Terry Regner
230021529

Modula-2

1 Introduction

In this paper, Modula-2, a general purpose, imperative programming language will be described and discussed.

In order to gain a full understanding of the language, I will discuss the history of Modula-2, including the key figures involved in its creation, and the obstacles it was designed to surmount.

My choice of compiler for the Modula-2 language will be discussed, and I will include in the discussion the simple commands I used to compile and execute the source code while learning the language.

Data types, loops, and control structures will be discussed in this paper, as will the basic structure of a Modula-2 MODULE. I utilized the “Hello World” program to illustrate the salient features of the Modula-2 language and to demonstrate the compilation and execution commands used to generate the assembly code and execute the program. Throughout this paper code selections will be provided in order to enrich the concepts under discussion.

The main properties of the language will be discussed in some detail and will include simplicity, orthogonality, and other pertinent concepts.

Detailed references to all materials used for the purposes of this paper have been separated for ease of review into “web resources” and “references and selected readings” sections respectively. The links to the compiler distribution and general modula-2 web sites have been referenced and are listed in the “web resources” section of my reference list

2 History

The creator of modula-2, Dr. Niklaus Wirth, was born in the February of 1934 in Winterthur, Switzerland. Dr. Wirth became a Professor of Informatics in 1968 at the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland. During sabbatical from ETH in 1976, Dr Wirth worked with the Xerox Palo Alto Research Center (PARC) in California. It was at PARC that Dr. Wirth was exposed to the Mesa programming language, a modular programming language that was being used for programming one of the first personal computers featuring a Graphical User Interface (GUI) named the Xerox Alto (N. Wirth, 1995) [3].

Upon return to ETH, Dr. Wirth took an integral role in the development of the workstation Lilith from 1978 - 1980. Lilith was quite revolutionary for the times, featuring a micro-programmed processor, a high-resolution display, and a mouse (N. Wirth, 1995). The entire software package for Lilith was programmed using a new language that Dr. Wirth was developing. The language, Modula-2, was to be a mixture of the Pascal language (also a creation of Dr. Wirth's), and the Mesa programming language Dr. Wirth was exposed to at PARC. Dr. Wirth, when discussing Modula-2, states: "*It featured the module with explicit interface specifications, and its implementation included the facility of separate compilation of modules with complete type checking*" (Wirth, 1995).

Dr. Wirth described the conditions in which Modula-2 was developed as being that the chief concern was constructing a language that could be used with limited programmer input which was at the same time powerful enough to institute an entire software system (Wirth, 1995).

Modula-2's design principles have continued to influence several object oriented languages including Oberon, Modula-3, Eiffel, and Java [3].

3 Compiler

For this paper I obtained the Native XDS-x86 (v2.51) optimizing ISO compiler for Intel x86 based platforms [xx]. Native XDS-x86 (v2.51) an almost fully featured development tool that I was able to install on my Windows XP system in less than 5 minutes. Native XDS-x86 (v2.51) features a GUI for its included text editor, which functions optimally, seamlessly, installing all needed files to compile and execute Modula-2 files via the command prompt on Windows XP.

The Modula-2 compiler requires that all “reserved words” (e.g. MODULE, BEGIN, END) are in uppercase text; otherwise they will not be recognized as reserved by the compiler. There are a total of 40 reserved words in the Modula-2 language [10].

To invoke XDS Environment, you must select the "XDS Environment" item from the XDS folder of your Start menu or type ***Xds*** at the command prompt and press ***<Enter>***.

To start the compiler from the command prompt, you must type ***xc {options | modes} module_or_project_file {options | modes}*** and press ***<Enter>***. If you were to specify “no arguments” a brief help message would be printed.

To set up a working directory for a new project, create a new directory, enter the directory, and type ***testwork*** in the command prompt***.*** Typing ***testwork*** will create sub-directories for source and output files, as well as a project file template, TEST.TPR.

4 Data Types

The following five types are denoted by standard identifiers and are predeclared [10].

