FH Braunschweig/Wolfenbüttel
University of Applied Sciences

Department of Computer Science

Extensible Programming in Oberon
A Tutorial

R. Rüdiger

Report 06/97 / revised 09/99
(excerpt)

Postal Address: Department of Computer Science, FH Braunschweig/Wolfenbüttel – University of Applied Sciences, Salzdahlumer Str. 46/48, D-38302 Wolfenbüttel, Germany

Contents

5 Structure of extensible systems

This section summarizes some prototypical programs in full-length source code which are to exemplify the general discussion of the preceding sections. Our primary concern is to illustrate the potential of Oberon's language constructs on a technical level by means of ad-hoc examples. In some cases, these might not represent good programming style; so, for example, in a lengthy program, it is in general not very reasonable to introduce procedures containing just one single statement. In a tutorial, however, some formalization—perhaps even over-formalization—might be helpful in explaining language constructs. Some of the programs are slightly over-simplified: if the user forgets to initialize a variable of procedure type a tentative procedure call will result in a program trap; this error situation will not be handled by any of our implementations.

Section 5.1 describes a traditional technique which might be termed *functional extensibility*. This can be realized by means of *module embedding*: this technique is demonstrated by a short program excerpt in Example 5.1.1. Section 5.2 mentions the technique of type extension (*inheritance*) which allows to extend records with additional fields without textual change of the original text. The section does not contain any examples because this technique is extensively used in the examples of the subsequent sections. The following Section 5.3 demonstrates by means of two examples (Example 5.3.1 and Example 5.3.2) the notion of *one-dimensional extensibility*, a "tricky" combination of several language constructs (type extension, procedure up-call) to obtain a module system with well-separated tasks of the individual modules ("isolation" or "separation of concerns" [RW92, p. 229]). In particular, by way of this technique, an abstract user interface can be constructed. Finally, Section 5.4 demonstrates in Example 5.4.1 the use of handlers and messages, a technique which has been used in the viewer subsystem of the Oberon system itself [RW92, p. 237]). Examples 5.3.1 and 5.3.2 use the Oberon-2 construct of type-bound procedures whereas Example 5.4.1 is based solely on Oberon-1 ("Oberon proper") constructs.

5.1 Functional Extensibility

Functional extensibility, which is achieved by *module embedding* (cf [Mö93, p. 61]), is an approach along the lines of the traditional use of large commercially available scientific subroutine packages, such as NAG or IMSL. For example, the Oberon module Math could be extended with additional functions by embedding Math in a new module, say MathX, which reproduces the functionality of module Math and adds some functions.

Example 5.1.1 (Module embedding)

```
MODULE MathX;
  IMPORT m := Math;

  PROCEDURE exp*(x: REAL): REAL;
  BEGIN RETURN m.exp(x)
  END exp;

  PROCEDURE sinh*(x: REAL): REAL;
  BEGIN RETURN 0.5*(m.exp(x) − m.exp(−x))
  END sinh;
  . . .

END MathX.
```

5.2 Extensibility of data structures

Type extension or, in the traditional terminology of objects-oriented languages, *inheritance*, is one of the key notions of object-oriented programming. The combination of this technique with other language constructs of Oberon is discussed extensively in the next sections. Therefore, this section does not present any specific example. In many other languages a presentation of object-oriented programming examples would have to end with this section.

5.3 One-dimensional extensibility: models with type-bound procedures

This section illustrates the technique of one-dimensional extensibility by way of an example which will be modified and extended in different ways. The basic structure is a static array where objects of type Item can be stored. There are some functions to manipulate the array; for example, the objects in the array can be shifted to the right. Objects will be specialized by the addition of keys of types Integer and String (ARRAY OF CHAR). Observe that these extensions keep the user interface completely abstract.

The first modification will add a sorting procedure allowing to abstractly sort the objects according to an (abstract) key, then, after extending the key, according to integer and string keys separately and, finally, according to a common key regardless of the key type. In this latter case, in order to compare integers and strings, the integer values are converted to strings.

The essential point to be noticed is the way how "incremental programming" is achieved in the examples: it would be a fairly simple exercise to write a monolithic program which does the job of sorting objects with integer or string keys. The real novelty which is Oberon's original contribution to program design is the way how things are separated from each other. The benefit of this approach is an exceptional degree of extensibility. So, in our example, we could use keys of other types without changing any part of the existing system or, in Example 5.4.1 which utilizes messages and handlers, we could add procedures which are meaningful to integer keys only without changing the existing system and—of course—without introducing errors into the system.

We have intentionally chosen a "toy example", which certainly does not represent a realistic data structure and which might seem somewhat artificial; in our opinion, however, the example is particularly lucid and may easily be modified even by people with little programming experience. There is one little educational side effect using empty objects. In the theory of formal languages the student is faced with the notion of the empty word which often appears to be some mysterious and highly confusing construction.[1] Empty objects bear some resemblance to empty words. Our example demonstrates how to handle empty (NIL) objects (empty places on a book shelf, see below) which, hopefully, thus might loose some of their mysteriousness.

Readers who do not like abstract constructions but prefer concrete pictures might imagine a situation like the following. Usually, books on a book shelf tend to get disordered with empty places appearing between the books.[2] So one needs an algorithm to reorder the books according to some key; the books could be ordered with increasing keys from left to right. The empty places (slots) are shifted to the right; so, formally, the key of empty objects is a generic maximum ($+\infty$), larger than any other key independently of the type of the key. In our program this has been implemented as NIL.

Figure 1 depicts the import relations of the modules of the following examples. The relations in Examples 5.3.2 and 5.3.1 are essentially the same except that, in the latter case, Module Comparables has not yet been introduced.

Example 5.3.1
Implementation:

```
     MODULE GenericArrays;
       IMPORT Out;
       CONST N = 10;
       TYPE
         Item* = POINTER TO ItemDesc;
         ItemDesc* = RECORD
         END;
1        GenericArray* = POINTER TO GenericArrayDesc;
         GenericArrayDesc* = RECORD
           n: INTEGER;
           a: ARRAY N OF Item
         END;

2      VAR
         newItem*: PROCEDURE;
         NewItem*: Item;

3      PROCEDURE (p: Item) Print*;
       BEGIN HALT(20)
       END Print;

4      PROCEDURE validRange*(i: INTEGER): BOOLEAN;
       BEGIN
         RETURN (0 <= i) & (i <= N-1)
       END validRange;

5      PROCEDURE Get*(G: GenericArray; i: INTEGER): Item;
         VAR p: Item;
       BEGIN
         p := G.a[i];
         IF p # NIL THEN DEC(G.n); G.a[i] := NIL END;
         RETURN p
       END Get;

       PROCEDURE Put*(G: GenericArray; i: INTEGER;
                                                    p: Item);
       BEGIN
         IF G.a[i] = NIL THEN INC(G.n) END;
         G.a[i] := p
       END Put;

6      PROCEDURE ShowAll*(G: GenericArray);
         VAR i: INTEGER;
       BEGIN
         FOR i := 0 TO N-1 DO
           Out.Int(i,1); Out.String(": ");
           IF G.a[i] = NIL THEN
             Out.String("no object "); Out.Ln
           ELSE
             G.a[i].Print; Out.Ln
           END
         END;
         Out.String("Number of objects: "); Out.Int(G.n, 1);
         Out.Ln; Out.Ln
       END ShowAll;

7      PROCEDURE ShiftRight*(G: GenericArray);
         VAR i: INTEGER; p: Item;
       BEGIN
         p := G.a[N-1];
         FOR i := N-1 TO 1 BY -1 DO
           G.a[i] := G.a[i-1]
         END;
         G.a[0] := p
       END ShiftRight;

       PROCEDURE Init*(VAR G: GenericArray);
         VAR i: INTEGER;
       BEGIN NEW(G);
         FOR i := 0 TO N-1 DO G.a[i] := NIL END;
```

[1] "Well, may be it's useful; but what's the *meaning* of a word which is empty and what's the way to pronounce something that doesn't exist?"

[2] It is not clear whether this is a fundamental law of nature; anyway, the author's books behave that way.

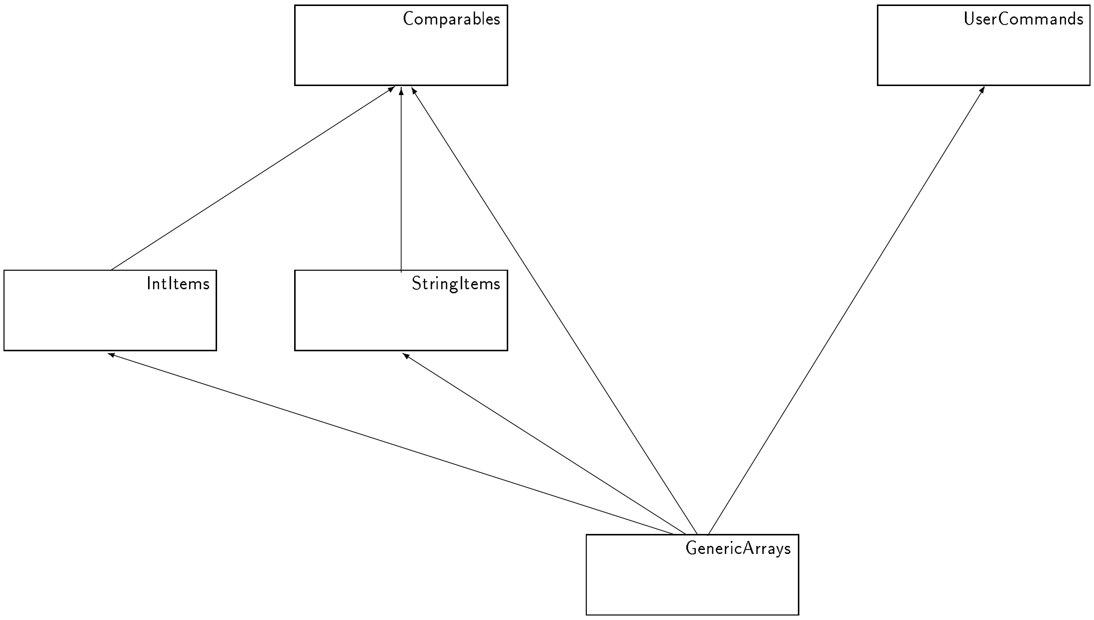


Figure 1 Import relations of modules in Examples 5.3.1, 5.3.2, and 5.4.1

```
    G.n := 0
  END Init;

END GenericArrays.
```

Notes:

1. GenericArray is a structure "containing" objects of type Item. In object-oriented terminology this is shorthand for describing the following situation: the array components are pointers which possibly (if # NIL) point to some heap storage places containing the data of the fields in record ItemDesc.

