

The Language JPI Modula-2 version 1

This chapter gives a concise definition of the TopSpeed Modula-2 language. The language definition is kept compact and should be read with care. This style of presentation is in contrast to the case studies, which give a more informal and explanatory description. Programs must conform to the syntax of Modula-2. The syntax specifies the basic textual structure of valid programs. Only syntactically correct programs are accepted by the compiler. Secondly, programs must conform to the compile-time (static) semantics of Modula-2. This concerns the meaning associated with program identifiers, and the way they are used. It involves, among other things, type-checking. Errors in these matters are detected by the compiler. Finally, programs should conform to the run-time (dynamic) semantics of Modula-2. This involves the actual behavior of an executing program. It is required, for example, that array-index values stay within certain ranges. It is optional whether these constraining rules are enforced during program execution. If not, violating them will generally produce undefined results, but can be used to achieve certain devious effects. The underlying machine architecture is the 8086-family with an address space of 220 (IM) bytes of 8 bits. A physical address is formed from a 16-bit segment and a 16-bit offset; the absolute byte-address is: $(16 * \text{segment}) + \text{offset}$. Therefore, address arithmetic is handled differently than specified by Wirth (in Programming in Modula-2). See the discussion of pointer constructors later in this chapter (page 109). Boldface will be used when new concepts are introduced, or when an ordinary phrase is given a special well-defined meaning. Examples will be used to illustrate each of the concepts described; some will refer to entities declared in previous examples.

Textual Topics

Tokens

Tokens are the basic textual elements on which the structure of Modula-2 is based. A token is a sequence of characters from the ASCII set. The tokens fall into four classes: keywords, delimiters, generic tokens, and separators. The keywords and delimiters consist of fixed sequences of characters, whereas for each generic token a multitude of character sequences are possible.

The keywords are:

```
AND
FOR
OR
ARRAY
BEGIN
BY
CASE
CONST
DEFINITION
DIV
DO
ELSE
ELSIF
END
EXIT
```

```
EXPORT
FORWARD
FROM
GOTO
IF
IMPLEMENTATION
IMPORT
IN
LABEL
LOOP
MOD
MODULE
NOT
OF
POINTER
PROCEDURE
QUALIFIED
RECORD
REPEAT
RETURN
SET
THEN
TO
TYPE
UNTIL
VAR
WHILE
WITH
```

The delimiters are:

```
+ - * / := & . , ;
: ( ) [ 1 { } ^ ~
= # <> < <= > >= << >>
```

The generic tokens are:

- Identifier: a list of letters ('A' to 'Z', 'a' to 'z' and and digits ('0' to '9') starting with a letter; the 43 keywords are excluded. Uppercase and lowercase letters are considered distinct.

Examples:

HelloThere Agent_007 _main

- Decimal literal: a list of digits.

Examples:

12345 0 255

- Octal literal: a list of octal digits ('0' to '7') followed by 'B'.

Examples:

10B (=8) 377B (=255)

- Hex literal: a list of digits and hexadecimal letters ('A' to 'F') followed by 'H'; it must start with a digit.

Examples:

10H (=16) OFFH (=255)

- Real literal: a list of digits, followed by optionally followed by a list of digits, optionally followed by an exponent part consisting of an 'E' followed by an optional sign, '+' or '-' followed by a list of digits.

Examples:

3.14 12.3E-3 (=0.0123)

- String literal: a list of characters enclosed in quotes (') or double-quotes ("). The enclosed list cannot contain the enclosing character, nor can the list extend over line breaks. A string of length one is alternatively called a character. A character can also be specified by its octal ASCII value as a list of octal digits followed by 'C'.

Examples:

'Hi' "that's ok" 101C (= 'A')

The separators are:

- White-spaces: any list of blanks, tabs and line breaks.
- Comments: any list of characters enclosed in '(*' and **)'. Comments can be nested and can extend over line breaks.

The tokens '#' and '<>' can be used interchangeably, as can '&' and 'AND'. The tokens '~' and 'NOT' can also be used interchangeably.

Identifiers are used to denote user-defined entities.

The decimal, octal, and hex literals denote whole numbers.

Real literals denote real numbers. The exponent part denotes multiplication by the specified power of 10.

As many characters as possible are fitted into each token: 123 is one three-digit literal, not three one-digit literals.

Separators, except comments starting by '(*\$' (see Chapter 7), have no influence on the meaning of the program except to separate tokens.

Syntax

The syntax of Modula-2 describes how sequences of tokens are grouped to form valid program text. The syntax contains a set of syntactical constructs and productions. The productions specify how constructs and

tokens are combined to form new constructs. Each construct can have several alternative productions, each specifying a possible expansion of that construct. The alternatives will be shown where relevant, rather than being shown collectively.

The following meta-symbols are used in the productions:

- square brackets ([and]) are used to enclose optional parts.
- curly braces ({ and }) are used to enclose parts that can be repeated zero or more times.
- bar (|) is used to separate alternatives.
- definition symbol (: =) separates the syntactical construct being defined from its expansion.

Mixed case is used for names of constructs, upper case for keywords. Delimiters are shown in single quotes. The generic tokens are named: Id, WholeNumber, RealNumber and String.

The productions form the backbone of the language definition, and should be 'read' like ordinary statements; the text following each production will implicitly refer to its constituents. Thus

- `IdList ::= Id { Id }`

defines a list of one or more identifiers separated by commas.

Examples:

`elloThere, _main , X, Agent_007`

Declarations and Visibility

Every identifier must either be declared or predefined. Declarations introduce programmer-defined entities and establish their properties. After an identifier is declared, it is used to name the declared entity:

- `Name ::= Id`

Declarations come in lists:

- `Del List ::= { Declaration }`

Identifiers must be declared before they are used. The only exception is types designated by pointers (see "Pointer Types" page 104), which must be declared by the end of the same declaration list. Before the declaration, operations requiring knowledge of the designated type are illegal - for example, anything involving de-referencing.

All identifiers declared in a declaration list must be distinct.

A declaration remains in effect throughout the scope of the declaration. The scope extends from the declaration itself, throughout the rest of the declaration list, and throughout a possible list of statements associated with the declaration list.

Declaration lists can be nested by means of procedure bodies and modules (see "Bodies" page 116 and "Modules" page 121), and it is legal to re-declare identifiers in such inner scopes. WITH-statements are another way of making nested scopes. A particular instance of an identifier denotes the entity declared previously in the innermost enclosing scope; it hides entities denoted, by the same identifier in outer scopes.

Example:

```

MODULE M;

VAR I,J: CARDINAL; (* Two variables belonging to M *)

PROCEDURE P;
VAR I,K: INTEGER; (* Two variable^ belonging to P *)
BEGIN
    I := 7;  (* P's I *)
    J := 8;  (* M's J *)
    K := 9;  (* P's K *)
END P;

BEGIN
    I := 10;  (* M's I *)
    J := 11;  (* M's J *)
    (* no K is visible here *)
END M;

```

Alias declarations do not declare new entities, but introduce alternative names for existing ones:

- Declaration ::= const { id Name }

The name can denote any entity.

Example:

```
CONST VisibleVersion : := AboutToBeHidden;
```

The predefined identifiers, listed below, are considered to be declared in an (outermost) scope, which is all-enclosing. The meaning of individual identifiers is explained later.

```

ABS DEC LONGCARD    ORD WORD
ADDRESS DISPOSE LONGINT PROC    VSIZE
ADR EXCL    LONGREAL    REAL
BITSET FALSE    LONGWORD    SHORTADDR
BOOLEAN FLOAT    MAX SHORTCARD
BYTE    HALT    MIN SHORTINT
CAP HIGH    NEW SIZE
CARDINAL    INC NIL TRUE
CHAR    INCL    NULLPROC    TRUNC
CHR INTEGER ODD VAL

```

Types

A type defines a set of values. Modula-2 contains a number of predefined types, and constructs for defining new types. New types can be given names in type declarations:

- Declaration type { Id '=' TypeDef}

Some types are called simple types:

- TypeDef ::= SimpleType

A type can be just the name of a type:

- SimpleType ::= Name

Note: this syntactical construct is used for naming any type, not just simple ones. If the definition of a type declaration is just the name of a type, the defined type is identical to the named one.