4.1 Integer Types

Integer types assume the value of an integer between the minimum and maximum values allowed, [minInt..maxInt] (Wirth, 1980). The value is represented in one memory word with the sign bit at bit 0 and the most significant bit at bit 1 (Bingham, et al.,1983).

Ex.

Range = [minInt ..(–minInt – 1)]

4.2 Cardinal Types

A cardinal value assumes a positive integer value < = 65535. Cardinal values are represented as one memory word and bit 0 is considered the most significant bit (Bingham et al., 1983).

Ex.

Range = -minInt + (–minInt – 1)

4.3 Real Types

A variable declared as a REAL assumes the value of a real number as represented by a two bit memory word (Bingham et al., 1983). The first bit of the second word is not stored as its value is always 1 (Bingham et al., 1983).

Ex.

First Word		
1 BIT Sign Bit	8 BIT Exponent	7 BIT High Part of Mantissa

Second Word
16 BIT Low Part of Mantissa

(Bingham et al., 1983)

4.4 Boolean Types

A Boolean value is a binary type that has the values of true (1) or false (0) only. The Boolean type is represented by one memory word (Bingham et al., 1983).

Ex.

Enumeration representation of a Boolean type

Enumeration (FALSE, TRUE) → false := 0 AND true := 1

4.5 Char Types

Char types assume the integer value of the ASCII (ISO) character that is to be represented (W Wirth, 1980). ASCII character sets have 2^7(0 - 127) actual characters, anything higher is reserved for special characters, such as “end-of-line” characters. Variables of type char are represented in one memory word per character (Bingham et al., 1983).

The following are not denoted by standard identifiers and the types are not predeclared.

4.6 Array Types

An array is a storage object that has a fixed number of locations as defined by a subrange. The index that defines the array is all of the same basic data type, which may include: INTEGER, CARDINAL, enumeration, boolean, or char data types [10]. The entries are stored in sequential order in memory and are recalled using the indexing system indirectly via a pointer that indicates the first element of the array (L. Bingham, et. al).

Ex:

Declaration of the array component and index type (N Wirth, 1980).

```
$   ArrayType = ARRAY SimpleType {"," SimpleType} OF type.
```

```
Examples of array types:

    ARRAY [0..N-1] OF CARDINAL
    ARRAY [1..10], [1..20] OF [0..99]
    ARRAY [-10..+10] OF BOOLEAN
    ARRAY WeekDay OF Color
    ARRAY Color OF WeekDay
```

4.7 Record Types

A record type is used to create a customized data structure that has a fixed number of components of potentially different types (Wirth, 1980). Records are usually accessed indirectly and have a pointer indicating the first field of the record. Each field in the record must be represented by at least one memory word (Bingham et al., 1983). The structure of a record type is similar to that of a struct in c and c++, and it can also be described as a table entry in a database.

Ex.

RECORD name: char
Age: [0..130]
Income: REAL
END

The above case (terry, 30, 150000) is a valid RECORD.

4.8 Enumeration types

An enumeration (enum) is simply a list of identifiers which represent values comprising a unique data type [10]. The identifiers chosen are the only values that belong to the type defined, and are used as constants throughout the program (Wirth, 1980). The location of a specific identifier in the list determines the attribute's value, and each subsequent value is increased from left to right as the number of attributes increases.

Ex:

Given the enum → (yes, maybe, no),

Yes < maybe < no, is true.

4.9 Subrange types

A subrange specifies the lower and upper bounds of an index, of which lowerbound < upperbound must be true. Real numbers are not valid types for defining subranges.

Ex:

[0..n] is a subrange, as is [0..n+1]

4.10 Set types

The set type is comprised of items of the same scalar and nonreal type, and do not distinguish between order. The range of the set is an integer value with a subrange of at most [0..wordsize-1], or in the case of an enum, the enum must have at most a wordsize value (Wirth, 1980). The set type is implemented in one memory word (Bingham et al., 1983).

