Modula-3

Modula-3 is a programming language conceived as a successor to an upgraded version of Modula-2 known as Modula-2+. While it has been influential in research circles (influencing the designs of languages such as Java, C#, and Python[7]) it has not been adopted widely in industry. It was designed by Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan (before at the Olivetti Software Technology Laboratory), Bill Kalsow and Greg Nelson at the Digital Equipment Corporation (DEC) Systems Research Center (SRC) and the Olivetti Research Center (ORC) in the late 1980s.

Modula-3's main features are simplicity and safety while preserving the power of a systems-programming language. Modula-3 aimed to continue the Pascal tradition of type safety, while introducing new constructs for practical real-world programming. In particular Modula-3 added support for generic programming (similar to templates), multithreading, exception handling, garbage collection, object-oriented programming, partial revelation, and explicit marking of unsafe code. The design goal of Modula-3 was a language that implements the most important features of modern imperative programming languages in quite basic forms. Thus allegedly dangerous and complicating features such as multiple inheritance and operator overloading were omitted.

Contents

Modula-3

Paradigm	imperative, structured, procedural, modular, concurrent
Family	Wirth Modula
Designed by	Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan; Bill Kalsow, Greg Nelson
Developers	DEC Olivetti elego Software Solutions GmbH
First appeared	1988
Stable release	5.8.6 / July 14, 2010
Preview release	5.8.6 / July 14, 2010
Typing discipline	strong, static, safe or if unsafe explicitly safe isolated
Platform	IA-32, x86-64, PowerPC, SPARC

OS	Cross-platform: FreeBSD, Linux, Darwin, SunOS
Website	www.modula3.org (http://www.modula3.org)
Major implementations	
SRC Modula-3, CM3,[1] PM3,[2] EZM3,[3] M3/PC Klagenfurt[4]	
Influenced by	
ALGOL, Euclid, Mesa, Modula-2, Modula-2+, Oberon, Pascal	
Influenced	
C#, Java, Nim,[5] OCaml, Python[6]	

Historical development

The Modula-3 project started in November 1986 when Maurice Wilkes wrote to Niklaus Wirth with some ideas for a new version of Modula. Wilkes had been working at DEC just prior to this point, and had returned to England and joined Olivetti's Research Strategy Board. Wirth had already moved on to Oberon, but had no problems with Wilkes's team continuing development under the Modula name. The language definition was completed in August 1988, and an updated version in January 1989. Compilers from DEC and Olivetti soon followed, and 3rd party implementations after that.

Its design was heavily influenced by work on the Modula-2+ language in use at SRC and at the Acorn Computers Research Center (ARC, later ORC when Olivetti acquired Acorn) at the time, which was the language in which the operating system for the DEC Firefly multiprocessor VAX workstation was written and in which the Acorn Compiler for Acorn C and Modula Execution Library (CAMEL) at ARC for the ARX operating system project of ARM based Acorn Archimedes range of computers was written. As the revised Modula-3 Report states, the language was influenced by other languages such as Mesa, Cedar, Object Pascal, Oberon and Euclid.[8]

During the 1990s, Modula-3 gained considerable currency as a teaching language, but it was never widely adopted for industrial use. Contributing to this may have been the demise of DEC, a key Modula-3 supporter (especially when it ceased to maintain it effectively before DEC was sold to Compaq in 1998). In any case, in spite of Modula-3's simplicity and power, it appears that there was little demand for a procedural compiled language with restricted implementation of object-oriented programming. For a time, a commercial compiler named CM3 maintained by one of the chief implementors prior at DEC SRC who was hired before DEC being sold to Compaq, an integrated development environment (IDE) named Reactor and an extensible Java virtual machine (licensed in binary code and source code formats and buildable with Reactor) were offered by Critical Mass, Inc., but that company ceased active operations in 2000 and gave some of the source code of its products to elego Software Solutions GmbH. Modula-3 is now taught in universities mostly in comparative programming language courses, and its textbooks are out of print. Essentially the only corporate supporter of Modula-3 is elego, which inherited the sources from Critical Mass and has since made several releases of the CM3 system in source and binary code. The Reactor IDE has been open source released after several years it had not, with the new name CM3-IDE. In March 2002, elego also took over the repository of another active Modula-3 distribution, PM3, until then maintained at the École Polytechnique de Montréal but which later continued by the work on HM3 improved over the years later until it was obsoleted.

