Seminar Report

Modula-2

Daniel Moosbrugger
daniel.moosbrugger@student.uibk.ac.at

22 February 2013

Supervisor: Assoc. Prof. Dr. Georg Moser

Abstract

In this short report the programming language Modula-2 will be presented by discussing its history and present, its creator Niklaus Wirth, some of its data types, its syntax, the concepts of separate compilation, Definition and Implementation modules, and Local modules. For a better understanding some code examples and pictures will be used.

Contents

1 Introduction

The **Modula-2** programming language was a very promising new language when it first came out. It introduced some new concepts that are unthinkable not to be included in any of todays more modern programming languages, e.g. separate compilation and higher order functions.

This report wants to serve as an introduction to this now over 30 year old programming language, without making any claim of being a complete introduction in any way, shape or form.

I will start the report off by talking about Niklaus Wirth, the creator of not only Modula-2, but many other, more or less well known, programming languages. Next, I will discuss the history of Modula-2, by talking about its predecessor, Pascal, and the circumstances around the inception of Modula-2, namely improving Pascal and the need for a new language while working on the Lilith computer. After that Modula-2's data types and syntax will be discussed, as well as the modules that were the namesake of Modula-2.

On the last few pages the present and possible future of Modula-2 will be talked about, especially the ways in which Modula-2 influenced languages that came after it, by briefly discussing concepts like Modula-2's modularity, separate compilation, and abstract data types.

2 Niklaus Wirth

Niklaus Wirth, the creator of Modula-2, was born in Winterthur, Switzerland, in 1934.

He studied at the Swiss Federal Institute of Technology Zürich (Eidenössische Technische Hochschule Zürich), also known as the ETH Zürich, where he received the degree of Electronics Engineering in 1959, the Université Laval in Canada, where he got his M.Sc. in 1960, and the University of Berkley in California, where he was awarded his Ph.D. in 1963. From 1963 to 1967 he was Assistant Professor of Computer Science at Stanford University and later at the University of Zürich.

After that, in 1968, Wirth became Professor for Informatics at the ETH Zürich where he kept teaching and researching for over 30 years (with two sabbaticals at the Xerox PARC in California) up to his retirement in 1999 [1].

Figure 1: Niklaus Wirth, 1969

Besides being known for creating many programming languages, Wirth also coined the phrase: *Software is getting slower more rapidly than hardware becomes faster*, now known as *Wirth's Law.*

Wirth was the chief designer of programming languages such as:

- Euler,
- Algol W,
- Pascal,
- Modula,
- **Modula-2**,
- Oberon,
- Oberon-2,
- Oberon-07.

The development of those languages (especially Pascal) gained him the *Turing Award* in 1984. As of today, Wirth still is the only German speaking winner of this prestigious award.

Later in 1988 he was awarded the *IEEE Computer Pioneer Award*, and in 2007 he became a member of the Academia Europaea Society. As of today, Niklaus Wirth was awarded with 10 honorary doctorates.

A more complete list of Wirth's honours, books and articles can be found at [1].

3 From Pascal to Modula

3.1 The History of Modula-2

In the early 1970s, a programming language previously thought to only be useful for students for an easy start to programming began to be used more and more in the professional realm as well. This language was *Pascal*, published by Niklaus Wirth in 1970. It was one of the first languages to make use of data types, and, even though it did not easily allow for very large programs, was heavily used in the following years – even today many students still learn it in schools around the world.

But of course Pascal had its problems as well, especially the aforementioned difficulties a programmer had to face when planning large programs. Because of downsides such as this, new languages with the intent to replace Pascal were developed. One of those languages was *Ada*, developed by the United States Department of Defense (DoD). At the same time, in the mid- to late 1970s, Wirth himself began to work on a successor of Pascal, also seeing its shortcomings: he created *Modula*. Modula is an acronym, standing for **Modu**lar Programming **La**nguage. Unfortunately, Modula was never actually released outside of Wirth's institute in the ETH Zürich, unlike it's successor – **Modula-2**.