4.11 Pointer types

A pointer variable is directed at a memory location in which the item of interest is located. In Modula-2 a procedure,"New", is used to generate a pointer to a variable. It may be the case that a pointer is directed at nothing, in which case a value of NIL will be assigned [10]. Pointers are represented in one memory word which contains the address to the element it is pointing at (Bingham et al., 1983).

4.12 Procedure types

Procedures may be assigned to variables for calculation of non-trivial equations; however, the returning values of the procedure must correspond to the values of the variables assigned. Procedures may have parameters, or may be free of parameters, and thus are represented in one memory word (Bingham et al., 1983).

Memory word	
8 BITS Module Number	8 BITS Procedure Number

(L. Bingham, et. al)

Ex.

A variable Proc is being assigned a value from the parameter free procedure termed Procedure.

```
var Proc = Procedure;
```

5 Loops and Control Structures

5.1 REPEAT ... UNTIL Loop

The repeat loop is designed to repeat until a predefined condition (a value of TRUE) is reached. The example increments index by one until the index value equals five. The loop will always terminate unless the original value of index is greater than five.

Ex.

```
REPEAT
        Index := Index + 1;
UNTIL Index = 5;
```

5.2 WHILE Loop

The while loop is similar to the repeat loop, however, the condition is tested at the loop's beginning rather than at the end. A value of FALSE is used to terminate the loop.

Ex.

```
WHILE Index < 5 DO
        Index := Index + 1;
 END;
```

5.3 FOR Loop

The FOR loop uses reserved words, including FOR, TO, END, BY, and DO. The FOR loop counts loops using any simple variable with the exception of REAL to count loops.

Ex.

```
FOR Index := 1 TO 5 DO
        WriteInt(Index,5);
END;
```

5.4 Infinite Loop

The infinite loop is unable to terminate itself without a programmer implementing a command, usually EXIT or END.

Ex.

```
1       LOOP
2               IF Index = 5 THEN
3                       EXIT;
4               END;
5               Index := Index + 1;
6       END;
```

5.5 IF Statement

An IF statement is a conditional statement which when evaluated as TRUE performs the operations specified within its' body.

Ex.

```
IF Index = 5 THEN
        WriteString("Index is 5");
END
```

5.6 The "ELSE" Clause

The ELSE clause can be utilized after an initial IF statement and performs the operation within it's body if the original IF statement evaluates as FALSE.

Ex.

```
IF Index = 5 THEN
        WriteString("Index is 5");
ELSE
        WriteString("Index1 is not 5");
END
```

5.7 The "ELSIF" Clause

The ELSIF clause is used between an IF and ELSE clause enabling the programmer to sequentially evaluate various conditions that may be present in the same manner as an IF statement.

Ex.

```
IF Index = 5 THEN
        WriteString("Index is 5");
ELSIF Index1 = 6 THEN
        WriteString("Index1 is 6");
ELSE
        WriteString("Index is not 5 or 6");
END
```

5.8 Case Statement

A case statement could be described as the root of a tree, from which different branches (or values) will take you to various outcomes depending on the evaluation.

Ex.

```
CASE Value OF

        1..5            : WriteString("The number is between  1- 5"); |
        6..9            : WriteString("The number is between 6 - 9"); |
        10,11           : WriteString("The number is 10 or 11"); |
        14..17          : WriteString("The number is between 14 - 17"); |
        18,20,22        : WriteString("The number is 18 or 20, or 22"); |
        19,21,23        : WriteString("The number is 19 or 21 or 23");

        ELSE
                WriteString("The number is not in the list");
 END;
```

6 Hello World

The following sequence is the code required for the ever popular hello world program.

```
1       (*
2               Hello World Program
3               Written by: Terry Regner
4               Date: November 26, 2005
5
6               To compile: xc =m HelloWorld
7
8               To run: HelloWorld
9       *)
10
11      MODULE HelloWorld;
12      FROM STextIO IMPORT WriteString, WriteLn;
13      BEGIN
14              WriteString ("hello world"); WriteLn;
15      END HelloWorld.
```

Line 1 - 9: Delineate the comment block as defined by the (* and *) brackets in lines 1-9. The information in the block is skipped by the compiler and is only needed to increase the readability of the code.

