followed by anything, but this is a bug.

Don't write a string surrounded by single quotes. Use double quotes around all strings.

Don't write a string containing two consecutive double-quote characters to denote a single double-quote character. Use the "\" escape followed by double-quote to signify a double-quote within a string.

Don't use the constants `System.MaxInt`, `System.MaxUnsigned`, and `System.MaxCard`. Use `LAST(INTEGER)` and `LAST(UNSIGNED)` and `LAST(CARDINAL)` instead.

In the following peculiar example, the compiler accepts a constant expression that (syntactically) contains a call to a user-defined procedure:

```
TYPE
  FixedSizeArrayType = ARRAY [1 .. 4] OF INTEGER;
PROCEDURE P(): FixedSizeArrayType;
  VAR j: FixedSizeArrayType;
  BEGIN
    RETURN j;
  END P;
CONST
  C = HIGH(P());  (* same as HIGH(FixedSizeArrayType) *)
```

Don't rely on this.

Don't write a type declaration for a procedure type without a formal parameter list (for declaring a parameterless procedure type). Write the keyword PROCEDURE followed by an empty argument list.

Don't write a procedure declaration without a formal parameter list (for declaring a parameterless procedure). Write the procedure name followed by an empty argument list.

Don't use the supplied type `PROC = PROCEDURE ();`

Don't write a module header in which the module name is followed by a number enclosed in square brackets (the interrupt priority level of the module). Don't use the Modula-2 type PROCESS. Use the `Thread` interface and the LOCK statement to express concurrent programs.

Don't write a type name followed by an expression in parentheses (the Modula-2 type transfer function). Use LOOPHOLE instead. This makes the type breach easier to notice.

Don't write `ORD(i)` where `i` is an INTEGER or `ORD(u)` where `u` is an

UNSIGNED. The results are peculiar (this is not simply a type-casting operation, the representation also changes). Use LOOPHOLE instead.

Similarly, don't write any of these:

```
VAL(INTEGER, e)
VAL(UNSIGNED, e)
VAL(System.Byte, e)
VAL(System.Word, e)
```

Use LOOPHOLE instead.

Don't write a type constructor of the form `BITS n FOR T`, where `n` is smaller than the "natural" size of type `T`. For instance, `BITS 15 FOR CARDINAL` is deprecated. Write `BITS 15 FOR [0..7fffH]` instead.

Don't write a type constructor of the form `BITS n FOR System.Word` or `BITS n FOR System.Byte`.

Don't write a type constructor of the form `ARRAY Index OF T`, where `Index` is INTEGER, CARDINAL, or UNSIGNED.

Don't pass a value whose size is less than 8 bits to a (value) `System.Byte` parameter, or pass a value whose size is not evenly divisible by 8 bits to a (value) `System.Byte` parameter.

Similarly, don't pass a value whose size is less than 32 bits to a (value) `System.Word` parameter, or pass a value whose size is not evenly divisible by 32 bits to a (value) `System.Word` parameter.

Don't write a procedure call statement without an argument list (for calling a parameterless procedure that returns no results). Write the procedure name (or procedure variable) followed by an empty argument list.

Don't make assignments to the control variable of a FOR-loop within the loop. In simple cases the compiler could detect and disallow this, but it does not. Use a WHILE-, UNTIL-, or LOOP-loop instead.

Don't write a RETURN statement in the body of a main module (meaning normal termination of the main program). Use the supplied procedure HALT instead (Chapter 11, page 55), since it expresses your intent more clearly.

Don't use the supplied procedure CAP. Use `Char.Upper` (there's also `Char.Lower`, but no supplied procedure UNCAP).

This one's complicated. Suppose that some name N is defined in a module because you've imported N, or is defined in a procedure because N is defined earlier in the containing module or procedure. If you're going to make a declaration that defines N, thereby overriding the existing definition, then make this declaration before using the name N in this module or procedure. If you use N before re-defining it, you won't be able to predict which definition of N is used.

For instance,

```
MODULE Silly;
TYPE
  T = INTEGER;
PROCEDURE P();
  VAR
    t: T;
  TYPE
    T = BOOLEAN;
  BEGIN
  END P;
END Silly.
```

The compiler won't complain (it would be nice if it would), but the type of variable t is not well-defined (different Modula-2+ compilers might well give different answers). Don't provoke this!

