An Overview of Objective Modula-2

Benjamin Kowarsch
Objective Modula-2 Project Founder
http://objective.modula2.net

Abstract

This document provides an overview of *Objective Modula-2*, a reflective, object oriented programming language which features both static and dynamic typing.

Objective Modula-2 is a hybrid of Modula-2 and Smalltalk, where Objective-C served as a design blueprint. It retains most of Modula-2's features and syntax, most importantly data encapsulation via modules, the use of explicit import lists and strong type checking for statically defined data objects, while all of its object oriented features are derived from Objective-C, a hybrid of C and Smalltalk. With the exception of the Smalltalk derived message passing syntax, which has been incorporated into *Objective Modula-2* as is, all other Objective-C derived language features have been cast into a Modula-2 like syntax.

Objective Modula-2 is designed to use the Objective-C runtime library and it has all the capabilities of Objective-C. Classes written in *Objective Modula-2* can be used within Objective-C and vice versa. Yet, by virtue of its core language Modula-2 being a type safe language, it can be considered a safer language than Objective-C.

Objective Modula-2 is predominantly aimed at Cocoa, Cocoa Touch and GNUstep development and in particular at software developers who would like to make use of the Cocoa, Cocoa Touch and GNUstep APIs from within a type-safe programming language in general or from within a language of the Algol family of programming languages in particular.

The Building Blocks of Objective Modula-2

Objective Modula-2 is a "thin" layer on top of Modula-2 R10, a modern revision of Modula-2 based on a subset of Wirth's fourth edition of *Programming in Modula-2*.

However, the *Objective Modula-2* language extension layer on top of the R10 core language has been designed to be dialect agnostic. It may be implemented on top of any other Modula-2 dialect without design conflicts and without inconsistencies. The extension layer is very lightweight, it may be added without significantly increasing the complexity of an implementation. The following sections are organised into two parts. Part I provides an overview of the changes in the core language relative to Wirth's fourth edition of *Programming in Modula-2* and Part II provides a description of the *Objective Modula-2* extension layer itself.

PART I • The Core Language

Objective Modula-2 is a superset of Modula-2 R10, a modern revision of Wirth's fourth edition of *Programming in Modula-2*. Various features have been omitted while others have been added in their place and some semantics have been revised.

Omitted Features

- No local modules
- No variant records
- No `EXPORT` statement
- No `WITH DO` statement
- No type conversion functions

Replaced Features

- Radix 2 and radix 16 replace radix 8 in literals
- Pragma delimiters `<*` and `*>` replace `(*$` and `*)`
- Type conversion operator `::` replaces type conversion functions

Added Features

- Atomic intrinsics
- Language defined pragmas
- Structured value constructors
- Extensible enumeration and record types
- Import qualifiers for import-all and re-export
- Built-in type `UNICHAR` for unicode characters
- Immutable function parameters and pointer target types
- Concatenation of string literals using the + operator
- Escape sequences `\0`, `\n`, `\r`, `\t`, `\\`, `\'`, `\"` in string literals
- Additional operators for type conversion, increment and decrement
- Type safe foreign function interface to C, incl. type safe variadic parameter passing

Revised Semantics

- Array indices always start at zero
- Variables are always exported immutable
- Strict name equivalence and reduced use of anonymous types
- `CHAR` literals are assignment compatible with `ARRAY OF CHAR`
- Named elements of sets and enumerations must always be qualified

Binary Literals

Binary literals have radix 2 and are denoted by prefix `0b` or suffix `B`:

```
VAR b : INTEGER;
b := 00100111B; b := 0b00100111;
```

Listing 1: binary literals

Character Code Literals

Character code literals have radix 16 and are denoted by prefix `0u` or suffix `U`:

```
VAR c : CHAR; u : UNICHAR;
c := 02EU; c := 0u2e;
u := 0B3F4U; u := 0ub3f4; (* values > 127 are of type UNICHAR *)
```

Listing 2: character code literals

String Literals

String literals may be concatenated and they may include escape sequences:

```
VAR s : ARRAY 80 OF CHAR;
s := “The quick brown fox\n” + “jumps over the lazy dog”;
```

Listing 3: string literal concatenation and use of escape sequences

Structured Literals

Structured literals may be created using structured value constructors:

```
CONST
  zeroVector = { 0.0, 0.0, 0.0, 0.0 };
  zeroArray = { 0 BY 10 };
TYPE
  Vector = RECORD a, b, c, d : REAL END;
  Array = ARRAY 10 OF INTEGER;
VAR
  v1, v2 : Vector; a1, a2 : Array;
BEGIN
  v1 := zeroVector; v2 := { 1.0, 2.0, 3.0, 4.0 };
  a1 := zeroArray; a2 := { 123, 456, 0 BY 8 };
```

