Texts and Monographs in Computer Science

PROGRAMMING IN MODULA-2

Niklaus Wirth

Second, Corrected Edition

F-27

PROGRAMMING IN MODULA-2

Niklaus Wirth

Second, Corrected Edition

Springer-Verlag
Berlin Heidelberg New York 1983

Professor Dr. Niklaus Wirth
Institut für Informatik, ETH, CH-8092 Zürich

Professor David Gries
Department of Computer Science, Cornell University
Ithaca, NY 14853, USA

The 1st edition was published as a monograph,
ISBN 3-540-11674-5
Springer-Verlag Berlin Heidelberg New York 1982

ISBN 3-540-12206-0 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-12206-0 Springer-Verlag New York Heidelberg Berlin

Library of Congress Cataloging in Publication Data
Wirth, Niklaus.
Programming in Modula-2.
Bibliography: p.
Includes index.
1. Modula-2 (Computer program language) I. Title.
QA76.73.M63W5713 1983 001.64'24 83-434
ISBN 0-387-12206-0 (New York)

Printing: Beltz, Hemsbach. Bookbinding: Schäffer, Grünstadt
2145/3140-543210

Contents

Preface

This text is an introduction to programming in general, and a manual for programming with the language Modula-2 in particular. It is oriented primarily towards people who have already acquired some basic knowledge of programming and would like to deepen their understanding in a more structured way. Nevertheless, an introductory chapter is included for the benefit of the beginner, displaying in a concise form some of the fundamental concepts of computers and their programming. The text is therefore also suitable as a self-contained tutorial. The notation used is Modula-2, which lends itself well for a structured approach and leads the student to a working style that has generally become known under the title of *structured programming.*

As a manual for programming in Modula-2, the text covers practically all facilities of that language. Part 1 covers the basic notions of the variable, expression, assignment, conditional and repetitive statement, and array data structure. Together with Part 2 which introduces the important concept of the procedure or subroutine, it contains essentially the material commonly discussed in introductory programming courses. Part 3 concerns data types and structures and constitutes the essence of an advanced course on programming. Part 4 introduces the notion of the module, a concept that is fundamental to the design of larger programmed systems and to programming as team work. The most commonly used utility programs for input and output are presented as examples of modules. And finally, Part 5 covers facilities for system programming, device handling, and multiprogramming. Practical hints on how and when to use particular facilities are included and are intended as guidelines for acquiring a sound style of programming and system structuring.

The language Modula-2 is a descendant of its direct ancestors Pascal [1] and Modula [2]. Whereas Pascal had been designed as a general purpose language and after implementation in 1970 has gained wide usage, Modula had emerged from experiments in multiprogramming and therefore concentrated on relevant aspects pertinent to that field of application. It had been defined and implemented experimentally by 1975.

In 1977, a research project with the goal to design a computer system (hardware and software) in an integrated approach, was launched at the Institut für Informatik of ETH Zürich. This system (later to be called Lilith) was to be programmed in a single high-level language, which therefore had to satisfy requirements of high-level system design as well as those of low-level programming of parts that closely interact with the given hardware. Modula-2 emerged from careful design deliberations as a language that includes all aspects of Pascal and extends them with the important module concept and those of multiprogramming. Since its syntax was more in line with that of Modula than with Pascal's, the chosen name was Modula-2. We shall subsequently use *Modula* as synonym for Modula-2.

The language's main additions with regard to Pascal are:

1. The *module* concept, and in particular the facility to split a module into a *definition part* and an *implementation part.*

2. A more systematic syntax which facilitates the learning process. In particular, every structure starting with a keyword also ends with a keyword, i.e. is properly bracketed.

3. The concept of the *process* as the key to multiprogramming facilities.

4. So-called *low-level facilities* which make it possible to breach the rigid type consistency rules and allow to map data with Modula-2 structure onto a store without inherent structure.

5. The *procedure type* which allows procedures to be dynamically assigned to variables.

A first implementation of Modula-2 became operational on the PDP-11 computer in 1979, and the language's definition was published as a Technical Report in March 1980. Since then the language has been in daily use in our institute. After a year's use and testing in applications, the compiler was released for outside users in March 1981. Interest in this compiler has grown rapidly, because it incorporates a powerful system design tool implemented on widely installed minicomputers. This interest had given rise to the need for this manual and tutorial. The defining report is included at the end of the manual, primarily for reference purposes. It has been left unchanged, with the exception that the chapters on standard utility modules and on the use of the compiler have been omitted.