Syntax

A common example of a language's syntax is the "Hello, World!" program.

```
MODULE Main;
IMPORT IO;
BEGIN
  IO.Put("Hello World\n")
END Main.
```

All programs in Modula-3 have at least a module file, while most also include an interface file that is used by clients to access data from the module. As in other languages, a Modula-3 program must export a Main module, which can either be a file named Main.m3, or a file can call `EXPORT` to export the Main module.

```
MODULE Foo EXPORTS Main
```

Module file names are advised to be the same as the name in source code. If they differ, the compiler only emits a warning.

Other conventions in the syntax include naming the exported type of an interface `T`, since types are usually qualified by their full names, so a type `T` inside a module named Foo will be named `Foo.T`. This aids in readability. Another similar convention is naming a public object `Public` as in the OOP examples above.

Language features

Modularity

First and foremost, all compiled units are either `INTERFACE` or implementation `MODULE`s, of one flavor or another. An interface compiled unit, starting with the keyword `INTERFACE`, defines constants, types, variables, exceptions, and procedures. The implementation module, starting with the keyword `MODULE`, provides the code, and any further constants, types, or variables needed to implement the interface. By default, an implementation module will implement the interface of the same name, but a module may explicitly `EXPORT` to a module not of the same name. For example, the main program exports an implementation module for the Main interface.

```
MODULE HelloWorld EXPORTS Main;
IMPORT IO;
BEGIN
  IO.Put("Hello World\n")
END HelloWorld.
```

Any compiled unit may `IMPORT` other interfaces, although circular imports are forbidden. This may be resolved by doing the import from the implementation MODULE. The entities within the imported module may be imported, instead of only the module name, using the `FROM Module IMPORT Item [, Item]*` syntax:

```
MODULE HelloWorld EXPORTS Main;
FROM IO IMPORT Put;
BEGIN
  Put("Hello World\n")
END HelloWorld.
```

Typically, one only imports the interface, and uses the 'dot' notation to access the items within the interface (similar to accessing the fields within a record). A typical use is to define one data structure (record or object) per interface along with any support procedures. Here the main type will get the name 'T', and one uses as in `MyModule.T`.

In the event of a name collision between an imported module and other entity within the module, the reserved word `AS` can be used as in `IMPORT CollidingModule AS X;`

Safe vs unsafe

Some ability is deemed unsafe, where the compiler can no longer guarantee that results will be consistent; for example, when interfacing to the C language. The keyword `UNSAFE` prefixed in front of `INTERFACE` or `MODULE`, may be used to tell the compiler to enable certain low level features of the language. For example, an unsafe operation is bypassing the type system using `LOOPHOLE` to copy the bits of an integer into a floating point `REAL` number.

An interface that imports an unsafe module must also be unsafe. A safe interface may be exported by an unsafe implementation module. This is the typical use when interfacing to external libraries, where two interfaces are built: one unsafe, the other safe.

Generics

A generic interface and its corresponding generic module, prefix the `INTERFACE` or `MODULE` keyword with `GENERIC`, and take as formal arguments other interfaces. Thus (like C++ templates) one can easily define and use abstract data types, but unlike C++, the granularity is at the module level. An interface is passed to the generic interface and implementation modules as arguments, and the compiler will generate concrete modules.

For example, one could define a GenericStack, then instantiate it with interfaces such as `IntegerElem`, or `RealElem`, or even interfaces to Objects, as long as each of those interfaces defines the properties needed by the generic modules.

The bare types `INTEGER`, or `REAL` can't be used, because they are not modules, and the system of generics is based on using modules as arguments. By comparison, in a C++ template, a bare type would be used.

FILE: IntegerElem.i3

```
INTERFACE IntegerElem;
CONST Name = "Integer";
TYPE T = INTEGER;
PROCEDURE Format(x: T): TEXT;
PROCEDURE Scan(txt: TEXT; VAR x: T): BOOLEAN;
END IntegerElem.
```

FILE: GenericStack.ig

```
GENERIC INTERFACE GenericStack(Element);
(* Here Element.T is the type to be stored in the generic stack. *)
TYPE
   T = Public OBJECT;
   Public = OBJECT
   METHODS
       init(): TStack;
       format(): TEXT;
       isEmpty(): BOOLEAN;
       count(): INTEGER;
       push(elm: Element.T);
       pop(VAR elem: Element.T): BOOLEAN;
   END;
END GenericStack.
```