There's a similar name scope issue in record declarations. Avoid doing obscure things like the following:

```
CONST
  Month = 12;
TYPE
  Foo = RECORD
    Month: [1..Month];
  END;
```

IV.2. DIV and MOD are weird

```
DIV and MOD with INTEGER arguments are weird. MOD is not the
mathematical mod, but it's worse than that. If the value of
the divisor can be computed at compile time, is positive,
and is a power of 2, different instructionse used. This causes
different results when the dividend is negative and the divisor
is positive.

divisor is constant, positive and power of 2, uses instructions:
  i DIV 4  =>  divl3     $4,(_test_i+0),r1
  i MOD 4  =>  movl      (_test_i+0),r1
               bicl2     $-4,r1
 otherwise uses instructions:
  i DIV j   =>  divl3    (_test_j+0),(_test_i+0),r1
  i MOD j   =>  movl     (_test_i+0),r1
                divl3    (_test_j+0),r1,r2
                mull2    (_test_j+0),r2
                subl2    r2,r1

results:
  i := 11; j := 4;    =>  i DIV j = 2          i MOD j = 3
  i := -11; j := 4;   =>  i DIV j = -2         i MOD j = -3 (**)
  i := 11; j := -4;   =>  i DIV j = -2         i MOD j = 3
  i := -11; j := -4;  =>  i DIV j = 2          i MOD j = -3

  i := 11;            =>  i DIV 4 =  2         i MOD 4 = 3
  i := - 11;          =>  i DIV 4 = -2         i MOD 4 = 1  (**)
  i := 11;            =>  i DIV -4 = -2        i MOD -4 = 3
  i := -11;           =>  i DIV -4 = 2         i MOD -4 = -3

  i := 11; j := 5;    =>  i DIV j = 2          i MOD j = 1
  i := -11; j := 5;   =>  i DIV j = -2         i MOD j = -1
  i := 11; j := -5;   =>  i DIV j = -2         i MOD j = 1
  i := -11; j := -5;  =>  i DIV j = 2          i MOD j = -1
```

```
DIV and MOD applied to UNSIGNED variable arguments seem to do the
right thing. As for INTEGER, different instructions are generated
if the divisor can be computed at compile time and is a power of 2,
but since the dividend and the divisor are both nonnegative, the
results are the same.

divisor is constant, positive and power of 2, uses instructions:
  u DIV 4    =>  movl      (_test_u+0),r1
                 bicl2     $3,r1
                 rotl      $-2,r1,r1
  u MOD 4    =>  movl      (_test_u+0),r1
                 bicl2     $-4,r1

otherwise uses instructions:
  u DIV v    =>  pushl     (_test_v+0)
                 pushl     (_test_u+0)
                 calls     $2,_runtime__udiv
  u MOD v    =>  pushl     (_test_v+0)
                 pushl     (_test_u+0)
                 calls     $2,_runtime__umod

results:
  u := LAST(INTEGER) + 1;
  v := LAST(INTEGER) - 1; =>   u DIV v = 1        u MOD v = 2

  u := 11;  v := 4;
  u DIV 4 = 2       u MOD 4 = 3
```

IV.3. Where the compiler accepts too little

Bugs where the compiler accepts too much are reported as deprecated features.

Bugs where the compiler accepts too little can't be called features.

This won't work:

```
PROCEDURE A(); END A;
PROCEDURE B(); END B;
CONST
  X = (A = B);
```

You also can't write

```
CONST
  X = (A = NIL)
```

or even

```
CONST
  X = (NIL = NIL)
```

which are strange things to disallow, though you seldom want to write them (!)

You should be able to write a vertical bar before the ELSE in a TRY EXCEPT that has only an ELSE, but you cannot.

Appendix V
Reserved Words and Standard Identifiers

Notation:

const means constant
proc means procedure
type means type
word means reserved word

no suffix means ordinary Modula-2
-WRL means WRL extension
-SRC means SRC extension

M2+ includes both WRL and SRC exensions.

Note that MAX and MIN are different procedures in vanilla Modula-2 and the WRL/SRC dialect.

