OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

The Design of the Programming Language X-2

David W. Sandberg
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

85-60-1

The Design of the Programming Language X-2

David W. Sandberg
Oregon State University

The design of an expermental object-based programming language is discussed. The language is intented for investigating techniques for organization of programs.

---

1. Introduction

The complexity of most current programming languages and their compilers makes it impossible to easily test new programming language concepts. It takes many years to design and implement a programming language. This paper describes the design of the programming language X-2 that is intended for testing new language concepts in object based programming. X-2 has few concepts to reduce its complexity but contains support for static strong typing, information hiding, parameterized types, and a programming environment. This support allows new ideas to be tested against current programming concepts. Other languages are either too complex like Ada[5] or omit some feature as Smalltalk[7] omits strong typing.

X-2 was designed primarily for testing concepts for organization of programs that will facilitate the reuse of their parts. Other goals were to gain experience with a Smalltalk like user interface and to investigate the feasibility of using a byte-code interpreter that treats activation records the same as other objects for the purpose of storage management. Obtaining activation records from the heap makes handling multiple stacks for multiple process easy. Smalltalk interpreters treat activation records as objects, but many tricks are needed to obtain reasonable speed on most processors[10].

In this paper the design requirements and philosophicall underpinnings of X-2 are given first followed by a brief description of the syntax and semantics of the language. The paper concludes with a discussion of specific constructs which were included in or omitted from X-2.

2. DESIGN REQUIREMENTS

To add a new concept to a language, one must understand its interaction with the rest of the language. The fewer concepts in a language, the easier the interaction will be

to understand. Hence, one design requirement of X-2 is that it be as small as possible. (There are also other reasons for desiring small languages[11].) A construct was not added to X-2 unless the absence of that construct became annoying. Two examples are enumeration types whose absence never became annoying enough for them to be added to X-2 and the by-reference parameter mode whose absence became annoying. The implementation and design of X-2 were interleaved. This allowed programs to be written in X-2 to determine which constructs would be needed in the language.

Several design requirements came from the experience of designing another language, Lithe[15]. Lithe has several programming concepts which worked very well and were to be retained: the class-instance model, access to low-level machine instructions, parameterized classes, and iterators. There were several things that were to be included in X-2 that are not in Lithe but should have been: garbage collection, loops with exit, and procedure types. A useful feature of Lithe similar to those found in LISP systems is a command interpreter that allowed statements of the language to be executed interactively. A programming environment more like that of Smalltalk was desired to carry these ideas further. X-2 was to allow experimentation with concurrency, although no processes have been added to X-2 yet. The user can greatly modify the syntax of Lithe. This allowed more flexibility than was needed, so X-2 was to have a more rigid syntax. The implementation was to be transportable and to demonstrate that a usable system could really be built.

3. THE PHILOSOPHY OF PROGRAMMING USED IN DESIGNING X-2

The task of programming can be viewed as writing a specification of a task for a computer to perform. The first draft of a specification usually is incomplete and incorrect and does not specify what was intended. The specification must then be modified to match the intent. A programming language should assist in writing these specifications. This view differs from the common view associated with abstract data types, in which the programming language is viewed mainly as a way to implement the specifications. The specification is assumed to be written in a different language.

There are several approaches to writing a specification; an existing specification can be modified, a specification can be built from large prefabricated pieces, or it can be built from primitive pieces. Programming is usually done by the third method. The other two methods require that an existing program be understood which is a difficult task if the program is written in most existing languages;

in fact, it is often easier to start over than to use what already exists. Starting over wastes effort in duplication. Several things can be done to make the programs easier to understand. Simple things, that is easy to understand, should be chosen over complex things whenever possible. Programs should be as uniform as possible within a single program and between different programs. If the same technique or part has been used many times, much less effort is needed to understand it in yet another program because it is familiar. The way abstractions organize a program greatly influences whether the pieces of a program can be reused. Abstraction has two components: suppression of detail and generalization. If the wrong details are suppressed or if the generalization takes the wrong direction, an abstraction can be of little use in other programs.

