Modula-2

Daniel Moosbrugger

99 Bottles of Beer

January 21st, 2013

Outline

The History of Modula-2: Niklaus Wirth

The Developer of Modula-2

Wirth was born in Switzerland in 1934. He studied at ETH Zürich (*Swiss Federal Institute of Technology Zürich / Eidgenössische Technische Hochschule Zürich*), Université Laval in Canada, and at the University of Berkley in California, where he was awarded his Ph.D. in 1963.

In 1968 Wirth became Professor of Informatics at ETH Zürich where he stayed until his retirement in 1999.

Niklaus Wirth, 1969

He also coined the phrase (now known as *Wirth's Law*):
"Software is getting slower more rapidly than hardware becomes faster."

The History of Modula-2: Niklaus Wirth

Languages and Awards

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, he still is the only German speaking winner of this prestigious award.
Later in 1988 he was awarded the *IEEE Computer Pioneer Award*.

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**
- one of the first languages to use data types

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**
- one of the first languages to use data types
- Pascal had its downsides, especially because it did not easily allow for large programs

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**
- one of the first languages to use data types
- Pascal had its downsides, especially because it did not easily allow for large programs
- United States Department of Defense (DoD) began to develop Ada

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**
- one of the first languages to use data types
- Pascal had its downsides, especially because it did not easily allow for large programs
- United States Department of Defense (DoD) began to develop Ada
- at the same time, Wirth started the development of Modula (**Modu**lar Programming **La**nguage)

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**
- one of the first languages to use data types
- Pascal had its downsides, especially because it did not easily allow for large programs
- United States Department of Defense (DoD) began to develop Ada
- at the same time, Wirth started the development of Modula (**Modu**lar Programming **La**nguage)
- first implementation of **Modula-2** in 1979, the first compiler was released in 1981

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**
- one of the first languages to use data types
- Pascal had its downsides, especially because it did not easily allow for large programs
- United States Department of Defense (DoD) began to develop Ada
- at the same time, Wirth started the development of Modula (**Modu**lar Programming **La**nguage)
- first implementation of **Modula-2** in 1979, the first compiler was released in 1981
- just like Pascal, Modula-2 was primarily intended as a language to learn programming (outside of the ETH *Lilith* project)

The History of Modula-2: From Pascal to Modula-2

Pascal and the Dawn of Modula-2

- early 1970s: **Pascal**
- one of the first languages to use data types
- Pascal had its downsides, especially because it did not easily allow for large programs
- United States Department of Defense (DoD) began to develop Ada
- at the same time, Wirth started the development of Modula (**Modu**lar Programming **La**nguage)
- first implementation of **Modula-2** in 1979, the first compiler was released in 1981
- just like Pascal, Modula-2 was primarily intended as a language to learn programming (outside of the ETH *Lilith* project)
- Modula-2 is Turing Complete

The History of Modula-2: The Lilith

Lilith (left) and Xerox Alto (right)

Pascal vs. Modula-2

The 5 Main Differences

Pascal vs. Modula-2

The 5 Main Differences

1. Modula-2 is, as its name already reveals, *modular*, unlike Pascal

Pascal vs. Modula-2

The 5 Main Differences

1. Modula-2 is, as its name already reveals, *modular*, unlike Pascal
2. its syntax is more systematic

Pascal vs. Modula-2

The 5 Main Differences

1. Modula-2 is, as its name already reveals, *modular*, unlike Pascal
2. its syntax is more systematic
3. the concept of processes was established

Pascal vs. Modula-2

The 5 Main Differences

1. Modula-2 is, as its name already reveals, *modular*, unlike Pascal
2. its syntax is more systematic
3. the concept of processes was established
4. low-level facilities [*maschinennahe Elemente*] - important for Lilith

Pascal vs. Modula-2

The 5 Main Differences

1. Modula-2 is, as its name already reveals, *modular*, unlike Pascal
2. its syntax is more systematic
3. the concept of processes was established
4. low-level facilities [*maschinennahe Elemente*] - important for Lilith
5. procedure type, allowing dynamic assignments

Symbols & Variables

Elementary Datatypes

- INTEGER
- CARDINAL
- REAL
- BOOLEAN
- CHAR
- BITSET

Symbols & Variables

Elementary Datatypes

- INTEGER
- CARDINAL
- REAL
- BOOLEAN
- CHAR
- BITSET

Constants and Variables

A constant will, as its name suggests, never change its value.
A variable on the other hand only has a fixed *type*, its value can change at any time in the program.

Symbols & Variables

Elementary Datatypes

- INTEGER
- CARDINAL
- REAL
- BOOLEAN
- CHAR
- BITSET

Constants and Variables

A constant will, as its name suggests, never change its value.
A variable on the other hand only has a fixed *type*, its value can change at any time in the program.

They are declared as follows:

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

Symbols & Variables

Symbols of the Modula-2 Vocabulary

- Identifiers
- Numbers: Integers, Real Numbers, Cardinals
- Strings: `"This is a String!"`
- Comments: `(* This is a Comment!  *)`
- Operators: `+`, `&`, `AND`, `OR`, `FROM`, `...`
- Assignments: `designator := expression`
- Statement Segments: `statement {";" statement}`

Symbols & Variables

Symbols of the Modula-2 Vocabulary

- Identifiers
- Numbers: Integers, Real Numbers, Cardinals
- Strings: `"This is a String!"`
- Comments: `(* This is a Comment!  *)`
- Operators: `+, &, AND, OR, FROM, ...`
- Assignments: `designator := expression`
- Statement Segments: `statement {";" statement}`

But wait, there's more!