FILE: GenericStack.mg

```
GENERIC MODULE GenericStack(Element);
< ... generic implementation details... >
```

```
  PROCEDURE Format(self: T): TEXT =
  VAR
     str: TEXT;
  BEGIN
     str := Element.Name & "Stack{";
     FOR k := 0 TO self.n -1 DO
         IF k > 0 THEN str := str & ", "; END;
             str := str & Element.Format(self.arr[k]);
     END;
     str := str & "};";
     RETURN str;
  END Format;
  < ... more generic implementation details... >
  END GenericStack.
```

FILE: IntegerStack.i3

```
INTERFACE IntegerStack = GenericStack(IntegerElem) END IntegerStack.
```

FILE: IntegerStack.m3

```
MODULE IntegerStack = GenericStack(IntegerElem) END IntegerStack.
```

Traceability

Any identifier can be traced back to where it originated, unlike the 'include' feature of other languages. A compiled unit must import identifiers from other compiled units, using an `IMPORT` statement. Even enumerations make use of the same 'dot' notation as used when accessing a field of a record.

```
INTERFACE A;

TYPE Color = {Black, Brown, Red, Orange, Yellow, Green, Blue, Violet, Gray, White};

END A;
```

```
MODULE B;

IMPORT A;
FROM A IMPORT Color;

VAR
  aColor: A.Color;  (* Uses the module name as a prefix *)
  theColor: Color;  (* Does not have the module name as a prefix *)
  anotherColor: A.Color;

BEGIN
  aColor := A.Color.Brown;
  theColor := Color.Red;
  anotherColor := Color.Orange;  (* Can't simply use Orange *)
END B.
```

Dynamic allocation

Modula-3 supports the allocation of data at runtime. There are two kinds of memory that can be allocated, `TRACED` and `UNTRACED`, the difference being whether the garbage collector can see it or not. `NEW()` is used to allocate data of either of these classes of memory. In an `UNSAFE` module, `DISPOSE` is available to free untraced memory.

Object-oriented

Object-oriented programming techniques may be used in Modula-3, but their use is not needed. Many of the other features provided in Modula-3 (modules, generics) can usually take the place of object-orientation.

Object support is intentionally kept to its simplest terms. An object type (termed a "class" in other object-oriented languages) is introduced with the `OBJECT` declaration, which has essentially the same syntax as a `RECORD` declaration, although an object type is a reference type, whereas RECORDs in Modula-3 are not (similar to structs in C). Exported types are usually named T by convention, and create a separate "Public" type to expose the methods and data. For example:

```
INTERFACE Person;

TYPE T <: Public;
  Public = OBJECT
  METHODS
    getAge(): INTEGER;
    init(name: TEXT; age: INTEGER): T;
  END;

END Person.
```

This defines an interface `Person` with two types, `T`, and `Public`, which is defined as an object with two methods, `getAge()` and `init()`. `T` is defined as a subtype of `Public` by the use of the `<:` operator.

To create a new `Person.T` object, use the built in procedure `NEW` with the method `init()` as

```
VAR jim := NEW(Person.T).init("Jim", 25);
```

Modula-3's `REVEAL` construct provides a conceptually simple and clean yet very powerful mechanism for hiding implementation details from clients, with arbitrarily many levels of *friendliness*. Use `REVEAL` to show the full implementation of the `Person` interface from above.

```
MODULE Person;

REVEAL T = Public BRANDED
OBJECT
  name: TEXT;   (* These two variables *)
  age: INTEGER; (* are private. *)
OVERRIDES
  getAge := Age;
  init := Init;
END;

PROCEDURE Age(self: T): INTEGER =
  BEGIN
    RETURN self.age;
  END Age;

PROCEDURE Init(self: T; name: TEXT; age: INTEGER): T =
  BEGIN
    self.name := name;
    self.age := age;
  RETURN self;
  END Init;

BEGIN
END Person.
```

Note the use of the `BRANDED` keyword, which "brands" objects to make them unique as to avoid structural equivalence. `BRANDED` can also take a string as an argument, but when omitted, a unique string is generated for you.

Modula-3 is one of a few programming languages which requires external references from a module to be strictly qualified. That is, a reference in module `A` to the object `x` exported from module `B` must take the form `B.x`. In Modula-3, it is impossible to import *all exported names* from a module.

Because of the language's requirements on name qualification and method overriding, it is impossible to break a working program simply by adding new declarations to an interface (any interface). This makes it possible for large programs to be edited concurrently by many programmers with no worries about naming conflicts; and it also makes it possible to edit core language libraries with the firm knowledge that no extant program will be *broken* in the process.