Line 11: The reserved word, MODULE, is used to declare a new module. The name following MODULE, in this case HelloWorld, is the name assigned to the module. A specification of Modula-2 requires that all modules defined are named. Modula-2 requires that this line is terminated by a semi-colon.

Line 12: Required system modules are also gathered in line 12 and are needed to perform the tasks of the program. Essentially, we are asking that the modules WriteString and WriteLn are loaded from the file named STextIOwhich allows a string to be written and the latter to produce a blank line. The blank line is ended by a semi-colon to indicate the end of the procedures needed from a particular location. The structure is as follows:

FROM *file* **IMPORT** *procedure procedure ;*

This is read “import this(these) module(s) from this file”.

Line 13&15: The reserved words BEGIN and END are used to mark where the actual code for the program is placed. After the work END, the module name follows. Once more it is used to specify to the compiler what to name the executable file created. The end of file deleter, the ".", is placed next in order to inform the compiler it has come to the end of the file.

Line 14: Line 14 contains a "WriteString" followed by a WriteLn" statement. These statements are quite elementary and may be considered self-explanatory. Each line is a call to a "procedure", which is a very important feature of Modula-2. A "procedure" is an external process that performs a specific job in a well defined way. In the case of the "WriteString", the procedure looks at the string of characters supplied to it and displays the string of characters on the monitor at the current cursor position [10]. In the case of the "WriteLn" procedure, it serves us by moving the cursor down one line on the monitor and moving it to the left side of the screen [10].

The parentheses are required immediately after WriteString because it has data following it. The data enclosed within the parentheses obtains the string of characters between the quotation marks or apostrophe delimiters, and display the string on the monitor. With Modula-2 you can switch between delimiters, allowing you to output the delimiters themselves. If you desire to output a quotation mark to the monitor, you must use apostrophes for delimiters, and if you wish to output apostrophes, you must use quotation marks.

Compilation, execution and Output from HelloWorld Program:

```
C:\Documents and Settings\regnert\My Documents>xc =m helloworld
O2/M2 development system v2.51 TS (c) 1999-2003 Excelsior, LLC. (build
10.05.205)
XDS Modula-2 v2.40 [x86, v1.50] - build 10.05.2005
Compiling "helloworld.mod"
no errors, no warnings, lines    5, time  0.03
New "tmp.lnk" is generated using template "C:/XDS/BIN/xc.tem"

XDS Link  Version 2.6 Copyright (c) 1995-2001 Excelsior
No errors, no warnings

C:\Documents and Settings\regnert\My Documents>helloworld
hello world
```

7 Language Properties and Evaluation

7.1 Simplicity

Niklaus Wirth's design philosophy is based on a simple scheme, namely, that anything that is not absolutely necessary can be left out (Polajnar, 2004). Modula-2 encompasses Wirth's focus on simplicity of design. Elements that emphasize simplicity are as follows.

Modula-2 features separately compiled library modules, allowing for platform dependencies to be isolated, portability to be increased [9], and cross-platform programming concepts to be implemented, regardless of environment. Modular designs can also reduce errors and cut down on maintenance time [9].

Modula-2 gives the programmer the ability to control name space and deconstruct a large project into smaller units with efficiency.

Identifiers in Modula-2 are all case sensitive (e.g. MODULE, IF, and ELSIF), providing normalization, readability, and elimination of compile time overhead by avoiding the need to check the case. Case sensitivity is also responsible for simplifying the definition of the Identifiers and reserved words [4], hence making the language simpler.

Another element of Modula-2 that increases simplicity is the fact that its implementation does not employ the use of the "goto label" [5]. Without the "goto label" the understanding of what the code is doing or intended to do is simplified.

Additionally, I/O is found in several type-specific modules, rather than being built in [4], as is commonly found in other languages. This allows for the linkers to only patch in the I/O code that's needed, thus making the programs smaller, simpler, and faster [9].

.