Since a programming language itself is a specification, it should have the same characteristics of simplicity and uniformity. The easiest way to get these characteristics is to choose a simple, powerful model of computation. For example, LISP[13] uses lambda calculus and lists as its model, Snobol[8] uses pattern matching, Prolog[4] uses theorem proving, and Smalltalk uses the class-instance model with message passing. X-2 also uses the class-instance model. This model has proved to be quite powerful in languages such as Simula[8], Smalltalk, CLU[12], and Lithe.

To understand the design of X-2, some knowledge of X-2 is needed. The next section gives an informal introduction to X-2 by means of an example.

4. A Brief Introduction to X2

X2 is an object based language that uses the class model. The classes replace the notion of a type in Pascal-like languages. The class model is an abstract model of the world. The class model has three characteristics:

1. Every entity of the world is an object, no matter what its function.

2. Every object is an instance of an unique class. The class of an object determines the object's behavior; that is, the operations defined on the object.

3. Each instance contains some state (or memory).

Smalltalk uses message passing to implement operations on objects. This is not a fundamental part of the class-instance model as the operations can be implemented as procedure calls just as well. Latter versions of Smalltalk

```
---------------------------------
class:
 t list
 first-t,tail-(t list)


add: x-t to: l-t list ref
free[t]   $ Allows t to be replaced by any class.
locals:
 k-t list
body:
 k <- create, k first <- x, k tail <- l
 l <- k

empty return: t list
free[t]
body:
 nil return

for: i-t ref in: l-t list do: s-statement delay
free[t]
body:
 loop:
  if: l isempty then: exit
  i <- l first
  s
  l <- l tail

(x-t list) isempty return: bool
free[t]
body:
 (x eq: nil) return
```

Figure 1

---

have a class hierarchy which allows many classes to share the same code. No such hierarchy exists in X-2. This is partly because the parameterized classes of X-2 removed some of the need for the hierarchy and partly because X-2 was intended for testing other possible organizations of classes.

Figure 1 defines a parameterized class for linked lists with operations for creating an empty list, adding to a list, testing for an empty list, and a "for" loop for stepping through the elements of a list. The details of figure 1 are explained in the next sections.

4.1. Syntax

The syntax of X-2 is similar to that of Smalltalk. The simplest expression is just a reference to some object which is an operator with no arguments. For example:

```
x       nil       empty
```

Unary operators follow a single argument:

```
x list                  l isempty
```

To conform to standard practice, X-2 has one unary prefix operator "-" which is equivalent to the postfix operator "minus"; that is, "-8" is equivalent to "8 minus".

General operators consist of one or more arguments separated by selectors. All the selectors are concatenated to form the operator name:

```
add: 3 to: 1              $ operator is "add:to:"
x eq: nil                 $ operator is "eq:"
free: t                   $ operator is "free:"
if: a then: b else: c     $ operator is "if:then:else:"
```

There are some binary operations that are abbreviations for general operators. For example:

```
8+x    is the same as    8 add: x
x<-3   is the same as    x gets: 3
8*3    is the same as    8 mult: 3
x=y    is the same as    x equal: y
```

These infix operators decrease the number of parentheses needed in expressions. There is also a notation to describe an operator that takes a variable number of arguments which are all of the same type:

```
list[3,4,8,9]      list["abc"]      block[a<-b, b<-c]
```

The order of precedence from highest to lowest of these different operators is:

```
unary operators
the operators "* and "/"
the prefix unary "-"
the operators "+" and "-"
relational binary operators such as "="
the general operators
the assignment operator, "<-"
```

To avoid confusion, precedence and associativity conform to normal usage.

The loop statement in figure 1 is an equivalent form to:

```
loop:block[if:1 isempty then:exit,i<-1 first,s,1<-1 tail]
```

Since indentation is significant, the above line can be expressed in two as:

```
loop:
 if: 1 isempty then: exit, i<- 1 first, s, 1<- 1 tail
```

When each statement is put on a separate line as in the figure, the comma must be left out between statements because they are automatically inserted at the end of the line. To suppress the automatic insertion of a comma, begin the line with a "#" at the proper indentation level. Automatic insertion of commas is included because they are often accidentally omitted.

4.2. Defining Lists in X-2.

Figure 1 defines linked lists over any class; that is, list of integers, list of reals, list of list of reals, or list of any other class. If only list of integers were to be defined, the following class definition would be used:

```
class:
 list, first-int,tail-list
```

This defines a new class named "list" whose instances have two parts to their state: an integer part named "first" and a list named "tail". This also automatically defines a special instance of the class list that has no state named "nil". In this example, nil will be used to represent the empty list.