Modula-2 was developed mainly between 1977 and 1980 and, even though it is a spiritual successor of Wirth's earlier work, i.e. Pascal, the name Modula-2 was chosen over, say, Pascal-2, as its syntax was primarily based on Modula.

The very first implementation of Modula-2 was in 1979, and was once again only used in Wirth's institute up until 1981, when the first compiler was released to the public.

Wirth tried to make Modula-2 as practically useful as possible, and by essentially merely expanding the already very popular Pascal, intended it to be an easy to comprehend and clean learning language for the wide public. And many universities, especially in the 1980s and 1990s, did use it as a first language for students to learn.[1]

Modula-2 is of course, just like its predecessor Pascal, Turing complete.

3.2 The Lilith

From 1978 to 1980, Modula-2 was used by Wirth and his team as the programming language for the custom built **Lilith** workstation – this was actually one of the main motivators for developing Modula-2 in the first place: it's operating system, *Medos-2*[2], was created only using Modula-2. The Lilith was a high resolution computer inspired by the Xerox Alto Wirth worked with during his two sabbatical years at the Palo Alto Research Institute [3].

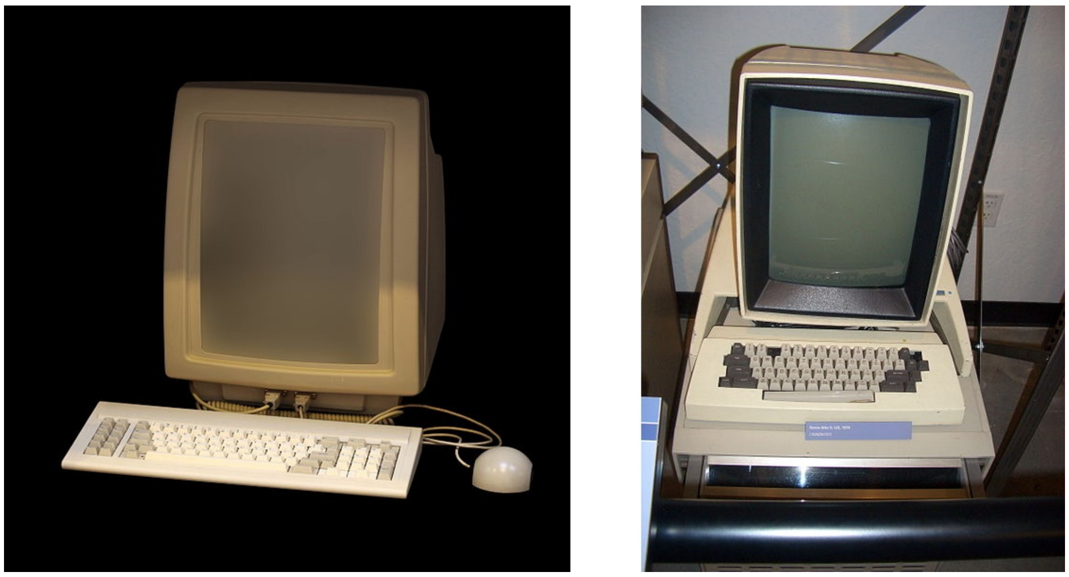

Figure 2: Lilith (left) and Xerox Alto (right)

[1] After some researching, I was not able to find any universities where Modula-2 was still being actively taught as of today.

[2] More on Lilith's operating system Medos-2 can be found at [2], also available at `http://e-collection.library.ethz.ch/eserv/eth:21975/eth-21975-01.pdf`.

Lilith was one of the first computers to have a bitmap display, utilize a mouse (as seen in Figure 2) and to have the now classic window based UI – just like the Xerox Alto machine Wirth wanted to emulate and was unable to obtain for the ETH Zürich, with the Xerox Alto "not being available on the market" [3]. Commercially, the Liltih was no success, not unlike the Xerox Alto, even though Wirth tried to market it properly.

