A Single-pass Modula-2 Compiler for Lilith

N. Wirth 1.5.84 / rev. 15.11.85

A new Modula-2 compiler for Lilith has been developed by J. Gutknecht and myself. It is based on the single-pass principle. This results in a significant gain in speed compared with earlier multi-pass compilers; it is about four times faster and recompiles itself on Lilith in about 80 seconds. The format of symbol and reference files differs from those generated by the previous compiler. A new debugger, designed by W. Heiz, is therefore necessary.

The compiler reads a source file (default extension MOD) and generates a symbol file (SBL) in the case of a definition module, or an object file (OBJ) and a reference file (RFC) otherwise. Output files automatically inherit the name of the compiled module, and not the name of the source file. Errors are reported on a log file called err.LST. The log shows short pieces of text preceding each error's point of detection, and a number characterizing the error. Brief explanations of the numbers are found on the file ErrList.DOK. If a compilation fails, the reference or symbol file is deleted, and no code is generated. The compiler then requests the name of a next source file; it terminates, if ESC is typed instead of a name. (Appending "/r" to a name causes the compiler to suppress range checking instructions in index computations and subrange assignments).

The reference file is used by the debugger only. The debugger merges information from the reference files and the source files. (A separate listing file is neither generated by the compiler nor needed by the debugger).

The compiler observes the revisions of the language made in 1983 which are summarized by the following paragraphs 1 to 3.

1. Restrictions and clarifications to the language

1.1. The type of a formal VAR-parameter and that of its corresponding actual parameter must be identical (and not merely compatible). This rule is relaxed in the cases of formal parameters of type ADDRESS, which are compatible with all pointer types, and of a formal parameters of type WORD, for which the compatible types are INTEGER, CARDINAL, BITSET, pointers and enumerations (for Lilith).

1.2. The types of the expressions specifying the starting and limiting values of the control variable in a for statement must be compatible (not merely assignment compatible) with the type of the control variable.

1.3. A process initiated in a module with priority level n must not call a procedure declared in a module with priority level m < n. Calls of procedures without declared priority are permitted.

1.4. Pointer types can be exported from definition modules as opaque types. Opaque export is also allowed for subrange types. Assignment and test for equality are permitted for all opaque types.

1.5. All modules imported to the main module are initialized before the importing module is initialized. If there exist circular references, the order of initialization is not defined.

1.6. Import of a module identifier does not imply that the identifiers local to this module become visible. However, those which are exported (from a definition module or in qualified mode from a local module) can be referenced by prefixing them with the module identifier.

2. Changes

2.1. Definition modules do not contain export lists. Instead, they are themselves regarded as export lists, and all identifiers defined are exported.

```
    DefinitionModule = DEFINITION MODULE ident ";" {import} {definition} END ident "." .
```

2.2. Variant record declarations without an explicit tag field now require a colon preceding the tag type.

```
    FieldList = [IdentList ":" type | CASE [ident] ":" qualident OF variant {"|" variant}
            [ELSE FieldListSequence] END ].
```

2.3. The type PROCESS and the procedures TRANSFER and NEWPROCESS are not required in module SYSTEM.

3. Extensions

3.1. Case statements and variant record declarations allow for the empty case.

```
    case = [CaseLabelList ":" StatementSequence].
    variant = [CaseLabelList ":" FieldListSequence].
```

3.2. Strings of length 1 are compatible with the type CHAR.
3.3. Subrange type definitions allow for an optional type identifier to specify the subrange's base type. Example: INTEGER [0 .. 99]

```
SubrangeType = [ident] "[" ConstExpression ".." ConstExpression "]".
```

3.4. Elements of sets can be specified by expressions instead of constants only. This compiler, however, requires constants, if the element is a range. For example, {k}, {i+j}, {i, j} are accepted, but not {i .. j}.

```
set = [qualident] "{" [element {"," element}] "}".
element = expression | ConstExpression ".." ConstExpression].
```

3.5. The character "~" is a synonym for NOT.
3.6. The new standard functions MIN(T) and MAX(T) take as argument any scalar type T (including REAL) and yield the minimal and maximal values of that type.

4. Further restrictions imposed by the compiler

The primary restriction imposed on the language by the new compiler is inherent in its single-pass structure: objects must be declared textually preceding their use. The exception to this rule are base types of pointer types. They may be declared following the pointer declaration within the same scope. (See also forward declarations below).

4.1. The index type of arrays must be a subrange type.
4.2. The least number of type INTEGER is -32767 (not -32768).
4.3. Code procedures cannot be exported from definition modules, and neither can objects declared within local modules.
4.4. The standard procedures NEW and DISPOSE are not available.
4.5. The standard function SIZE(t) yields the amount of storage (in words) needed by the variable t or by variables of type t. The function TSIZE is identical to SIZE, but is defined in module SYSTEM. Neither SIZE nor TSIZE accept parameters indicating variants of records.

5. Extensions

5.1. A forward procedure declaration serves to allow forward references which are in general prohibited by the new compiler. Forward declarations are needed in the case of mutual recursion only. Example:

```
PROCEDURE Q; FORWARD;

PROCEDURE P;
BEGIN ... Q ... END P;

PROCEDURE Q;
BEGIN ... P ... END Q;

BEGIN ... P ... Q ... END
```

In the forward declaration, the procedure body is substituted by the symbol (new reserved word) FORWARD. The heading is repeated in the later, full declaration. Forward declarations are accepted for global procedures (level 0) only.

5.2. There exists a new type LONGINT, supporting arithmetic with 32-bit integers to a limited extent. The available operators are assignment, addition, subtraction, and comparison. The second comparand must not be negative. (Use constants only!) Constants are decimal numbers (followed by the character D) in the range -2147483648 .. 2147483647. The function

```
PROCEDURE LONG(a, b: CARDINAL): LONGINT
```

imported from module SYSTEM yields the value $a*2^{16} + b$, if $a < 2^{15}$.

6. Integer Arithmetic

We define in Modula-2 two kinds of integer arithmetic: Euler's integer arithmetic and modulo arithmetic. The differences concern division with negative arguments only.

1. Euler's arithmetic: we define $q = m/n$ and $r = m$ REM n satisfying the relations

$$q*n + r = m \quad \text{and} \quad 0 <= \mathrm{ABS}(r) < \mathrm{ABS}(n)$$

The sign of the remainder r is that of the dividend m (or $r = 0$). Euler's division is symmetric relative to 0:

$$(-m)/n = m/(-n) = -(m/n)$$

2. Modulo arithmetic: we define $Q = m$ DIV n and $R = m$ MOD n satisfying the relations

$$Q*n + R = n \text{ and } 0 <= R < n$$

For Lilith, negative arguments of DIV and MOD are prohibited, and therefore MOD and REM are synonyms, as are DIV and / ; REM is a new reserved word.

File: Modula.Rev.DOK