This text has been produced in camera-ready form by a Lilith minicomputer connected to a Canon LBP-10 laser printer. Concurrently with the writing of the book, the author designed the programs necessary for automatic text formatting (and controlling the printer) and designed the interface connecting the printer. Naturally, all these programs have been written in Modula (for Lilith).

It is impossible to properly acknowledge all the influences that contributed to the writing of this text or the design of Modula. However, I am particularly grateful for the inspiring influence of a sabbatical year (1976) at the research laboratory of Xerox Corporation (PARC), and for the ideas concerning modules presented by the language Mesa [3]. Perhaps the most important insight gained was the feasibility of implementing a high-level language effectively on minicomputers. My thanks are also due to the implementors of Modula, notably L. Geissmann, A. Gorrengourt, Ch. Jacobi and S.E. Knudsen, who not only have turned Modula into an effective, reliable tool, but have often wisely consulted against the inclusion of further fancy facilities.

Zürich, February 1982 N. W.

References

1. N.Wirth. The programming language PASCAL.
Acta Informatica 1, 35-63 (1971).

2. N.Wirth. Modula: a language for modular multiprogramming.
Software - Practice and Experience, 7, 3-35 (1977).

3. J.G.Mitchell, W. Maybury, R.Sweet. Mesa Language Manual.
Xerox PARC, CSL-78-1, (1978).

1. Introduction

Although this manual assumes that its reader is already familiar with the basic notions of computer and programming, it may be appropriate to start out with the explanation of some concepts and their terminology. We recognize that - with rare exceptions - programs are written - more appropriately: designed - with the purpose of being interpreted by a computer. The computer then performs a process, i.e. a sequence of actions, according to the specifications given by that program. The process is also called a computation.

The program itself is a *text*. Since it specifies a usually fairly complex process, and must do so with utmost precision and care for all details, the meaning of this text must be specified very precisely. Such precision requires an exact formalism. This formalism has become known as a *language*. We adopt this name, although a language is normally spoken and much less precisely defined. Our purpose here is to learn the formalism or language called **Modula-2** (henceforth simply called Modula).

A program usually specifies a process that causes its interpreter, i.e. the computer, to read data (the so-called *input*) from some sources and to vary its subsequent actions according to the accepted data. This implies that a program does not only specify a (single) process, but an entire - usually unbounded - class of computations. We have to ensure that these processes act according to the given specifications (or should we say expectations) in all cases of this class. Whereas we could verify that this specification is met in the case of a single computation, this is impossible in the general case, because the class of all permitted processes is much too large. The conscientious programmer ensures the correctness of his program by careful design and analysis. Careful design is the essence of professional programming.

The task of designing a program is further complicated by the fact that the program not only must describe an entire class of computations, but often should also be interpreted (executed) by different interpreters (computers). At earlier times, this required the manual transcription of the program from its source form into different computer codes, taking into account their various characteristics and limitations. The difficulties have been drastically reduced, albeit not eliminated, by the creation of high level languages with formal definitions and the construction of automatic translators converting the program into the codes of the various computers.

In principle, the formal language should be defined in an abstract, perhaps axiomatic fashion without reference to an actual computer or interpretation mechanism. If this were achieved, the programmer would have to understand the formal language only. However, such generality is costly and often restrictive, and in many cases the programmer should still know the principal characteristics of his computer(s). Nevertheless, the qualified programmer will make as little reference to specific computer characteristics as possible and rely exclusively on the rules of the formal language in order to keep his program general and portable. The language Modula assists in this task by confining computer dependencies to specific objects, used in so-called *low-level programming* only.

From the foregoing it follows that a translation process lies between the program's formulation and its interpretation. This process is called a *compilation*, because it condenses the program's source text into a cryptic computer code. The quality of this compilation may be crucial to the efficiency of the program's ultimate interpretation. We stress the fact that there may be many compilers for a given language (even for the same computer). Some may be more efficient than others. We recognize that efficiency is a characteristic of implementations rather than the language. It therefore is important to distinguish between the concepts of language and implementation.

We summarize:

A program is a piece of *text*.

The program specifies *computations* or processes.

A process is performed by an interpreter, usually a *computer*, interpreting (executing) the program.

The meaning of the program is specified by a formalism called *programming language*.

A program specifies a *class of computations*, the input data acting as parameter of each individual process.

Prior to its execution, a program text is translated into computer code by a *compiler*. This process is called a *compilation*.

