Is Oberon as Simple as Possible? A Smaller Object-Oriented Language Based on the Concept of Module Type

Atanas Radenski

Department of Computer Science
Winston-Salem State University, P.O.Box 13027
Winston-Salem, North Carolina 27110, U.S.A.
E-mail: radenski@ecsvax.uncecs.edu

Abstract. The design of the programming language Oberon was led by the quote by Albert Einstein: 'make it as simple as possible, but not simpler'. The objective of this paper is to analyze some design solutions and propose alternatives which could both simplify and strengthen the language without making it simpler than possible.

The paper introduces one general concept, the module type, which can be used to represent records, modules, and eventually procedures. Type extension is redefined in terms of component nesting and incomplete designators. As a result, type extension supports multiple inheritance.

1 Introduction

The design of the programming language Oberon was led by the quote by Albert Einstein: 'make it as simple as possible, but not simpler'. The objective of this paper is to analyze some design solutions and propose alternatives which could both simplify and strengthen the language without making it simpler than possible.

The object orientation of Oberon is based on the concept of type extension. Section 2 of this paper outlines a problematic point in this concept as defined in Oberon: type extension applies to record and pointer types, but does not apply to procedure types. For this reason, procedures cannot be directly and conveniently redefined for extended types. As a consequence, method overriding may seem somewhat unnatural and tedious. This problematic point is eliminated with the concept of module type defined in Section 3. It is a generalization of record and procedure types and a single substitute for these types. As shown in Section 3, instances of module types can be used as record variables, or as procedures, or as Oberon modules. Overriding a method can be easily implemented by changing the module assigned to a field in an extension. Type extension itself is redefined in terms of component nesting and incomplete designators; as a result, it supports multiple inheritance.

Module types and type extension are integrated in an experimental object-oriented language that evolved from Oberon. The experimental language does not include record types, procedure types, procedures and modules, since all they are implemented by means of module types or module variables. The paper represents those features of the experimental language that are relevant to module types and type extension. The object orientation of this language is outlined in the end of Section 3.

2 The Need for Improvement

2.1 Type Extension as a Base of the Object Orientation of Oberon

Classes are implemented in Oberon as pointer types bound to record types with procedure variables. Objects are dynamic variables of such record types. For instance:

```
TYPE
 Class = POINTER TO ClassDesc;
 ClassDesc =  RECORD
  x : INTEGER;
  method : PROCEDURE (self : Class; v : INTEGER);
 END;
VAR
 ptr : Class;
```

Note that *ptr.x* and *ptr.method* designate the fields *x* and *method* of the dynamic record variable *ptr^*.

Methods are implemented in Oberon as procedures. For example, a method may look like this:

```
PROCEDURE Method (self : Class; v : INTEGER);
BEGIN self.x := v END Method;
```

To create a new object, one has to assign specific procedures to all procedure variables:

```
NEW (ptr); ptr.method := Method;
```

Messages are calls of procedure variables, as, for instance:

```
ptr.method(ptr, 1);
```

Inheritance in Oberon is based on the concept of type extension [2, 3]. It permits the construction of new record types by adding fields to existing ones. For instance, type *SubclassDesc* extends type *ClassDesc* with the data field *y*:

```
TYPE
```

```
 Subclass = POINTER TO SubclassDesc;
 SubclassDesc =  RECORD (ClassDesc)
   y : INTEGER
 END;
VAR
 subPtr : SubClass;
```

Type *SubclassDesc* is said to be a direct extension of type *ClassDesc*. Type *ClassDesc* is the direct base type of type *SubclassDesc*.

The fields of a record variable of an extended type can be referenced by usual field designators. For instance, *subPtr.x, subPtr.y, subPtr.method* are designators referencing the fields of the record variable *subPtr^*. A new object that belongs to *Subclass* can be created as follows:

```
NEW (subPtr); subPtr.method := Method;
```

An extended type is assignment compatible with its base type. For instance, the assignment *ptr^ := subPtr^* is legal and acts as a projection of record *subPtr^* onto record *ptr^*. The field *y* does not participate in the assignment. In contrary, the assignment *subPtr^ := ptr^* is illegal.