Example:

```
TYPE JustCardinal = CARDINAL;
```

Numeric Types

- CARDINAL Types Modula-2 has three predefined cardinal types, whose values are unsigned whole numbers in the specified ranges:

CARDINAL: 0 to 65535 (0 to $2E16 - 1$) SHORTCARD: 0 to 255 (0 to $2E8 - 1$) LONGCARD: 0 to 4294967295 (0 to $2E32 - 1$)

INTEGER Types Similarly, there are three predefined integer types, whose values are signed whole numbers in the specified ranges:

INTEGER: -32768 to +32767 ($-2E15$ to $2E15 - 1$) SHORTINT: -128 to +127 ($-2E7$ to $2E7 - 1$) LONGINT: -2147483648 to +2147483647 ($-2E31$ to $2E31 - 1$)

Collectively, the integer and cardinal types are called whole number types.

- REAL Types There are two predefined real types, whose values are the real numbers to a certain precision:

REAL: +/- 1.2E-38 to 3.4E+38 6 digits precision LONGREAL: +/- 2.3E-308 to 1.7E+308 15 digits precision

Collectively, the integer, cardinal and real types are called numeric types.

Ordinal Types

- CHAR Type The predefined type CHAR contains 256 values. The first 128 are the characters of the ASCH set, the last 128 are special graphic characters.
- Enumeration Types Modula-2 provides a mechanism for defining enumeration types by giving a complete list of the values in the type. The values, enumeration literals, are represented by identifiers, which are declared by the type definition:

- SimpleType ::= '(' IdList ')'

Example:

```
TYPE
  Color = (Red, Yellow, Green, Brown, Blue, Pink, Black);
  Gender = (Male, Female);
```

Modula-2 has one predefined enumeration type containing truth values:

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

Collectively, the integer, cardinal, character, and enumeration types are called ordinal types; they have whole-number ordinal values. Enumeration literals are numbered consecutively starting from zero. Likewise for the character type. The short ordinal types exclude LONGCARD and LONGINT.

Subrange Types

Given an ordinal type, it is possible to define a subrange type of that base type:

- SimpleType ::= [Name] '[' Expr '..' Expr ']'

The two expressions must be constant and of the same type. They restrict the values of the type by specifying a lower and upper bound (lower <= upper). If the Name is present it names the base type; otherwise, if the expression values are of unspecified whole-number type, the base type is assumed to be INTEGER if the first expression is negative, otherwise CARDINAL.

Examples:

```
TYPE Year = [1900..2001];    (* Base is CARDINAL *)
MyInt = [-1000..+1000];    (* Base is INTEGER *)
Digits = ['0'..'9'];        (* Base is CHAR *)
IntYear = INTEGER [1900..2001]; (* Base is INTEGER *)
LongYear = LONGCARD[101900..102001];
HighColor = [Blue..Black];  (* Base is Color *)
```

Subrange types are themselves ordinal types.

Set Types

Given any short ordinal type, it is possible to define a set type whose values are (unordered) sets of values of that short ordinal type:

- TypeDef ::= set of SimpleType

A set type contains any subset of values of the set element type.

Example:

```
TYPE Chars = SET OF CHAR;  
There is one predefined set type:  
TYPE BITSET = SET OF [0..15];
```

Array Types

Array types provide mappings from a short ordinal index type onto any array element type:

- $\text{TypeDef} ::= \text{array IndexList of TypeDef}$
- $\text{IndexList} ::= \text{SimpleType } \{ ', ' \text{ SimpleType } \}$

An array type definition with more than one index type is equivalent to the expanded type definition:

```
ARRAY Index1 OF ARRAY Index2 ... OF TypeDef
```

All explanations assume a single index type.

A value of an array type contains an ordered collection of values of the element type - one for each value in the index type.

Examples:

```
TYPE  
  Namestring = ARRAY [0..24] OF CHAR; (* A person's name *)  
  IntArray = ARRAY BOOLEAN OF INTEGER;
```

Record Types

Record types provide an aggregation of individual fields:

- $\text{TypeDef} ::= \text{record FieldDefList end}$
- $\text{FieldDefList} ::= \text{FieldDef } \{ \text{Field Def } \}$
- $\text{FieldDef} ::= [\text{Id ListTypeDef }]$

All the identifiers, called field names, must be distinct, but they are local to the record type definition and need not be distinct from other identifiers.

A record value contains one value of the relevant type for each field.

Examples:

```
TYPE Person = RECORD  
    First, Last: Namestring;
```

```

        Age:  SHORTCARD  [0..125];
    END;

    AddrPair = RECORD
        Ofz, Seg:  CARDINAL;
    END;

```

Variant record

Variant record types allow for alternative groups of fields, variants, to be present in a record value. Variant parts can be arbitrarily nested.

- Field Def ::= Variant Part';
- VariantPart ::= case [Id] Name of Variant { '|' Variant } [else FieldDefList] END
- Variant ::= [ChoiceList FieldDefUst]
- ChoiceList ::= Choice { ',' Choice }
- Choice ::= Expr ['..' Expr]

The optional tag identifier after CASE is followed by the name of a short ordinal tag type. If the tag identifier is present, it defines an actual field of that type. The variants and the optional ELSE part each specify a list of field definitions, but only the fields of one of these variants are present in any value of the variant record type.

The choice expressions must be constant and of the tag type, and must not have overlapping values. A choice with two expressions denotes all values in the range. Each choice list should specify the tag values for which the corresponding variant is present; the ELSE-part covers any remaining values.

The presence or absence of variant fields is merely logical; they are always all accessible, but they share storage.

Example:

```

TYPE Location = RECORD
    Value: BYTE;
    CASE Simple: BOOLEAN OF
        | TRUE: ByteOfz: LONGCARD;
        | FALSE: SegOfz: AddrPair;
    END;
END;

```

Pointer Types

The values of a pointer type are access paths to objects of a designated type:

- TypeDef ::= pointer [Expr] to TypeDef

If the expression is omitted, an absolute pointer type is defined; the values of the type are complete 32-bit segment/offset physical addresses.

By including a CARDINAL expression (after POINTER), a based pointer type is defined. Such values only contain the offset part of a physical address. The segment part is obtained by evaluating the expression every time the designated object is accessed. Based pointers are thus short, 16-bit.

Note: Based pointers allow relocated objects to be referenced by just modifying the base (segment) value (leaving the offset unaffected).

Examples:

```
TYPE
  ListPtr = POINTER TO ListNode; (* Note: ListNode not defined yet *)
  ListNode = RECORD
    Value:    CARDINAL; (* element value *)
    NextNode: ListPtr; (* next element *)
  END;
  ShortPtr = POINTER Base TO ListNode;
```

There are two predefined pointer types:

```
TYPE
  ADDRESS = POINTER TO WORD;
  SHORTADDR = POINTER 0 TO WORD; (* Zero base segment *)
```

The values in a pointer type either are obtained as the (absolute) addresses of existing objects of the designated type, or they can be raw storage addresses obtained from a storage manager (see Chapter 8).

Array, record and pointer types are collectively called compound types; the rest, except for set types, are the simple types.

Type Compatibility

Numerous situations require types to be compatible; there are three levels of compatibility, each of decreasing strength.

The most restrictive - and hence, the strongest - is to require types to be identical; that is, they must denote the same type definition.

Next comes compatible. This includes subrange types being compatible with their base types and other subrange types of the same base type. ADDRESS is compatible with any absolute pointer type, and SHORTADDR with any based pointer type.

Finally assignment compatibility also holds between the pairs CARDINAL / INTEGER, SHORTCARD / SHORTINT and LONGCARD / LONGINT.