The definition of the class in the figure defines more general lists by parameterizing the class with a parameter named "t". When a variable of class list is defined the parameter must be specified, for example, "1-int list" or "1-int array list". The procedure definitions in figure 1 have a line: "free[t]". This line indicates that when the procedure is used the t in the definition can be replaced by any type, as long as all occurrences in a single use are replaced by the same type.

The procedure "add:to:" in figure 1 uses "create" which creates a new instance of the class. Note that "k first" is used instead of the more familiar "k.first" to access components.

Parameters in X-2 are normally passed by-value. In the "add:to" procedure the second operator is passed by-reference by adding "ref" as an unary operator to the type. If "delay" was used instead of "ref", the parameter would be passed by name as in Algol 60.

Using delay parameters allows CLU-like iterators to be defined. An iterator is a generalization of a "for" statement. Instead of stepping through a sequence of integers, an iterator steps through the elements of a data structure such as a tree or list. The "for:in:do:" procedure of figure 1 is an iterator for lists. This iterator works by assigning each value in the list to the parameter i and then evaluating the delay parameter s. Since i is passed by reference, the parameter s is evaluated with a different value for i each time it is evaluated.

Here is a sample procedure using lists:

```
test
locals:
 L-int list, i-int
body:
 L<-empty
 for: i from: 1 to: 10 do:
  add: i to: L
 for: i in: L do:
  write: i, write: " "
```

This procedure will output:

```
10 9 8 7 6 4 3 2 1
```

5. DESIGN ISSUES

5.1. Encapsulation

Any modern programming language must deal with the issue of information hiding. An encapsulation unit (which we refer to as a package) is a mechanism that walls off a section of a program and controls what names are visible inside and outside the wall. Ada's packages, Modula's modules[15], and CLU's clusters are examples of encapsulation units. One issue in designing an encapsulation mechanism is whether there should be an one-to-one correspondence between the encapsulation units and the class definitions. Sometimes it is useful to wall off a set of procedures that define no classes. Also, allowing only one class per package is awkward when defining closely related classes, such as a class for the header of a list and one for the elements

of a list, since a wall is introduced between the classes[13]. A more useful encapsulation unit will allow any number of classes per unit.

Another issue in encapsulation is whether one should control both imports and exports from a package. Controlling the imports, that is, what identifiers are visible inside an unit, limits the amount of information needed to understand the package. If the package only imports a few other packages then only those packages need to be examined to understand the package which uses them. If the imports are not controlled the whole system must be considered. Exports are controlled to hide the internal details of the package so they are not a concern in understanding the rest of the system. By default nothing should be imported and nothing exported to encourage the programmer to only import what is needed inside the package and only export what is needed in the rest of the system.

Yet another issue in encapsulation is whether the import and export lists should be lists of packages or lists of specific procedures. One of the advantages of using a list of packages is that one obtains a whole set of related procedures whereas if one is required to list each procedure separately, one procedure is likely to be forgotten. Another problem with requiring a list of specific procedures is that these lists become very long. Also if overloading is present, a procedure name may not be enough to uniquely identify a procedure. To uniquely specify the procedure the types of the parameters also must be given.

Encapsulation is really a successor to nesting of procedures for scope control[3]. For this reason nesting was left out of X-2.

The encapsulation in X-2 is achieved by the use of contexts, which are sets containing other contexts, classes, and procedures. The same procedure or context may appear in more than one context. These contexts are built using a context editor provided in the programming environment. To encapsulate a section of program two contexts are used: one for the imports and any procedures that are defined and a second for the exports. The use of contexts in X-2 gives a much finer control over visibility than do the encapsulation units found in languages such as Ada, CLU, and Modula. Usually a package has only one view from the outside and one from the inside. In X-2 it is possible to construct a context which contains any set of identifiers. This allows several contexts to be constructed so a package can have many different views.