Listing 4: structured value constructors

Extensible Enumeration Types

Enumeration types are extensible.

```
TYPE
  Foo = ( foo, bar, baz );
  Bar = ( +Foo, bam, boo );
  (* Bar is defined as ( foo, bar, baz, bam, boo ) *)
```

Listing 5: declaring an enumeration type by extending another enumeration type

Extensible Record Types

Record types are extensible.

```
TYPE
  Foo = RECORD foo, bar, baz : INTEGER END; (* Foo *)
  Bar = RECORD ( Foo )   (* inherits fields from Foo *)
    bam, boo : CHAR  (* adds own fields bam and boo *)
  END; (* Bar *)
```

Listing 6: declaring a record type by extending another record type

Referencing Named Set and Enumeration Type Elements

Due to a design flaw in PIM and ISO Modula-2, name conflicts can occur when importing enumeration types. This problem has been fixed by requiring named elements of sets and enumeration types to always be referenced qualified.

Given the following type declarations:

```
TYPE
  Status = ( failure, success );
  Colour = SET OF ( red, green, blue );
```

Listing 7: declaring an enumeration and and a set of an anonymous enumeration

the elements of types `Status` and `Colour` must be referenced qualified:

```
foo := Status.failure; bar := Status.success;
baz[Colour.red] := TRUE; baz[Colour.green] := FALSE;
CONST AllColours = { Colour.red, Colour.green, Colour.blue };
```

Listing 8: qualifying enumeration and named set elements

Import-all Wildcard and Re-export Qualifier

```
FROM Cocoa IMPORT *; (* import all items from library Cocoa *)
IMPORT Foo+, Bar+; (* import and re-export Foo and Bar *)
```

Listing 9: using import -all wildcard and re-export qualifier

Immutable Export of Variables

Variables declared within a definition module are exported as immutable objects. An imported variable may not be assigned to within the scope of an importing module.

Immutable Procedure Parameters

Formal parameters may be declared immutable in the header of a procedure. An immutable parameter may not be assigned to within the scope of the procedure.

```
PROCEDURE Foobar (CONST foo : Foo; CONST bar : Bar)
```

Listing 10: immutable procedure parameters

Immutable Pointer type targets

Target types of a pointer type may be declared immutable. Variables of pointer types which point to immutable targets may be modified but the compiler will enforce that the data such variables point to cannot be modified through such pointer variables.

```
TYPE FooPtr = POINTER TO CONST Foo;
```

Listing 11: declaring a pointer type pointing to an immutable target

Alias Types and Derived Types

```
TYPE INT = ALIAS OF INTEGER; (* compatible types *)
TYPE Celsius = REAL; Fahrenheit = REAL; (* incompatible types *)
```

Listing 12: declaration of alias types and derived types

Safe Type Conversion

```
r := i :: REAL; (* converting the value of i to type REAL *)
i := i * r :: INTEGER; (* converting r to type INTEGER *)
```

Listing 13: using the type conversion operator

Variadic Procedures with Counter Terminated Variadic Parameter Lists

The number of variadic parameters of a counter terminated variadic parameter list is automatically passed as a hidden parameter immediately before the variadic list.

```
PROCEDURE Variadic(v : VARIADIC OF (p1 : Bar; p2 : Baz));
BEGIN
  (* iterate over all variadic tuples by index *)
  FOR n OF CARDINAL := 0 to HIGH(v) DO
    DoSomethingWithBar(v[n].p1);
    DoSomethingWithBaz(v[n].p2);
  END; (* FOR *)
END Variadic;
```

Listing 14: procedure with a counter terminated variadic parameter list

Variadic Procedures with Value Terminated Variadic Parameter Lists

A value terminated variadic parameter list is terminated by a given constant value.

```
PROCEDURE Variadic(v : VARIADIC [-1] OF (p1 : INT; p2 : Baz));
BEGIN (* v points to first variadic tuple *)
  WHILE v # NIL DO (* while there are tuples *)
    DoSomethingWithINT(v^.p1);
    DoSomethingWithBaz(v^.p2);
    v := NEXTV(v); (* next variadic tuple *)
  END; (* WHILE *)
END Variadic;
```

Listing 15: procedure with a value terminated variadic parameter list

Calling a Variadic Procedure

Index arguments and value terminating arguments are never supplied in the actual parameter list of a variadic procedure call. The compiler automatically determines and inserts these arguments into the procedure's activation record.

```
(* calling procedure Variadic from Listing 14 *)
Variadic( (* counter omitted *) foo1, bar2, foo2, bar2 );
(* calling procedure Variadic from Listing 15 *)
Variadic( foo1, bar1, foo2, bar2 (* -1 omitted *) );
```

Listing 16: variadic procedure calls omit index and value terminating arguments

Pervasive Identifiers

Pervasive identifiers are identifiers which are language defined and thus built-in. They do not need to be declared nor imported and are available anywhere in a program or library. Unlike reserved words, pervasive identifiers may be redefined.