There are three predefined types called BYTE, WORD and LONGWORD. They correspond to 1, 2 and 4 bytes of memory, respectively, and are assignment compatible with all other types of equal size.

Special compatibility rules apply to formal parameters (see "Calling Procedures," page 118).

Objects and Values

Objects have types and hold values exclusively of that type. There are two kinds of objects: variables and formal parameters (see page 115). Objects of array and record types can contain several component objects.

Variables have to be declared:

- Declaration ::= VAR { VarId { ',' VarId } ":" TypeDef ";" }
- VarId ::= id

Each declaration declares all the identifiers of the list to be variables of the specified type. Their initial values are undefined.

Examples:

```
VAR
  CARDINAL;
  Base: CARDINAL;
  L:    LONGCARD;
  X,Y:  LONGREAL;
  P:    POINTER TO INTEGER;
  S:    ARRAY [1..100] OF CHAR;
  R:    RECORD
        X,Y:    INTEGER;
      END;
  C:    CHAR;
  Z:    Chars;
  Bad : BOOLEAN;
  Loc:  Location;
  Persons: ARRAY BOOLEAN OF POINTER TO Person;
```

A variable can be placed at a fixed physical address, by specifying constant expressions of type CARDINAL for the segment and offset:

- VarId ::= Id '[' Expr ':' Expr ']'

Example:

```
VAR
  ColorScreen [0B800H:0] : ARRAY [1..25] OF
                                ARRAY [1..80] OF
                                    RECORD
                                        Chr: CHAR;
                                        Atr: SHORTCARD;
                                    END;
```

Constants

Constant literals denote the most basic values:

- Value ::= WholeNumber
- Value ::= String

The whole numbers are possible values for the integer and cardinal types, the real numbers for the real types, and strings for the character type (in which case, the string must have length 1) and for types of the form: ARRAY ... OF CHAR.

Constant record and array values are formed with aggregates:

- Value ::= Name '(' Expr ',' Expr { ',' Expr } ')'

The expressions, of which there must be at least two, are constant and give the component values for the named type.

For array types, one value must be given for each value in the index range.

For record types, one value is given for each field present. Values must be specified even for absent tag fields, in order to determine to which variant the following values belong.

Named constants can be declared, introducing identifiers that represent constant values:

- Declaration const { Id '=' Expr }

There is a predefined constant, NIL, compatible with any absolute pointer type.

Neither literals nor named constants are objects.

Examples:

```
CONST
  Pi = 3.14159;
  Ratio = 360.0 / (2.0 * Pi) ;
  K = 1024;
  P = Person("Donald", "Duck", 50) ;
  LastLoc = Location(0, FALSE, AdrPair(0FFFFH, 0FH));
  X = IntArray(-12345, 16*K);
  S = Chars { 'a', ' e', 'i', 'o', 'u'};
```

Set Values

Set values are formed with a set constructor:

- Value ::= [Name] '{' [ChoiceList] '}'

The expressions of the choices are values of the set element type; they need not be constant. The name denotes the set type; omitting it means BITSET.

Example:

```
Chars { 'A'..'2' , 'a'..'z' , '_' }
```

Designators

Designators are used to denote objects. Component objects of a compound object are designated by supplying suffixes to the designator of the compound object. Suffixes are also used to designate components of named aggregate constants.

Using a designator as a value means the value of the designated object:

- Value ::= Designator

The simplest form of designator is just the name of an entity:

- Designator ::= Name

This is also the way to use enumeration literals and named constants as values - just name them.

Indexing is used to designate components of objects of array types:

- Designator ::= Designator '[' Expr { ',' Expr } ']'

A list of index expressions is equivalent to a list of separate indices: X [A,B, C] is equivalent to X[A] [B] [C]. All explanations assume a single index expression.

The value of the expression must have a type assignment compatible with the index type; the designator selects the corresponding component object.

Example:

```
S[I+7]
```

Field selection is used to designate components of objects of record types:

- Designator ::= Designator '.' Id

The identifier is any field identifier of the record type; the resulting designator designates that field of the record object.

Example:

```
R.Y
```

Dereferencing is used to designate the object pointed at by objects of pointer types:

- Designator ::= Designator '^'

If the pointer type is based, this includes evaluation of the base expression.

Example:

```
P^
```

Indexing, field selection and dereferencing can be mixed.

Example:

```
Persons[TRUE]^Last[0] (* First letter of last name *)
```

A pointer constructor is provided to combine CARDINAL segment and offset values into a physical address:

- Designator ::= '[' Expr ':' Expr [Name] ']'

The name specifies the resulting absolute pointer type; omitting it means ADDRESS.

Example:

```
[ListSeg:FirstNode+N ListPtr]^Value]
```

Expressions

Expressions specify the computation of values. Within expressions, operators are used to combine operands, which are themselves expressions. Expressions, as well as values, have types unless all the operands are numeric literals; in that case, determination of types for such expressions is deferred until the context requires a specific type. This is how the same numeric literals (or named constants) can be used for the different numeric types. Until a context is introduced, the only distinction is between whole numbers and real numbers.

When the operands of an operator or predefined function are constant, the result is also constant, and is calculated at compile-time.

Precedence of operators is described in the syntax below; association is left-to-right. Parentheses can be used to enforce any grouping:

- Expr ::= SimpleExpr [RelOp SimpleExpr]
- SimpleExpr ::= [SignOp] Term { AddOp Term }
- Term ::= Factor { MulOp Factor }
- Factor ::= '(' Expr ')'
- Factor ::= NOT Factor
- Factor ::= Value
- RelOp ::= '=' | '#' | '<' | '<=' | '>' | '>=' | IN
- SgnOp ::= '+' | '-'
- AddOp ::= '+' | '-' | OR

- `MulOp ::= '*' | '/' | DIV | MOD | AND | '<<' | '>>'`

The operation indicated by an operator depends both on the operator and on the operand types.

All operators, except `IN`, require operands of compatible types.

The relational operators and `IN` deliver a result of type `BOOLEAN`. The rest deliver a result of the same type as the operand(s).

The operators `'='` and `'#'` are defined for all types, and compare for equality and inequality, respectively.

Operators `'<'`, `'<='`, `'>'` and `'>='` are defined for the ordinal and real types, and compare for relative ordering. `'<='` and `'>='` are defined for set types, and compare for subset and superset.

The sign operator `'+'` is defined for numeric types; it does nothing. The sign operator `'-'` is defined for integer and real types, negating the operand.

The adding operators `'+'` and `'-'` are defined for numeric types and indicate addition and subtraction, respectively. They are also defined for set types and indicate set union and set difference, respectively. Finally, `'+'` is also defined for any combination of constant characters and string literals, concatenating them to produce one string constant. The `OR` operator takes `BOOLEAN` operands and computes the logical sum; the right operand is only evaluated if the left is `FALSE`.

The multiplying operators `DIV` and `MOD` are defined for integer and cardinal types, computing product, quotient (truncated towards zero) and remainder (after division). `'<<'` and `'>>'` are defined for cardinal types, computing logical left and right shift of the left operand by the amount of the right operand. `'*'` and `'/'` are defined for real types, computing (approximate) product and quotient. `'*'` and `'/'` are defined for set types, computing set intersection and symmetric set difference. The `AND` operator is defined for `BOOLEAN` operands, computing the logical product; the right operand is only evaluated if the left is `TRUE`.

The `NOT` operator takes a `BOOLEAN` operand and complements it.

The `IN` operator takes values of set types as right operands and values of the set element type as left operands; it tests whether the element value is in the set value.

Note: Sets are represented as bitmaps with one bit for each possible element value (indicating its presence), so the set operators `'+', '**', '/'` and correspond to bitwise boolean operations `OR`, `AND`, `XOR` and `AND NOT`, respectively.

Examples:

```
(J = 0) OR (I MOD J = I - (I DIV J)*J)  (* Always TRUE *)
X * Y / Ratio
R.X + 1
S[1] IN Z * (Chars{'A'..'F'} + Chars{'0'..'9' })
'Line terminated with cr/lf' + CHR(13) + CHR(10)
```

Statements

Programs achieve their effect by executing (possibly nested) statements.