2. Variables newItem and NewItem are crucial to the notion of extensibility in the sense discussed here. newItem has to be set by an upcall, NewItem is a (generic) buffer variable to temporarily store newly created objects.

   Alternatively, newItem could be declared as a function procedure which returns new objects thus making the variable NewItem obsolete. The graphics editor is implemented this way (see Example 5.5.1). The implementation shown here is typical for the gadgets of Oberon System 3.

3. Print is an abstract procedure which has to be overridden. A direct call to procedure Print of the present module terminates the program. ("The program traps.")

4. An alternative to introducing validRange would be to export N and to check the validity of the range of i in other modules.

5. Put and Get are procedures to store objects in the array and to retrieve them from the array at some later time, respectively. Getting an object also means to remove it; this is simply our design decision.

6. The implementation of ShowAll has to take into account that, for a given array index, there might exist no object.

7. ShiftRight makes obvious that the array is heterogeneous: although the static type of any array component is Item and remains unchanged at run-time (of course, it is static!) the actual (dynamic) type will in general change when elements are being shifted.

Implementation:

```
  MODULE IntItems;
    IMPORT GenArr := GenericArrays, In, Out;
    TYPE
1     IntItem* = POINTER TO IntItemDesc;
      IntItemDesc* = RECORD(GenArr.ItemDesc)
        key: INTEGER
      END;

2   PROCEDURE (p: IntItem) Print*;
    BEGIN Out.Int(p.key, 5)
    END Print;
```

```
3     PROCEDURE New*;
        VAR p: IntItem;
      BEGIN
        NEW(p); In.Int(p.key);
        GenArr.NewItem := p
      END New;

4     PROCEDURE Set*;
      BEGIN
        GenArr.newItem := New
      END Set;

    END IntItems.
```

Notes:

1. IntItem is a specialization of GenericArrays.Item or, in technical terms, IntItem extends GenericArrays.Item.
2. This procedure Print overrides the abstract Print in module GenericArrays.
3. Procedure New is central to the mechanism of extensibility. New creates an object of type IntItem, initializes the object, and stores the result in GenericAray.NewItem. Note that the variables are assignment compatible because type IntItem extends Item.
4. Activation of Set puts the whole system into a state where all (abstract) actions on Item refer to IntItem until a Set command of some other module will be activated. Observe that here we are faced with a surprising trick: any module M which imports GenericArrays can activate GenericArrays.New. Effectively, however, it activates a procedure of some other module (e.g. New in IntItems) although this will in general not be visible to M. So, in a sense, M oversteps the module walls encapsulating names in IntItems, but this is achieved in a well-controlled fashion and without violating any of the strict rules of the language Oberon.

Implementation:

```
1   MODULE StringItems;
      IMPORT GenArr := GenericArrays, In, Out;
      TYPE
        String = ARRAY 32 OF CHAR;
        StringItem* = POINTER TO StringItemDesc;

        StringItemDesc* = RECORD(GenArr.ItemDesc)
          key: String
        END;

      PROCEDURE (p: StringItem) Print*;
      BEGIN Out.String(p.key)
      END Print;

      PROCEDURE New*;
        VAR p: StringItem;
      BEGIN
        NEW(p); In.String(p.key);
        GenArr.NewItem := p
      END New;

      PROCEDURE Set*;
      BEGIN
        GenArr.newItem := New
      END Set;

    END StringItems.
```

Notes:

1. In this example module StringItems has been printed in full detail to show that it is constructed the same way as IntItems. In the following examples we shall not repeat analogous modules. It is important that StringItems is neither aware of the existence of IntItems nor of UserCommands or, in technical terms, identifiers of these modules cannot be used in StringItems. This forms part of the concept which realizes a high degree of independence of the individual modules.

Implementation:

```
1   MODULE UserCommands;
      IMPORT In, Out, GenArr := GenericArrays;
      VAR G: GenArr.GenericArray;

      PROCEDURE Get*;
        VAR i: INTEGER; p: GenArr.Item;
      BEGIN In.Open;
        In.Int(i);
        IF GenArr.validRange(i) THEN
          p := GenArr.Get(G, i); Out.Int(i,1); Out.String(": ");
          IF p = NIL THEN Out.String("no object ")
          ELSE p.Print
          END
        ELSE
          Out.String("index out of range")
        END;
        Out.Ln
      END Get;

      PROCEDURE Put*;
        VAR i: INTEGER; p: GenArr.Item;
      BEGIN
        In.Open; In.Int(i);
        GenArr.newItem; p := GenArr.NewItem;
        WHILE In.Done DO
          IF GenArr.validRange(i) THEN GenArr.Put(G, i, p)
          ELSE Out.String("index out of range"); Out.Ln
          END;
          In.Int(i); GenArr.newItem; p := GenArr.NewItem
        END
      END Put;

      PROCEDURE ShiftRight*;
      BEGIN GenArr.ShiftRight(G)
      END ShiftRight;

      PROCEDURE ShowAll*;
      BEGIN GenArr.ShowAll(G)
      END ShowAll;

      PROCEDURE Init*;
      BEGIN GenArr.Init(G);
```

```
    END Init;

  END UserCommands.
```

Some sample commands:

```
    UserCommands.Init ~

    IntItems.Set ~ UserCommands.Put
      3 17
      5 22
      9 77
      1 123 ~

2   StringItems.Set ~ UserCommands.Put
      4 "eppe"
      7 "peppe"
      9 "kakke" ~

    UserCommands.Get 3 ~
    UserCommands.ShiftRight ~
    UserCommands.ShowAll ~
```

Notes:

1. UserCommands presents to the user a set of commands to manipulate the system. Note that all of the commands are abstract: they refer to Items, although, ultimately, they act on IntItems, StringItems, or possibly any future addition.
2. The command Put inputs pairs of values: the first is the array index i and the second the value characterizing the object to be stored in the array at i. For ease of readability each pair has been written in one line.[3]

Example 5.3.2
Implementation:

```
  MODULE GenericArrays;
```

... see example 5.3.1 starting on page 2 ...

```
    PROCEDURE (p: Item) LessThan*(q: Item): BOOLEAN;
    BEGIN HALT(22)
    END LessThan;

    PROCEDURE Sort*(G: GenericArray);

1     PROCEDURE LessThan(u, v: Item): BOOLEAN;
      (* returns the value of the boolean expression
         u < v *)
      BEGIN
        IF (u # NIL) & (v # NIL) THEN
          RETURN u.LessThan(v)
        ELSIF (u # NIL) & (v = NIL) THEN
          RETURN TRUE
        ELSE
          RETURN FALSE
        END
      END LessThan;

2     PROCEDURE QuickSort(L, R: INTEGER);
        VAR i, j: INTEGER; w, x: Item;
      BEGIN
        i := L; j := R; x := G.a[(L+R) DIV 2];
        REPEAT
          WHILE LessThan(G.a[i], x) DO INC(i) END;
          WHILE LessThan(x, G.a[j]) DO DEC(j) END;
          IF i <= j THEN
            w := G.a[i]; G.a[i] := G.a[j]; G.a[j] := w;
            INC(i); DEC(j)
          END
        UNTIL i > j;
        IF L < j THEN QuickSort(L, j) END;
        IF i < R THEN QuickSort(i, R) END;
      END QuickSort;

    BEGIN QuickSort(0, N-1)
    END Sort;

  END GenericArrays.
```

Notes:

1. Note that the trivial cases where one of the operands in the relation is NIL (u < NIL, etc) should be handled in this module.
2. When comparing this formulation of the algorithm with classical texts such as [Wir86, p. 96] the reader will notice that there are only minor changes of the code, a truly remarkable fact because the present implementation is completely generic. In the variants of Oberon under discussion the relational operator < has to be replaced with a Boolean-valued procedure (LessThan). In the language variant *Active Oberon* ([FM98, App. B], [Gut96]) the mechanism of overloading has been implemented; therefore, in this language variant, LessThan could be replaced by the operator < so that the difference to the conventional formulation would even be smaller.

Implementation:

```
  MODULE IntItems;
```

... see example 5.3.1 ...

```
    PROCEDURE (p: IntItem) LessThan*(
                                q: GenArr.Item): BOOLEAN;
    BEGIN
      RETURN p.key < q(IntItem).key
    END LessThan;

  END IntItems.
```

[3] **CAUTION**: Beware of the strings which are read after StringItems.Set has been activated: there is some powerful charm on the words. Therefore, saying these words may be most dangerous unless you are an experienced and well-trained sorcerer or sorceress; for more information see [Bau98].