Built-in Constants

```
NIL : null pointer value
TRUE, FALSE : boolean values
```

Listing 17: built-in constants

Built-in Types

```
BOOLEAN, BITSET, LONGBITSET, CHAR, UNICHAR,
OCTET, CARDINAL, INTEGER, REAL, LONGCARD, LONGINT, LONGREAL;
```

Listing 18: built-in types

Built-in Functions

```
ABS(x) returns the absolute value of x
NEG(x) returns the sign reversed value of x
ODD(x) returns TRUE if x is an odd number
ORD(x) returns ordinal value of x, x is of ordinal type
CHR(x) returns character, CHAR if x < 128, else UNICHAR
PRED(x, n) returns n-th predecessor of x
SUCC(x, n) returns n-th successor of x
SIZE(v) returns allocated size of variable v
HIGH(a) returns highest index of open array or variadic list
COUNT(s) returns number of items in set or collection
LENGTH(s) returns length of string s
NEXTV(v) returns pointer to next variadic tuple of v
TMAX(T) returns the maximum value for type T
TMIN(T) returns the minimum value for type T
TSIZE(T) returns storage size needed for type T
```

Listing 19: built-in functions

Built-in Procedures

```
NEW(p) expands to ALLOCATE(p, TSIZE(p))
DISPOSE(p) expands to DEALLOCATE(p, TSIZE(p))
READ(f, x) expands to TYPEOF(x).Read(f, x)
WRITE(f, x) expands to TYPEOF(x).Write(f, x)
WRITEF(f, fmtStr, x) expands to TYPEOF(x).WriteF(f, fmtStr, x)
```

Listing 20: built-in procedures

Even though `NEW` and `DISPOSE` are built-in, the procedures `ALLOCATE` and `DEALLOCATE` must nevertheless be imported from a library module before use. Likewise, built-in `READ`, `WRITE` and `WRITEF` rely on the import of a module that provides the `Read`, `Write` and `WriteF` functions for the type of argument `x`.

Built-in Lexical Macros

```
MIN(c1, c2, c3 …) inserts smallest constant from argument list
MAX(c1, c2, c3 …) inserts largest constant from argument list
```

Listing 21: built-in macros

Built-in Intrinsics

More built-in constants, types, procedures and functions are available from pseudo modules `SYSTEM`, `ATOMIC` and `COROUTINES`. Although built-in, these entities must be imported before use. Module `SYSTEM` provides access to system dependent resources. Its use is potentially unsafe and *may* render a program non-portable:

```
OctetsPerByte, BytesPerWord, BitsPerMachineByte,
MachineBytesPerMachineWord, OctetsPerMachineWord,
BYTE, WORD, MACHINEBYTE, MACHINEWORD, ADDRESS,
ADR, CAST, SHL, SHR, ROTL, ROTR, INC, DEC,
BWNOT, BWAND, BWNAND, BWOR, BWNOR, BWXOR;
```

Module `ATOMIC` provides portable intrinsics for atomic operations:

```
INTRINSIC, AVAIL,
SWAP, CAS, INC, DEC, BWAND, BWNAND, BWOR, BWXOR;
```

Module `COROUTINES` provides portable intrinsics for concurrent programming:

```
Coroutine, CoroutineProc, New, Yield, Dispose;
```

Binding to Built-in Functions and Operators

To allow library defined types to be used in the same manner as built-in types, library defined procedures may be bound to built-in functions and operators. For instance, a library module may define a binary coded decimals type as follows:

```
DEFINITION MODULE BCD [RTYPE]; (* Binary Coded Decimals *)
FROM FileIO IMPORT File;
TYPE
  BCD = OPAQUE RECORD
      value : ARRAY 8 OF OCTET
  END;
```

Listing 22: library module defining type BCD for binary coded decimals

The specified prototype `[RTYPE]` in the module header determines which type of literal is assignment compatible with the type and what bindings are required. Prototypes are user definable but the standard library contains a number of predefined prototypes. Examples of predefined prototypes are `ZTYPE` for whole numbers, `RTYPE` for real numbers, `CTYPE` for complex numbers and `DateType` for date-time types.