Exceptions

Exception handling is based on a `TRY...EXCEPT` block system, which has since become common. One feature that has not been adopted in other languages, with the notable exceptions of Delphi, Python[1] (https://www.python.org/doc/faq/general/#why-was-python-created-in-the-first-place), Scala[2] (http://scala.epfl.ch) and Visual Basic.NET, is that the `EXCEPT` construct defined a form of switch statement with each possible exception as a case in its own EXCEPT clause. Modula-3 also supports a `LOOP...EXIT...END` construct that loops until an `EXIT` occurs, a structure equivalent to a simple loop inside a `TRY...EXCEPT` clause.

Multi-threaded

The language supports the use of multi-threading, and synchronization between threads. There is a standard module within the runtime library (*m3core*) named Thread, which supports the use of multi-threaded applications. The Modula-3 runtime may make use of a separate thread for internal tasks such as garbage collection.

A built-in data structure `MUTEX` is used to synchronize multiple threads and protect data structures from simultaneous access with possible corruption or race conditions. The `LOCK` statement introduces a block in which the mutex is locked. Unlocking a `MUTEX` is implicit by the code execution locus's leaving the block. The `MUTEX` is an object, and as such, other objects may be derived from it.

For example, in the input/output (I/O) section of the library *libm3*, readers and writers (Rd.T, and Wr.T) are derived from MUTEX, and they lock themselves before accessing or modifying any internal data such as buffers.

Summary

In summary, the language features:

- Modules and interfaces
- Explicit marking of unsafe code
- Generics
- Automatic garbage collection
- Strong typing, structural equivalence of types
- Objects

- Exceptions
- Threads

Modula-3 is one of the rare languages whose evolution of features is documented.

In *Systems Programming with Modula-3*, four essential points of the language design are intensively discussed. These topics are: structural vs. name equivalence, subtyping rules, generic modules, and parameter modes like `READONLY`.

Standard library features

Continuing a trend started with the C language, many of the features needed to write real programs were left out of the language definition and instead provided via a standard library set. Most of the interfaces below are described in detail in[9]

Standard libraries providing the following features. These are called standard interfaces and are required (must be provided) in the language.

- Text: Operations on immutable string references, called `TEXT`s
- Thread: Operations relating to threading, including `MUTEX`, condition variable, and thread pausing. The threading library provides pre-emptive threads switching
- Word: Bitwise operations on unsigned integers (or machine words). Normally implemented directly by the compiler
- Floating-point interfaces

Some recommended interfaces implemented in the available implementations but are not required

- Lex: For parsing number and other data
- Fmt: Formatting various datatypes for printing
- Pkl (or Pickle): Object serialization of any reference types reachable by the garbage collector
- Table: Generic modules for maps

As in C, I/O is also provided via libraries, in Modula-3 called `Rd` and `Wr`. The object-oriented design of the Rd (readers) and Wr (writers) libraries is covered in detail in the book by Greg Nelson. An interesting aspect of Modula-3 is that it is one of few programming languages which standard libraries have been formally verified not to contain various types of bugs, including locking bugs. This was done under the auspices of the Larch/Modula-3 (see Larch family)[10] and Extended static checking[11] projects at DEC Systems Research Center.

Implementations

Several compilers are available, most of them open source.

- DEC-SRC M3, the original.[12]
- Olivetti Research Center (ORC) Modula-3 toolkit, originally a compiler, now available as a library for syntactic, lexical and semantic analysis of Modula-3 programs.[13]
- Critical Mass CM3, a different successor of DEC-SRC M3
- Polytechnique Montreal Modula-3 PM3, a successor of DEC-SRC M3, currently merging with CM3

- EzM3, an independent lightweight and easily portable implementation, developed in connection with CVSup
- HM3, a successor of the pm3-1.1.15 release of PM3, with support of native threading using NPTL
- CM3, the successor to Critical Mass CM3. This is the only up to date, maintained and developed implementation. Releases are available from http://www.opencm3.net/releng/.

Since the only aspect of C data structures that is missing from Modula-3 is the union type, all extant Modula-3 implementations are able to provide good binary code compatibility with C language type declarations of arrays and structs.

Books

None of these books are still in print, although used copies are obtainable and some are digitized, partly or fully, and some chapters of one them have prior or posterior versions obtainable as research reports from the web.