Implementation:

```
MODULE StringItems;
```

... see example 5.3.1 ...

```
  PROCEDURE (p: StringItem) LessThan*(
                               q: GenArr.Item): BOOLEAN;
  BEGIN
    RETURN p.key < q(StringItem).key
  END LessThan;

END StringItems.
```

Implementation:

```
MODULE UserCommands;
```

... see example 5.3.1 ...

```
  PROCEDURE Sort*;
  BEGIN GenArr.Sort(G);
  END Sort;

END UserCommands.
```

Some sample commands:

```
  UserCommands.Init ~

  IntItems.Set ~ Comparables.SetInt ~
  UserCommands.Put
    2 70
    3 75
    6 63
    9 81 ~

  StringItems.Set ~ Comparables.SetString ~
  UserCommands.Put
    1 "hillo"
    4 "hollo"
    7 "hello" ~

  UserCommands.Get 3 ~
  UserCommands.ShiftRight ~
  UserCommands.ShowAll ~
  UserCommands.Sort ~
```

Now, in the following module, we shall use both integer and string keys simultaneously:

Implementation:

```
MODULE Comparables;
  IMPORT
    In, Strings, IntItems, StringItems,
    GenArr := GenericArrays;
  TYPE
    IntItem* = POINTER TO IntItemDesc;
    IntItemDesc* = RECORD(IntItems.IntItemDesc) END;
    StringItem* = POINTER TO StringItemDesc;
    StringItemDesc* =
      RECORD(StringItems.StringItemDesc)
      END;

  PROCEDURE (p: IntItem) LessThan*(q: GenArr.Item):
                                             BOOLEAN;
    VAR str: ARRAY 64 OF CHAR;
  BEGIN
    IF q IS IntItem THEN
      RETURN p.LessThan↑(q(IntItem))
    ELSIF q IS StringItem THEN
      Strings.IntToStr(p.key, str);
      RETURN str < q(StringItem).key
    ELSE HALT(23)
    END
  END LessThan;

  PROCEDURE (p: StringItem) LessThan*(
                               q: GenArr.Item): BOOLEAN;
    VAR str: ARRAY 64 OF CHAR;
  BEGIN
    IF q IS IntItem THEN
      Strings.IntToStr(q(IntItem).key, str);
      RETURN p.key < str
    ELSIF q IS StringItem THEN
      RETURN p.LessThan↑(q(StringItem))
    ELSE HALT(24)
    END
  END LessThan;

  PROCEDURE NewInt*;
    VAR p: IntItem;
  BEGIN
    NEW(p); In.Int(p.key);
    GenArr.NewItem := p;
  END NewInt;

  PROCEDURE SetInt*;
  BEGIN
    GenArr.newItem := NewInt
  END SetInt;

  PROCEDURE NewString*;
    VAR p: StringItem;
  BEGIN
    NEW(p); In.String(p.key);
    GenArr.NewItem := p;
  END NewString;

  PROCEDURE SetString*;
  BEGIN
    GenArr.newItem := NewString
  END SetString;

END Comparables.
```

5.4 Two-dimensional extensibility: model with messages and handlers

The following and last program of this series of examples demonstrates the use of *messages* and *handlers*. This technique permits to construct programs along the lines of a concept which has been called *two-dimensional extensibility* [RW92, p. 234], [Fra94], [Szy96]. Parts of the Oberon system itself have been implemented this way.

Example 5.4.1
Implementation:

```
MODULE GenericArrays;
  IMPORT Out;
  CONST N = 10;
  TYPE
    Item* = POINTER TO ItemDesc;

1   Message* = RECORD END;
    PrintMsg* = RECORD (Message) END;
    ReadMsg* = RECORD (Message) END;
```

```
    CmpMsg* = RECORD (Message)
      q*: Item;
      lessthan*: BOOLEAN (* result to be returned *)
    END;

2   Handler* = PROCEDURE (p: Item;
                                   VAR msg: Message);
    ItemDesc* = RECORD
      handle*: Handler
    END;

    GenericArray* = POINTER TO GenericArrayDesc;
    GenericArrayDesc* = RECORD
      n: INTEGER;
      a: ARRAY N OF Item
    END;

  VAR
    newItem*: PROCEDURE;
    NewItem*: Item;

  PROCEDURE validRange*(i: INTEGER): BOOLEAN;
  BEGIN
    RETURN (0 <= i) & (i <= N−1)
  END validRange;

  PROCEDURE Get*(G: GenericArray; i: INTEGER): Item;
    VAR p: Item;
  BEGIN
    p := G.a[i];
    IF p # NIL THEN DEC(G.n); G.a[i] := NIL END;
    RETURN p
  END Get;

  PROCEDURE Put*(G: GenericArray; i: INTEGER;
                                          p: Item);
  BEGIN
    IF G.a[i] = NIL THEN INC(G.n) END;
    G.a[i] := p
  END Put;

  PROCEDURE ShowAll*(G: GenericArray);
    VAR i: INTEGER; print: PrintMsg;
  BEGIN
    FOR i := 0 TO N−1 DO
      Out.Int(i,1); Out.String(": ");
      IF G.a[i] = NIL THEN
        Out.String("no object "); Out.Ln
      ELSE
        G.a[i].handle(G.a[i], print); Out.Ln
      END
    END;
    Out.String("Number of objects: "); Out.Int(G.n, 1);
    Out.Ln; Out.Ln
  END ShowAll;

  PROCEDURE ShiftRight*(G: GenericArray);
    VAR i: INTEGER; p: Item;
  BEGIN
    p := G.a[N−1];
    FOR i := N−1 TO 1 BY −1 DO
      G.a[i] := G.a[i−1]
    END;
    G.a[0] := p
  END ShiftRight;

  PROCEDURE Init*(VAR G: GenericArray);
    VAR i: INTEGER;
  BEGIN NEW(G);
    FOR i := 0 TO N−1 DO G.a[i] := NIL END;
    G.n := 0
  END Init;

  PROCEDURE Sort*(G: GenericArray);
    PROCEDURE LessThan(u, v: Item): BOOLEAN;
    (* returns the value of the boolean expression
         u < v *)
      VAR compare: CmpMsg;
    BEGIN
      IF (u # NIL) & (v # NIL) THEN
        compare.q := v;
        u.handle(u, compare);
        RETURN compare.lessthan
      ELSIF (u # NIL) & (v = NIL) THEN
        RETURN TRUE
      ELSE RETURN FALSE
      END
    END LessThan;

    PROCEDURE QuickSort(L, R: INTEGER);
      ... see page 5 ...
    END QuickSort;

  BEGIN QuickSort(0, N−1)
  END Sort;

END GenericArrays.
```

Notes:

1. Messages are sent to objects to perform certain actions. So, messages play the part of (function) procedures. The fields of the messages can be considered as function parameters or return values.

2. There is only one procedure, which may be type-bound or object-bound, a kind of universal procedure with the task of interpreting the "incoming" messages. Whenever new objects are being created long lists for initialization of record fields of procedure type are not required; this is different to the technique to be described in Example 5.5.2 (see page 13).

Implementation:

```
MODULE IntItems;
  IMPORT GenArr := GenericArrays, In, Out;
  TYPE
    IntItem* = POINTER TO IntItemDesc;
    IntItemDesc* = RECORD(GenArr.ItemDesc)
      key*: INTEGER;
    END;

1 PROCEDURE IntHandler*(p: GenArr.Item;
                              VAR M: GenArr.Message);
  BEGIN
    WITH p: IntItem DO
      IF M IS GenArr.PrintMsg THEN
        Out.Int(p.key, 5)
      ELSIF M IS GenArr.CmpMsg THEN
        WITH M: GenArr.CmpMsg DO
          IF M.q IS IntItem THEN
            M.lessthan := p.key < M.q(IntItem).key
          END
        END
      ELSIF M IS GenArr.ReadMsg THEN
        In.Int(p.key)
2     ELSE (* forward the message *)
      END
    END
  END IntHandler;

  PROCEDURE InitIntItem*(p: IntItem);
  BEGIN
```

```
    (* call init procedure of lower level? *)
    (* In this case we don't have any. *)
    p.handle := IntHandler (* installation of handler *)
  END InitIntItem;

3 PROCEDURE New*;
    VAR p: IntItem; read: GenArr.ReadMsg;
  BEGIN
4   NEW(p); InitIntItem(p); p.handle(p, read);
    GenArr.NewItem := p
  END New;

  PROCEDURE Set*;
  BEGIN
    GenArr.newItem := New
  END Set;

END IntItems.
```

Notes:

1. Within this concept writing a program primarily means to code the handler. The program shows the typical form of a With assignment in Oberon-1, which is somewhat clumsy. To avoid some textual repetitions Oberon-2 permits the following condensed form:

   ```
   PROCEDURE (p: Type1) Handle(
                          VAR msg: Message)
   BEGIN
     WITH
         msg: MsgType1 DO ...
       | msg: MsgType2 DO ...
       ......
     ELSE
     END
   END Handle;
   ```

2. Some people suspect that administrations do their jobs similarly to the way handlers work: in general, the job is delegated to a lower-level handler ("That's not in my department.") who on his part will try to delegate the job and so on. Eventually, either somebody has to do it because nobody is left or, if nobody knows what to do, the job will disappear in a trash box (the concluding Else clause in Oberon).[4]

3. Essentially, New and Set are not changed compared to Examples 5.3.1 and 5.3.2. The main advantage of messages and handlers is this: in a heterogeneous structure messages can be sent to all objects regardless whether each of the objects understands the message or not; if it does not the message will simply be ignored. Therefore, messages can be extended by the mechanism of type extension, which amounts to introducing new functions into an existing system without the need of changing or even recompiling any parts of this system.

4. Typically, a function like read is activated by calling p.handle(p, read): message read has to be sent to object p in order to read the values of its fields from external input.