The `BCD` module is then required to define a set of procedures for operations on values of type `BCD`, and bind them to the appropriate built-in functions and operators:

Binding to Type Range Operations

```
PROCEDURE [TMIN] minValue : BCD; (* TMIN(BCD) => minValue *)
PROCEDURE [TMAX] maxValue : BCD; (* TMAX(BCD) => maxValue *)
```

Listing 23: binding procedures to type range operations

Binding to the Assignment Operator

```
PROCEDURE [:=] assign ( VAR assignTo  : BCD;
                        CONST literal : ARRAY OF CHAR );
```

Listing 24: binding procedures to the assignment operator

Binding to the Type Conversion Operator

```
PROCEDURE [::] toCARD  ( a : BCD ) : CARDINAL;
PROCEDURE [::] toINT ( a : BCD ) : INTEGER;
PROCEDURE [::] toREAL ( a : BCD ) : REAL;
```

Listing 25: binding procedures to the type conversion operator

Binding to Unary Arithmetic Operations

```
PROCEDURE [ABS] abs ( a : BCD ) : BCD;
PROCEDURE [NEG] neg ( a : BCD ) : BCD;
PROCEDURE [ODD] odd ( a : BCD ) : BOOLEAN;
```

Listing 26: binding procedures to unary arithmetic operations

Binding to Binary Arithmetic Operators

```
PROCEDURE [+] add ( a, b : BCD ) : BCD;
PROCEDURE [-] sub ( a, b : BCD ) : BCD;
PROCEDURE [*] multiply ( a, b : BCD ) : BCD;
PROCEDURE [/] divide ( a, b : BCD ) : BCD;
```

Listing 27: binding procedures to binary arithmetic operators

Binding to Relational Operators

```
PROCEDURE [=] isEqual ( a, b : BCD ) : BOOLEAN
PROCEDURE [<] isLess ( a, b : BCD ) : BOOLEAN
PROCEDURE [>] isGreater ( a, b : BCD ) : BOOLEAN
```

Listing 28: binding procedures to binary arithmetic operators

Operations not-equal (#), less-or-equal (<=) and greater-or-equal (>=) are derived automatically from the procedures bound to operators equal, less and greater.

IO Operations

To facilitate the use of the built-in macros `READ`, `WRITE` and `WRITEF`, with library defined types, a set of corresponding IO procedures may be defined in the same module that defines the type. For instance:

```
PROCEDURE Read( infile : File; VAR a : BCD );
PROCEDURE Write( outfile : File; a : BCD );
PROCEDURE WriteF( f : File; fmtStr : ARRAY OF CHAR;
                  itemList : VARIADIC OF a : BCD );
```

Listing 29: defining IO procedures to be invoked in place of built-in IO macros

In the above example, any invocation of the built-in macros `READ`, `WRITE` and `WRITEF` with arguments of type `BCD` are then replaced by the compiler with the corresponding procedure calls `BCD.Read()`, `BCD.Write()` and `BCD.WriteF()`.

Pragmas

The following pragmas are language defined:

```
IF, ELSIF, ELSE, ENDIF, INFO, WARN, ERROR, FATAL,
ALIGN, FOREIGN, MAKE, INLINE, NOINLINE, VOLATILE;
```

Pragmas are always delimited by `<*` and `*>`.

Pragmas for Conditional Compilation

Pragmas `IF`, `ELSIF`, `ELSE` and `ENDIF` are provided for conditional compilation:

```
<*IF foo > bar *>
  foobar(foo);
<*ELSIF foo = bar *>
  barbaz(bar);
<*ELSE*>
  bam(baz);
<*ENDIF*>
```

Listing 30 conditional compilation

Pragmas to Generate Compile-Time Messages

Pragmas `INFO`, `WARN`, `ERROR` and `FATAL` are provided for compile-time messages:

```
<*FATAL "Error: Constant FOOBAR must not be larger than 100" *>
```

Listing 31: generating a compile-time message and abort compilation

Pragmas to Direct or Influence Code-Generation

```
<*ALIGN = TSIZE(SYSTEM.BYTE) *>  (* use byte alignment *)
<*FOREIGN = "C" *>  (* force C calling convention *)
<*INLINE*> PROCEDURE foo;  (* suggest to inline procedure *)
VAR <*VOLATILE*> port [0EFH] : OCTET;  (* volatile variable *)
```

Listing 32: compiler directives

Pragma to Direct the Modula-2 Make Utility or Build System

```
<*MAKE = "expand(file:stack, module:IntStack, type:INTEGER)" *>
```

Listing 33: make directive

PART II - The Objective Modula-2 Language Extension

Language extensions for object-oriented features, including method invocation syntax are derived from Objective-C, which itself derived its object model and associated method invocation syntax from Smalltalk.