[[Some question about who gets credit for UNSIGNED, WRL or us.]]

ABS	proc-WRL	LOOP	word		
AND	word		LOOPHOLE	proc-SRC	
ARRAY	word		MAX	proc-WRL	
ASSERT	proc-WRL		MIN	proc-WRL	
BEGIN	word		MOD	word	
BITS	word-SRC		MODULE	word	
BITSET	type		NARROW	proc-SRC	
BOOLEAN	type		NEW	proc	
BY	word		NIL	const	
@C	word-WRL		@NOCOUNT	word-WRL	
CAP	proc		NOT	word	
CARDINAL	type		NUMBER	proc-WRL	
CASE	word		ODD	proc	
CHAR	type		OF	word	
CHR	proc		OR	word	
CONST	word		ORD	proc	
DEC	proc		@PASCAL	word-WRL	
DEFINITION	word		PASSING	word-SRC	
DISPOSE	proc		POINTER	word	
DIV	word		PROC	type[8]	
DO	word		PROCEDURE	word	
ELSE	word		QUALIFIED	word	
ELSIF	word		RAISE	proc-SRC	
END	word		RAISES	word-SRC	
EXCEPT	word-SRC		REAL	type	
EXCEPTION	word-SRC		RECORD	word	
EXCL	proc		REF	word-SRC	
EXIT	word		REFANY	type-SRC	
EXPORT	word		REPEAT	word	
@EXTERNAL	word-WRL		RETURN	word	
FALSE	const		SAFE	word-SRC	
FINALLY	word-SRC		SET	word	
FIRST	proc-WRL		THEN	word	
FLOAT	proc		TO	word	
FOR	word		TRUE	const	
FROM	word		TRUNC	proc	
HALT	proc		TRY	word-SRC	
HIGH	proc		TYPE	word	
IF	word		TYPECASE	word-SRC	
IMPLEMENTATION	word		UNSIGNED	type-WRL	
IMPLEMENTS	word		UNTIL	word	
IMPORT	word		VAL	proc	
IN	word		VAR	word	
INC	proc		WHILE	word	
INCL	proc		WITH	word	
INTEGER	type				

[8]deprecated

LAST	proc-WRL
LOCK	word-SRC
LONGFLOAT	proc-WRL
LONGREAL	type-WRL

Appendix VI
Wirth's EBNF

VI.1. Notation

Wirth's notation is an extended Backus-Naur formalism called EBNF. [4] Square brackets [] mean that the enclosed form is optional, and curly brackets {} mean that the enclosed form is repeated (possibly 0 times). Non-terminals have names that are meant to express their intuitive meaning and written in uppers and lowers. Terminal symbols, symbols of the language vocabulary, are either strings enclosed in quotes or reserved words, written in capital letters.

[[All you people who love EBNF are invited to come up with a volunteer from among yourselves to maintain this representation of the grammar. So far I've heard lots of testimonials but no offers of assistance.]]