Program design includes ensuring that all members of this class of computations act according to specification. This is done by careful *analytic verification* and by selective *empirical testing* of characteristic cases.

Programs should refrain from making reference to characteristics of specific interpreters (computers) whenever possible. Only the lack of such reference ensures that their meaning can be derived from rules of the language.

A *compiler* is a program translating programs from their source form to specific computer codes. Programs need to be compiled before they are executed. Programming in the wider sense not only includes the formulation of the program, but also the concrete preparation of the text, its compilation, correction of errors, so-called *debugging*, and the planning of tests. The modern programmer uses many tools for these tasks, including text editors, compilers, and debuggers. He also has to be familiar with the environment of these components. We shall not describe these aspects, but concentrate on the *language* Modula.

2. A first example

Let us follow the steps of development of a simple program and thereby explain some of the fundamental concepts of programming and of the basic facilities of Modula. The task shall be, given two natural numbers x and y, to compute their *greatest common divisor* (gcd).

The mathematical knowledge needed for this problem is the following:

1. if x equals y, x (or y) is the desired result
2. the gcd of two numbers remains unchanged, if we replace the larger number by the difference of the numbers, i.e. subtract the smaller number form the larger one.

Expressed in mathematical terms, these rules take the form

1. $\gcd(x,x) = x$
2. if $x > y$, $\gcd(x,y) = \gcd(x-y,y)$

The basic recipe, the so-called *algorithm*, is then the following: Change the numbers x and y according to rule 2 such that their difference decreases. Repeat this until they are equal. Rule 2 guarantees that the changes are such that gcd(x,y) always remains the same, and rule 1 guarantees that we finally find the result.

Now we must put these recommendations into terms of Modula. A first attempt leads to the following sketch. Note that the symbol # means "unequal".

```
WHILE x # y DO
  "apply rule 2, reducing the difference"
END
```

The sentence within quotes is plain English. The second version refines the first version by replacing the English by formal terms:

```
WHILE x # y DO
  IF x > y THEN
    x := x-y
  ELSE
    y := y-x
  END
END
```

This piece of text is not yet a complete program, but it shows already the essential characteristic of a *structured* programming language. Version 1 is a *statement*, and this statement contains another, subordinate statement (within quotes). In version 2 this is elaborated, and yet further subordinate statements emerge (expressing the replacement of a value x by another value x-y). This hierarchy of statements expresses the underlying structure of the algorithm. It becomes explicit due to the structure of the language, allowing the nesting of components of a program. It is therefore important to know the language's structure (syntax) in full detail. Textually we express nesting or subordination by appropriate indentation. Although this is not required by the rules of the language, it helps in the understanding of a text very considerably.

Reflecting an algorithm's inherent structure by the textual structure of the program is a key idea of structured programming. It is virtually impossible to recognise the meaning of a program when its structure is removed, such as done by a compiler when producing computer code. And we should keep in mind that a program is worthless, unless it exists in some form in which a human can understand it and gain confidence in its design.

We now proceed towards the goal of producing a complete program from the above fragment. We realize that we need to specify an action that assigns initial values to the variables x and y, as well as an action that makes the result visible. For this purpose we should actually know about a computer's facilities to communicate with its user. Since we do not wish to refer to a specific machinery, and particularly not in such a frequent and important case as the generation of output, we introduce *abstractions* of such communication facilites, postulating that they will be available - and realized in some appropriate fashion - on all computers where Modula programming is said to be possible. These abstractions take the form of standard statements, as shown below. The input of data is called a *Read* operation, their output as a *Write* operation. We may, for example, assume that data are read from a keyboard and written on a display.

```
ReadCard(x);
ReadCard(y);
WHILE x # y DO
  IF x > y THEN x := x-y
       ELSE y := y-x
  END
END;
WriteCard(x,6)
```

The procedure *ReadCard* reads a cardinal (a whole, non-negative number) and assigns it to its parameter (x). The procedure *WriteCard* outputs a cardinal as specified by its first parameter (x). The second parameter (6) indicates the number of digits available for the representation of this value on the output medium.

In the next and final version we complete our text such that it becomes a genuine Modula program.

```
MODULE gcd;
  FROM InOut IMPORT ReadCard, WriteString, WriteLn, WriteCard;

  VAR x,y: CARDINAL;
BEGIN
  WriteString("x = "); ReadCard(x); WriteLn;
  WriteString("y = "); ReadCard(y); WriteLn;
  WHILE x # y DO
    IF x > y THEN x := x-y
    ELSE y := y-x
    END
  END;
  WriteString("gcd = "); WriteCard(x,6); WriteLn;
END gcd.
```

The essential additions in this step are so-called *declarations*. In Modula, all names of objects occuring in a program, such as variables and constants, have to be declared. A declaration introduces the object's *identifier* (name), specifies the kind of the object (whether it is a variable, a constant, or something else) and indicates general, invariant properties, such as the type of a variable or the value of a constant.