Additional Pervasive Identifiers

Objective Modula-2 adds three additional pervasive identifiers:

Boolean Constants

```
YES, NO : synonyms for boolean values TRUE and FALSE
```

Listing 34: Objective Modula-2 adds built-in constants YES and NO

Type OBJECT

```
OBJECT : corresponds to type id in Objective-C
```

Listing 35: Objective Modula-2 adds built-in type OBJECT

Message Passing

Objective Modula-2 uses the Objective-C object model, which is based on the Smalltalk object model and thus distinct from the object model of Simula, followed by C++ and other programming languages. This distinction is semantically important. The main differences are that instead of "calling a method", one "sends a message" and parameters are interleaved within the method name. An object called `obj` whose class has a method `doSomething` implemented is said to "respond" to the message `doSomething`. The Objective-C based syntax for sending a `doSomething` message to `obj` is `[obj doSomething];` As an alternative, *Objective Modula-2* also provides a more Smalltalk-like syntax which does not use brackets:

```
foo := [[FooClass alloc] init]; (* Objective-C style syntax *)
foo := `FooClass alloc init; (* Smalltalk style alternative *)
```

Listing 36: message passing syntax

Dynamic Typing

A major difference to statically typed languages such as C++ and Java is that in *Objective Modula-2* it is possible to send messages to objects that do "not" respond to them. This is because the object oriented part of *Objective Modula-2* is dynamically typed, just like Objective-C. This means that it is possible to send a message to an object which does not have a method specified in its interface to respond to that message. This may seem like a bad idea, but in fact this allows for a great level of flexibility - in *Objective Modula-2* an object can "capture" this message, and depending on the object, it can pass the message on to a different object which can respond to the message correctly and appropriately, or pass the message on yet again. In Objective-C parlance this is called delegation, also referred to as "message forwarding". An error handler can be used in case the message cannot be forwarded. However if the object does not respond to the message, nor forward it, nor handle the error, then a runtime error occurs. An example of a dynamically typed object in *Objective Modula-2* is shown below:

```
VAR anObject : OBJECT;
```

Listing 37: declaring a dynamically typed object

The type `OBJECT` in *Objective Modula-2* is the equivalent of type `id` in Objective-C. Like other dynamically typed languages, there is the potential problem of an endless stream of runtime errors that come from sending the wrong message to the wrong object. However, *Objective Modula-2* allows the programmer to optionally specify the class of an object, and in such cases the compiler will treat the object as statically typed. An example of a statically typed object in *Objective Modula-2* is shown below:

```
VAR aString : NSString;
```

Listing 38: declaring a statically typed object

Unlike Objective-C where the declaration of a variable of a class type requires the class identifier to be referenced with a preceding `*` operator, in *Objective Modula-2* the class identifier is not preceded by `POINTER TO`.

Class Interfaces and Implementations

In *Objective Modula-2* interface and implementation of a class are located in separate files. A class' interface consists of a class declaration followed by the class' public method headers both of which are located in a definition module. The implementation of a class consists of corresponding method declarations and any method declarations for private methods which are located in the corresponding implementation module. Multiple classes may be defined within a single definition module. However, methods defined within a definition module must always be implemented within the corresponding implementation module.

Class Interfaces

The interface of a class is represented by a class declaration followed by method headers for the class' public methods. An example of a class declaration defining a class called `FooBar` as a descendant of superclass `NSObject` with initialiser, accessor and mutator methods is shown below:

```
DEFINITION MODULE FoobarLib;
FROM Cocoa IMPORT NSObject;
(* Define class FooBar as a subclass of NSObject *)
TYPE
  FooBar = CLASS ( NSObject )
  (* instance variables *)
    foo : Foo;
    bar : Bar;
  END;

  (* constructor and initialiser *)
  CLASS METHOD (self : FooBar) newWithFoo: (foo : Foo)
                                   andBar: (bar : Bar) : OBJECT;

  (* accessor for foo *)
  METHOD (self : FooBar) foo : Foo;

  (* mutator for foo *)
  METHOD (self : FooBar) setFoo: (foo : Foo);

END FoobarLib.
```

Listing 39: definition module with class interface

The Objective-C object model distinguishes between class methods and instance methods. Class methods operate on the class object of a class while instance methods operate on instance objects of the class. In *Objective Modula-2* a class method is declared by prepending `CLASS` before the method header. A method declaration without the `CLASS` modifier is always an instance method declaration.