Finally, this is an extremely simple language to learn by reason of its small vocabulary, few sentence structures, and the fact that it features consistent, simple rules of syntax [6], as well as a simple conceptual model of semantics [7].

The main point against simplicity that was evident was the fact that the sub-ranges and unsigned types seem to complicate the whole-number type compatibility rules without adding any important functionality to the language.

7.2 Generality

Immediately noticeable when first learning to program in Modula-2 is the fact that the END reserved word is used to terminate all defined control structures as demonstrated in Section 5, Loops and Control Structures. No special cases will change the fact that the END is used to terminate all control structures. A second notable feature of the language was the VAR (variant) type. The VAR type allows you to specify a new variant without actually defining what data type it belongs to, simplifying the programming process.

Also of note is the fact that Modula-2 does not support Generic modules, although this can be simulated using an untyped pointer technique [5].

There were few constructs that seemed to generalize to any degree. I would say that already the extensive focus on safety as previously discussed in section 7.1 was a good indication that generality would not be a prominent feature of the language, as simplicity is often at odds with generality (Polajnar, 2004).

7.3 Orthogonality

Modula-2 does not employ the automatic conversion between integers and real numbers. The lack of automatic conversion provides a predictable outcome when one is faced with performing mathematical operations on variables of different types. All conversions are to be done manually, thus ensuring validity of results.

Another constant feature of Modula-2 is the fact that boolean expressions are short-circuit [5]. Through the employ of short circuit boolean expressions, as seen in the first example below, when a conditional statement is expressed with the AND or the OR operators, the left hand side of the expression is evaluated first. If the left statement is true it will continue on and evaluate the next expression for correctness and so on until a final true or false value is determined. On the other hand, if a condition is found to be false and there are still expressions that have yet to be evaluated, the control statement will “short-circuit” and return a final value of false for the IF statement. This “short-circuit” stops any of the remaining expressions from evaluating, which also serves to enhance performance by eliminating the number of instructions when performing most tasks. Non short circuit booleans will evaluate all expressions regardless of intermediate results.

Ex. Short Circuit programming

```
IF ( !IsEmpty( Data ) AND ( Data.name = "grades" ) )
      Data.getanAplus;
END
```

Additionally, Modula-2 has the facilities to allow for programmers to define custom data structures. Modula-2 fulfills the four mechanisms relied on for defining new data domains (Polajnar, 2004, p.21). The four mechanisms are satisfied by reserved words: RECORD, VAR, ARRAY, and RECURSION. These mechanisms allow for the creation of types that are customizable, readable, and understandable.

Modula-2 has the ability to be strongly modular, and therefore, it re-uses the same code for common tasks then packages its own module. By reusing the same code productivity is greatly increased and uniformity is promoted with a predictable effect [6].

Declarations can be given in any order as long as the names are declared before they are called for use. The exception to this rule is the procedure, which can be declared in any order regardless of where it is used [5].

Lastly the implementation of the reserved word END is as orthogonal as it is uniform, predictable, and can be composed in a free and non interfering manner. END is used to terminate all control structures, as demonstrated in the examples in their respective sections. END serves to demonstrate complete uniformity of use as we always know what it will do and it is present whenever a control structure is used. The only exception to END's uniformity of use is when the END represents the end of a MODULE. This line is then ended with a period instead of the usual semicolon.

7.4 Uniformity

As mentioned in 7.3, the uniformity in the use of the reserved word END is quite evident as there is an END statement associated with every loop and control structure. The only variance occurs when the END is associated with a MODULE, in which case the line is ended with a period. The two different END statements are shown below.

Ex.

```
MODULE nada
      IF( x := 7 )
            x := 8;
      END;
```

```
END nada .
```

Modula-2 displayed its uniformity through the fact that all keywords are in capitals. Capitalized keywords are a requirement of the language; any deviation from this standard will cause a syntactic error at compile time.

An example of a non-uniform property of Modula-2 is the Instance variable when compared with the parameter less function as shown in the following example.

Ex.

```
function( variable );
function( procedure );
```

The above function calls are indistinguishable from each other although they are evaluations of completely different operations.