3.3 Pascal vs. Modula-2: The Main Differences

Modula-2 is, as mentioned before, largely based on Pascal. But as its successor, there are of course some major differences between the two, or rather *improvements* Modula-2 had over Pascal, Wirth himself counts at least five [4, p 3-4]:

1. Modula-2 is, as its name already reveals, *modular*, unlike Pascal.
2. Modula-2's syntax is more systematic than it was in Pascal – for example almost every statement is closed by the keyword `END`.
3. The concept of processes was established, allowing multiprogramming facilities.
4. Low-level facilities made it possible to breach rigid type consistency rules and allowed to map data with Modula-2 structure onto a store without inherent structure – a very important feature for the development of the Lilith.
5. And lastly, Modula-2 had a procedure type, allowing processes to be dynamically assigned to variables.

4 Modula-2's Data Types

Modula-2 has 6 elementary data types: integers, cardinal numbers, real numbers, a Boolean type, characters, and bitsets.[3]

4.1 Elementary Data Types

4.1.1 `INTEGER`

This type represents whole numbers. Operators applicable to these integers include: addition (+), subtraction (-), multiplication (*), division (**DIV**), and the modulus operation (**MOD**).

4.1.2 `CARDINAL`

Just like integers, cardinal numbers represent whole numbers, but with the difference that cardinal numbers must be non-negative (this includes 0). Therefore, the operators for integers cardinal numbers are the same.

Mixed expressions, i.e. the use of an operator on an `INTEGER` and a `CARDINAL` in the same expression, are *not* allowed in Modula-2.

[3]Most examples are taken from [4, p 28ff].

4.1.3 `REAL`

Real, or floating point, numbers can utilize the same basic operators as the `INTEGER` and `CARDINAL` types. Because integers and real numbers are represented differently internally, a strict distinction has to be made. One can however convert from `INTEGER` to `REAL`, and vice versa with the `FLOAT` and `TRUNC` functions respectively.

4.1.4 `BOOLEAN`

Boolean values, or truth values, are evaluated either as `TRUE` or `FALSE` and their operators are `AND`, `OR` and `NOT`, and the relations are: ”`<>`”, ”`#`”, ”`<=`”, ”`<`”, ”`=`”, ”`>`”, ”`>=`”, and ”`IN`”, whereas both ”`<>`” and ”`#`” are synonyms for $\neq$.

4.1.5 `CHAR`

Modula-2 uses the standard ASCII table for its characters. This allows for functions such as `ORD(ch)`, that returns the ordinal number of a character `ch`. It would, for example return the number 65 if the value of `ch` was a capital ’A’. Strings are of course nothing other than sequences of characters enclosed in quote marks – both double and single quote marks are applicable, although the opening and closing marks have to either be both double or single quotation marks: `"A String"` is just fine, whereas `"Not a String'` would lead to an error.

4.1.6 `BITSET`

`BITSET`s are probably the most unusual data type in Modula-2, as they are missing in, for example, both C or Java, although they can of course easily be simulated. They are nothing else than sets of integers with a range between 0 and N-1, whereas the N depends on the machine Modula-2 is running on, i.e. the wordlength. Legal examples for a `BITSET` would be:

```
1 {1, 2, 3, 4, 5}
2 {2, 0,12313, 9}
3 {1}
4 {1 .. 12}
```

The notation {`n ..  m`} as seen above in line 4 is used as a shortcut for writing {`n, n+1, n+2, ..., m-2, m-1, m`} – so line 4 would be evaluated to {`1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12`}. Operators for the `BITSET` type include: set union (+), set difference (`-`), set intersection (`*`), and the symmetric set difference (/).

4.2 Constants and Variables

Besides the various types of variables shown above, Modula-2 also supports the use of constants. These constants are declared similarly to variables, with the exception of not declaring them as a certain type, e.g. `INTEGER` or `BITSET`.