The entire program is called a *module*, given a name (gcd), and has the following format:

```
MODULE name;
  <import lists>
  <declarations>
BEGIN
  <statements>
END name.
```

A few more comments concerning our example are in order. The procedures *WriteLn*, *WriteString*, *ReadCard*, and *WriteCard* are not part of the language Modula itself. They are defined in another module called *InOut* which is presumed to be available. A collection of such useful modules will be listed and explained in later parts of this text. Here we merely point out that they need to be imported in order to be known in a program. This is done by including the names of the needed objects in an import list and by specifying from which module they are requested.

The procedure *WriteString* outputs a *string*, i.e. a sequence of characters (enclosed in quotes). This output makes the computer user aware that an input is subsequently requested, an essential feature of conversational systems. The procedure *WriteLn* terminates a line in the output text.

And this concludes the discussion of our first example. It has been kept quite informal. This is admissible because the goal was to explain an existing program. However, programming is designing, creating new programs. For this purpose, only a precise, formal description of our tool is adequate. In the next chapter, we introduce a formalism for the precise description of correct, "legal" program texts. This formalism makes it possible to determine in a rigorous manner whether a written text meets the language's rules.

3. A notation to describe the syntax of Modula

A formal language is an infinite set of sequences of *symbols*. The members of this set are called sentences, and in the case of a programming language these sentences are *programs*. The symbols are taken from a finite set called the *vocabulary*. Since the set of programs is infinite, it cannot be enumerated, but is instead defined by rules for their composition. Sequences of symbols that are composed according to these rules are said to be syntactically correct programs; the set of rules is the *syntax* of the language.

Programs in a formal language then correspond to grammatically correct sentences of spoken languages. Every sentence has a structure and consists of distinct parts, such as subject, object, and predicate. Similarly, a program consists of parts, called syntactic entities, such as statements, expressions, or declarations. If a construct A consists of B followed by C, i.e. the concatenation BC, then we call B and C syntactic *factors* and describe A by the syntactic formula

```
        A = BC.
```

If, on the other hand, an A consists of a B or, alternatively, of a C, we call B and C syntactic *terms* and express A as

```
        A = B|C.
```

Parentheses may be used to group terms and factors. It is noteworthy that here A, B, and C denote syntactic entities of the formal language to be described, whereas the symbols =, |, parentheses, and the period are symbols of the meta-notation describing syntax. The latter are called *meta-symbols*, and the meta-notation introduced here is called ***Extended Backus Naur-Formalism*** (EBNF).

In addition to concatenation and choice, EBNF also allows to express option and repetition. If a construct A may be either a B or nothing (empty), this is expressed as

```
        A = [B].
```

and if an A consists of the concatenation of any number of Bs (including none), this is denoted by

```
        A = {B}.
```

This is all there is to EBNF! A few examples show how sets of sentences are defined by EBNF formulas:

```
        (A|B)(C|D)      AC AD BC BD
        A[B]C           ABC AC
        A{BA}           A ABA ABABA ABABABA ...
        {A|B}C          C AC BC AAC ABC BBC BAC ...
```

Evidently, EBNF is itself a formal language. If it suits its purpose, it must at least be able to describe itself! In the following definition of EBNF in EBNF, we use the following names for entities:

statement:	a syntactic equation
expression:	a list of alternative terms
term:	a concatenation of factors
factor:	a single syntactic entity or a parenthesized expression

The formal definition of EBNF is now given as follows:

```
syntax       = {statement}.
statement    = identifier "=" expression ".".
expression   = term {"|" term}.
term         = factor {factor}.
factor       = identifier | string | "(" expression ")" |
                 "[" expression "]" | "{" expression "}".
```

Identifiers denote syntactic entities; strings are sequences of symbols taken from the defined language's vocabulary. For the denotation of identifiers we adopt the widely used conventions for programming languages, namely:

An identifier consists of a sequence of letters and digits, where the first character must be a letter. A string constists of any sequence of characters enclosed by quote marks (or apostrophes).

A formal statement of these rules in terms of EBNF is given in the subsequent chapter.