Type extension applies also to pointer types. By definition, the pointer type *Class* is extended by *Subclass* (see their declarations above), since the pointer base type *ClassDesc* of *Class* is extended by the pointer base type *SubclassDesc* of *Subclass*.

Since *subPtr* is an extension of *ptr*, the assignment *ptr := subPtr* is legal. After the assignment, *ptr* points to a dynamic variable of type *SubclassDesc*. After the assignment, *ptr* is said to be of dynamic type *Subclass*, while its declared (static) type continues to be *Class*. Thus, only *ptr.x* and *ptr.method* are accepted by the compiler as legal field designators. The field *y* can be referenced through *ptr* by means of a type guard, as illustrated by the following example:

```
ptr(Subclass).y := 0;
```

An attempt to execute the above statement when *ptr* does not actually point to a dynamic record of type *SubclassDesc* results in an abnormal halt. An abnormal halt can be prevented by a type test:

```
IF ptr IS Subclass THEN ptr(Subclass).y := 0 END;
```

2.2 What is Problematic with Type Extension

Overriding a method in Oberon can be implemented by changing the procedure assigned to a field in an extension [1]. Unfortunately, procedures cannot be directly and conveniently redefined for extended types. For this reason, method overriding may seem somewhat unnatural and tedious. Consider, for example, the following procedure:

```
PROCEDURE OverridingMethod (self : Subclass; v : INTEGER);
BEGIN
  self.x := v; self.y := v
END OverridingMethod;
```

To override *Method* with *OverridingMethod*, one may wish to use the assignment *subPtr.method := OverridingMethod*. However, the definition of Oberon implies that *OverridingMethod* is not assignment compatible with *method*, and this assignment is not allowed.

More precisely, field method of *SubclassDesc* is inherited from *ClassDesc* and has the following procedure type:

```
PROCEDURE (self : Class; v : INTEGER)
```

Besides, the heading of the newly created *OverridingMethod* is

```
PROCEDURE OverridingMethod (self : Subclass; v : INTEGER);
```

The type of the formal parameter *self* of *OverridingMethod*, namely *Subclass*, is an extension of the type indicated in the declaration of *method*, namely *Class*. According to the definition of type extension, the type of *OverridingMethod* is not an extension of the type of *method*. Thus, *OverridingMethod* is not assignment compatible with *subPtr.method*.

The following implementation of *OverridingMethod* can be assigned to *subPtr.method*, since now *OverridingMethod* and *subPtr.method* have a single formal parameter of the same type:

```
PROCEDURE OverridingMethod (self : Class; v : INTEGER);
BEGIN
  self.x := v;
  IF self IS SubClass THEN
    self(SubClass).y := v;
  END
END OverridingMethod;
```

Despite of the fact that the formal parameter of *OverridingMethod* is *Class*, it can and has to be called with actual parameters of type *SubClass*. By means of a type test and type guard, the overriding method treats the parameter as a variable of type *Subclass*. On the other end, the type of formal parameter of *OverridingMethod* is the same as that indicated for *subPtr.method*, and *OverridingMethod* can be assigned into *subPtr.method*. Such implementation of *OverridingMethod* seems somewhat unnatural and tedious.

3 Our Approach

A major problem with the object orientation of Oberon is that type extension applies to record and pointer types, but does not apply to procedure types. For this reason, methods cannot be directly and conveniently overridden for subclasses (see Section 2.2). The problem can be eliminated with the concept of module type defined in this section. Module types can be viewed as generalized record types. As shown in what follows, instances of module types can be used as record variables, or as procedures, or as Oberon modules. Overriding a method can be easily implemented by changing the module assigned to a field in an extension.