Here's a quick example on how to declare various constants (line 1), right above the declaration of some variables (line 2):

```
1 CONST N = 16; EOL = 36C; M = N-1; empty = {};
2 VAR   i,j,k: CARDINAL; x,y,z: REAL; ch: CHAR;
```

As one can see, the declaration of variables and constants looks similar, insofar that the constant's value is taken by the variable's type, which can in this case be seen as the variable's *constant property* [4, p 36].

4.3 Arrays

Modula-2 also supports the data structure array. It allows to store data (of the same type) into one single structure, which is of course very useful if those variables, for example, will all be treated the exact same way henceforth. They are declared and used as follows:

```
CONST N = 100;
VAR a: ARRAY [0 .. N-1] OF INTEGER
    (* stores 100 integer values into the array a - note
     * that no other data types, e.g. real numbers may
     * be stored in this array after its declaration as
     * an integer array. *)

i := 0;
REPEAT a[i] := i; i := i+1;
UNTIL i = N;
```

In the code example above, `a` is the *array*, `a[i]` is called the *designator*, and `i` the *selector*.

4.4 Procedure Types

Lastly[4], there is the procedure type. Procedures can not only be regarded as program parts: in Modula-2, they may also be seen as objects that can be assigned to variables, and therefore even be parameters for other procedures, i.e. *higher order functions*. They are declared as follows[5]:

```
MyFunction = PROCEDURE(INTEGER) : REAL
```

[4]At least for this report; there would be much more to talk about – the interested reader is advised to take a look at [4] or [5] for a more complete introduction to Modula-2.

[5]This function takes one integer as a parameter and returns one real number, e.g. the square root of the parameter.

4.4.1 Evaluation Types of Modula-2

When talking about procedures in Modula-2, one should also mention its as of today rather dated way of differentiating between a *call-by-value* and *call-by-reference* evaluation – both types of evaluating are used in Modula-2:

```
PROCEDURE PrintDataOut(Puppy : INTEGER);
BEGIN
   WriteString("The␣value␣of␣Puppy␣is␣␣␣␣␣");
   WriteInt(Puppy,5);
   WriteLn;
   Puppy := 12;
END PrintDataOut;
```

```
PROCEDURE PrintAndModify(VAR Cat : INTEGER);
BEGIN
   WriteString("The␣value␣of␣Cat␣is␣␣␣␣␣␣␣");
   WriteInt(Cat,5);
   WriteLn;
   Cat := 37;
END PrintAndModify;
```

The two procedures (taken from [6]) above show both evaluation strategies: the first one, the procedure `PrintDataOut` gets the parameter `Puppy`, an integer. The second procedure, `PrintAndModify` also gets an integer as a parameter, but with the added `VAR` in front of it. By adding `VAR`, the compiler is signaled that not just the value of the integer is passed to the procedure, but the actual variable, which means that the very variable the caller passed to the procedure will be altered in the same way as the variable in the procedure `PrintAndModify` – a *call-by-reference* is made, whereas the procedure `PrintDataOut` merely gets the value of the integer – thereby a *call-by-value* is made here.

5 Syntax

A Modula-2 program consists of at least one module. A module is made up of three parts:

- the module head including the module name,
- import lists & declaration of variables and constants,
- the program body.

```
MODULE Name;
  <import lists>
  <declarations>
  BEGIN
    <statements>

END Name.
```

Every module's end is marked by the keyword `END`, followed by the module name and a period. Here's a *Hello, World!* program to see the basic structure of a simple Modula-2 program put to use:

```
MODULE HelloWorld;
  FROM InOut IMPORT
       WriteString, WriteLn;
  BEGIN
    WriteString('Hello␣world!');
    WriteLn;
END HelloWorld.
```

6 The Module

As mentioned before, arguably the main difference between Modula-2 and its predecessor Pascal is Modula-2's *modularity*. It exposes the details of the simplest subtasks while simultaneously expressing the relationship of each subtask to the rest of the program – this way a program gains a lot of clarity, much like with an explosion diagram [7, Preface viii] seen in Figure 3. The concept of the module was something very new and absolutely revolutionary in the world of computer sciences – the idea of separating code into different modules and thereby also different files allowed for a far more flexible way of programming than ever before.