Statements come in lists and are executed one at a time.

- StmtList ::= [Stmt] { [Stmt] }

The after the last statement is optional.

Assignment Statement

The assignment statement is used to change the value of an object:

- Stmt ::= Designator ':=' Expr

The expression is evaluated, and its value replaces the old value of the designated object. The expression must be assignment compatible with the designated object's type.

String literals are a special case: they can be assigned to any object whose type is ARRAY ... OF CHAR and is long enough to contain the string; if the object is longer than the literal being assigned, a null character (0C) is included after the string value.

Examples:

```
X := 0.0;
I := J + 1;
Z := Z - Chars{' a' . . ' z' } ;
Persons[NOT Bad]^First := "Kurt";
```

IF Statement

The IF statement is used to select a statement list conditionally depending on BOOLEAN expressions, conditions:

- Stmt ::= IF Expr THEN StmtList { ELSIF Expr THEN StmtList } [ELSE [StmUSt] END

At most one of the statement lists is executed, namely the one following the first expression evaluating to TRUE, where ELSE is treated as ELSIF TRUE THEN.

Example:

```
IF I > J THEN
  M := I;
ELSIF I < J THEN
  M := J;
ELSE (* I must be = J *)
  B := TRUE;
  M := I;
END;
```

CASE Statement

The CASE statement selects between alternative statement lists depending on the value of a case expression of short ordinal type:

- Stmt ::= CASE Expr OF Case { '|' Case } [ELSE StmtList] END
- Case ::= [ChoiceList ':' StmtList]

The '|' before the first case is optional.

The types of the choice expressions must be compatible with the case expression type; they must be constant and their values must not overlap.

The statement list executed is the one whose choice list includes the value of the case expression. If none do, and there is an ELSE, that statement list is executed; otherwise, execution continues with the statement following the case statement.

Example:

```
CASE S[I] OF
| '.' : I := 999;
| 'A'..'Z' : L := L + 1;
           U := U + 1;
| 'a'..'z' : L := L + 1;
           X := X + 1;
ELSE
  TheEnd := TRUE;
END;
```

WHILE Statement

The WHILE statement is used to execute a statement list zero or more times depending on the value of a condition:

- Stmt ::= WHILE Expr DO StmtList END

The expression is evaluated before each execution of the statement list; repetition stops as soon as the expression is FALSE.

Example:

```
WHILE (I > 0) AND (I MOD 2=0) DO
  I := I DIV 2;
END;
```

REPEAT Statement

The REPEAT statement is used to execute a statement list one or more times, depending on the value of a condition:

- Stmt ::= REPEAT StmtList UNTIL Expr

The expression is evaluated after each execution of the statement list; repetition stops as soon as the expression is TRUE.

Example:

```
REPEAT
  I := I MOD N + 1;
UNTIL S[I] = ' ';
```

LOOP and EXIT Statements

The LOOP statement is used to execute a statement list repeatedly, with several exit points possible:

- Stmt ::= LOOP StmtList END
- Stmt ::= EXIT

An EXIT statement is only legal inside LOOP statements; it terminates the innermost enclosing LOOP.

Example:

```
LOOP
  IF I = J THEN EXIT; END;
  WHILE I < J DO
    I := (I * J) MOD N;
    IF I = 0 THEN
      I := J;
      EXIT; (* exit LOOP, not just WHILE *)
    END;
  END;
  I := I DIV 2;
END; (* The LOOP has now been EXITed *)
```

FOR Statement

The FOR statement is used to execute a statement list a precalculated number of times, with an ordinal control variable taking on a progressing series of values:

- Stmt ::= FOR Id Expr TO Expr [by Expr] DO StmtList END

The first two expressions are evaluated to obtain the start and stop values; they must have types compatible with the ordinal type of the identifier. The expression after BY, the step value, must be a constant whole number; omitting it means +1.

The control variable takes on ordinal values beginning with the start value and spaced by the step value. The FOR statement terminates when the next value would exceed the stop value. If the start

value exceeds the stop value, the statement list is not executed at all. The direction of progression is determined by the sign of the step value.

The value of the control variable should not be changed inside the statement list, and it is undefined after the FOR statement.

Example:

```
FOR Ball := Black TO Yellow BY -2 DO
  LastBall := Ball; (* Takes on: Black, Blue, Green *)
END;
```

WITH Statement

The WITH statement is used to create a local scope wherein the field names of a record type can be used directly to denote the fields of a particular designated record object.

- Stmt ::= WITH Designator DO StmtList END

Example:

```
WITH Persons[TRUE]^ DO
  IF Age < 4 THEN
    First := "baby";
  ELSIF Age < 15 THEN
    First := "junior";
  END;
END;
```

GOTO Statement

The GOTO statement is used to alter the flow of execution explicitly:

- Stmt ::= GOTO Id

The target of the jump is indicated by the label identifier, which must be located somewhere in the same body (see "Bodies" page 116):

- Stmt ::= Id ':' [Stmt]

Labels must be declared in the body in which they are used:

- Declaration ::= LABEL IdList

Procedures

Procedures are used to group commonly performed operations into isolated blocks. Proper procedures are used like statements, whereas functions compute values and are used in expressions.

Procedures:

- are declared like other entities, and are subsequently invoked by calls
- can have parameters which are objects or values supplied at the call
- finish execution by returning

Procedures can also be handled without being called; they can be valid values of a procedure type and can be manipulated as such.

The characteristics of a procedure are specified in a procedure heading:

- $\text{ProcHead} ::= \text{PROCEDURE Id [FormalList [':' Name]] ';'}$
- $\text{FormalList} ::= '(' [\text{FormalSection} \{ ';' \text{FormalSection} \}] ')'$
- $\text{FormalSection} ::= [\text{VAR}] \text{IdList} ':' \text{FormalType}$
- $\text{FormalType} ::= [\text{ARRAY OF}] \text{Name}$

A procedure heading declares the identifier denoting the procedure. It has a (possibly empty) list of formal parameters declared similarly to variables. The absence of the and the name in the procedure heading indicates a proper procedure; their presence indicates a function, in which case the name states the return type (which can be a structured type).

Each formal section does the following:

- declares a list of formal parameters
- states their formal type
- indicates whether they are variable, or VAR parameters or value parameters by the presence/absence of the keyword,VAR

All the parameter identifiers in a formal list must be distinct.

Examples:

```
PROCEDURE NewLine;
PROCEDURE PrintNumber ( N: INTEGER; Width: SHORTCARD );
PROCEDURE Accumulate ( VAR X: LONGREAL; Delta: LONGREAL );
PROCEDURE HypSquare ( A,B: LONGREAL ): LONGREAL;
PROCEDURE PrintMessage ( M: ARRAY OF CHAR );
PROCEDURE GetChar (): CHAR;
```

Bodies

A procedure heading is combined with a body specifying the internal workings of the procedure; a FORWARD declaration is used to delay specifying the body:

- $\text{Declaration} ::= \text{ProcHead Body Id ';'}$
- $\text{Declaration} ::= \text{ProcHead FORWARD ';'}$
- $\text{Body} ::= \text{DeclList [BEGIN StmtList] END}$

The identifier repeats the procedure name.

A FORWARD declared procedure must be completed later in the same declaration list by a full procedure declaration repeating the procedure heading and supplying a body.

The declaration list declares entities that are local to the procedure; they must have names distinct from the formal parameters.

Formal parameters of type ARRAY OF Type, open array parameters, are considered to be local arrays indexed with CARDINALS starting from 0; the upper index bound is obtained by the HIGH function.

Local variables come into existence when procedures are called and vanish when they return. Procedures can call themselves directly or indirectly, causing several incarnations of local variables to be in existence simultaneously. Designating any local variable refers to the instance in the most recent active invocation of that procedure.

A formal value parameter is considered an ordinary local variable whose value is initialized when the procedure is called. Formal VAR parameters denote actual objects, which are identified by the caller when the procedure is called.