VI.2. Syntax in Wirth's EBNF

These numbers are production numbers, not line numbers.

```
 1 ident = letter {letter | digit}.
 2 number = integer | real.
 3 integer =
     digit {digit}
   | octalDigit {octalDigit} ("B" | "C")
   | digit {hexDigit} "H".
 4 real = digit {digit} "." {digit} [ScaleFactor].
 5 ScaleFactor = "E" ["+" | "-"] digit {digit}.
 6 hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
 7 digit = octalDigit | "8" | "9".
 8 octalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
 9 string = "'" {character} "'" | '"' {character} '"'.
10 qualident = ident {"." ident}.
11 ConstantDeclaration = ident "=" ConstExpression.
12 ConstExpression = expression.
13 TypeDeclaration = ident "=" type.
14 type =
     SimpleType | ArrayType | RecordType | SetType
   | PointerType | ProcedureType | REF ident | REF type
   | BITS ConstExpression FOR type.
15 SimpleType = qualident | enumeration | SubrangeType.
16 enumeration = "(" IdentList ")".
17 IdentList = ident {"," ident}.
18 SubrangeType =
   [qualident] "[" ConstExpression ".." ConstExpression "]".
19 ArrayType = ARRAY [SimpleType {"," SimpleType}] OF type.
20 RecordType = RECORD FieldListSequence END.
21 FieldListSequence = FieldList {";" FieldList}.
22 FieldList =
   [ IdentList ":" type
   | CASE [ident] ":" qualident OF variant {"|" variant}
     [ELSE FieldListSequence] END
   ].
23 variant = [CaseLabelList ":" FieldListSequence].
24 CaseLabelList = CaseLabels {"," CaseLabels}.
25 CaseLabels = ConstExpression [".." ConstExpression].
26 SetType = SET OF SimpleType.
27 PointerType = POINTER TO type.
28 ProcedureType = PROCEDURE [FormalTypeList] [RAISES raisees].
29 FormalTypeList =
   "(" [[VAR] FType {"," [VAR] FType}] ")" [":" qualident].
30 FType = FormalType | (qualident ":=" ConstExpression).
31 VariableDeclaration = IdentList ":" type.
32 designator = qualident {"." ident | "[" ExpList "]" | "^"}.
33 ExpList = expression {"," expression}.
34 expression = SimpleExpression [relation SimpleExpression].
35 relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN.
36 SimpleExpression = ["+" | "-"] term {AddOperator term}.
37 AddOperator = "+" | "-" | OR.
38 term = factor {MulOperator factor}.
39 MulOperator = "*" | "/" | DIV | MOD | AND.
40 factor =
     number | string | set | designator [ActualParameters]
   | "(" expression ")" | NOT factor.
41 set = [qualident] "{" [element {"," element}] "}".
```

```
42 element = expression [".." expression].
43 ActualParameters = "(" [ExpList] ")".
44 statement =
   [  assignment | ProcedureCall | IfStatement
    | CaseStatement | WhileStatement | RepeatStatement
    | LoopStatement | ForStatement | WithStatement | EXIT
    | RETURN [expression] | TryStatement
    | TypecaseStatement | LockStatement
   ].
45 assignment = designator ":=" expression.
46 ProcedureCall = designator [ActualParameters].
47 StatementSequence = statement {";" statement}.
48 IfStatement =
   IF expression THEN StatementSequence
   {ELSIF expression THEN StatementSequence}
   [ELSE StatementSequence]
   END.
49 CaseStatement =
   CASE expression OF case {"|" case}
   [ELSE StatementSequence]
   END.
50 case = [CaseLabelList ":" StatementSequence].
51 WhileStatement =
   WHILE expression DO StatementSequence END.
52 RepeatStatement =
   REPEAT StatementSequence UNTIL expression.
53 ForStatement =
   FOR ident ":=" expression TO expression
   [BY ConstExpression] DO StatementSequence END.
54 LoopStatement = LOOP StatementSequence END.
55 WithStatement = WITH designator DO StatementSequence END.
56 TryStatement = TRY StatementSequence TryTail END.
57 TryTail =
    FINALLY StatementSequence | PASSING raisees
   | PASSING raisees ";"
   | EXCEPT [HandlerArm {"|" HandlerArm}]
     [ELSE ["(" ident ")"] StatementSequence].
58 HandlerArm =
   [QualidList ["(" ident ")"] ":" StatementSequence].
59 raisees = "{" [QualidList] "}".
60 QualidList = qualident {"," qualident}.
61 TypecaseStatement =
   TYPECASE expression OF tcase {"|" tcase}
   [ELSE StatementSequence]
   END.
62 tcase =
    [QualidList ":" StatementSequence]
   | qualident ["(" ident ")"] ":" StatementSequence.
63 LockStatement = LOCK designator DO StatementSequence END.
64 ProcedureDeclaration = ProcedureHeading ";" block ident.
65 ProcedureHeading =
   PROCEDURE ident [FormalParameters] [RAISES raisees].
66 block = {declaration} [BEGIN StatementSequence] END.
67 declaration =
    CONST {ConstantDeclaration ";"}
   | TYPE {TypeDeclaration ";"}
   | VAR {VariableDeclaration ";"}
   | ProcedureDeclaration ";"
   | ModuleDeclaration ";"
```

```
   | EXCEPTION {IdentList ["(" qualident ")"] ";"}.
68 FormalParameters =
   "(" [ FPSect {";" FPSect}] ")" [":" qualident].
69 FPSection = [VAR] IdentList ":" FormalType.
70 FPSect =
     FPSection
   | (identlist ":" qualident ":=" ConstExpression).
71 FormalType = [ARRAY OF] qualident.
72 ModuleDeclaration =
   MODULE ident [priority] ";" {import} [export] block ident.
73 priority = "[" ConstExpression "]".
74 export = EXPORT [QUALIFIED] IdentList ";".
75 import = [FROM ident] IMPORT IdentList ";".
76 DefinitionModule =
   DEFINITION MODULE ident ";" {import}
   {definition} END ident ".".
77 definition =
     CONST {ConstantDeclaration ";"}
   | TYPE {ident ["=" type | "=" REF] ";"}
   | VAR {ident ":" type ["=" qualident] ";"}
   | ProcedureHeading ["=" qualident] ";"
   | EXCEPTION
       {IdentList ["(" qualident ")"] ["=" qualident] ";"}.
78 ProgramModule =
   MODULE ident [priority] ";" {import} block ident ".".
79 CompilationUnit =
     [SAFE] DefinitionModule
   | [SAFE] [IMPLEMENTATION] ProgramModule.
```