It is because of this new paradigm, that many classic elements such as I/O routines, semaphores, or a FILE data type are not part of the original Modula-2 package. They are stored in external library modules, and are like most other things only imported into a program when needed – just like in the *Hello, World!* program above, when the `InOut` module was imported to utilize the methods `WriteString` and `WriteLn`.

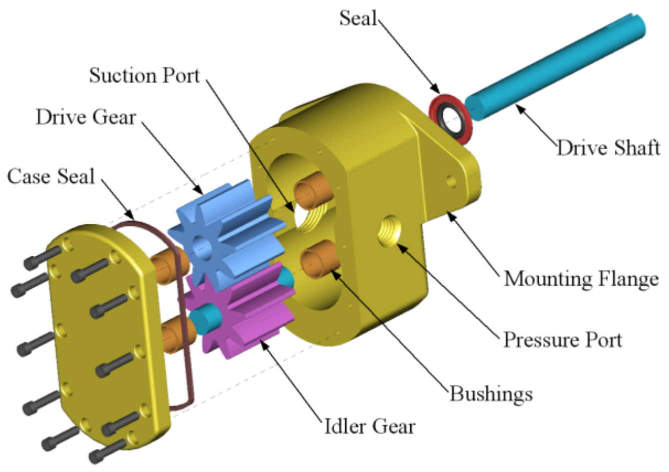

Figure 3: Exploded View Drawing

6.1 Separate Compilation

Each and every used module can be compiled by itself, an idea called *separate compilation*, and therefore exists on its own abstraction level. An example would be the module `InOut`, which was used above. That module was compiled by itself and could therefore be used in the *Hello, World!* program without having to compile it together with yet uncompiled modules. They are stored in the program library in already compiled form and are merely *linked* with the main program.[6]

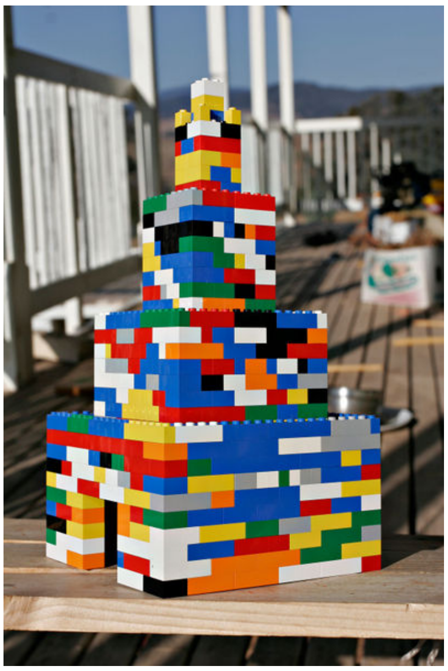

Figure 4: Many different modules form a single large program

This feature was new to Modula-2, and it was actually one of the first languages to utilize this now essential feature, and one of the main advantages over Pascal. With it, a large number of programmers could work simultaneously on one complex program, consisting of many single modules, each programmer working on their own modules. This of course, made their work a lot easier as far less communication between them was necessary and the debugging process was simplified enormously. Not to forget the augmentation of the reusability of a single module.

The importance of this feature can not be emphasized enough – it is one of the main reasons why Modula-2 was (and is, in a more historical sense) such an important language.

[6]A feature missing in other major programming languages at the time, such as Fortran, Pascal and assembler codes, which used *independent compilation* [4, p 81].

6.2 Importing and Exporting Modules

Each module has, at least theoretically, two interfaces that enable it to interact with other methods it wants to utilize or be utilized by, respectively – an *import-* and an *export interface*. Because of these interfaces, a programmer only needs to know the specifications of another module's export interface to know what it offers and how to make use of it. This allows for an relatively easy construction of larger programs as well as minimizing code duplication, and affirms the principle of single responsibilities for modules. Niklaus Wirth writes:

> *The principal motivation behind the partitioning of a program into modules is – beside the use of modules provided by other programmers – the establishment of a* hierarchy of abstractions. [...] [W]*e merely wish to have them available, but do not need to know - or rather do not wish to bother to learn – how these procedures function in detail* [4, p 82].