There are naturally more data types, symbols, etc. in Modula-2, e.g.: the procedure as a data type, arrays, enumeration types, set types, dynamic structures and pointers, record types, the possibility of higher order functions, ...

Syntax

Modula-2's syntax

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

END Name.
```

Syntax

Modula-2's syntax

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

END Name.
```

A module is made up of three parts:

Syntax

Modula-2's syntax

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

END Name.
```

A module is made up of three parts:

- the module head including the module name,

Syntax

Modula-2's syntax

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

END Name.
```

A module is made up of three parts:

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

Syntax

Modula-2's syntax

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

END Name.
```

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.

Syntax: Hello World!

An example: *Hello World!*

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

END Name.
```

Syntax: Hello World!

An example: *Hello World!*

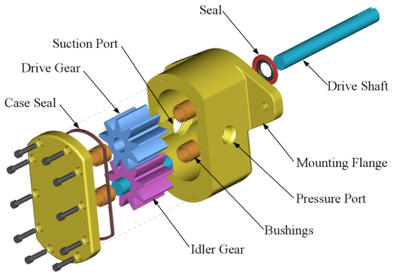
```
MODULE Name;
  <import lists>
  <declarations>
  BEGIN
    <statements>

END Name.
```

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

Syntax: The Module

The Module

The main difference between Pascal and Modula-2 is its *modularity*:



Modula-2 is like an exploded diagram. It exposes the details of the simplest subtasks while simulatneously expressing the relationship of each subtask to the rest of the program.

Syntax: The Module cont.

Divide and Conquer!

Many different modules can be used by importing them into a program. Each module can be compiled by itself (**separate compilation**, a possibility missing in Pascal) and exists on its own *abstraction level*, e.g.: `InOut`

Syntax: The Module cont.

Divide and Conquer!

Many different modules can be used by importing them into a program. Each module can be compiled by itself (**separate compilation**, a possibility missing in Pascal) and exists on its own *abstraction level*, e.g.: `InOut`

Each module has two interfaces: import- and export interface.

Because a module only needs to know another module's export interface, very large programs can be built easily by putting together (importing) a number of different modules.

Syntax: The Module cont.

The Module

Niklaus Wirth, from *Programming with Modula-2* (4th ed., 1988):

> *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.*

Syntax: The Module cont.

The Module

Niklaus Wirth, from *Programming with Modula-2* (4th ed., 1988):

> *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.*

- necessary for this kind of abstraction: splitting a module into a *definition part* and an *implementation part*

Syntax: The Module cont.

The Module

Niklaus Wirth, from *Programming with Modula-2* (4th ed., 1988):

> *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.*

- necessary for this kind of abstraction: splitting a module into a *definition part* and an *implementation part*
- two modules, one using the other, can therefore be compiled separately, and are called *compilation units*

Syntax: The Module cont.

Definition and Implementation Modules of Buffer

```
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;

  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.
```

Syntax: Importing a Module

Definition and Implementation Modules: Importing

Importing via:

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

Qualified identifiers: If a module A imports a, b, c from a module B, they can be referenced by the designators B.a, B.b, B.c

`FROM ModuleName` is not necessary when the module is nested inside the exporting module.

Syntax: Importing a Module

Definition and Implementation Modules: Importing

Importing via:

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

Qualified identifiers: If a module A imports a, b, c from a module B, they can be referenced by the designators B.a, B.b, B.c

`FROM ModuleName` is not necessary when the module is nested inside the exporting module.

Local Modules

An alternative to using modules via an Implementation- and Definition module is using a local module. They are not separately compilable.

Syntax: Importing a Module

Local Modules cont.

An example on local modules and their scope:

```
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 *)
```

Modula-2 Today

Modula-2: A dead language?

Even though Modula-2 was relatively successful, it never really reached full mainstream.

A number of programming languages succeeded it, most notably:

Modula-2 Today

Modula-2: A dead language?

Even though Modula-2 was relatively successful, it never really reached full mainstream.

A number of programming languages succeeded it, most notably:

- Modula-2+
- Modula-3
- Oberon
- Oberon-2
- Modula-2 R10

Modula-2 also influenced, of course, later iterations of Pascal.

99 Bottles of Beer (Modula-2)

```
MODULE BottlesOfBeer;
FROM InOut IMPORT WriteCard, WriteString, WriteLn;

CONST BOTTLESOFBEER = 99;
VAR counter : CARDINAL;

BEGIN
  counter := BOTTLESOFBEER;

  LOOP
    IF (counter > 9) THEN WriteCard(counter,2)
      ELSE WriteCard(counter,1) END;
    WriteString(" bottles of beer on the wall, ");
    IF (counter > 9) THEN WriteCard(counter,2)
      ELSE WriteCard(counter,1) END;
    WriteString(" bottles of beer,"); WriteLn;
    WriteString("take one down, pass it around, ");
    DEC(counter);
    IF (counter = 1) THEN EXIT END;
    IF (counter > 9) THEN WriteCard(counter,2)
      ELSE WriteCard(counter,1) END;
    WriteString(" bottles of beer on the wall.");
    WriteLn; WriteLn;
  END;

  WriteString("1 bottle of beer on the wall."); WriteLn; WriteLn;
  WriteString("1 bottle of beer on the wall, 1 bottle of beer,"); WriteLn;
  WriteString("take it down, pass it around, no more bottles of beer on the wall!");
  WriteLn;

END BottlesOfBeer.
```

Thank you for your attention!

Questions? / Fragen?