There are several difficulties with contexts in X-2. Each procedure can have a different context, which produces

an overwhelming number of contexts. To keep the number of contexts manageable the same context should be shared by many procedures. Another problem arises when trying to produce a linear, textual form of a program. The context structure is a general graph structure which does not have a simple linear representation. Also, because a procedure can appear in many contexts there is no way to tell in which context a procedure is defined. This makes it difficult to distinguish between the application program and the system.

5.2. Parameterized Classes.

One powerful mechanism for abstraction is parameterization. This allows a single abstraction to be used in more places because the parameters allow variations. In languages like Modula-2 and Pascal[8], a list of integers and a list of reals must be defined with two separate types. The only real difference between these definitions is that the type name has changed from integer to real. It would be useful to parameterize this type and thus only have one definition. Then one could use a "int list" and a "real list". These parameterized types should behave much like unparameterized types since they are only a generalization. The Ada generic type concept does not behave like the type in Ada. An extra "instantiation" step is required with the generic type.

X-2 only allows types to be used as parameters to types. Other parameters to types such as integers were not allowed because the interactions became difficult to understand. Arrays can be considered as parameterized types. Most languages require the size of an array to be specified in the declaration of an array. This requires an integer parameter to the type. Thus arrays in X-2 can only have the base type as a parameter; the size of an array must be a runtime field of the array. This allows the size of the array to be easily specified at run time instead of at compile time as is required in Pascal. This makes arrays more useful, for often the array bounds are unknown at compile time.

Parameterized types also are useful in defining the basic concept of a variable. A parameterized class "name" is used to represent the address of a variable. The parameter of name gives the type of value stored at that address. When a variable of type T is declared, it is given a class "T name". "T name" represents the l-value of the variable and "T" the r-value. A statement such as "x<-x+1" can be considered as a set of procedure calls, namely: "x gets: (x

dref add: 1)". The headings of these procedures would be:

```
x return: int name

(a-int add: b-int) return: int

(a-y name) dref return: y
free[y]

a-y name gets: b-y
free[y]
```

The extra clause on the last two headings means that the y can be replaced by any type to get a valid procedure which allows one rule to describe all assignments. If x were a global variable, a procedure could be written that would allow access to the value only:

```
xrvalue return: int
body: x return
```

or a procedure that would allow getting or changing the value:

```
xlvalue return: int name
body: x return
```

The procedure xlvalue can be treated just as any other variable in a assignment statement: "xlvalue<-xlvalue+1". Dereferencing is done implicitly. Thus procedures can return l-values.

The parameterized type name can also be used to pass a parameter by reference. For example, consider a procedure to increment an integer:

```
inc: x-int name
body:
 x<-x dref dref +1
```

In X-2 parameters are treated as initialized local variables. Thus a parameter definition "y-int" will produce a procedure to reference it like:

```
y return: int name
```

The parameter definition "x-int name" will produce:

```
x return: int name name
```

Thus "x" must be dereferenced twice to get the value. X-2 only implicitly dereferences a variable once to simplify the implementation. Using one implicit and one explicit

dereference does not work because X-2 does not know whether to apply the explicit dereference before or after the implicit one and will complain about the ambiguity. Even if X-2 allowed two implicit dereferences there would still be ambiguities in some cases, for example, with "x<-x" which can be interpreted as "x dref <- x dref dref" or "x <- x dref".

To avoid having to use explicit dereferencing with call-by-reference parameters an explicit call-by-reference parameter mode was added. This allows the increment procedure to be written as:

```
inc: x-int ref
body:
 x<-x+1
```

The procedure will still be viewed when it is used as if it were declared as before, but in the definition of the procedure the parameter "x-int ref" will produce a procedure which returns the value of the parameter:

```
x return: int name
```

This allows a reference parameter to be used like other local variables.

An alternate way to remove explicit dereferences from by-reference parameters is to treat parameters as constants instead of variables, but this introduces a distinction between where value parameters and local variables can be used.

5.3. ITERATORS

When defining abstract data types iterators are very useful. Consider the follow two program fragments to write out the elements of a linked list.

```
loop:
 if: (p eq: nil) then: exit
 write: p first, p<-p tail
```

```
i<-0
loop:
 if: i<=p size then: exit
 write: (p index: i), i<-i+1
```

In the second fragment, "index:" is the subscript operator for arrays. The two underlying implementations of linked lists are quite obvious from the code fragments. On the other hand, if an iterator is used, the same program

fragment would be used no matter what the implementation:

```
for: i in: p do:
 write: i
```

Call-by-name parameters were added to X-2 mainly for defining iterators as was done in figure 1.