Open arrays can be used as actual parameters to other open array formal parameters; otherwise they can only be manipulated element-wise.

The statement list specifies the actions of the procedure.

Executing a RETURN statement is the only legal way of leaving a function. A proper procedure can also return by just reaching the end of the statement list.

- Stmt ::= RETURN [Expr]

The expression must be present for functions exclusively, and the expression's type must be assignment compatible with the return type; it is the value returned to the caller.

Examples:

```
PROCEDURE Max ( X,Y: LONGREAL ): LONGREAL;  
  
BEGIN  
  IF X > Y THEN  
    RETURN X;  
  ELSE  
    RETURN Y;  
  END;  
END Max;  
  
PROCEDURE HypSquare ( A,B: LONGREAL ): LONGREAL; FORWARD;  
  
PROCEDURE Accumulate ( VAR X: LONGREAL; Y: LONGREAL );  
  VAR  
    Z: LONGREAL;  
  
BEGIN  
  Z := HypSquare( Y , 10.0-Y );
```

```

    X := X + Max(Z * Z , -999.99);
END Accumulate;

PROCEDURE HypSquare ( A,B: LONGREAL ): LONGREAL;
BEGIN
    RETURN A*A + B*B;
END HypSquare;

```

Calling Procedures

Proper procedures are invoked in call statements:

- Stmt ::= Designator [ActualList]
- ActualList ::= '(' [Expr { ',' Expr }] ')'

The actual parameter list must supply one value or designator for each corresponding formal parameter. For a VAR parameter the expression must be a designator of an object with type identical to the formal type; for value parameters any expression of assignment compatible type is valid. String literals are valid parameters for any value parameter of type ARRAY ... OF CHAR.

A formal type ARRAY OF Type is considered identical to any array type with that element type; the index range of the actual parameter is mapped onto the CARDINALS starting from 0. An expression of the element type is also valid (and is treated as an array with one element).

The formal types BYTE, WORD and LONGWORD are compatible with any type of identical size. The formal types ARRAY OF BYTE, ARRAY OF WORD and ARRAY OF LONGWORD are compatible with anything, allowing the procedure to treat the actual parameter as unstructured storage.

Examples:

```

NewLine;
PrintMessage("Don't panic");
Accumulate( X , 7.0 * Y );

```

Functions are invoked in expressions:

- Value ::= Designator ActualList

The parameter rules are as for proper procedures.

Examples:

```

X := HypSquare( 10.0 , Y );
C := GetChar ();

```

Some procedures can alternatively be called using infix notation:

- `Infix ::= " Designator "`

The designated procedure must take two parameters.

Functions are applied like operators of the lowest possible expression precedence:

- `Expr' ::= Expr { Infix Expr }`

Proper procedures are called similarly to assignment statements:

- `Stmt ::= Expr Infix Expr'`

Examples:

```
X \Accuraulate\ 1.0 + (5.0 \HypSquare\ Y-1.0); (* infix *)
Accumulate( X , 1.0 + HypSquare( 5.0 , Y-1.0 ) ); (* same *)
```

Procedure Types

A procedure type denotes a family of procedures with identical calling characteristics:

- `TypeDef ::= PROCEDURE ['( [ FormalTypeLst ] ')'] [':' Name ]]`
- `FormalTypeLst ::= [ VAR ] FormalType { [ VAR ] FormalType }`

The procedures belonging to the type are those with matching procedure headings: each parameter must be of identical type for VAR parameters, and of identical or assignment compatible type for value parameters; return types must be identical for functions. Predefined procedures and procedures nested within other procedures are excluded.

Example:

```
TYPE
  PutProc = PROCEDURE ( ARRAY OF CHAR );
VAR
  G: ARRAY BOOLEAN OF PROCEDURE () : CHAR;
  P: PutProc;
```

There is one predefined procedure type:

```
TYPE PROC = PROCEDURE; (* Procedures without parameters *)
```

One predefined procedure value, `NULLPROC`, is compatible with any procedure type. Calling it causes a run-time error.

Procedure values are denoted by designators without parameter lists.

Examples:

```

P := PrintMessage; (* P now denotes PrintMessage *)
P("Hi, there"); (* Call it *)
G[TRUE] := GetChar; (* G[TRUE] now denotes GetChar *)
C := G[TRUE] ();    (* Call it *)

```

Predefined Procedures

Modula-2 contains a number of predefined procedures. Some are generic in the sense that they are valid for several parameter types and can take one or two parameters.

Predefined Function Procedures The predefined function procedures are:

ABS (X)	Absolute value of numeric operands.
ADR(X)	The physical ADDRESS of object X.
CAP (C)	Character C, changing *a'..'z' to *A'..'Z'.
CHR(X)	The CHAR with ordinal value X.
FLOAT(C)	REAL value of CARDINAL C.
HIGH (A)	The upper index bound of open array A.
MAX (T)	Maximum value of ordinal/real type T.
MIN(T)	Minimum value of ordinal/real type T.
ODD(X)	TRUE if ordinal value X is not even.
ORD (X)	CARDINAL, short ordinal value of X.
SIZE(T)	Size in bytes of type or object T.
TRUNC(R)	CARDINAL, truncated value of real R.
VAL(T,X)	The value X converted to type T.
VSIZE(R.F)	Size of record type R if it contained just the fields up to and including field F. See the following example:

Example: To illustrate how VSIZE works, consider the following RECORD definition:

```

TYPE
  VSTest = RECORD
    A : CARDINAL;
    B : INTEGER;
    C : CARDINAL;
    D : LONGCARD;
  END;

```

With this RECORD definition, a call to VSIZE (VsTest.C) would return the size of the record including only the first three fields - A, B, and C.

VAL can convert values between any two numeric or ordinal types.

Any type name can be used as a type transfer function, taking one value parameter of any type. The result is that value interpreted as a value of the named type. The actual bit pattern of the value remains unchanged,

except if the value type and transfer type are numeric or ordinal, when a proper VAL conversion is performed.

A type transfer is considered well-behaved if the sizes of the value type and the transfer type are equal, or a proper VAL conversion is performed. Such type transfers can be used freely in expressions.

If the value type size is larger than the transfer type size, the remaining last bytes of the value are ignored. If it is shorter, the last bytes of the resulting value are undefined.

Such type transfers should only be used to circumvent the type requirements of assignment and parameter passing.

Examples:

```
I := CARDINAL( BITSET(I) * BITSET(J) ); (* bitwise AND *)
L := LONGCARD( SegOfz );                (* record -> cardinal *)
SegOfz := AddrPair( L );                (* cardinal -> record *)
P := ADDRESS ( L );                     (* Not address of L!!! *)
```

Predefined Proper Procedures The predefined proper procedures are:

DEC(X)	Decrement ordinal object X.
DISPOSE! X)	When the compiler encounters the DISPOSE procedure, This is replaced by a call to DEALLOCATE (X, SIZE(XA)).
DEC(X,N)	Decrement ordinal object X by amount N.
EXCL(S,E)	Exclude element E from set object S.
HALT	Terminate program execution successfully.
INC(X)	Increment ordinal object X.
INC(X,N)	Increment ordinal object X with amount N.
INCL(S,E)	Include element E in set object S.
NEW (X)	This is replaced by a call to ALLOCATE (X, SIZE(XA)) when the compiler encounters the NEW procedure.

Modules

Modules encapsulate related declarations. An executing program consists of a main module and a number of server modules. The server modules are partitioned into a definition part, which is visible to clients, and an implementation part, which hides the internal details from clients.

Modules provide a general facility for implementing features not supported explicitly within the language. Such features include: input and output, string handling, storage management, concurrency, operating system access, etc. (see Chapter 8).

Modules are the basic units of compilation:

- Compilation ::= DefModule '.'
- Compilation ::= [implementation] Module '.'
- Module ::= MODULE Id [Priority]';' {Import} [Export] Body Id

- `DefModule ::= DEFINITION MODULE Id ';' { Import } DclList END Id`
- `Priority ::= '[' Expr ']`

Each identifier names the module defined.