7.5 Abstraction

The custom data structures and modularity (described in detail in section 7.3), allow exceptional abstract concepts to become possible. Modula-2 demonstrates an immediate sense of abstraction by virtue of it's ability to separate the problem itself from tangible methods of response.

The ability to call recursive routines in Modula-2 allows problems to be solved from a more natural approach. This approach could be compared to mathematics, although more complex problems sometimes require a level of knowledge with respect to recursion that the average programmer may not possess; therefore, although relevant to abstraction, the benefits of recursion are negligible for this reason.

During the development of Modula-2, Dr. Wirth determined the benefits of constructing incomplete modules that can only be implemented together with different modules, a concept similar to that of constructing a jigsaw puzzle. Using incomplete modules to form a comprehensive whole when placed together allow enhancement of pieces of code while not changing the entire original module, making upgrades easier to accomplish [8]. From this perspective, abstractness and an expected interface differ only in the intended style of composition. Modules allow for separate compilation and separate type checking, which is an important aspect of programming language abstraction [8].

The type transfer function implemented in Modula-2 is an element of the language that defies the concept of abstraction. It allows the type identifier to be used in expressions as a function identifier (Wirth, 1980).

Ex.

type_identifier(x) := x, where x is of type type_identifier

The above interpretation inherently depends on the architecture of the system in question and in the way it represents type x and type_identifier in memory (e.g. binary representation) (N. Wirth, 1995). Making every program implementation dependent is a clear contradiction of the fundamental goal of high-level languages; however, there is no indication of such dependency in a module's heading (N. Wirth, 1995).

The final property of the language in which it was evident that it clearly defied the concept of abstraction was the variant record VAR and its ability to be declared without use of the tag field. The tag field, used to determine the structure of the record, does not necessarily need to be included in the declaration (Wirth, 1980). The record VAR can be misused to access record fields with "wrong" types and skew results by mixing types that are incompatible. The lack of a tagfield raises an issue with portability, as the possibility of previously wrong types being introduced to a new architecture with a different bitset representation certainly exists.

7.6 Clarity

The Modula-2 language makes it easy to write clear, well documented code that is as straight forward to read it is to follow the flow of execution. Small portions of code are securely stored in their respective modules, allowing for encapsulation of data for use as needed. Programs written in Modula-2 are quite easy to follow and understand as they have a very simple language, small vocabulary [6], and follow a sequential and logical order. The lack of goto statements, as discussed in section 7.1, helps to facilitate the element of clarity, as does the orthogonality of the END clause described in section 7.3.

Modula-2 does not employ the automatic conversion between integers and real numbers as described in section 7.3, thus increasing clarity by providing a predictable outcome by allowing the compiler to disallow operations on variables of different types with static type checking, which is employed religiously to ensure all errors are detected at compile time, avoiding runtime errors.

The number of issues detracting from clarity is similar in number to those of the clear esthetic qualities which are obtainable in the structure and flow of the execution in Modula-2.

Of note is the type transfer function, described in section 7.5, which may be hard to follow if it is used repeatedly within a module or procedure. Also described in

7.5 are VAR's that may be declared without the use of the tag field, making it unclear as to which type of variable the VAR represents.

The Instance variable in comparison with the parameter less, function as described in section 7.5, is completely unclear. It is impossible from first glance to determine the type of parameter without investigating its source.

7.7 Modularity

As stated in Wirth, 1995, Modula-2 was originally created due to *" … the need for a structured programming language for building large, complex systems, and the solution, or rather the feature needed, was a construct for expressing units of separately compiled system components, now known as modules*". As the name suggests, this language is all about modularity, which reduces errors by allowing compilation in small independent pieces, and can greatly reducing maintenance time [9].

The Lilith workstation for which Modula-2 was designed to support was developed in such a way as to ensure the ability to construct large programs designed by many different people. The modules written by programmers have well-specified interfaces that can be declared independently of their actual implementations, utilizing only those resources that are relevant to each specific interface. Encapsulation is also made possible by the module concept.

Modula-2 is "modular", in every sense of the word.