This makes it possible not only to *hide* other functions, but also, much more important, lets the programmer change the implementation of a module, the *Implementation Module*, and still have it working for every other module it is implemented by, simply by not changing the interface (and, of course its offered services), or the so called *Definition Module*, which also can be understood as the export interface.

Each of those two modules is put in its own file. By separating the code into theses two parts, Modula-2 achieves encapsulation, and, as mentioned above, levels of abstraction. Therefore, both modules can be compiled separately – they are called *compilation units*.

Here's an example on a module split into a definition and an implementation part, taken from Wirth's book *Programming in Modula-2* [4, p 83, 84]:

```
DEFINITION MODULE Buffer;
  VAR nonempty, nonfull: BOOLEAN;
  PROCEDURE put(x: INTEGER);
  PROCEDURE get(VAR x: INTEGER);
END Buffer.
```

```
IMPLEMENTATION MODULE Buffer;
  CONST N = 100;
  VAR in, out: [0 .. N-1];
    n: [0..N];
    buf: ARRAY [0 .. N-1] OF INTEGER;
  PROCEDURE put(x: INTEGER);
  BEGIN
    IF n < N THEN
      buf[in] := x; in := (in+1) MOD N;
      n := n+1; nonfull := n < N; nonempty := TRUE
    END
  END put;
  (* continued on the next page *)

  PROCEDURE get(VAR x: CARDINAL);
  BEGIN
    IF n > 0 THEN
      x := buf[out]; out := (out+1) MOD N;
      n := n-1; nonempty := n > 0; nonfull := TRUE
    END
  END get;

  BEGIN n := 0; in := 0; out := 0;
    nonempty := FALSE; nonfull := TRUE
END Buffer.
```

Above is the `DEFINITION MODULE Buffer`, right below its `IMPLEMENTATION MODULE Buffer`. As one can see, `Buffer` lets other modules utilize its procedures `put` and `get`, as well as two Boolean variables telling the user is the buffer is full or empty. It does not, however, have to offer all its variables or procedures for exporting, some can stay *private* and therefore only in the *scope* of the `Buffer` module.[7]

Here's how exactly a module is imported by another module:

```
$ import = ["FROM" identifier]
           "IMPORT" IndentList ";".
```

As the reader can see from the EBNF clause, the `FROM ModulName` clause obviously is not always necessary – this is the case when using so called *Local Modules*.

6.3 Local Modules and Levels of Abstraction

Local modules are not separately compilable and their only purpose is the hiding of details of their internally declared objects [4, p 94]. They are *nested* inside of another module and are especially useful as they restrict and are restricted on the scope of variables and procedures.

Here's an example on a local module and the scope of some variables inside it [4, p 94]:

[7]Modula-2 also offers the possibility of using qualified identifiers – if one imports a, b, and c from a module B, they can be referenced by B.a, B.b, and B.c, respectively.

```
VAR a, b: INTEGER;
MODULE M;
  IMPORT a; EXPORT w, x;
  VAR u, v, w: INTEGER;
  MODULE N;
    IMPORT u; EXPORT x, y;
    VAR x, y, z: INTEGER;
    (* u, x, y, z visible here *)
  END N;
  (* a, u, v, w, x, y visible here *)
END M;
(* a, b, w, x visible here *)
```

As seen in the code above, a module `N` is nested inside of a module `M`, effectively hiding everything it does not explicitly export from `M` and the outside – each module only sees the things it actually imports, and of course its own variables. This means that any exported variable is visible *one abstraction layer above* a nested, or local, module.