Modules optionally have a CARDINAL priority used with the SYSTEM module (see Chapter 8).

A compilation module without IMPLEMENTATION is a main module.

Server Modules

The definition part of a module declares the entities available to clients. Only CONST, TYPE and VAR declarations are legal, in addition to the following two, which are legal only in definition parts:

- `Declaration ::= ProcHead`
- `Declaration ::= TYPE { Id [ '=' TypeDef ] ';' }`

The first form declares the existence of a procedure. The second form, when the type definition is omitted, declares the existence of an opaque pointer type with unknown designated type; the only allowed operations on objects of opaque type are assignment and test for equality. Corresponding full declarations for objects of the opaque type must appear in the implementation part of the module.

Objects declared in compilation modules are called global and exist throughout the execution of the program (as opposed to variables local to procedures).

Example:

```
DEFINITION MODULE Str; (* A bit of string handling *)

CONST
  MaxWidth = 20;

TYPE
  Buffer = ARRAY [1..MaxWidth] OF CHAR;

VAR
  Width: [1..MaxWidth];

PROCEDURE Put ( C: CARDINAL ): Buffer;
PROCEDURE SetFill( C: CHAR );

END Str.
```

The implementation part contains declarations private to the module. These declarations are not accessible to client modules. The optional list of statements, initialization code, is executed before any statements of clients are executed. If this is impossible to satisfy because of circularity, the initialization order is undefined within the circle. The TopSpeed Modula-2 TechKit contains a program for determining the module dependencies in a program, and for detecting circular dependencies.

The declarations in the implementation part form a single scope together with those in the definition part. Consequently every name from the definition part is visible, and additional names declared must be distinct from those. Reimporting identifiers is allowed.

Example:

```
IMPLEMENTATION MODULE Str;

VAR
  FillChar: CHAR;

PROCEDURE Put ( C: CARDINAL ): Buffer;
VAR
  P: [0..MaxWidth];
  S: Buffer;
BEGIN
  P := Width;
  REPEAT
    S[P] := CHR( ORD('0') + C MOD 10 );
    C := C DIV 10;
    DEC( P );
  UNTIL (C = 0) OR (P = 0);
  WHILE P > 0 DO
    S[P] := FillChar;
    DEC( P );
  END;
  RETURN S;
END Put;

PROCEDURE SetFill ( C: CHAR );
BEGIN
  FillChar := C;
END SetFill;

BEGIN
  FillChar := ' ';
END Str.
```

Importing

Clients gain access to a server module by importing from it:

- Import ::= [FROM Id] IMPORT IdLst

The FROM form takes one module and a list of identifiers naming entities in it. Those names are considered declarations of those identifiers and thus achieve direct visibility of the named entities. Importing an enumeration type also imports the enumeration literals.

The FROM-less form imports each of the modules named. Qualified names are used to denote the entities in those modules:

- Name ::= Name '.' Id

The name denotes a module; the second component - the Id - an entity therein. The same qualified notation can be used to name a module's own entities, but only within the implementation part.

Example:

```
FROM Str IMPORT Put,Width;
IMPORT Str;

VAR
  S: Str.Buffer;
...
Str.SetFill(' ');
Width := Str.MaxWidth DIV 2;
S := Put( 1+7 );
```

Local Modules

In addition to their use as compilation units, local modules can be declared;

- Declaration ::= Module

Local modules obey special scope rules. Any required entity (except the predefined ones) from outside the local module must be imported, and must be visible immediately outside the local module. Thus FROM-less import can name any entity (not only server modules). Using the FROM form requires the named server module to be visible (and thus already imported) immediately outside the local module.

Exportation is valid only in local modules:

- Export ::= EXPORT [QUALIFIED] IdList

The listed identifiers are declared in the enclosing scope to make those entities directly available.

Qualification can be used to access any entity.

If the export list contains QUALIFIED, then the export statement has no effect.

The optional statement lists of local modules are executed (in the sequence they appear) before the statement list of the enclosing body.

This concludes the TopSpeed Modula-2 language definition.

Sources for Language Examples

K. N. King's TopSpeed Modula-2 Language Tutorial, included with your TopSpeed Modula-2 package, contains a more detailed discussion of the language, as well as numerous examples. The discussion of library procedures in Chapter 8 provides additional exposure to the language's constructs and how to use them. The

case studies in Chapter 4 also show examples of how certain Modula-2 types and constructs are used. You can refer to the source code for the library modules to see how various Modula-2 constructs are used, and also to see how modules are constructed.

In Chapter 7, you will find out about options you can use to check that your program conforms to the run-time semantics of the language.

Chapter 7

The Compiler

The TopSpeed Modula-2 compiler has a one-pass architecture, meaning that it outputs the generated code simultaneously with reading the source text; no temporary files are used.

Importing is handled by (re)compiling module definition files (DEF-files) whenever required. This eliminates the need for special 'symbol-table' files, and means that module definitions need not be compiled explicitly; it also means that such definitions affect subsequent compilations as soon as they are modified. This strategy also eliminates the compilation-order restrictions that usually apply to module definitions; module implementations can be compiled in any order.

The output from the compiler is a relocatable object file (OBJ-file) in Intel/Microsoft format. A collection of such files is combined into one executable file (EXE-file) by a linker.

The OBJ-file

The OBJ-file created by the compiler contains special information that makes linking easier, checks consistency, and minimizes EXE-file size.

Include-records are inserted into the OBJ-files. These records tell the linker the names of the files used by the module. This means that only the name of the main module needs be supplied when linking. The linker will find the remaining files through the include-records.

Version-records are also inserted. These record the date and time of the DEF-files used. The linker will check that all modules agree on these versions. If they don't, there is the possibility of a fatal inconsistency which should be eliminated by recompiling (see the Make option in "Making a Program" page 69).

The OBJ-file is built as a library, meaning that the module is split into sections which are only included by the linker if they are used. This eliminates the overhead for large general-purpose library modules. Thus, your EXE-file is as small as possible. In the OBJ-file, each global procedure is in a section by itself, and so are groups of data-declarations belonging to the same VAR or CONST part.

Data Representation

The basic storage unit is an 8-bit byte. Any Modula-2 object is represented in a number of bytes determined by the object's type. Subrange types are represented as their base types.

Taking the ADDRESS of an object yields the address of the first byte of its storage. For multi-byte numeric types, the least significant byte of the object is stored at the lowest address.

Cardinal types are represented as unsigned binary numbers, and have the following storage requirements:

```
SHORTCARD:  1  byte
CARDINAL:   2  bytes

LONGCARD:   4  bytes
```

Integer types are represented as 2's-complement binary numbers, and require the following storage:

```
SHORTINT:   1  byte
INTEGER:    2  bytes
LONGINT:    4  bytes
```

Real types are represented as 8087 mantissa/exponent pairs. (See the relevant 8087 documentation for your machine.) These types require the following amounts of storage.

```
REAL:      4 bytes
LONGREAL:  8 bytes
```

Enumeration types are represented as unsigned binary (ordinal) numbers:

```
<= 256 values:  1 byte
> 256 values:   2 bytes
BOOLEAN:        1 byte
CHAR:           1 byte
```

Absolute pointer types occupy 4 bytes. The first two bytes hold an offset value and the last two bytes hold a segment value. NIL is represented as (0,0). Based pointers occupy 2 bytes, which hold an offset.

FAR procedure variables are represented as absolute pointers to the first instruction of the procedure's code. NEAR procedure variables are represented as 16-bit offsets within the applicable code segment.

Sets are represented as packed bit-maps with one bit for each potential member in the element type, say [0 . . N]. The number of bytes required to represent a set is given by $1 + (N \text{ DIV } 8)$. For example, if you have a set which can have up to 100 elements, you need 13 bytes - that is, $(1 + 100 \text{ DIV } 8)$ -to represent this set. The set element E, where $(0 \leq E \leq N)$ is represented by bit number $E \text{ MOD } 8$ in byte number $E \text{ DIV } 8$. The formula assumes you start counting with byte 0 and with bit 0.