7.8 Safety

Some languages allow the programmer to define constants and variables in a random fashion, and then proceed to combine data in a further random manner. Modula-2, on the other hand, requires that its programmer set up their constants and variables in a very precise manner, while additionally providing many cross checks at compile time, thus reducing the possibility of receiving meaningless output [10].

Modula-2 does not employ the goto or jump commands, thus avoiding complicated program flow of execution, so-called "spaghetti code" [3].

7.9 Expressiveness

The enumeration type employed by Modula-2 allows for expressiveness. Enumeration enables the programmer to specify an ordered list which can be used in place of a sequential number to describe various aspects of program elements.

Ex.

ENUMERATION(red, black, blue) is a more natural way to represent the type of color than ARRAY[3], which contains { 0, 1, 2 }.

7.10 Efficiency

Due to the extreme simplicity and Orthogonality of the Modula-2 language, the compiler is able to generate extremely efficient code. Separate compilation techniques of Modula-2 allow the distribution of code as binary modules.

Modula-2 was developed to be essentially machine-independent, the exception being limitations due to word size. If machine dependant features are needed they can be confined into separate modules, isolating them from the population of regular modules. The language provides the possibility of relaxed rules about data type compatibility in cases of machine dependency, has facilities to express input/output conversion procedures, file handling routines, storage allocators, and process schedulers as low-level modules which are components of most programs written [10].

Efficiency is a hard problem in the language Modula-2 to gauge, as various compiler producers offer different module libraries, affecting the possible efficiency of the language. Although generally speaking, because of the small size of the language and its constructs, efficient code should be expected when using Modula-2 9 (Wirth, 1995).

8 Web Resources Used

8.1 Compiler

1. Excelsior, Native XDS-x86 @ http://www.excelsior-usa.com/xdsx86.html

8.2 General

2. Lilith and Modula-2 @ http://cfbsoftware.com/modula2/

3. Wikipedia Online Encyclopedia @ http://en.wikipedia.org/wiki/Modula-2

4. Some reflections on Modula-2 standardization @ http://www.scifac.ru.ac.za/cspt/sc22wg13.htm

5. Christian Collberg, Principles of Programming Languages, Handout 20 @ http://www.cs.arizona.edu/~collberg/Teaching/520/2004/Handouts/Handout-20.pdf

6. Modula-2: Modula-2 Programming Language @ http://www.engin.umd.umich.edu/CIS/course.des/cis400/modula2/modula2.html

7. http://www.cs.williams.edu/~kim/cs334.97/Lec23.html#RTFToC8

8. http://prog.vub.ac.be/~wdmeuter/PostJava/Herrmann.pdf

8.3 Online Shareware Texts

9. Richard J. Sutcliffe**,** Modula-2: Abstractions for Data and Programming Structures (Using ISO-Standard Modula-2) 2004-2005 Edition @ http://www.arjay.bc.ca/Modula-2/Text/index.html

8.4 Tutorial

10. Modula-2 @ http://www.modula2.org/tutor/index.php

9 References and Selected Readings

N. Wirth , Oberon-2 and Modula-2 Technical Publication
Ubaye's First Independent Modula-2 & Oberon-2 Journal! Nr. 0, Jan-1995
http://www.modulaware.com/mdlt52.htm

N. Wirth , Modula-2, March 1980
Retrieved from http://www.fh-jena.de/~kleine/history/languages/Wirth-Modula2.pdf

L Bingham, L Geissman, C Jacobi, S E Knudsen, R L Riggs, N Wirth, Modula-2 Handbook: A Guide for Modula-2 Users and Programmers, 1983, retrieved from http://cfbsoftware.com/files/Modula2%20Handbook.pdf

N Wirth, Programming in Modula-2, 1982, Springer-Verlag, Heidelberg, New York, ISBN 3-540-12206-0

N Wirth, The personal computer Lilith, 1981, Proc. 5th International Conference on Software Engineering, IEEE Computer Society Press

J. Polajnar, 2004, Lecture Notes on Programming Languages