4. Representation of Modula programs

The preceding chapter has introduced a formalism, by which the structures of well-formed programs will subsequently be defined. It defines, however, merely the way in which programs are composed as sequences of symbols, in contrast to sequences of characters. This "shortcoming" is quite intentional: the representation of symbols (and thereby programs) in terms of characters is considered too much dependent on individual implementations for the general level of abstraction appropriate for a language definition. The creation of an intermediate level of representation by symbol sequences provides a useful decoupling between language and ultimate program representation. The latter depends on the available character set. As a consequence, we need to postulate a set of rules governing the representation of symbols as character sequences. The symbols of the Modula vocabulary are divided into the following classes:

identifiers, numbers, strings, operators and delimiters, and comments.

The rules governing their representation in terms of the standard ISO character set are the following:

1. *Identifiers* are sequences of letters and digits. The first character must be a letter.

```
$  identifier = letter {letter|digit}.
```

Examples of well-formed identifiers are

```
Alice   likely   jump   BlackBird   SR71
```

Examples of words which are no identifiers are

sound proof	(blank space is not allowed)
sound-proof	(neither is a hyphen)
2N	(first character must be a letter)
Miller's	(no apostrophe allowed)

Capital and lower-case letters are considered as distinct.

Sometimes an identifier has to be qualified by another identifier; this is expressed by prefixing i with j and a period (j.i); the combined identifier is called a ***qualified identifier*** (abbreviated as ***qualident***). Its syntax is

```
$  qualident = {identifier "."} identifier.
```

2. *Numbers* are either integers or real numbers. The former are denoted by sequences of digits. Numbers must not include any spaces. Real numbers contain a decimal point and a fractional part. In addition, a scale factor may be appended. It is specified by the letter E and an integer which is possibly preceded by a sign. The E is pronounced as "times 10 to the power of".

Examples of well-formed numbers are

```
1981   1   3.25   5.1E3   4.0E-10
```

Examples of character sequences that are not recognized as numbers are

1,5	no comma may appear
1'000'000	neither may apostrophs
3.5En	no letters allowed (except the E)

The exact rules for forming numbers are given by the following syntax:

```
$  number = integer | real.
$  integer = digit {digit}.
$  real = digit {digit} "." {digit} [ScaleFactor].
$  ScaleFactor = "E" ["+"|"-"] digit {digit}.
```

Note: Integers are taken as octal numbers, if followed by the letter B, or as hexadecimal numbers if followed by the letter H.

3. *Strings* are sequences of any characters enclosed in quote marks. In order that the closing quote is recognized unambiguously, the string itself evidently cannot contain a quote mark. To allow strings with quote marks, a string may be enclosed within apostrophes instead of quote marks. In this case, however, the string must not contain apostrophes.

```
$  string = '"' {character} '"' | "'" {character} "'".
```

Examples of strings are

```
"no comment"
"Buck's Corner"
'he said "do not fret", and fired a shot'
```

4. *Operators and delimiters* are either special characters or *reserved words*. These latter are written in capital letters and must not be used as identifiers. Hence is it advantageous to memorize this short list of words.

The operators and delimiters composed of special characters are

+	addition, set union
-	subtraction, set difference
*	multiplication, set intersection
/	division, symmetric set difference
:=	assignment
&	logical AND
=	equal
# <>	unequal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
( )	parentheses
[ ]	index brackets
{ }	set braces
(* *)	comment brackets
↑	dereferencing operator
, . ; : .. |	punctuation symbols

The reserved words are enumerated in the following list; their meaning will be explained throughout the subsequent chapters:

AND	ELSIF	LOOP	REPEAT
ARRAY	END	MOD	RETURN
BEGIN	EXIT	MODULE	SET
BY	EXPORT	NOT	THEN
CASE	FOR	OF	TO
CONST	FROM	OR	TYPE
DEFINITION	IF	POINTER	UNTIL
DIV	IMPLEMENTATION	PROCEDURE	VAR
DO	IMPORT	QUALIFIED	WHILE
ELSE	IN	RECORD	WITH

It is customary to separate consecutive symbols by a space , i.e. one or several blanks. However, this is mandatory only in those cases where the lack of such a space would merge the two symbols into one. For example, in "IF x = y THEN" spaces are necessary in front of x and after y, but could be omitted around the equal sign.

5. *Comments* may be inserted between any two symbols. They are arbitrary sequences of characters enclosed in the comment brackets (* and *). Comments are skipped by compilers and serve as additional information to the human reader. They may also serve to signal instructions (options) to the compiler.