Implementation:

```
MODULE StringItems;
  IMPORT GenArr := GenericArrays, In, Out;
  TYPE
    StringItem* = POINTER TO StringItemDesc;
    StringItemDesc* = RECORD(GenArr.ItemDesc)
      key*: ARRAY 64 OF CHAR;
    END;

  PROCEDURE StringHandler*(p: GenArr.Item;
                                VAR M: GenArr.Message);
  BEGIN
    WITH p: StringItem DO
      IF M IS GenArr.PrintMsg THEN
        Out.String(p.key)
      ELSIF M IS GenArr.CmpMsg THEN
        WITH M: GenArr.CmpMsg DO
          IF M.q IS StringItem THEN
            M.lessthan := p.key < M.q(StringItem).key
          END
        END
      ELSIF M IS GenArr.ReadMsg THEN
        In.String(p.key);
      ELSE (* forward the message *)
      END
    END
  END StringHandler;

  PROCEDURE InitStringItem*(p: StringItem);
  BEGIN
    p.handle := StringHandler (* installation of handler *)
  END InitStringItem;

  PROCEDURE New*;
    VAR p: StringItem; read: GenArr.ReadMsg;
  BEGIN
    NEW(p); InitStringItem(p); p.handle(p, read);
    GenArr.NewItem := p
  END New;

  PROCEDURE Set*;
  BEGIN
    GenArr.newItem := New
  END Set;

END StringItems.
```

Implementation:

```
MODULE UserCommands;
  IMPORT In, Out, GenArr := GenericArrays;

  VAR G: GenArr.GenericArray;

  PROCEDURE Get*;
    VAR
      i: INTEGER; p: GenArr.Item;
```

[4] Of course, administrations never work this way; if you believe you know of any such cases you must certainly be wrong.

```
    print: GenArr.PrintMsg;
  BEGIN In.Open;
    In.Int(i);
    IF GenArr.validRange(i) THEN
      p := GenArr.Get(G, i); Out.Int(i,1); Out.String(": ");
      IF p = NIL THEN Out.String("no object ")
      ELSE (* p.Print *) p.handle(p, print)
      END
    ELSE
      Out.String("index out of range")
    END;
    Out.Ln
  END Get;

  PROCEDURE Put*;
    VAR i: INTEGER; p: GenArr.Item;
  BEGIN
    In.Open; In.Int(i);
    GenArr.newItem; p := GenArr.NewItem;
    WHILE In.Done DO
      IF GenArr.validRange(i) THEN GenArr.Put(G, i, p)
      ELSE Out.String("index out of range"); Out.Ln
      END;
      In.Int(i); GenArr.newItem; p := GenArr.NewItem
    END
  END Put;

  PROCEDURE ShiftRight*;
  BEGIN GenArr.ShiftRight(G)
  END ShiftRight;

  PROCEDURE ShowAll*;
  BEGIN GenArr.ShowAll(G)
  END ShowAll;

  PROCEDURE Init*;
  BEGIN GenArr.Init(G);
  END Init;

  PROCEDURE Sort*;
  BEGIN GenArr.Sort(G);
  END Sort;


END UserCommands.


  UserCommands.Init ~

  IntItems.Set ~ Comparables.SetInt ~
  UserCommands.Put
    1 11
    2 22
    3 17
    5 22
    9 77 ~

  StringItems.Set ~ Comparables.SetString ~
  UserCommands.Put
    1 "zizzy"
    2 "zuzzy"
    4 "zik" ~

  UserCommands.Get 3 ~
  UserCommands.ShiftRight ~
  UserCommands.ShowAll ~
  UserCommands.Sort ~
```

Now, in close analogy to Example 5.3.2, both kinds of keys will be used at the same time.

Implementation:

```
MODULE Comparables;
IMPORT
  Strings, IntItems, StringItems,
  GenArr := GenericArrays;
TYPE
  IntItem* = POINTER TO IntItemDesc;
  IntItemDesc* =
    RECORD(IntItems.IntItemDesc)
    END;
  StringItem* = POINTER TO StringItemDesc;
  StringItemDesc* =
    RECORD(StringItems.StringItemDesc)
    END;

PROCEDURE IntHandler*(p: GenArr.Item;
                      VAR M: GenArr.Message);
  VAR str: ARRAY 64 OF CHAR;
BEGIN
  WITH p: IntItem DO
    IF M IS GenArr.CmpMsg THEN
      WITH M: GenArr.CmpMsg DO
        IF M.q IS IntItem THEN (* forward *)
        IntItems.IntHandler(p, M);
        ELSIF M.q IS StringItem THEN
          Strings.IntToStr(p.key, str);
          M.lessthan := str < M.q(StringItem).key
        ELSE HALT(23)
          (* situation should never occur *)
        END
      END
    ELSE (* forward the message *)
      IntItems.IntHandler(p, M);
    END
  END
END IntHandler;

PROCEDURE StringHandler*(p: GenArr.Item;
                         VAR M: GenArr.Message);
  VAR str: ARRAY 64 OF CHAR;
BEGIN
  WITH p: StringItem DO
    IF M IS GenArr.CmpMsg THEN
      WITH M: GenArr.CmpMsg DO
        IF M.q IS StringItem THEN (* forward *)
          StringItems.StringHandler(p, M)
        ELSIF M.q IS IntItem THEN
          Strings.IntToStr(M.q(IntItem).key, str);
          M.lessthan := p.key < str
        ELSE HALT(23) (* situation should never occur *)
        END
      END
    ELSE
      StringItems.StringHandler(p, M)
    END
  END
END StringHandler;

PROCEDURE InitIntItem*(p: IntItem);
BEGIN
  IntItems.InitIntItem(p);
  p.handle := IntHandler (* installation of handler *)
END InitIntItem;

PROCEDURE InitStringItem*(p: StringItem);
BEGIN
  StringItems.InitStringItem(p);
  p.handle := StringHandler (* installation of handler *)
END InitStringItem;

PROCEDURE NewInt*;
  VAR p: IntItem; read: GenArr.ReadMsg;
BEGIN
  NEW(p); InitIntItem(p); p.handle(p, read);
  GenArr.NewItem := p
END NewInt;

PROCEDURE NewString*;
  VAR p: StringItem; read: GenArr.ReadMsg;
```

```
  BEGIN
    NEW(p); InitStringItem(p); p.handle(p, read);
    GenArr.NewItem := p
  END NewString;

  PROCEDURE SetInt*;
  BEGIN
    GenArr.newItem := NewInt
  END SetInt;

  PROCEDURE SetString*;
  BEGIN
    GenArr.newItem := NewString
  END SetString;

END Comparables.
```

5.5 *Tema con Variazioni* on "The Graphics Editor"

In the textbooks [RW92] and [Mö93] a graphics editor is the main example to demonstrate the potential of the technique of extensible programming in Oberon. These books present program fragments which demonstrate the use of some of the technical variants (type-bound and object-bound procedures, use of messages and handlers etc). For the newcomer to the world of Oberon it is not completely trivial to form complete programs out of the fragments because these appear in slightly varying contexts.

Therefore, in order to compare these techniques, it is a useful exercise to compose complete programs with essentially identical functionality. This gives some insight into the differences between programs implemented in Oberon-1 vs Oberon-2 or the use of record fields of procedure type vs messages and handlers.

Using some of these techniques we shall present details of the graphics editor in full-length program code. In order to ease comparisons between the program variants we have chosen the same names, functions, etc in the following examples. Example 5.5.1 uses the concept of type-bound procedures of Oberon-2, cf [RW92, p. 266], Example 5.5.2 is confined to language constructs of Oberon-1: this corresponds to the program fragments in [RW92, p. 212]. Example 5.5.3 uses messages and handlers with Oberon-1, see [RW92, p. 230] and, finally, Example 5.5.4 is based upon two special modules, Modules and Types, and type-bound procedures of Oberon-2.

Module Draw implements the abstract user interface; in each of the examples this module is the same and will therefore not repeatedly be shown.

With this tutorial we do by no means intend to replace any of the cited textbooks on Oberon. So, we shall not repeat the notes given therein but merely add some remarks which might possibly be helpful to Oberon beginners. To summarize the essentials of extensibility we state that the basic import structure of the modules comprising the graphics editor as depicted in Figure 2 is the same in each of the program variants. All of the commands in module Draw are abstract actions in the sense that they act on abstract figures. By putting the whole system into a state where actions refer to some specialization of Figure we can make these action concrete. To these ends, procedure Set in modules Rectangles or Ellipses has to be activated; this will assign at run-time a procedure New to the variable of procedure type, newFigure.

Extensibility results from a clever decoupling of details in modules Graphics, Ellipses, Rectangles, etc. In particular, none of the names Ellipse or Rectangle needs to be known—and even can be known—in one of the modules Figures, Graphics, or Draw. Therefore, we could write another module, Lines e.g., with its own procedure Set, which could be used in complete analogy to Ellipses or Rectangles. This extension would not require any parts of the existing system to be changed or even be recompiled. It would even not be necessary to have the source code of these modules.

The foregoing examples have been constructed in close analogy to the graphics editor; therefore, some of the following notes are merely repetitions.

The first variant in our series of examples is an implementation using type-bound procedures of Oberon-2.

Example 5.5.1
Implementation:

```
  MODULE Graphics;
    IMPORT In, Out;
    CONST fmt = 15;
    TYPE
      Figure* = POINTER TO FigureDesc;
      FigureDesc* = RECORD
        next: Figure
      END;
1     Graph* = POINTER TO GraphDesc;
      GraphDesc* = RECORD
2       list: Figure
      END;

3   VAR newFigure*: PROCEDURE (): Figure;

    PROCEDURE (f: Figure) Draw*;
    BEGIN HALT(20)
    END Draw;

    PROCEDURE(f: Figure) Move*(dx, dy: INTEGER);
    BEGIN HALT(21)
    END Move;

4   PROCEDURE (f: Figure) GetArea*(): REAL;
    BEGIN HALT(22)
    END GetArea;
```