One would like to generalize the above writing of a list by using a procedure that would write out a list no matter what the element type of the list. The following does not quite work:

```
write: l- t list
free[t]
local:
 i-t list
body:
 for: i in: p do:
  write: i          $ will produce an error message
```

The "write: i" in this procedure is referring to a specific procedure for a specific type. Some way is needed to specify the properties of t that must be present to use this procedure. CLU and Lithe provide such mechanisms, but no such mechanism has yet been added to X-2.

5.4. OTHER SEMANTIC ISSUES.

X-2 assumes there is a good garbage collector built into the system. Garbage collection errors are among the most difficult errors to detect and find. Errors tend to be detected long after they occur and then small changes to the program may make the evidence of the error disappear completely. Also, the programmer may produce a much less readable program if he deals with the garbage collection himself.

Loop-with-exit is the only primitive looping construct contained in X-2. The loop-with-exit produces more readable programs than just while loops. At the moment while loops are not include in X-2. It may be that while loops are used often enough and are enough clearer than a loop-with-exit to justify their inclusion.

A case statement is included in X-2 because it is more readable than a sequence of if-then's and can more easily be compiled into efficient code.

Any large system needs some way to recover from addressing errors and arithmetic traps. Languages such as CLU and ADA provide extensive exception handling mechanisms,

but there is some doubt as to the wisdom of this[2]. X-2 provides only a very limited mechanism. A way is provided to pair a block of code with an error handler. If any error arises while executing this block of code, control is passed to the error handler. A string is saved telling what error occurred. The user is allowed to generate an error and pass a string telling what error occurred.

X-2 runs on an interpreter. Access is given to the machine instructions of this interpreter. This has two advantages: new instructions can be added to the interpreter without

---

```
class:
 t array

(a-t array) size return: int
free[t]
instruction: 6 arg: 15  $ this instruction gives the size of
                        $ the space allocated

(newarray: size-int) return: t array
free[t]
$creates a new array
instruction: 6 arg: 18  $ This instruction allocates
                        $ space off the heap.

(a-t array index: i-int) return: t name
free[t]
instruction: 37 arg: 0  $ Forms the address of the ith
$ word in the segment of space allocated off the heap.

(expand: a-t array by:i-int) return: t array
free[t]
$This procedure creates a bigger array and
$copies the old array to the bigger one.
locals:
 b-t array,j-int
body:
   b <- newarray: a size+i
   for: j from: 0 to: a size-1 do:
    b index: j <- a index: j
   b return
```

Figure 2

---

changing the compiler for X-2 at all; and access to the instruction allows some basic data types to be defined instead of including them in the language definition. For example, there is no array type in the definition of X-2. Figure-2 shows how an array type can be defined. The use of numbers instead of mnemonics as the opcodes of the machine is poor practice, but has not been annoying enough to change, since access to the machine is used only infrequently.

Access to the machine also allows the strong typing of the language to be defeated. Type checking is valuable for catching many mistakes in a program, but sometimes type miss-matches is what the programmer really wanted. Some way should then be provided to defeat the strong typing.

A procedure type was included in X-2. A procedure can be stored in a data structure and later retrieved and called. A good example of where this is useful is in a window manager in a Smalltalk-like programming environment. The window manager needs a different set of instructions for displaying each different kind of window. The best representation of each set of instructions is a procedure. The window manager needs to store these procedures for later retrieval. In languages without a procedure type such as Ada and Pascal, a good window manager can be very difficult to write. Smalltalk does not use a procedure type but uses the fact that binding of procedure names to procedures is done at runtime.

5.5. Syntax

Several things were considered in designing the syntax for X-2. The user must be allowed to redefine the meaning of any existing syntax in the programming language. This is needed to blend user-defined extensions into the language and to allow the user to change the actual implementation of an abstraction type without changing the syntax. This is often impossible in Pascal; for example, if one used an array to start with, but later decided to use a sparse representation of an array, one is forced to go back and change the syntax because the meaning of "a[1]" can not be changed.