VI.3. Alpha Order

The numbers refer to the production numbers above.

```
43 ActualParameters = "(" [ExpList] ")".
37 AddOperator = "+" | "-" | OR.
19 ArrayType = ARRAY [SimpleType {"," SimpleType}] OF type.
45 assignment = designator ":=" expression.
66 block = {declaration} [BEGIN StatementSequence] END.
50 case = [CaseLabelList ":" StatementSequence].
24 CaseLabelList = CaseLabels {"," CaseLabels}.
25 CaseLabels = ConstExpression [".." ConstExpression].
49 CaseStatement =
   CASE expression OF case {"|" case}
   [ELSE StatementSequence]
   END.
79 CompilationUnit =
    [SAFE] DefinitionModule
   | [SAFE] [IMPLEMENTATION] ProgramModule.
11 ConstantDeclaration = ident "=" ConstExpression.
12 ConstExpression = expression.
67 declaration =
    CONST {ConstantDeclaration ";"}
   | TYPE {TypeDeclaration ";"}
   | VAR {VariableDeclaration ";"}
   | ProcedureDeclaration ";"
   | ModuleDeclaration ";"
   | EXCEPTION {IdentList ["(" qualident ")"] ";"}.
77 definition =
    CONST {ConstantDeclaration ";"}
   | TYPE {ident ["=" type | "=" REF] ";"}
   | VAR {ident ":" type ["=" qualident] ";"}
   | ProcedureHeading ["=" qualident] ";"
   | EXCEPTION
     {IdentList ["(" qualident ")"] ["=" qualident] ";"}.
76 DefinitionModule =
   DEFINITION MODULE ident ";" {import}
   {definition} END ident ".".
32 designator = qualident {"." ident | "[" ExpList "]" | "^"}.
 7 digit = octalDigit | "8" | "9".
42 element = expression [".." expression].
16 enumeration = "(" IdentList ")".
33 ExpList = expression {"," expression}.
74 export = EXPORT [QUALIFIED] IdentList ";".
34 expression = SimpleExpression [relation SimpleExpression].
40 factor =
    number | string | set | designator [ActualParameters]
   | "(" expression ")" | NOT factor.
22 FieldList =
   [ IdentList ":" type
   | CASE [ident] ":" qualident OF variant {"|" variant}
     [ELSE FieldListSequence] END
   ].
21 FieldListSequence = FieldList {";" FieldList}.
68 FormalParameters =
   "(" [ FPSect {";" FPSect}] ")" [":" qualident].
71 FormalType = [ARRAY OF] qualident.
29 FormalTypeList =
   "(" [[VAR] FType {"," [VAR] FType}] ")" [":" qualident].
```

```
53 ForStatement =
   FOR ident ":=" expression TO expression
   [BY ConstExpression] DO StatementSequence END.
70 FPSect =
    FPSection
   | (identlist ":" qualident ":=" ConstExpression).
69 FPSection = [VAR] IdentList ":" FormalType.
30 FType = FormalType | (qualident ":=" ConstExpression).
58 HandlerArm =
   [QualidList ["(" ident ")"] ":" StatementSequence].
 6 hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
 1 ident = letter {letter | digit}.
17 IdentList = ident {"," ident}.
48 IfStatement =
   IF expression THEN StatementSequence
   {ELSIF expression THEN StatementSequence}
   [ELSE StatementSequence]
   END.
75 import = [FROM ident] IMPORT IdentList ";".
 3 integer =
    digit {digit}
   | octalDigit {octalDigit} ("B" | "C")
   | digit {hexDigit} "H".
63 LockStatement = LOCK designator DO StatementSequence END.