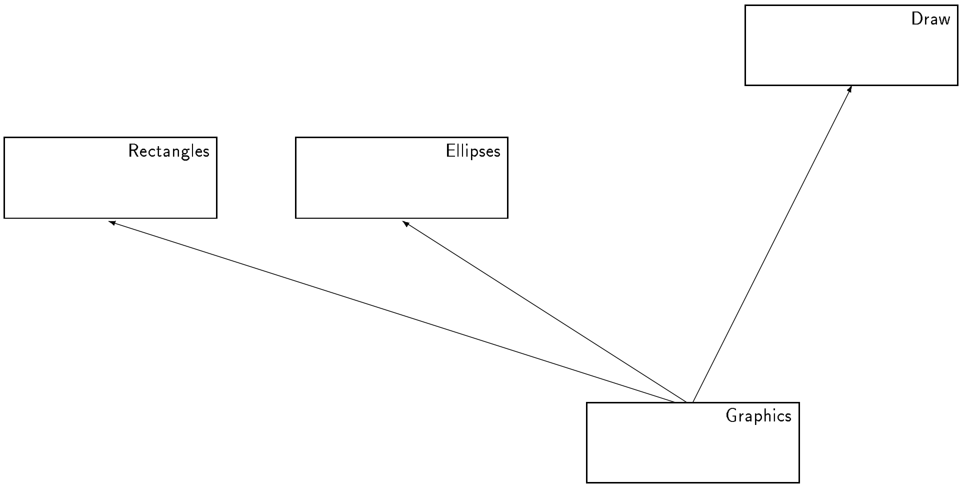


Figure 2 Import relations of the graphics editor

```
5    PROCEDURE InsertLast(g: Graph; f: Figure);
       VAR fig: Figure;
     BEGIN f.next := NIL;
       IF g.list = NIL THEN g.list := f
       ELSE
         fig := g.list;
         WHILE fig.next # NIL DO fig := fig.next END;
         fig.next := f
       END
     END InsertLast;

6    PROCEDURE CreateFigure*(g: Graph);
       VAR f: Figure;
     BEGIN
       f := newFigure(); f.Draw; InsertLast(g, f)
     END CreateFigure;

     PROCEDURE DrawAll*(g: Graph);
       VAR f: Figure;
     BEGIN f := g.list;
       WHILE f # NIL DO f.Draw; f := f.next END
     END DrawAll;

     PROCEDURE PrintAreas*(g: Graph);
       VAR f: Figure; area: REAL;
     BEGIN f := g.list;
       WHILE f # NIL DO
         Out.Real(f.GetArea(), fmt); Out.Ln;
         f := f.next
       END
     END PrintAreas;

7    PROCEDURE MoveAll*(g: Graph);
       VAR f: Figure; dx, dy: INTEGER;
     BEGIN f := g.list;
       In.Int(dx); In.Int(dy);
       WHILE f # NIL DO f.Move(dx, dy); f := f.next END
     END MoveAll;

     PROCEDURE Open*(VAR g: Graph);
     BEGIN NEW(g); g.list := NIL
     END Open;

   END Graphics.


8  MODULE Rectangles;
     IMPORT G := Graphics, In, Out;
     CONST fmt = 8;
     TYPE
       Rectangle = POINTER TO RectDesc;
       RectDesc = RECORD (G.FigureDesc)
         x, y, w, h: INTEGER
       END;

     PROCEDURE (rect: Rectangle) Draw;
     BEGIN Out.String(" drawing rectangle ... ");
       Out.Int(rect.x, fmt); Out.Int(rect.y, fmt);
       Out.Int(rect.w, fmt); Out.Int(rect.h, fmt);
       Out.Ln
     END Draw;

     PROCEDURE (rect: Rectangle) Move(dx, dy: INTEGER);
     BEGIN Out.String(" moving rectangle ... "); Out.Ln;
       rect.x := rect.x + dx; rect.y := rect.y + dy
     END Move;

     PROCEDURE (rect: Rectangle) GetArea(): REAL;
     BEGIN RETURN rect.w * rect.h
     END GetArea;

9    PROCEDURE New(): G.Figure;
       VAR rect: Rectangle; x, y, w, h: INTEGER;
     BEGIN NEW(rect);
       In.Int(rect.x); In.Int(rect.y); In.Int(rect.w); In.Int(rect.h);
       RETURN rect
     END New;

     PROCEDURE Set*;
     BEGIN G.newFigure := New
     END Set;

   END Rectangles.


10 MODULE Ellipses;
     IMPORT G := Graphics, In, Out;
     CONST fmt = 8;
     TYPE
       Ellipse = POINTER TO EllipseDesc;
       EllipseDesc = RECORD (G.FigureDesc)
         x, y, a, b: INTEGER
       END;

     PROCEDURE (ellip: Ellipse) Draw;
     BEGIN Out.String(" drawing ellipse ... ");
       Out.Int(ellip.x, fmt); Out.Int(ellip.y, fmt);
```

```
    Out.Int(ellip.a, fmt); Out.Int(ellip.b, fmt);
    Out.Ln
  END Draw;

  PROCEDURE (ellip: Ellipse) Move(dx, dy: INTEGER);
  BEGIN Out.String(" moving ellipse ... "); Out.Ln;
    ellip.x := ellip.x + dx; ellip.y := ellip.y + dy
  END Move;

  PROCEDURE (ellip: Ellipse) GetArea(): REAL;
    CONST pi = 3.1415;
  BEGIN RETURN pi * ellip.a * ellip.b
  END GetArea;

  PROCEDURE New(): G.Figure;
    VAR ellip: Ellipse; x, y, a, b: INTEGER;
  BEGIN NEW(ellip);
    In.Int(ellip.x); In.Int(ellip.y);
    In.Int(ellip.a); In.Int(ellip.b);
    RETURN ellip
  END New;

  PROCEDURE Set*;
  BEGIN G.newFigure := New
  END Set;

END Ellipses.


11 MODULE Draw;
  IMPORT G := Graphics, In, Out;
  VAR g: G.Graph;

  PROCEDURE CreateFigure*;
  BEGIN In.Open; G.CreateFigure(g)
  END CreateFigure;

  PROCEDURE DrawAll*;
  BEGIN G.DrawAll(g);
    Out.Ln; Out.Ln; Out.Ln
  END DrawAll;

  PROCEDURE PrintAreas*;
  BEGIN G.PrintAreas(g); Out.Ln
  END PrintAreas;

  PROCEDURE MoveAll*;
  BEGIN In.Open; G.MoveAll(g)
  END MoveAll;

  PROCEDURE Open*;
  BEGIN G.Open(g)
  END Open;

BEGIN G.Open(g)
END Draw.


Sample list of commands:

  Rectangles.Set ~
  Ellipses.Set ~

Rectangle:
  Draw.CreateFigure 7 7 11 9 ~
Ellipse:
  Draw.CreateFigure 1 2 4 6 ~

  Draw.DrawAll ~
  Draw.MoveAll −1 −2 ~
  Draw.Open ~
  Draw.PrintAreas ~
  Draw.MoveAll −1 −2 ~
```

Notes:

1. Figure and Graph are abstract data types (ADTs). Graph is a kind of container structure: objects of type Figure can be stored in a Graph object.

2. The record fields next and list are purely technical fields serving to form the structures; they should not be exported.

3. In contrast to Example 5.3.1 we have one global variable only, a function procedure, the return value of which is a new object of type Figure.

4. Draw, Move, and GetArea are abstract procedures each demonstrating some typical situations: e.g. differing number of parameters, return value. In Example 5.5.3 (with messages and handlers) theses differences will become more important than in the present situation.

5. InsertLast inserts figure f into graph g. Reiser and Wirth [RW92, p. 225] use a FIFO-queue. Of course, this is just one possible choice; a LIFO-queue would do the job as well. Perhaps a FIFO-queue seems a bit more intuitive because it keeps the order in which figures are inserted.

6. CreateFigure creates and draws an abstract figure and inserts it into the list.

7. Each of the procedures DrawAll, PrintAreas, and MoveAll implements a traversal of the list of figures. A so-called broadcast could simplify this implementation.

8. Module Rectangles implements a specialization of Figure to Rectangle. Particularly, this requires overwriting the (abstract) procedures Draw, Move, and GetArea of module Graphics.

9. The procedures New and Set are the main constituents of the structure: New creates and initializes a new instance of Rectangle; Set initializes the variable newFigure. So, calling procedure newFigure effectively activates New. This is a procedure activation in the opposite direction of module imports, a so-called up-call.

10. All of the details in module Ellipses are analogous to those in module Rectangles. The module demonstrates the notion of extensibility as conceived by Wirth and Gutknecht:

the system can be extended without recompiling existing modules. In this example this can easily be recognized because module Ellipses does not rely on any details of module Rectangles. In the following examples the detailed code of module Ellipses will not be reproduced.

11. Module Draw implements one object of type Graph (the global variable g) and presents to the user a set of Oberon commands to manipulate g. The essential point is that none of the identifiers of modules Rectangles or Ellipses appears in Draw; so, in a sense, Draw is an abstract interface. Therefore, when extending the system by another module, say Lines, we do not have to change or to recompile module Draw. Of course, actually, any action has to refer to concrete objects. This is achieved by putting the system into a state where abstract operation on figures refer to Rectangle, Ellipse, etc (command Set in Rectangles, Ellipses, etc).

    In each of the variants of the graphics editor described here module Draw is exactly the same; therefore, we shall not repeat the code in the following examples.

The following program variant demonstrates the use of record fields of procedure type. This is the starting point of the introduction of *Extensible Programming* in [RW92, p. 212].

Example 5.5.2
Implementation:

```
  MODULE Graphics;
    IMPORT In, Out;
    CONST fmt = 15;
    TYPE
      Figure* = POINTER TO FigureDesc;
1     FigureDesc* = RECORD
        draw*: PROCEDURE (f: Figure);
        move*: PROCEDURE (f: Figure; dx, dy: INTEGER);
        getArea*: PROCEDURE (f: Figure): REAL;
        next: Figure
      END;
      Graph* = POINTER TO GraphDesc;
      GraphDesc* = RECORD
        list: Figure
      END;

    VAR newFigure*: PROCEDURE (): Figure;

    PROCEDURE InsertLast(g: Graph; f: Figure);
      ... see page 10...
    END InsertLast;

    PROCEDURE CreateFigure*(g: Graph);
      VAR f: Figure;
    BEGIN
2     f := newFigure(); f.draw(f); InsertLast(g, f)
    END CreateFigure;

    PROCEDURE DrawAll*(g: Graph);
      VAR f: Figure;
    BEGIN f := g.list;
      WHILE f # NIL DO f.draw(f); f := f.next END
    END DrawAll;

    PROCEDURE PrintAreas*(g: Graph);
      VAR f: Figure; area: REAL;
    BEGIN f := g.list;
      WHILE f # NIL DO
        Out.Real(f.getArea(f), fmt); Out.Ln;
        f := f.next
      END
    END PrintAreas;

    PROCEDURE MoveAll*(g: Graph);
      VAR f: Figure; dx, dy: INTEGER;
    BEGIN f := g.list;
      In.Int(dx); In.Int(dy);
      WHILE f # NIL DO f.move(f, dx, dy); f := f.next END
    END MoveAll;

    PROCEDURE Open*(VAR g: Graph);
    BEGIN NEW(g); g.list := NIL
    END Open;

  END Graphics.


  MODULE Rectangles;
    IMPORT G := Graphics, In, Out;
    CONST fmt = 8;
    TYPE
      Rectangle = POINTER TO RectDesc;
      RectDesc = RECORD (G.FigureDesc)
        x, y, w, h: INTEGER
      END;

    PROCEDURE Draw(f: G.Figure);
    BEGIN Out.String(" drawing rectangle ... ");
      WITH f: Rectangle DO
        Out.Int(f.x, fmt); Out.Int(f.y, fmt);
        Out.Int(f.w, fmt); Out.Int(f.h, fmt)
      END;
      Out.Ln
    END Draw;

    PROCEDURE Move(f: G.Figure; dx, dy: INTEGER);
    BEGIN Out.String(" moving rectangle ... "); Out.Ln;
      WITH f: Rectangle DO
        f.x := f.x + dx; f.y := f.y + dy
      END
    END Move;

    PROCEDURE GetArea(f: G.Figure): REAL;
    BEGIN
      WITH f: Rectangle DO RETURN f.w * f.h END
    END GetArea;

    PROCEDURE New(): G.Figure;
      VAR rect: Rectangle; x, y, w, h: INTEGER;
    BEGIN NEW(rect);
3     rect.draw := Draw; rect.move := Move;
      rect.getArea := GetArea;
      In.Int(rect.x); In.Int(rect.y); In.Int(rect.w); In.Int(rect.h);
      RETURN rect
    END New;

    PROCEDURE Set*;
    BEGIN G.newFigure := New
    END Set;

  END Rectangles.
```

```
MODULE Ellipses;
   same structure as in example 5.5.1
END Ellipses.

MODULE Draw;
... see page 12 ...
END Draw.
```