Too much user flexibility in the syntax has been avoided. If the user is allowed to radically change the syntax, the syntax must first be deciphered before the program can be understood. Changing the syntax will usually make little impact on a program because the semantics of the constructs is much more important.

A uniform syntax for both control structures and expressions was desired, so the programmer would not need to deal with a different syntax mechanism when defining control structures. Also the syntax of user-defined control structures should be no different from the predefined control structures.

Overloading of the meaning of a specific syntax was desired. This allows more concise and uniform naming conventions. For example, with overloading one name "write:" can be used for:

```
writeInt: i
writeBool: B
writeStack: s
writeReal: r
```

Which procedure is meant by "write:" is resolved by the types and number of the argument.

"<-" was used as the infix assignment operator instead of the more common ":=" to limit the amount of look-ahead the scanner must use. A two character look-ahead would be needed for ":=". Consider "x:=y" , "x: y", "xy+z", and "x+y". If "x" is the current character, one character look-ahead is needed to determine if the identifier continues. If the look-ahead is a colon, the character following the colon must be examined to determine whether the colon is part of the selector "x:" or an assignment operator.

Indentation was made significant to the compiler. The reason behind this is that because indentation is significant to the human reader it should be significant to the compiler. This prevents errors arising from the compiler interpreting the indentation one way and the reader another way. Using a more conventional treatment of indentation would affect the language very little.

6. Conclusion

X-2 has been implemented on a Motorola-68000-based system with a bit-mapped display. It took the author about one calendar year to write a simple programming environment for X-2 including a window manager, a text editor, separate compilation at the procedure level, a Baker-style garbage collector[1], and the compiler. This is all written in X-2 itself except for 20 pages of machine language that implement the basic interpreter. The time it took to write this implementation demonstrates that X-2 did attain some measure of simplicity.

X-2 is meeting the original design goals. The implementation is of reasonable speed which demonstrates the feasibility of using an interpreter that uses the same storage mechanism for activation records as other objects. X-2 has helped discover some problems with contexts mentioned earlier. Plans have been made to use X-2 further in investigating organization of programs.

7. REFERENCES

[1] Baker, H.G. List Processing in Real Time on a Serial Computer. Commun. ACM 21,4 (April 1978), 280-294.

[2] Black, A.P. Exception Handling: The Case Against. Thesis, Univ. of Oxford, January 1982. Also Tech. Rep. 82-01-02, Department of Computer Science, Univ. of Washington.

[3] Clarke, L.A., Wileden, J.C., and Wolf, A.L. Nesting in Ada Programs is for the Birds. Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language. Boston, December, 1980, 139-145. Published as Vol 15 No. 11 of SIGPLAN Notices.

[4] Clocksin, W.F., and Mellish, C.S. Programming in Prolog. Springer-Verlag, Berlin, 1981.

[5] Ada Programming Language, Department of Defense, Military Standard MIL-STD-1815A, January, 1983.

[6] Franta, W.R. The Process View of Simulation. North-Holland, New York, 1977.

[7] Goldberg, A., and Robson D. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, 1983.

[8] Griswold, R.E., Poage, J.F., and Polonsky, I.P. The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, 1971.

[9] Jensen, K., and Wirth, N. Pascal User Manual and Report, 2nd ed. Springer-Verlag, New York, 1974.

[10] Krasner, G., Ed. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley, 1983.

[11] Ledgard, H.F., and Singer, A. Scaling Down Ada. Commun. ACM 25,2(Feb. 1982), 121-125.

[12] Liskov, B, Snyder, A., Atkinson R., and Schaffert, C. Abstraction Mechanisms in CLU. Commun. ACM 20,8(Aug. 1977), 564-576.

[13] McCarty, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levein, M.I. LISP 1.5 Programmer's Manual, 2nd ed. MIT Press, Cambride, 1965.

[14] Rowe, L.A. Data Abstraction from a Programming Language Viewpoint. Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling, June 1980,29-35. Published as Vol 16 No. 1 of SIGPLAN Notices.

[15] Sandberg, D. W. A Language Combining a Flexible Syntax with Classes. Thesis, University of Washington, 1982. Also Tech. Rep. 82-12-03, Department of Computer Science, Univ. of Washington.

[16] Wirth, N. Programming in Modula-2, Springer-Verlag, New York, 1983.