Module types and type extension are integrated in an experimental object-oriented language named K2 that evolved from Oberon. K2 does not include record types, procedure types, procedures and modules, since all they are implemented by means of module types or module variables. This section represents all features of the experimental language that are relevant to module types and type extension. The object orientation of this language is outlined in the end of the section.

3.1 Module Types

A *module type* consists of a definition, and optionally, a body. A *module definition* is a collection of declarations of constants, types, and variables. A *module body* is a collection of declarations, other bodies, and a sequence of statements. The statements are executed when the body is activated through a module call (Section 3.6). The definition of a global identifier and/or its body may include an import list (Section 3.7). A module type allows a body only if its definition contains a forward body declaration. Then a body can be declared within the same scope, or it can be left undefined (Section 3.4).

```
ModuleDefinition =
        "("[ImportList]
          DeclarationSequence
          [ForwardBodyDeclaration]
        ")"
DeclarationSequence = {declaration ";"}
declaration = ConstantDeclaration | TypeDeclaration |
VariableDeclaration
ForwardBodyDeclaration = BODY
BodyDeclaration =
        BODY ident ";"
          [ImportList]
          DeclarationSequence
          BodySequence
         [BEGIN
          StatementSequence]
        END ident
BodySequence = {BodyDeclaration ";"}
```

Examples:

```
TYPE Date = (day, month, year: INTEGER);
TYPE PersonalRecord = (
  CONST length = 32;
  TYPE Name = ARRAY length OF CHAR;
  name, firstName: Name;
  age: INTEGER
);
```

Constants, types and variables declared in a module definition are called *public components*, while those declared in the corresponding body are referred to as *local components*. Public components that are variables are also referred to as *parameters* (see Section also 3.6). Public types and constants are not parameters.

Example:

```
TYPE Sample = (
  publicVar: INTEGER;
  BODY
);
BODY Sample;
  localVar: INTEGER;
BEGIN (* ... *) END Sample;
```

The *scope* of an identifier which denotes a public component includes the module definition itself and the whole body, if any. Such an identifier is also visible within component designators. An identifier which declares a local component is not visible outside of the body that contains its declaration. Local variables keep their values between two successive calls of the body.

In addition to its public components and locally declared components, the entities declared in the environment of the body and its definition are also visible in the body. A local component hides non-local entities that have the same name. Hidden entities can still be referred to by component designators.

A variable declared in a module type definition can be followed by the read-only mark "-". Such a variable can be assigned values only from within the module body.

The identifier list of a variable declaration may contain the word *RESULT*. In this case, the type of the declared variable(s) can be neither a module type, nor an array type. Refer to Section 3.6 for the use of variables named *RESULT*.

Example:

```
TYPE Log2 = (
 x: INTEGER;
 RESULT - : INTEGER;
 BODY
);
```

3.2 Type Extension

A module type T_{ext} *directly extends* a module type T_{base} if T_{ext} has exactly one component of type T_{base}. T_{ext} *extends* a type T_{base} if it equals T_{base} or if it directly extends an extension of T_{base}.

Examples:

```
TYPE Module1 = (x : INTEGER);
TYPE Module2 = (ancestor : Module1; y : INTEGER);
TYPE Module3 = (ancestor : Module2; z : INTEGER);
```

In the examples above, *Module3* directly extends *Module2* with component *z*. *Module3* is an indirect extension of *Module1*. *Module1* is a direct base type of *Module2* which is a direct base type of *Module3*. Nested components of *Module3* can be referenced by incomplete designators that do not contain the identifier *ancestor*, as explained below.

Components of module variables can be denoted by *incomplete designators* according to the following rules. It is said that *c* is a *nested component* of a module variable *m*, if *c* is a component of *m*, or *c* is a nested component of some component of *m*. Then, if the module variable *m* does not have a component *c*, then *m.c* designates a nested component of *m* determined by left-to-right level-order search among all nested components of *m*. If *p* designates a pointer, then *p.c* stands for *p^.c* and *p[e]* stands for *p^[e]* (that is, the dot and the opening bracket imply dereferencing).