Notes:

1. In Oberon-1 record fields of procedure type represent the functionality of objects. In analogy to the type-bounds procedures of Oberon-2 procedures assigned to these fields could be termed "object-bound".
2. f.draw(f) is a typical example of a procedure call in Oberon-1.
3. In New the record fields of procedure type are initialized at run time; the procedures assigned to these fields must be assignment compatible, i.e. the formal parameter lists must match.

The next program variant uses the technique of messages and handlers, see [RW92, p. 230].

Example 5.5.3
Implementation:

```
  MODULE Graphics;
    IMPORT In, Out;
    CONST fmt = 15;
    TYPE
1     Message* = RECORD END;
      DrawMsg* = RECORD (Message) END;
      MoveMsg* = RECORD (Message)
        dx*, dy*: INTEGER;
      END;
      GetAreaMsg* = RECORD (Message)
        value*: REAL;
      END;
      Figure* = POINTER TO FigureDesc;

2     Handler* = PROCEDURE (f: Figure;
                                     VAR msg: Message);
      FigureDesc* = RECORD
        handle*: Handler;
        next: Figure
      END;

      Graph* = POINTER TO GraphDesc;
      GraphDesc* = RECORD
        list: Figure
      END;

    VAR newFigure*: PROCEDURE (): Figure;

    PROCEDURE InsertLast(g: Graph; f: Figure);
      ... see page 10...
    END InsertLast;

    PROCEDURE CreateFigure*(g: Graph);
      VAR f: Figure; draw: DrawMsg;
    BEGIN
3     f := newFigure(); f.handle(f, draw); InsertLast(g, f)
    END CreateFigure;

    PROCEDURE DrawAll*(g: Graph);
      VAR f: Figure; draw: DrawMsg;
    BEGIN f := g.list;
      WHILE f # NIL DO f.handle(f, draw); f := f.next END
    END DrawAll;

    PROCEDURE PrintAreas*(g: Graph);
      VAR f: Figure; area: REAL; getArea: GetAreaMsg;
    BEGIN f := g.list;
      WHILE f # NIL DO
        f.handle(f, getArea);
        Out.Real(getArea.value, fmt); Out.Ln;
        f := f.next
      END
    END PrintAreas;

    PROCEDURE MoveAll*(g: Graph);
      VAR f: Figure; dx, dy: INTEGER; move: MoveMsg;
    BEGIN f := g.list;
      In.Int(move.dx); In.Int(move.dy);
      WHILE f # NIL DO f.handle(f, move); f := f.next END
    END MoveAll;

    PROCEDURE Open*(VAR g: Graph);
    BEGIN NEW(g); g.list := NIL
    END Open;

  END Graphics.


  MODULE Rectangles;
    IMPORT G := Graphics, In, Out;
    CONST fmt = 8;
    TYPE
      Rect = POINTER TO RectDesc;
      RectDesc = RECORD (G.FigureDesc)
        x, y, w, h: INTEGER
      END;

    PROCEDURE HandleRect(f: G.Figure;
                                     VAR msg: G.Message);
    BEGIN
      WITH f: Rect DO
        IF msg IS G.DrawMsg THEN
          WITH msg: G.DrawMsg DO
            Out.String("Drawing rectangle ... ");
            Out.Int(f.x, fmt); Out.Int(f.y, fmt);
            Out.Int(f.w, fmt); Out.Int(f.h, fmt);
            Out.Ln
          END
        ELSIF msg IS G.MoveMsg THEN
          WITH msg: G.MoveMsg DO
            Out.String("Moving rectangle"); Out.Ln;
            f.x := f.x + msg.dx; f.y := f.y + msg.dy;
          END;
        ELSIF msg IS G.GetAreaMsg THEN
          WITH msg: G.GetAreaMsg DO
            msg.value := f.w * f.h
          END;
        ELSE
          Out.String(" unknown message: no action ... ");
          Out.Ln
        END;
      END;
    END HandleRect;

    PROCEDURE New(): G.Figure;
      VAR rect: Rect; x, y, w, h: INTEGER;
    BEGIN NEW(rect);
4   rect.handle := HandleRect; (* installation of handler *)
      In.Int(rect.x); In.Int(rect.y);
      In.Int(rect.w); In.Int(rect.h);
      RETURN rect
```

```
  END New;

  PROCEDURE Set*;
  BEGIN G.newFigure := New
  END Set;

END Rectangles.


MODULE Ellipses;
  same structure as in example 5.5.1
END Ellipses.


MODULE Draw;
... see page 12 ...
END Draw.
```

Notes:

1. At a first glance it might seem a little irritating that a message is extended by other messages all of which are empty (i.e. contain no fields). The point is, however, that the names themselves carry some information.

2. The handler is a universal procedure which handles (interprets) messages.

3. The standard interpretation of the statement f.handle(f, draw) is: a draw message is sent to figure f advising the figure to draw itself. The activation of the handler with parameter draw is the way a procedure is called within the context of messages and handlers.

4. This assignment is termed "installation of the handler".

Alternatively to up-calls the following variant makes use of the functionality of two modules, Modules and Types, which form part of the Oberon system. The implementation of Oberon-0 in [Mö93] has been based upon this technique. Although this approach does not stay within the bounds set by the Oberon *language*, it is quite remarkable because a similar structure can be mimicked by other languages, e.g. Java, which are not embedded in an operating system with the potential of the Oberon system. Detailed information concerning these modules may be found in the textbook of Mössenböck [Mö93], the documentation distributed with the Oberon system, and in [Ste97].

Example 5.5.4
Implementation:

```
MODULE Graphics;
IMPORT In, Out, Types, Modules;
CONST fmt = 15;
TYPE
  Figure* = POINTER TO FigureDesc;
  FigureDesc* = RECORD
    next: Figure
  END;
  Graph* = POINTER TO GraphDesc;
  GraphDesc* = RECORD
    list: Figure
  END;

PROCEDURE (f: Figure) Draw*;
BEGIN HALT(20)
END Draw;

PROCEDURE(f: Figure) Move*(dx, dy: INTEGER);
BEGIN HALT(21)
END Move;

PROCEDURE (f: Figure) GetArea*(): REAL;
BEGIN HALT(22)
END GetArea;

PROCEDURE (f: Figure) Init*;
BEGIN HALT(23)
END Init;

PROCEDURE InsertLast(g: Graph; f: Figure);
  ... see page 10...
END InsertLast;

PROCEDURE newFigure(): Figure;
  VAR curModule, curType: ARRAY 64 OF CHAR;
  mod: Modules.Module; type: Types.Type;
    f: Figure;
BEGIN
  In.String(curModule); In.String(curType);
  mod := Modules.ThisMod(curModule);
  IF mod # NIL THEN
    type := Types.This(mod, curType);
  END;
  IF type # NIL THEN
    Types.NewObj(f, type);
    f.Init;
  ELSE f := NIL
  END;
  RETURN f
END newFigure;

PROCEDURE CreateFigure*(g: Graph);
  VAR f: Figure;
BEGIN
  f := newFigure(); f.Draw; InsertLast(g, f);
END CreateFigure;

PROCEDURE DrawAll*(g: Graph);
  VAR f: Figure;
BEGIN f := g.list;
  WHILE f # NIL DO f.Draw; f := f.next END
END DrawAll;

PROCEDURE PrintAreas*(g: Graph);
  VAR f: Figure; area: REAL;
BEGIN f := g.list;
  WHILE f # NIL DO
    Out.Real(f.GetArea(), fmt); Out.Ln;
    f := f.next
  END
END PrintAreas;

PROCEDURE MoveAll*(g: Graph);
  VAR f: Figure; dx, dy: INTEGER;
BEGIN f := g.list;
  In.Int(dx); In.Int(dy);
  WHILE f # NIL DO f.Move(dx, dy); f := f.next END
END MoveAll;
```

```
   PROCEDURE Open*(VAR g: Graph);
   BEGIN NEW(g); g.list := NIL
   END Open;

 END Graphics.


 MODULE Rectangles;
   IMPORT G := Graphics, In, Out;
   CONST fmt = 8;
   TYPE
     Rectangle = POINTER TO RectDesc;
     RectDesc = RECORD (G.FigureDesc)
       x, y, w, h: INTEGER
     END;

   PROCEDURE (rect: Rectangle) Draw;
   BEGIN Out.String(" drawing rectangle ... ");
     Out.Int(rect.x, fmt); Out.Int(rect.y, fmt);
     Out.Int(rect.w, fmt); Out.Int(rect.h, fmt);
     Out.Ln
   END Draw;

   PROCEDURE (rect: Rectangle) Move(dx, dy: INTEGER);
   BEGIN Out.String(" moving rectangle ... "); Out.Ln;
     rect.x := rect.x + dx; rect.y := rect.y + dy
   END Move;

   PROCEDURE (rect: Rectangle) GetArea(): REAL;
   BEGIN RETURN rect.w * rect.h
   END GetArea;

   PROCEDURE (rect: Rectangle) Init*;
   BEGIN
     In.Int(rect.x); In.Int(rect.y);
     In.Int(rect.w); In.Int(rect.h)
   END Init;

 END Rectangles.


 MODULE Ellipses;
     ...same structure as in Module Rectangles ...
 END Ellipses.


 MODULE Draw;
   ... see page 12 ...
 END Draw.
```