Elements of arrays are stored in consecutive memory locations, and ordered according to increasing indices. The total array size is the size of a single element multiplied by the number of elements.

Fields of records are likewise stored in consecutive memory locations, and are ordered as they appear in the type declaration. Each field is stored according to its own type. The total record size is the sum of sizes of the individual fields. Variant record parts are special: the fields of different alternatives are overlayed (share storage). The total size of a variant part is the sum of the field sizes of the biggest alternative.

Example:

```

TYPE R = RECORD
    F1:  INTEGER;      (* Bytes 0-1 *)
    CASE F2: BOOLEAN OF  (* Byte 2 *)
    | FALSE:  F3, (* Byte 3 *)
              F4, (* Byte 4 *)
              F5: CHAR;  (* Byte 5 *)
    | TRUE:  F6: CARDINAL; (* Bytes 3-4 *)
    END
    F7:  SET OF [0..20]; (* Bytes 6-8 *)
END;    (* SIZE ( R ) = 9,  VSIZE( R.F6 ) = 5 *)

```

This record requires nine bytes:

- two bytes (0-1) for field F1
- one byte (2) for field F2
- three bytes (3-5) for fields F3, F4, and F5 together
- three bytes (6-8) for field F7

Note that field F6 is subsumed by the larger storage allocated for the fields when F2 is FALSE.

Calling Conventions

This discussion of the calling conventions used in TopSpeed Modula-2 code files assumes familiarity with the 8086-architecture. We will refer to the registers by the following names: AX(, AL), BX, CX, DX, SI, DI, BP, SP, CS, SS, DS, ES, IP, and Flags. For most programming tasks, you won't need to be concerned with these conventions, since the default conventions will suffice.

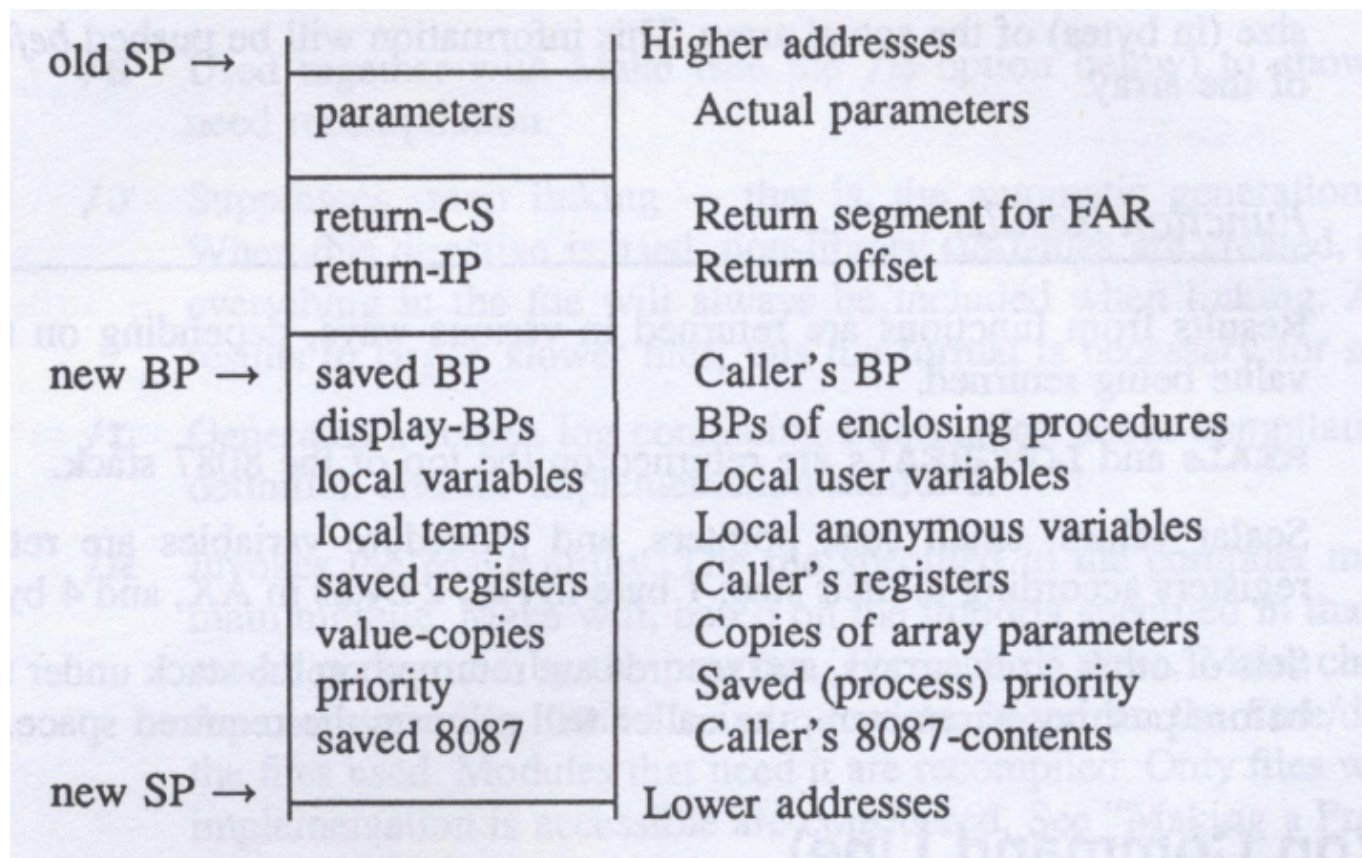
The Stack Frame

Procedure activation makes use of the hardware stack (defined by registers SS and SP) for several purposes:

- Passing parameters.
- Saving return addresses.
- Saving registers which are used and must be preserved.
- Saving 8087 contents if not enough 8087 stack space is left for local floating point computations.
- Building 'displays' for accessing variables local to surrounding procedures.
- Storing local variables (whether user or compiler generated).
- Storing copies of value open array parameters, so they can be modified without modifying the actual parameters.
- Saving the process priority of the caller.

The parameters are pushed onto the stack by the calling procedure; the rest of the information on the stack is built by the called procedure on entry. The BP register points to the base of the stack-frame, and is used to access the local variables and parameters.

The general picture is:



Only the parts required are actually built; for a very simple procedure, saving BP will be sufficient.

When returning, a procedure will restore the stack and preserved registers (see the `$C` directive) to their state before the call. This restoration includes popping the space used by parameters.

Parameter Passing

Parameters are pushed onto the stack in the order they appear in the procedure declaration. Precisely what is pushed depends on the corresponding formal parameter's type and whether it is a VAR parameter or not.

All VAR parameters and any open array parameters are passed by reference. That is, the full segment-offset address for the parameter or array is pushed, and the procedure uses this address to access the storage for the actual parameter. Such a parameter takes up 4 bytes, regardless of the parameter's base type.

Value parameters, except open arrays, are passed by pushing the actual value of the parameter onto the stack. The space taken up is determined by the parameter's type, except that any 1-byte type occupies 2 bytes when pushed.

For open array value parameters, the procedure will optionally make a copy of the passed value into local storage, and will modify the address passed to point at the copy rather than at the original array. (See the `$V-` directive in "Directives (in source text)" page 133.)

For open array parameters (whether VAR or not), the caller also will push the (2-byte) size (in bytes) of the actual array. This information will be pushed before the address of the array.

Function Results

Results from functions are returned in various ways, depending on the type of the value being returned.

REALS and LONGREALs are returned on the top of the 8087 stack.

Scalar values, small sets, pointers, and procedure variables are returned in 8086 registers according to their size: 1 byte in AL, 2 bytes in AX, and 4 bytes in DX:AX.

Sets of other sizes, arrays, and records are returned on the stack under the parameters: before pushing parameters, the caller will allocate the required space.