Examples:

```
m3 : Module3;
m3.z
m3.y                  (stands for m3.ancestor.y)
m3.x                  (stands for m3.ancestor.ancestor.x)
```

3.3 Pointers

Variables of a *PointerType* assume as values pointers to variables of some *BaseType*. The *PointerType* is said to be bound to its *pointer BaseType*. Pointer types inherit the extension relation of their base types. A pointer type *P* bound to T_{base} is extended by any pointer type P_{ext} bound to an extension T_{ext} of T_{base}. For instance, type *Ptr3* extends type *Ptr1*, because *Module3* extends *Module1*:

```
TYPE Ptr1 = POINTER TO Module1;
TYPE Ptr3 = POINTER TO Module3;
p1 : Ptr1;          p3 : Ptr3;
```

The type with which a pointer variable is declared is called its *static type* (or simply its type). The type of the value assumed by a pointer variable at run time is called its *dynamic type*. The dynamic type of a pointer variable may be an extension of its static type (see examples in Section 3.5).

The *type guard PointerVariable(DynamicType)* asserts that the *PointerVariable* has the quoted *DynamicType*. If the assertion fails, the program execution is aborted, otherwise the *PointerVariable* is regarded as having the *DynamicType*. The guard is applicable only if the *DynamicType* is an extension of the static type of the *PointerVariable*.

The type test *v* **IS** *T* stands for "the dynamic type of *v* is *T*" and is called a *type test*. It is applicable if

(1) *T* is an extension of the declared type *T0* of *v*, and

(2) *v* is a pointer variable.

The monadic *address operator* "@" applies to an operand which is a variable of any type. The type of the result is a pointer to the operand's type. This operator is used to implement variable parameters (See an example in Section 3.6.)

Examples:

i	(*INTEGER*)	*@i*	(**POINTER TO** *INTEGER*)

3.4 Bodies for Module Variables

If a module type definition does not include a forward body declaration, variables of this type are not allowed to have bodies. If the definition does include a forward body declaration, two options exist.

First, let *T* be a module type for which a body *B* has been declared. The variable declaration *M …: T* defines *B* as a body of *M*.

Second, let *T* be a module type which body has been left undefined. The declaration *M …: T* does not define a body for *M*. An individual body B_m may be defined for *M* in the scope of *M*. In this way, module variables of the same type can have completely different bodies.

In all cases, a whole module assignment (Section 3.5) can be used to give a new value and a new body to a module variable.

Examples (refer to examples in Section 3.1):

log2: Log2;

```
BODY log2; (* assume x > 0 *)
BEGIN RESULT := 0;
  WHILE x > 1 DO x := x DIV 2; INC (RESULT) END;
END log2;
myLog2: Log2;
BODY myLog2; (* assume x > 0 *)
  y: INTEGER;
BEGIN RESULT := 0; y := 1;
  WHILE x > y DO ASH (y); INC (RESULT) END;
END myLog2;
```

3.5 Assignments

Assignments replace the current value of a variable by a new value specified by an expression. The expression must be assignment compatible with the variable. In particular, an expression e of type T_e is *assignment compatible* with a variable v of type T_v if:

- T_e and T_v are the same type, as specified below;
- T_e and T_v are pointer types and T_e is an extension of T_v;

Some less important cases of type compatibility (numeric types, strings, **NIL** and pointer types) need not to be discussed here.

T_a is the *same type* as T_b if:

- T_a and T_b are both denoted by the same type identifier, or
- T_a and T_b are denoted by type identifiers and T_a is declared to equal T_b in a declaration of the form **TYPE** $T_a = T_b$, or
- T_a and T_b are types of variables a and b which appear in the same identifier list in a variable declaration, provided T_a and T_b are not open arrays.

Note that module variables of the same type may have different bodies.