- Greg Nelson, ed., *Systems Programming with Modula-3* The definitive reference on the Modula-3 language with interesting articles on object-oriented systems software construction and a documentation of the discussion leading to the final features of the language. There are some formerly (see[8] for Chapter two,[14] for chapter four,[15] for chapter five,[16] for chapter six) and some posteriorly (see[17] for Chapter one and more updated two, thus of both prior versions of language definition[8] and,[9] for chapter three and[18] for chapter seven) of publishing versions of the majority of its eight chapters individually available from prior DEC Systems Research Center (SRC) as research reports for download.
- Samuel P. Harbison, *Modula-3* Easy to use class textbook.
- Robert Sedgewick, *Algorithms in Modula-3*
- Laszlo Boszormenyi & Carsten Weich, *Programming in Modula-3: An Introduction in Programming with Style*
- Renzo Orsini, Agostino Cortesi *Programmare in Modula-3: introduzione alla programmazione imperativa e a oggetti* an Italian book of the language explaining its main features.

Projects using Modula-3

Software which is programmed Modula-3 includes:

- The SPIN operating system
- The CVSup software repository synchronizing program
- The Obliq language, which uses Modula-3 network objects ability to migrate objects over local networks transparently, allowing a distributed ability to Modula-3 object-oriented programming paradigm. It has been used to build distributed applications, computer animations, and web programming applications in the form of scripting extension to Modula-3.

Influences on other programming languages

Although Modula-3 did not gain mainstream status, several parts of the DEC-SRC M3 distribution did. Probably the most influential part was the Network Objects library, which formed the basis for Java's first Remote Method Invocation (RMI) implementation, including the network protocol. Only when Sun moved from the Common Object Request Broker Architecture (CORBA) standard to the IIOP based protocol was it

dropped. The Java documentation on garbage collection of remote objects still refer to the pioneering work done for Modula-3 Network Objects.[19] Python's implementation of classes was also inspired by the class mechanism found in C++ and Modula-3.[20] Also the language Nim makes use of some aspects of Modula-3, such as traced vs untraced pointers.

References

1. "Critical Mass Modula-3 (CM3)" (https://modula3.elegosoft.com/cm3/). *Critical Mass Modula-3*. elego Software Solutions GmbH. Retrieved 2020-03-21.
2. "Polytechnique Montréal Modula-3 (PM3): What is it" (https://modula3.elegosoft.com/pm3/). *Polytechnique Montréal Modula-3*. elego Software Solutions GmbH. Retrieved 2020-03-21.
3. Polstra, John D. (November 9, 2006). "Ezm3: An Easier Modula-3 Distribution" (https://web.archive.org/web/20130410151327/http://www.cvsup.org/ezm3/). *CVSup.org*. Archived from the original (http://www.cvsup.org/ezm3/) on April 10, 2013. Retrieved 2020-03-21.
4. Weich, Carsten. "M3/PC Klagenfurt 96: a Modula-3 environment for MS-DOS" (https://web.archive.org/web/20000520073936/http://www.ifi.uni-klu.ac.at/Modula-3/m3pc/m3pc.html). *Department of Informatics*. University of Klagenfurt. Archived from the original (http://www.ifi.uni-klu.ac.at/Modula-3/m3pc/m3pc.html) on 20 May 2000. Retrieved 2020-03-21.
5. Picheta, Dominik; Locurcio, Hugo. "Frequently Asked Questions" (http://nim-lang.org/question.html). Retrieved 2020-03-21.
6. van Rossum, Guido (May 1996). "Programming Python: Foreword (1st ed.)" (https://www.python.org/doc/essays/foreword/). *Python.org*. Retrieved 2020-03-21.
7. "Design and History FAQ: Why must 'self' be used explicitly in method definitions and calls?" (https://docs.python.org/3/faq/design.html#why-self). *Python.org*. March 21, 2020. Retrieved 2020-03-21.
8. Modula-3 report (revised) (http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-52.html) Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. DEC Systems Research Center (SRC) Research Report 52 (November 1989)
9. Some Useful Modula-3 Interfaces (http://apotheca.hpl.hp.com/ftp/pub/dec/SRC/research-reports/SRC-113.pdf) Jim Horning, Bill Kalsow, Paul McJones, Greg Nelson. DEC Systems Research Center (SRC) Research Report 113 (December 1993)
10. LM3 (http://apotheca.hpl.hp.com/ftp/pub/dec/SRC/research-reports/SRC-072.pdf) Kevin D. Jones. DEC Systems Research Center (SRC) Research Report 72 (June 1991)
11. Extended Static Checking (http://apotheca.hpl.hp.com/ftp/pub/dec/SRC/research-reports/SRC-159.pdf) David L. Detlefs, K. Rustan M. Leino, Greg Nelson, James B. Saxe. Compaq SRC Research Report 159 (December 1998)
12. SRC Modula-3 3.3 (ftp://ftp.u-aizu.ac.jp/pub/lang/Modula/m3/faq/document/src-m3-doc/SRCm3-3.3.ps.gz) Bill Kalsow and Eric Muller. Digital Equipment Corporation (January 1995)
13. Jordan, Mick (1990). "An extensible programming environment for Modula-3" (http://portal.acm.org/citation.cfm?id=99285). *SIGSOFT Softw. Eng. Notes*. **15** (6): 66–76. doi:10.1145/99278.99285 (https://doi.org/10.1145%2F99278.99285). Retrieved 2009-09-08.
14. An Introduction to Programming with Threads (http://apotheca.hpl.hp.com/ftp/pub/dec/SRC/research-reports/SRC-035.pdf) Andrew D. Birrell. DEC Systems Research Center (SRC) Research Report 35 (January 1989)
15. Synchronization Primitives for a Multiprocessor: A Formal Specification (http://apotheca.hpl.hp.com/ftp/pub/dec/SRC/research-reports/SRC-020.pdf) A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin. DEC Systems Research Center (SRC) Research Report 20 (August 1987)
16. IO Streams: Abstract Types, Real Programs (http://apotheca.hpl.hp.com/ftp/pub/dec/SRC/research-reports/SRC-053.pdf) Mark R. Brown and Greg Nelson. DEC Systems Research Center (SRC) Research Report 53 (November 1989)