Class Implementations

The implementation of a class is represented by an implementation module with method declarations of the class' public and private methods. The implementation must correspond to the definition module in which the class' interface was defined. An example of a class implementation corresponding to the class interface of the previous example is shown below:

```
IMPLEMENTATION MODULE FoobarLib;

(* constructor and initialiser *)
CLASS METHOD (self : FooBar) newWithFoo: (foo : Foo)
                                 andBar: (bar : Bar) : OBJECT;
VAR
  thisInstance : FooBar;
BEGIN
  thisInstance := [[FooBar alloc] init];
  thisInstance^.foo := foo;
  thisInstance^.bar := bar;
  RETURN thisInstance;
END newWithFoo: ;

(* accessor for foo *)
METHOD (self : FooBar) foo : Foo;
BEGIN
  RETURN self.foo;
END foo ;

(* mutator for foo *)
METHOD (self : FooBar) setFoo: (foo : Foo);
BEGIN
  self.foo := foo;
END setFoo: ;

END FoobarLib.
```

Listing 40: implementation module with class implementation

Method Invocation

The syntax for method headers is different from that of procedures. A procedure header in Modula-2 follows this general form:

```
PROCEDURE ProcedureName (parameter : FormalType) : ReturnedType;
```

Listing 41: procedure header

This syntax cannot be used for methods because of the way in which parameters are interleaved with the method name in the Smalltalk derived method invocation syntax. Therefore, additional syntax for declaring class and instance methods has been added as shown in the `FooBar` class example. This syntax allows to define methods which follow the Smalltalk derived notation for sending messages. An example of how the methods in the `FooBar` class are used is shown below:

```
MODULE UseFoobar;
FROM FoobarLib IMPORT FooBar;
CONST
  someFoo = 1; someBar = 2;
VAR
  foobar : FooBar;
BEGIN
  (* allocate and initialise a new instance of FooBar *)
  foobar := [FooBar newWithFoo:someFoo andBar:someBar];

  (* sending a foo message to foobar *)
  someFoo := [foobar foo];

  (* sending a setFoo: message to foobar *)
  [foobar setFoo:someFoo];
END UseFoobar.
```

Listing 42: program module using a class and its methods

Class Refinement

The Objective-C object model defines an instrument to refine an existing class by overloading existing methods or adding additional methods. In Objective-C parlance this is called a category. Categories permit the programmer to modify or add methods to an existing class without the need to recompile that class or even have access to its source code. Since all methods are added to a class at runtime, methods declared within categories are indistinguishable from methods declared within a class' interface, they have full access to all of the instance variables within the class, including private instance variables.

In *Objective Modula-2* a reference to the receiver class is part of a method declaration and therefore, method declarations can target any class regardless of where the class declaration is located. There is no need for special syntax to declare categories. The equivalent of an Objective-C category can be declared simply by placing method headers in a separate definition module and their declarations in a corresponding implementation module. An example of a definition module to declare a category is shown below:

```
DEFINITION MODULE AdditionsToNSString;
(* import the target class *)
FROM Cocoa IMPORT NSString;
  (* declare a new method to be added to the target class *)
METHOD (self:NSString) stringByCollapsingWhitespace : NSString;
END AdditionsToNSString.
```

Listing 43: definition module defining a method to be added to class NSString

An example of a corresponding implementation module for the category is shown below:

```
IMPLEMENTATION MODULE AdditionsToNSString;
FROM Cocoa IMPORT NSString;
METHOD (self:NSString) stringByCollapsingWhitespace : NSString;
    (* code to implement the method *)
  END stringByCollapsingWhitespace;
END AdditionsToNSString.
```

Listing 44: implementation module implementing a method to be added to class NSString

Multiple Inheritance

The Objective-C object model defines an instrument for multiple inheritance of specification (but not implementation). In Objective-C parlance this is called a protocol. There are two types of protocols: ad-hoc protocols, called *informal protocols*, and compiler enforced protocols called *formal protocols*.

An informal protocol is a list of methods that a class may implement. It is specified in the documentation and has no presence in the language. Informal protocols often include optional methods, where implementing the method can change the behavior of a class. For example, a text field class might have a delegate that should implement an informal protocol with an optional autocomplete method. When the text field discovers that the delegate implements that method, it calls the method to support autocomplete. Informal protocols are typically implemented using categories.