7 Modula-2 Today and a Short Resume

Even though Modula-2 was and is widely regarded as a very good and simple programming language, it never really reached a complete mainstream status, even though for example big companies such as General Motors used to have their own Modula-2 compiler, *M-GM*, "to program the engine/transmission controllers in millions of cars and trucks per year beginning in the mid 90's" [8].

A reason for this surely was the rapid improvement and development of new languages in the years following Modula-2's creation and publication.

Modula-2 has a rather large number of successors, such as:

- Modula-2+, with added exception handling and a garbage collector,
- Modula-3, an evolution of Modula-2+,
- Oberon, created in 1986 by Niklaus Wirth, much like Modula-2, but a lot smaller and with an added garbage collector,
- Oberon-2,
- Modula-2 R10, an open source revision of Modula-2 started in 2009, but seemingly abandoned at the moment.

Modula-2 was of course a big influence on later iterations of Pascal, as well as the most part of "modern" programming languages – most languages would, for example, be unthinkable without the possibility of separate compilation.

The use of modules not only simulated some sort of object orientated paradigm, in combination with, e.g., the possibility of higher order functions it also implicates concepts such as *abstract data types*: code can be definite in *what* it does, but not *how* it does it – Modula-2 was, together with Ada, one of the first modular programming languages, that made use of this form of abstraction by encapsulating modules and procedures, i.e. by the strict distinction made between the Definition - and the Implementation Module. Each of those encapsulations represents an abstract data structure, and by importing and implementing such structures, one can use them as abstract data types and not be concerned about the actual implementation of those those types. This of course was a huge influence on programming languages to follow – abstract data types are considered a standard now.

Also the idea and application of modules, and by that the possibility of separate compilation of these modules, its absence almost unthinkable now, can not be overemphasized.

Merely the concept of local modules and its influence on object-oriented inheritance would probably be enough to secure Modula-2 its well deserved historical importance.

Today, more and more, Modula-2 is being pushed aside as a learning language, especially in Europe where it was used most, by more popular languages such as C, C++, and Java, just as much as a language for professional programmers. Specific reasons for this trend may be things like Modula-2's evaluation strategy (which can be call-by-value or call-by-reference, as shown in 4.4.1), which is considered antiquated as of today.

Modula-2 is, as I hope I could show, a rather easy to read and comprehend programming language, what makes it a top choice especially as a first language to learn – but its time in the mainstream seems to be over, and today it is probably more of a "special interest" than something to put on ones CV.

References

[1] ETH Department Informatik. `http://www.inf.ethz.ch/personal/wirth/`.

[2] Svend Erik Knudsen. *Medos-2: A Modula-2 Oriented Operating System for the Personal Computer Lilith.* PhD thesis, Swiss Federal Institute of Technology Zürich, 1983.

[3] Niklaus Wirth. *A Brief History of Modula and Lilith.* `http://www.modulaware.com/mdlt52.htm`, January 1995.

[4] Niklaus Wirth. *Programming in Modula-2, 4th Edition.* Texts and monographs in computer science. Springer, 1988.

[5] Niklaus Wirth. *Programmieren in Modula-2, 2. Auflage.* Springer, 1991.

[6] modula2.org. *Chapter 5 - Modula-2 Procedures.* `http://www.modula2.org/tutor/chapter5.php`.

[7] Kaare Christian. *A guide to Modula-2.* Texts and monographs in computer science. Springer, 1986.

[8] Free Modula-2 Pages. *Compilers and Translators.* `http://freepages.modula2.org/compi.html#disap`.

Figures

Figure 1, page 1: `http://en.wikipedia.org/wiki/File:Niklaus_Wirth_large.jpg`

Figure 2, page 3: `https://en.wikipedia.org/wiki/File:Diser_Lilith-IMG_1729.jpg` and `https://en.wikipedia.org/wiki/File:Xerox_Alto.jpg`

Figure 3, page 8: `https://en.wikipedia.org/wiki/File:Gear_pump_exploded.png`

Figure 4, page 9: `https://en.wiktionary.org/wiki/File:Lego_tower.jpg`