17. Modula-3 Reference Manual (http://www.minet.uni-jena.de/www/fakultaet/schukat/Inf1/Praktikum/Modula-3-refman.ps) Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. DEC Systems Research Center (SRC) (February 1995)
18. Trestle Tutorial (http://apotheca.hpl.hp.com/ftp/pub/dec/SRC/research-reports/SRC-069.pdf) Mark S. Manasse and Greg Nelson. DEC Systems Research Center (SRC) Research Report 69 (May 1992)
19. *Garbage Collection of Remote Objects* (http://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmi-arch4.html), Java Remote Method Invocation Documentation for Java SE 8.
20. *Classes* (https://docs.python.org/3/tutorial/classes.html#classes), Official Python Documentation.

External links

- Official website (http://www.modula3.org)
- Modula3 (https://github.com/modula3) on GitHub
- CM3 Implementation Website (http://www.opencm3.net)
- Modula-3 Home Page (http://www.research.compaq.com/SRC/modula-3/html/home.html) (now long dead, mirror (https://web.archive.org/web/20050220025439/http://www.research.compaq.com/SRC/modula-3/html/home.html))
- Modula-3: Language definition (http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-52.html)
- elego Software Solutions (http://www.elegosoft.com)
- Modula-3 newsgroup (news:comp.lang.modula3), mostly deserted
- Modula-3 Development Mailing List (https://web.archive.org/web/20120326171529/https://mail.elegosoft.com/cgi-bin/mailman/listinfo/m3devel), active
- Notes from Caltech's CS2 class, taught in Modula-3 in 2002 and 2003 (https://web.archive.org/web/20091203151748/http://www.mikanystrom.com/cs2/)
- Caltech's CS3 class 2009 (https://web.archive.org/web/20130523220501/http://www.ugcs.caltech.edu/~se/) at the Wayback Machine (archived May 23, 2013)
- mirror *Programming in Modula-3*: program examples (https://web.archive.org/web/19970814162826/http://www.ifi.uni-klu.ac.at/Modula-3/m3book/examples.html)
- *Building Distributed OO Applications: Modula-3 Objects at Work*. Michel R. Dagenais. Draft Version (January 1997) (https://web.archive.org/web/20120227030339/http://www.professeurs.polymtl.ca/michel.dagenais/pkg/BDAM3alpha.ps)
- *Modula-3: Language, Libraries and Tools*. Presentation on Modula-3 over 120 slides. Michael R. Dagenais (ftp://ftp.u-aizu.ac.jp/pub/lang/Modula/m3/faq/document/LangToolsLibs/root.ps), dead
- *Object-Oriented Data Abstraction in Modula-3*. Joseph Bergin (1997) (http://csis.pace.edu/~bergin/M3text)
- Computerworld Interview with Luca Cardelli on Modula-3 (http://www.techworld.com.au/article/252531/a-z_programming_languages_modula-3/)