A formal protocol is a collection of method declarations that any given class may adopt by implementing the methods declared by the protocol. A class which adopts a protocol must implement all methods declared by that protocol unless the methods are explicitly declared optional. To allow the declaration of formal protocols, additional syntax has been added in *Objective Modula-2*. An example of a protocol module which declares a formal protocol called `Foobaring` is shown below:

```
PROTOCOL Foobaring;
(* declaration of a required method *)
METHOD (self : *) foo;
(* declarations of an optional method *)
OPTIONAL METHOD (self : *) bar;
END Foobaring.
```

Listing 45: protocol module

To declare classes adopting formal protocols, a comma separated list of references to the adopted protocols is placed after the super class reference in the class declaration. An example of a class declaration declaring a class `Foo` as a subclass of `NSObject` adopting the `Foobaring` protocol is shown below:

```
TYPE
  Foo = CLASS ( NSObject, Foobaring )
  (* instance variables *)
  END;
(* method declarations *)
```

Listing 46: class definition module adopting a protocol

Instance Variable Access Modes

By default, the instance variables of a class are visible and accessible to classes residing within the same link image. This is not always desirable.

The class itself and its categories always need to have direct access to the instance variables, but outside of the class and its categories, it is often desirable that they should only be accessible via accessor and mutator methods. In some cases it may however, be desirable to make them public to all.

In order to accommodate the afore mentioned scenarios, different access modes for instance variables are provided.

In *Objective Modula-2* there are at present four access modes[1] for instance variables: `PUBLIC`, `MODULE`, `PROTECTED` and `PRIVATE`. The default access mode is `MODULE`.

Access Mode	Qualifier	Visibility
Public	`PUBLIC`	anywhere
Module (aka Package)	`MODULE` (default)	within the class itself, its categories, subclasses and any classes declared in the same module
Protected	`PROTECTED`	within the class itself, its categories and subclasses
Private	`PRIVATE`	within the class itself and its categories

An example of instance variables with different access modes is shown below:

```
TYPE
  Foobar = CLASS ( NSObject )
    foo : Foo; (* Classes in the same module can access foo *)
    PUBLIC bar : Bar; (* Any class can access bar *)
    PROTECTED baz : Baz; (* Subclasses can access baz *)
    PRIVATE bam : Bam; (* Only class itself can access bam *)
  END; (* CLASS *)
```

Listing 47: instance variables with different access modes

[1] The equivalent access modes in Objective-C are called `@public`, `@package`, `@protected` and `@private`. However, the `@package` access mode was only introduced with Objective-C 2.0. Instance variables declared with access mode `@package` will instead have access mode `@public` when their classes are used from classes compiled under Objective-C prior to version 2.0.

Objective-C Runtime Exception Handling

Further syntax has been added in *Objective Modula-2* in order to support Objective-C runtime exception handling. An example is shown below:

```
TRY
  (* statements during which a runtime exception may occur *)
ON exception DO
  (* statements to be executed if specified exception occurs *)
CONTINUE
  (* statements to be executed in any event *)
END;
```

Listing 48: exception handling

Synchronising Thread Execution

Further syntax has been added in *Objective Modula-2* in order to support Objective-C runtime compatible thread synchronisation. An example is shown below:

```
METHOD (self : Foo) doSomeCriticalStuff : Bar;
  (* non-critical code *)
  CRITICAL ( self )
    (* critical code *)
  END;
  (* non-critical code *)
END doSomeCriticalStuff;
```

Listing 49: synchronising a thread

Mapping of Modula-2 Name Spaces to the Global Objective-C Name Space

In Modula-2, name conflicts are avoided by qualified import of objects with identical names, which are then referenced by qualified identifier. An example is shown below:

```
IMPORT FooLib, BarLib, BazLib;
FooLib.write(); BarLib.write(); BazLib.write();
```

Listing 50: qualified import in Modula-2

To avoid name conflicts, Modula-2 compilers export all names in a qualified fashion, that is, they combine the name of the compilation unit with the identifier of constants, types, variables and procedures, a process referred to as name mangling.

Unfortunately, there is a compatibility issue with qualified class identifiers and the Objective-C runtime because Objective-C does not qualify class names and it does not understand mangled names. For compatibility with Objective-C class libraries, an Objective Modula-2 compiler needs to export and import class names unqualified by default. This means that two classes of the same name, declared in different library modules will cause a name collision when imported into the same library or program:

```
DEFINITION MODULE FooLib;
FROM Cocoa IMPORT NSObject;
TYPE FooClass = CLASS ( NSObject ) … END;
END Foolib.

DEFINITION MODULE BarLib;
FROM Cocoa IMPORT NSObject;
TYPE FooClass = CLASS ( NSObject ) … END;
END BarLib.

MODULE Foobar;
FROM FooLib IMPORT FooClass;
FROM BarLib IMPORT FooClass; (* expected name collision *)
END Foobar.

MODULE Barbaz;
IMPORT FooLib, BarLib; (* unexpected name collision *)
END Barbaz.
```