Options (on Command Line)

When invoking the compiler, you can use options to specify what you want from the compiler. You can invoke any of these options on the DOS command line; you can also specify some of them from menus when compiling in the TopSpeed Modula-2 environment. To specify an option from the appropriate menu, select the Options Compiler menu, then set entries to the desired values. See Chapter 5 for more details about the TopSpeed Modula-2 options.

When you invoke the options from the command line, you must precede the name of the file you want to compile with /C, to tell the system that you want to invoke the compiler on the file that follows as the next command line argument. (Command line options are not case sensitive. Thus, /C and /c are equivalent.)

Example:

```
M2 /C MyMain /ML
```

The compiler options are:

- /B This option lets you specify whether compiler directives that do checking at run-time are ON or OFF. This option affects only the default settings for these options. You can override these settings with directives in the source code, as described in the next section.
- /D If selected, the compiler generates a debug information file for the module being compiled. This file is required by the TopSpeed Modula-2 source level debugger.
- /F Allows file names to be different from module names.
- /H Used together with Make (see the /M option below) to show which files need recompilation.
- /J Suppresses smart linking - that is, the automatic generation of libraries. When this directive is used, non-library OBJ files are created, meaning that everything in the file will always be included when linking. Although this results in larger, slower files, this file format is necessary for some linkers.
- /L Generates a screen log containing information about compilation speed for definition and for implementation modules.
- /M Invokes the Make utility. The file specified to the compiler must contain a mam module. Make will, based on the imports specified in that file, find all modules required by the program. For each of these, Make checks whether the corresponding OBJ-file is up-to-date, based on the time/date values of the files used. Modules that need it are recompiled. Only files whose module implementation is accessible are considered. See "Making a Program" page 69, for more information about Make.

- /N Include line numbers in the OBJ-file. This enables a program such as a debugger to determine the correspondence between code addresses and source program lines.
- /O Where possible, the compiler will try to produce the fastest code it can. To do this, the compiler will sometimes reorganize expressions, in order to speed up their evaluation. The /O option disables reorganization of expressions. This ensures that expressions are evaluated left-to-right, but results in suboptimal code.
- /P Tells the compiler to display the name of each procedure as it is compiled, to indicate progress.
- /R Used together with Make to recompile everything, regardless of checks. This option may be necessary because the Make facility only considers the actual time/dates values of files; it does not read OBJ-files to find the version-records.
- /V Makes all variables volatile: instead of trying to keep variables in registers, they are kept in memory. This can help when debugging. This option results in slower code than if registers are used.

Directives (in Source Text)

Directives, which appear in the source text, are used to specify various details about how code should be generated. Directives are specified in special comments, whose first character is \$ after the comment opening characters, (*. Each directive is indicated with a single letter, optionally followed by a parameter; several directives can be given in a single comment by separating them with commas.

Most of the directives are flags. In such cases, the parameter is a + or - to indicate whether to enable the feature or not. The flags can be reset to their value before the most recent use of the directive by specifying the flag with = as the parameter.

Example:

- (\$V-,M mycode,N) (* set \$V feature to regardless of current value) (code here *)
- (\$V=,F) (* restore \$V feature to value it had before \$V- *)

Other directives require a number or a name as their parameters - for example,

- M mycode

above. (Notice that the \$ appears only once in the comment line.) Finally, a few directives are set simply by including the directive, without a parameter.

Directives take effect from the spot in which they first occur, and remain in effect until the end of the source file or until revoked by another directive. The current setting for a directive is sampled at the place where its value is relevant.

If certain directives are specified in a module definition, their settings override any settings in the module implementation. These directives are: C,G,F,K,N.

The following list summarizes the compiler directives that are available in TopSpeed Modula-2. Where applicable, default values are specified in boldface.

- \$A+/- Enable/disable aliased behavior on global variables. The compiler will assume that variables declared with the directive disabled cannot be accessed directly and indirectly at the same time.

- `$B+/-` Include/exclude a control-break handler in the program. If included, pressing `[ctrl][Break]` will terminate the program. The handler can be dynamically enabled and disabled with library procedures. The directive must be in the main module.
- `$C h` Specifies which registers will be preserved by procedures. The hexadecimal number, `h`, specifies the registers, based on the following values:

`AX=1, CX=2, DX=4, BX=8, DS=10, ES=20, SI=40, DI=80.`

The BP register is always preserved. The default configuration is

`FO = DS+ES+SI+DI`

This would be specified as:

`($C FO)`

- `$D n` Specifies the name, `n`, of the data segment in which to put the global variables declared by the module. The default is to use the module name. In any case, the name specified is prefixed by `D_`. If used, this directive must appear before the `MODULE` keyword in both definition and implementation files.
- `$E+/-` Enable/disable relaxed alias treatment of variant records. Because of overlapping, the compiler will normally consider fields of variant records volatile, causing them to be kept in memory.
- `$F` Procedures will be called with FAR calls. Likewise, procedure types are 32-bit. This is the default, and applies only to global procedures; local procedures are always called with NEAR calls.
- `$G+/-` Enable/disable module prefixes in external names. Enabling such prefixes guarantees that external names will be unique.
- `$H+/-` Enable/disable treating constant aggregates as variables, thereby allowing them to be modified. This is possible because the values are in memory anyway.
- `$I+/-` Enable/disable index checking. If enabled, accessing a non-existent array element will produce a run-time error. All variables are assumed to hold legal values corresponding to their type (see `$R` below). Likewise, type transfer on indices is assumed not to cause problems.
- `$J+/-` Enable/disable interrupt procedures, by generating IRET returns instead of the usual RET instruction.
- `$K+/-` Enable/disable the C language calling convention for procedures. This specifies that caller (not called) pops parameters. Across calls, the DS register is set to the group named 'DGROUP.'
- `$M n` Specifies the name, `n`, of the code segment in which to put code. The default is to use the module name. In any case, the name is prefixed by `C_`. When used, this directive must appear before the `MODULE` keyword in both definition and implementation files.
- `$N` When this directive is used, procedures will be called with NEAR calls. This requires callers to be in the same code segment. With this directive, procedure types are 16-bit.
- `$O+/-` Enable/disable overflow checking on whole number operations. If this directive is enabled, a numeric overflow will cause a run-time error.

- `$P+/-` Enable/disable generating external names for local procedures. Enabling eases debugging but can cause name clashes.
- `$Q+/-` Enable/disable procedure tracing. If enabled, procedures will execute an `INT 60H` instruction on entry and `INT 61H` on exit. Handlers for these interrupts must be installed explicitly (see "Interrupt Handlers" page 140). Library module `ProcTrace` enables you to do this.
- `$R+/-` Enable/disable subrange checking. If enabled, a run-time error is generated by assignment or parameter passing if the value is outside the bounds of the receiver's type.
- `$S h` The hex number, `h`, specifies the amount of stack allocated for the program. If used, this directive must be in the main module.
- `$S+/-` Enable/disable stack overflow checking. If this directive is enabled, a run-time error is generated if stack space is exhausted.
- - `$V+/-` Enable/disable copying of open array value parameters. Disabling such copying increases efficiency but is potentially incorrect.
- `$W+/-` Enable/disable the use of volatile variables. Volatile variables are not kept in registers across statements. The ability to use volatile variables can be essential if concurrent processes communicate via shared global variables, and can make debugging easier. The `/V` command line option selects this globally.
- `$X+/-` Enable/disable 8087 stack spilling for procedures. Spilling is necessary if nested function calls exhaust the 8087 stack. Spilling involves saving excess values from floating point computations on the hardware stack - because there is no longer room on the 8087 stack.
- `$Y+/-` Enable/disable coinciding variant fields in a record. If enabled, it is legal to use the same name for fields in distinct alternatives, provided that these fields have the same type and are at the same offset in the record.
- `$Z+/-` Enable/disable checks for dereferencing of NIL pointers, which then generate a run-time error. When the feature is enabled, all local variables are initialized to zero. If the feature is enabled in the main module, all global variables are also zeroed.