54 LoopStatement = LOOP StatementSequence END.
72 ModuleDeclaration =
   MODULE ident [priority] ";" {import} [export] block ident.
39 MulOperator = "*" | "/" | DIV | MOD | AND.
 2 number = integer | real.
 8 octalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
27 PointerType = POINTER TO type.
73 priority = "[" ConstExpression "]".
46 ProcedureCall = designator [ActualParameters].
64 ProcedureDeclaration = ProcedureHeading ";" block ident.
65 ProcedureHeading =
   PROCEDURE ident [FormalParameters] [RAISES raisees].
28 ProcedureType = PROCEDURE [FormalTypeList] [RAISES raisees].
78 ProgramModule =
   MODULE ident [priority] ";" {import} block ident ".".
10 qualident = ident {"." ident}.
60 QualidList = qualident {"," qualident}.
59 raisees = "{" [QualidList] "}".
 4 real = digit {digit} "." {digit} [ScaleFactor].
20 RecordType = RECORD FieldListSequence END.
35 relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN.
52 RepeatStatement = REPEAT StatementSequence UNTIL expression.
 5 ScaleFactor = "E" ["+" | "-"] digit {digit}.
41 set = [qualident] "{" [element {"," element}] "}".
26 SetType = SET OF SimpleType.
36 SimpleExpression = ["+" | "-"] term {AddOperator term}.
15 SimpleType = qualident | enumeration | SubrangeType.
44 statement =
   [  assignment | ProcedureCall | IfStatement
    | CaseStatement | WhileStatement | RepeatStatement
    | LoopStatement | ForStatement | WithStatement | EXIT
    | RETURN [expression] | TryStatement
    | TypecaseStatement | LockStatement
   ].
47 StatementSequence = statement {";" statement}.
```

```
 9 string = "'" {character} "'" | '"' {character} '"'.
18 SubrangeType =
   [qualident] "[" ConstExpression ".." ConstExpression "]".
62 tcase =
    [QualidList ":" StatementSequence]
   | qualident ["(" ident ")"] ":" StatementSequence.
38 term = factor {MulOperator factor}.
56 TryStatement = TRY StatementSequence TryTail END.
57 TryTail =
    FINALLY StatementSequence | PASSING raisees
   | PASSING raisees ";"
   | EXCEPT [HandlerArm {"|" HandlerArm}]
      [ELSE ["(" ident ")"] StatementSequence].
14 type =
    SimpleType | ArrayType | RecordType | SetType
   | PointerType | ProcedureType | REF ident | REF type
   | BITS ConstExpression FOR type.
61 TypecaseStatement =
   TYPECASE expression OF tcase {"|" tcase}
   [ELSE StatementSequence]
   END.
13 TypeDeclaration = ident "=" type.
31 VariableDeclaration = IdentList ":" type.
23 variant = [CaseLabelList ":" FieldListSequence].
51 WhileStatement =
   WHILE expression DO StatementSequence END.
55 WithStatement = WITH designator DO StatementSequence END.
```

References

[1] Danny Cohen.
Blah Blah Blah Blah ... On Holy Wars and a Plea for Peace.
XXXX XX(X), XXXX.

[2] J. G. Mitchell et al.
Mesa Language Manual.
Technical Report Report CSL-79-3, Xerox PARC, 1979.

[3] N. Wirth.
Programming in Modula-2.
Springer-Verlag, Third Edition, 1985.

[4] N. Wirth.
What Can We Do about the Unnecessary Diversity of Notation etc.
CACM 20(11):822-823, November 1977.

Index