If an expression is assigned to a variable, the value of the variable becomes the same as the value of the expression. Besides:

(1) If the expression is of a module type, both its value and its body (if any) are assigned into the variable. If the body of the expression is undefined, the body of the variable becomes undefined.

(2) If the variable and the expression are of pointer types, the dynamic type of the variable becomes the same as the dynamic type of the expression.

Examples (refer to examples in Sections 3.3 and 3.4):

```
p1 := p3;        p1(Ptr3).z := 0; log2 := myLog2;
```

Compared to Oberon, K2 offers a restricted form of assignment compatibility: In K2, an extended module type is not assignment compatible with its base type, while in Oberon an extended record type is assignment compatible with its base type.

3.6 Module calls

A *module call* consists of a module variable designator, followed by a (possibly empty) list of arguments. For the execution of the call, the arguments are assigned (Section 3.5) to the parameters (Section 3.1), then the body of the module variable (if any) is executed. The association between the arguments and the parameters is positional, but the list of arguments may have less members than the total number of parameters. Module calls can appear as individual statements; they also can be used in expressions, as specified later in this section.

```
ModuleCall = designator "(" Arguments ")"
```

Examples:

```
Subroutine: (
  valuePar: INTEGER;
  variablePar: POINTER TO INTEGER;
  BODY
);
BODY Subroutine;
BEGIN valuePar := valuePar + 1;
  variablePar^ := variablePar^ + 1
END Subroutine;

i := 0; Subroutine(0, @i);            (* ... *)
Subroutine.valuePar := 0; Subroutine.variablePar := @i;
            Subroutine();             (* ... *)
Subroutine(); i := Subroutine.ValuePar + 1;
```

In an expression, a designator of a module variable which is not followed by an argument list refers to the current value of that variable. If it is followed by a (possibly empty) argument list, the designator implies the activation of the module body and stands for the value of the module variable resulting from the execution.

A factor of the form

F(Arguments)

where *F* is a designator of a module variable which contains a component named *RESULT*, is evaluated as follows:

(1) the module call *F(Arguments)* is executed first;

(2) the value of *F.RESULT* is returned as value of *F(ARGUMENTS)*.

Example (refer to the examples in Section 3.4):

log2(k) + 1

If *designator* is a pointer variable with value NIL, the call *designator^(Arguments)* is executed as follows:

(1) NEW(*designator*) allocates a dynamic module which is thereafter called and executed;

(2) DISPOSE(*designator*) deallocates the dynamic module assigning NIL into designator.

An implementation may use a stack rather than a heap for such implicit module allocation/deallocation.

Example:

```
TYPE Factorial = (
  n: INTEGER; RESULT - : INTEGER;
  BODY
);
BODY Factorial;
  localFactorial: POINTER TO Factorial;
BEGIN
  IF n = 0 THEN RESULT := 1
  ELSE RESULT := n * localFactorial^(n - 1);
  END
END Factorial;
```

3.7 Compilation Units

A *compilation unit* is either a module type declaration eventually followed by a body, or a module variable declaration eventually followed by a body.

```
CompilationUnit =
        TypeDeclaration [";"BodyDeclaration]
        | VariableDeclaration [";"BodyDeclaration]
```

A compilation unit declares a single global identifier which is exported by the declaring unit. The exported identifier can be imported and used by other compilation units by means of an *import list* (see also Section 3.1).

```
ImportList = IMPORT ident [":="ident] {","ident [":="ident]}";"
```

Each identifier *I* from the import list of a module definition can be used in the definition itself, and in the type's body, if the type has a body. It the import list belongs to a module body, *I* can only be used in the body. If the form *I1 := I* is used in the import list, then the imported entity is referred as *I1* rather than *I*.

A main program can be implemented as a compilation unit which consists of a module variable declaration and a body. A conventional module (or a package) is a also a compilation unit consisting of a module variable declaration plus eventually a body. A separately compiled class is a compilation unit which consists of a module type declaration and, in most cases, a body.