Listing 51: by default classes are exported unqualified

However, *Objective Modula-2* provides a compiler pragma to override the default and export classes qualified like constants, non-class types, variables and procedures:

```
DEFINITION MODULE FooLib;
FROM Cocoa IMPORT NSObject;
TYPE FooClass = <*QUALIFIED*> CLASS ( NSObject ) … END;
END FooLib.

MODULE Barbaz;
IMPORT FooLib, BarLib; (* no name collision *)
END Barbaz.
```

Listing 52: import of qualified classes in Objective Modula-2

Classes which have been declared qualified can then be imported qualified and referenced by qualified identifiers:

```
DEFINITION MODULE FooLib;
FROM Cocoa IMPORT NSObject;
TYPE FooClass = <*QUALIFIED*> CLASS ( NSObject ) … END;
END FooLib.

DEFINITION MODULE BarLib;
FROM Cocoa IMPORT NSObject;
TYPE FooClass = <*QUALIFIED*> CLASS ( NSObject ) … END;
END BarLib.

MODULE Foobar;
IMPORT FooLib, BarLib;
VAR
  foo : FooLib.FooClass := [[FooLib.FooClass alloc] init];
  bar : BarLib.FooClass := [[BarLib.FooClass alloc] init];
END Foobar.
```

Listing 53: qualified class identifiers

Unfortunately, the resulting qualified class identifiers cannot be referenced the same way from classes written in Objective-C. Any such reference to the class within Objective-C code needs to use the mangled class name verbatim:

```
#import “FooLib.h”
#import “BarLib.h”
FooLib$FooClass *foo = [[FooLib$FooClass alloc] init];
BarLib$BarClass *bar = [[BarLib$FooClass alloc] init];
```

Listing 54: inability of Objective-C to unmangle name-mangled symbols

However, the C preprocessor may be used to define a better readable alias name:

```
#import “ClassLib.h”
#define FooClass FooLib$FooClass
#define BarClass BarLib$FooClass
FooClass *foo = [[FooClass alloc] init];
BarClass *bar = [[BarClass alloc] init];
```

Listing 55: using C preprocessor macros to unmangle name-mangled symbols in Objective-C

Appendix A: Outlook on Future Removals

The following language features are being considered for removal:

FOR TO BY statement

The `FOR TO BY` statement of the Modula-2 R10 core language is being considered for removal in favour of an enhanced `FOR IN` statement.

Appendix B: Outlook on Future Additions

Further additions to the extension layer are being considered, most notably Smalltalk-style blocks (aka closures) and per-object selection of garbage collection.

Blocks, aka Closures

To maintain orthogonality with Objective-C 2.0, an additional built-in data type `BLOCK` is likely to be introduced in support of Smalltalk-style blocks (aka closures):

```
VAR b : BLOCK;
b := DO foobar(x); x++ END;
```

Listing 56: possible closure support

Per Object Selection of Garbage Collection

It is desirable to support automatic garbage collection through adding methods and classes, similar to the way autorelease pools are implemented in Objective-C. This would allow the programmer to choose a memory management scheme on a per object basis. Objects could be manually managed, autoreleased or garbage collected simply by sending a message. An example how objects could be declared using different coexisting memory management schemes is shown below:

```
VAR
  (* declaring a garbage collected object *)
  gcString : NSString := [[[NSString alloc] init] gc];
  (* declaring an autorelease pool object *)
  arString : NSString := [[[NSString alloc] init] autorelease];
```

Listing 57: possible per object garbage collection

With Objective-C 2.0, Apple Inc. has introduced automatic garbage collection into its Objective-C runtime and it is possible to mix reference counted memory management and automatic garbage collection, but whether or not per-object selection of the memory management is feasible will require further study.

Appendix C: Project Related Information

Modula-2 R10 Language Description

A language description of the Modula-2 R10 dialect is available online at:

http://modula2.net/resources/M2R10.pdf

http://bitbucket.org/trijezdci/m2r10stdlib/src/tip/_LANGUAGE/Language.txt

Modula-2 R10 Standard Library Reference

A full reference of the Modula-2 R10 standard library is available online at:

http://bitbucket.org/trijezdci/m2r10stdlib/src

Objective Modula-2 Reference Compiler

Experimental translators for *Objective Modula-2* for the purpose of testing language design concepts have been under development since 2006. As the language definition has matured, a new compiler, derived from earlier experimental compilers, is being developed with the aim to produce a reference implementation for the general public. The compiler uses a recursive descent parser and it will generate Objective-C source code. An LLVM back-end is also under development.

Source code for the reference compiler is being made available under a peer-review license until the grammar has been finalised and the front-end implementation is complete. The source code will then be relicensed under a BSD style license.

Further Reading

http://objective.modula2.net

http://objective.modula2.net/grammar.shtml

Document License

This document is released under the Creative Commons License.