Examples:

```
TYPE ClassDesc = (
 TYPE Class = POINTER TO ClassDesc;
 x : INTEGER;
 method : (v : INTEGER; BODY);
 BODY
);

BODY ClassDesc;
 BODY method;
 BEGIN
  x := v;
 END method;
END ClassDesc;

MainProgram: (BODY);
BODY MainProgram;
 IMPORT ClassDesc;
 ptr: ClassDesc.Class;
 (* ... *)
BEGIN  (* MainProgram *)
 NEW (ptr);    ptr.method (1);
 (* ... *)
END MainProgram;
```

3.8 Module Types and Object Orientation

In K2, a pointer type bound to a module type represents a class (see *Class* and *ClassDesc* in Section 3.7). A variable (such as *ptr^*) of that module type is an object. A module component of that module type is a method. A call of a module component (such as *ptr.method(1)*) is a message.

Type extension implements inheritance in K2. For instance, *SubclassDesc* inherits field *x* from *ClassDesc* extending *ClassDesc* with a field *y*:

```
TYPE SubclassDesc = (
 IMPORT ClassDesc;
 superclass : ClassDesc;
 y : INTEGER;
```

```
  BODY
);
```

A module variable declared in the body of an extension can be used to override an inherited method:

```
BODY SubclassDesc;
  overridingMethod : (v : INTEGER; BODY);
  BODY overridingMethod;
  BEGIN x := v; y := v END;
BEGIN (* SubclassDesc *)
  superclass.method := overridingMethod;
END SubclassDesc;

subPtr : POINTER TO SubclassDesc;
NEW (subPtr);   subPtr^();
```

The module call *subPtr^()* executes the assignment *superclass.method := overridingMethod* from the body of *SubclassDesc*. This assignment overrides (in *subPtr^*) the method inherited from *ClassDesc*. Thus, overriding a method is simply a module variable assignment. The difficulty with Oberon outlined in Section 2.2 does not exist in K2.

Note finally that the fields of an extension can be referred by incomplete designators. For instance:

```
subPtr.x              (stands for subPtr.superclass.x)
subPtr.method                 (stands for subPtr.superclass.method)
```

4 Conclusion

A problematic point in Oberon is that procedure fields of records cannot be directly and conveniently redefined for extensions. From a standard object-oriented point of view, method overriding in Oberon may seem unnatural and tedious (see Section 2.2). To cure this problem, Oberon-2 [4] extends Oberon with the new concept of type bound procedures. Besides, Oberon-2 adds to Oberon open array variables, FOR loops, and read-only export of data. (Object Oberon [5] is an experimental predecessor of Oberon-2.) In fact, Oberon-2 implants the standard concept of method in Oberon. The resulting language is not so simple and clean as Oberon was intended to be. In particular, it supports too many different structures related to procedures: type bound procedures, traditional constant procedures, procedure types, and procedure variables.

K2 evolved from Oberon by introducing only one new feature, the module type. Grace to the generality of the new concept, several features of Oberon were eliminated. Namely, K2 does not contain record and procedure types (because they are special kinds of module types), and does not need procedures and modules (because they are modeled by module variables). While record extension is supported by a specially designated language feature in Oberon, it is simply achieved by module nesting and use of incomplete module component designators in K2.

The body of a K2 module that is a component of a larger module has access to the components of the enclosing module. Thus, syntactical binding is as simple as module nesting, and there is no need for a special concept such as the type-bound procedure of Oberon-2.

One more advantage of K2 compared to Oberon is that a module type that implements a class can be compiled separately and need not be enclosed in a package or Oberon module.

Most features of K2 have been tested by an experimental compiler implemented as a Turbo Pascal 6.0 program of about 5000 lines. A K2 compilation unit (a module type or variable declaration, eventually followed by a body) is translated into a Turbo Pascal unit; then this unit is compiled by the Turbo Pascal compiler. The K2 compiler extracts all constant and type declarations from module definitions and generates Turbo Pascal representations for those declarations. Module definitions are compiled into record types. Turbo Pascal objects are not used in the implementation. At present, type tests and type guards are not supported by the experimental compiler.

This paper describes an approach to the design of a small and simple, yet practically convincing object-oriented language. Our approach can be characterized as *simplicity through generality*. While we present a solution, we do not consider it as a final one. The absence of procedures as a special language feature and their implementation by means of module variables is a point that is widely open for criticism. Although a pointer variable of a module base type can be used as a conventional procedure (as illustrated in Section 3.6), programmers may wish to have procedures explicitly included in the language. Fortunately, our solution can be relatively easily modified to include procedures, while merging record types and modules in the same concept. A careful evaluation of this alternative is a subject of future work.

References

1. M. Reiser, N. Wirth: Programming in Oberon. Steps beyond Pascal and Modula.
Wokingham: Addison-Wesley 1992

2. N. Wirth: The Programming Language Oberon. *Software - Practice and Experience* 18, 671-690 (1988)

3. N. Wirth: Type Extensions. *ACM Transactions on Programming Languages and Systems* 10, 204-214 (1987)

4. H. Moessenboeck, J. Templ: Object Oberon - A Modest Object-Oriented

Language. *Structured Programming* 10, 44-46 (1989)

5. H. Moessenboeck: The Programming Language Oberon-2 Report. Computer Science Report 160, ETH Zurich 1991

Appendix: Syntax Description

```
declaration = ConstantDeclaration | TypeDeclaration | VariableDeclaration
ConstantDeclaration = CONST ident "=" ConstExpr
TypeDeclaration = TYPE ident "=" type "
type = ArrayDefinition | ModuleDefinition | PointerDefinition | TypeDesignator
TypeDesignator = qualident
qualident = {ident "."} ident
ArrayDefinition = ARRAY [ConstExpr {"," ConstExpr}] OF type
ModuleDefinition = "(" [ImportList] DeclarationSequence [BODY] ")"
ImportList = IMPORT ident [":=" ident] {","ident [":=" ident]}";"
DeclarationSequence = {declaration ";"}
BodyDeclaration =
        BODY ident ";" [ImportList] DeclarationSequence BodySequence
        [BEGIN StatementSequence] END ident
BodySequence = {BodyDeclaration ";"}
PointerDefinition = POINTER TO Type
VariableDeclaration = ident["-"] ["," ident["-"]] ":" type
expression = SimpleExpression [relation SimpleExpression]
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS
SimpleExpression = ["+" | "-"] term {AddOperator term}
AddOperator = "+" | "-" | "OR"
term = factor {MulOperator factor}
MulOperator = "*" | "/" | DIV | MOD | "&"
factor = number | CharConstant | string | NIL | set | "~" factor | "@"
designator
        | designator ["("[ExprList]")"] | "("expression")"
designator = qualident {"."ident | "[" ExprList "]" | "("TypeDesignator")" | "^"}
set = "{" [element {"," element}] "}"
element = expression [".." expression]
statement = [assignment | ModuleCall | IfStatement | CaseStatement
        | WhileStatement | RepeatStatement LoopStatement | WithStatement
        | EXIT | RETURN]
assignment = designator ":=" expression
ModuleCall = designator "(" [ExprList] ")"
IfStatement = IF expression THEN StatementSequence {ELSIF expression
        THEN StatementSequence} [ELSE StatementSequence] END
CaseStatement =
        CASE expression OF case {"|"case} [ELSE StatementSequence] END
```

case = [CaseLabels {"," CaseLabels} ":" StatementSequence]

Caselabels = ConstExpr [".." ConstExpr]

WhileStatement = WHILE expression DO StatementSequence END

RepeatStatement = REPEAT StatementSequence UNTIL expression

LoopStatement = LOOP StatementSequence END

WithStatement = WITH qualident ":" typeDesignator DO StatementSequence END

CompilationUnit = TypeDeclaration [";"BodyDeclaration]
          | VariableDeclaration [";"BodyDeclaration]