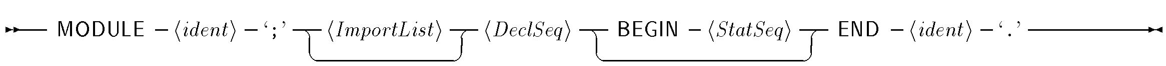
Sample list of commands:

```
Rectangle:
  Draw.CreateFigure
    Rectangles RectDesc 7 7 11 9 ~

Ellipse:
  Draw.CreateFigure
    Ellipses EllipseDesc 1 2 4 6 ~

  Draw.DrawAll ~
  Draw.MoveAll −1 −2 ~
  Draw.Open ~
  Draw.PrintAreas ~
```

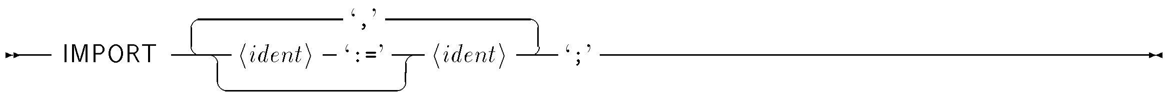
A Syntax of Oberon-2

A.1 Parser

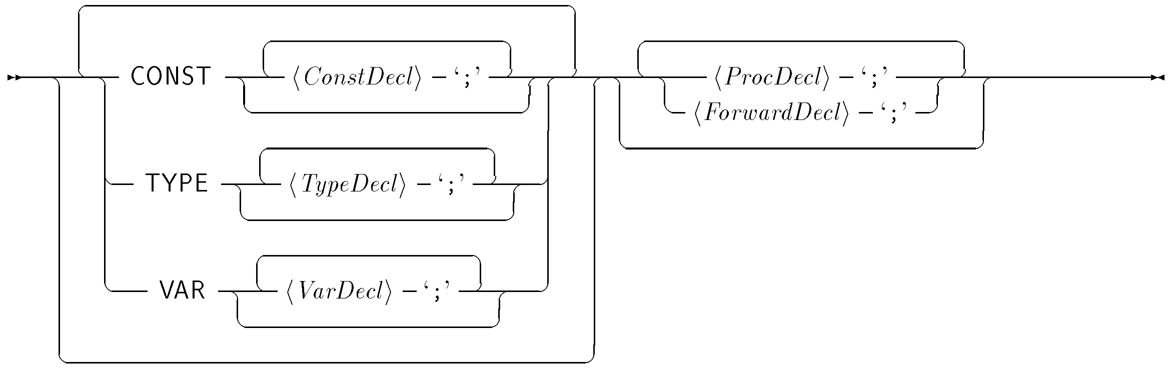
Module:



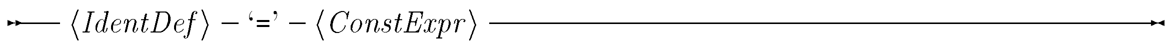
ImportList:



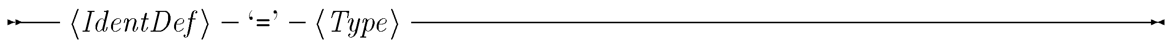
DeclSeq:



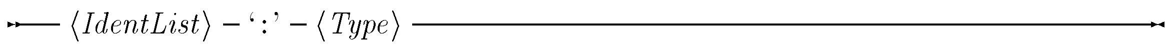
ConstDecl:



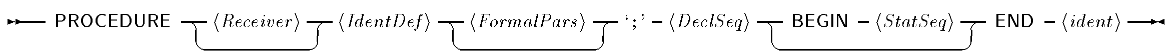
TypeDecl:



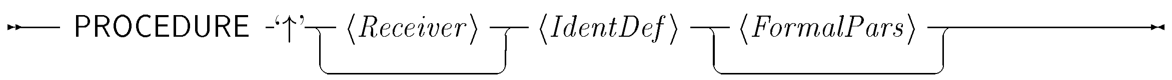
VarDecl:



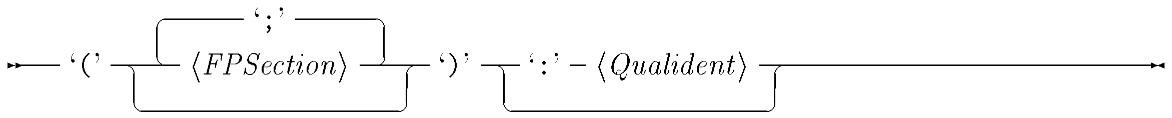
ProcDecl:



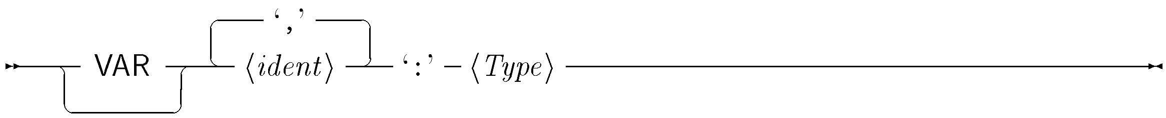
ForwardDecl:



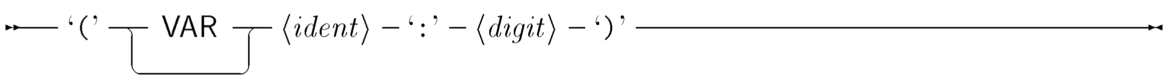
FormalPars:



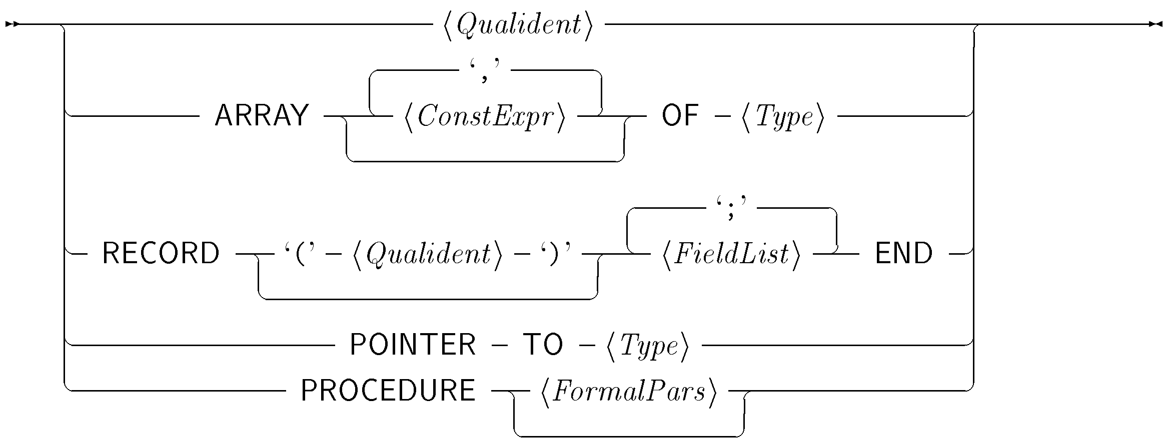
FPSection:



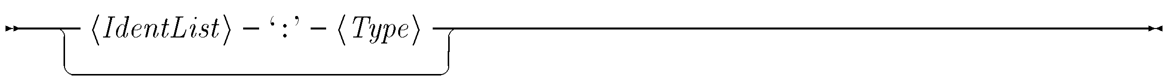
Receiver:



Type:

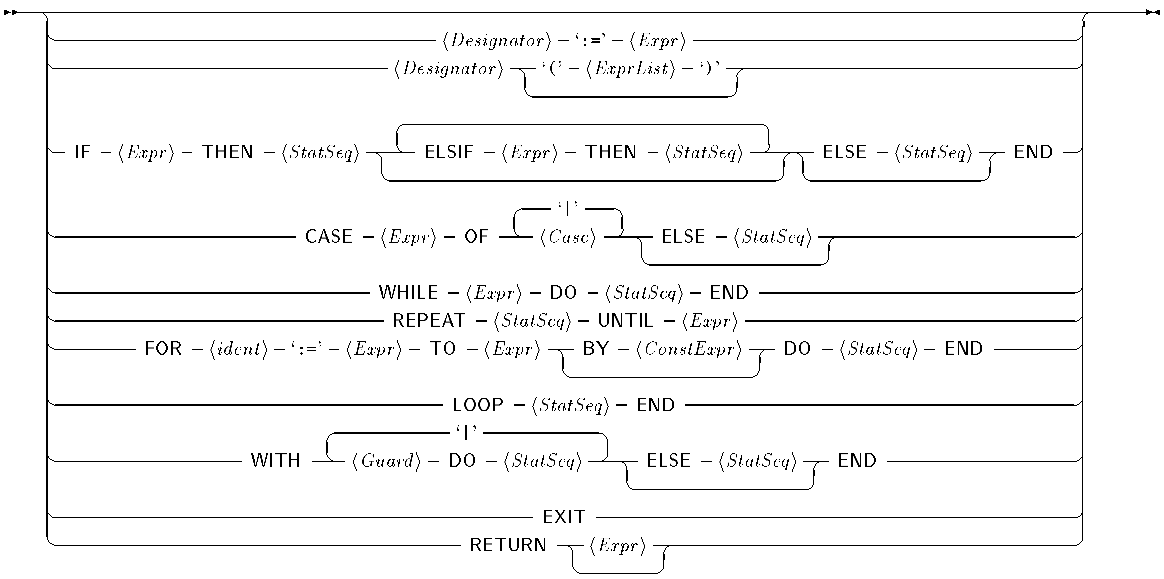


FieldList:

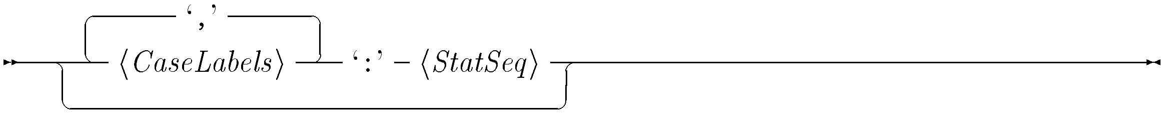


StatSeq:

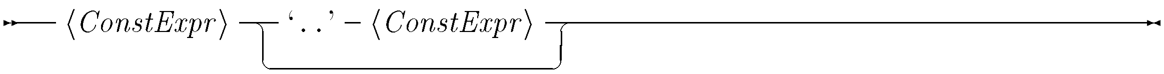
Statement:



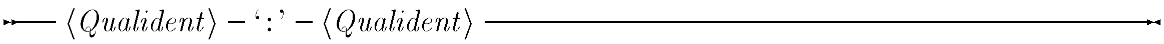
Case:



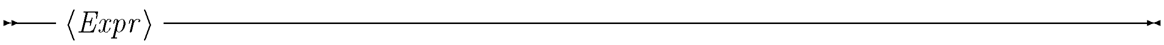
CaseLabels:



Guard:

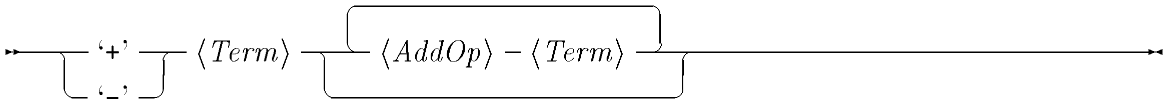


ConstExpr:

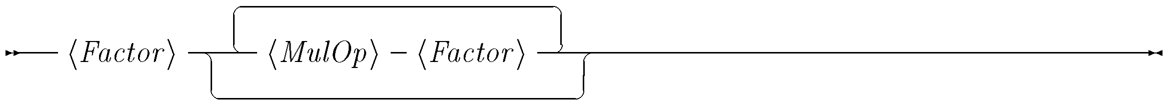


Expr:

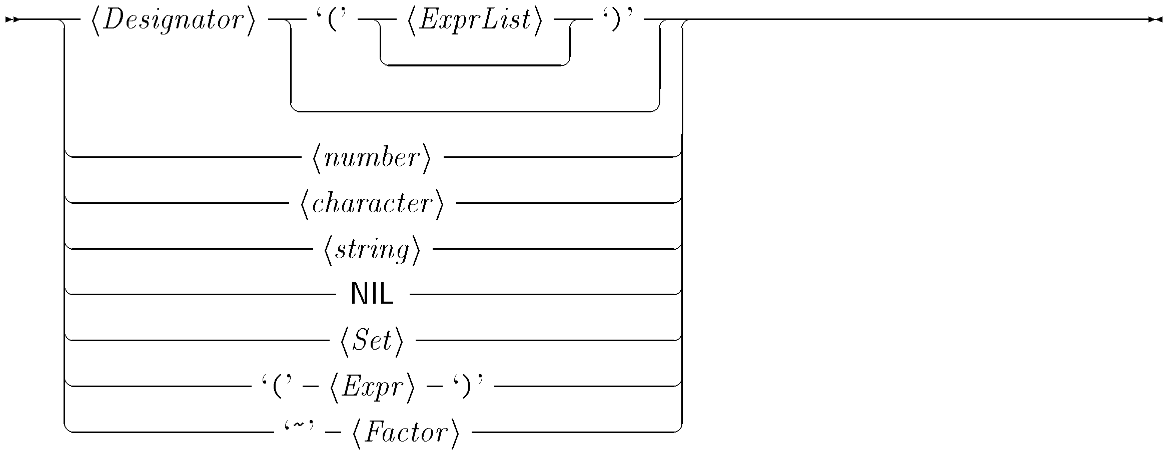
SimplExpr:



Term:

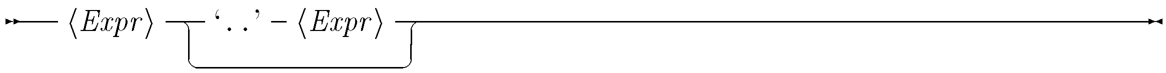


Factor:



Set:

Element:

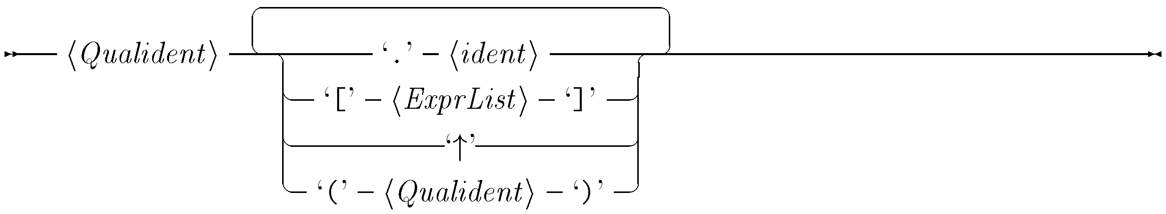


Relation:

AddOp:

MulOp:

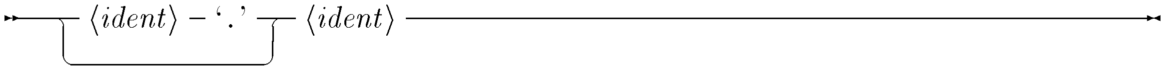
Designator:



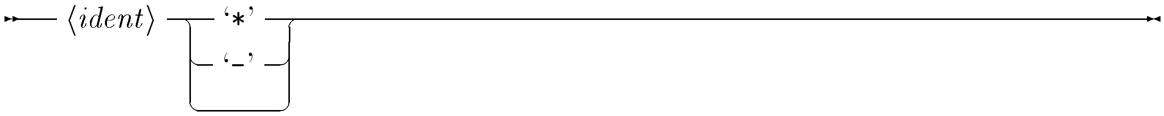
ExprList:

IdentList:

Qualident:

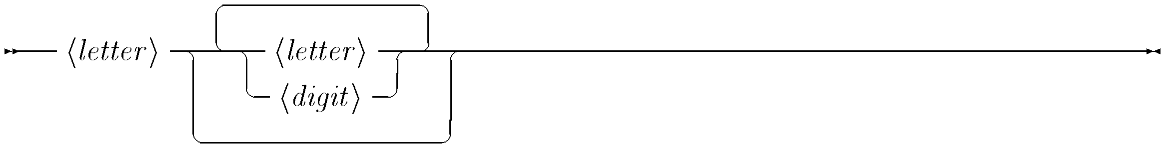


IdentDef:



A.2 Scanner

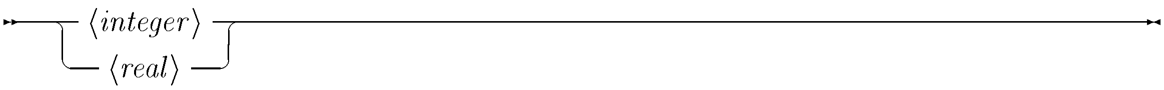
ident:



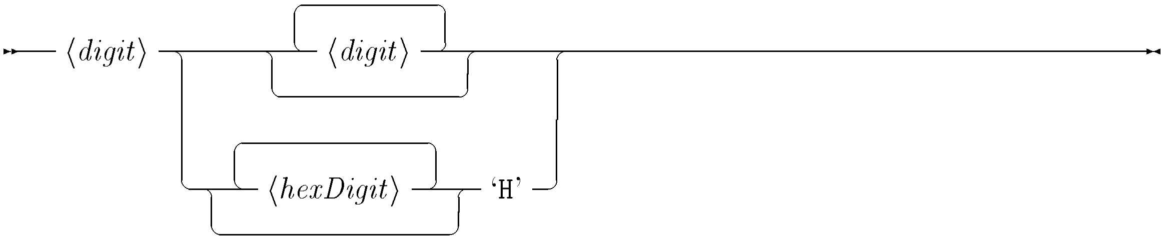
letter:

digit:

number:

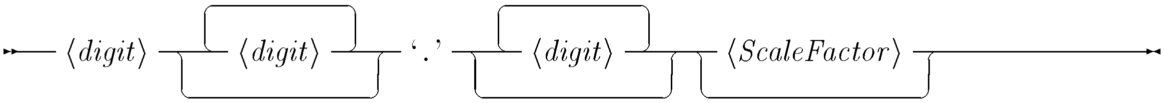


integer:

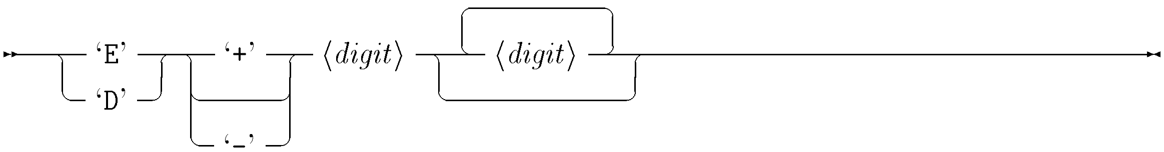


hexDigit:

real:



ScaleFactor:



References

[Bau98] L. Frank Baum. The Wizard of Oz. In *The Wonderful World of Oz*. Penguin Books, 1998.

[FM98] André Fischer and Hannes Marais. *The Oberon Companion: A Guide to Using and Programming Oberon System 3*. vdf Hochschulverlag AG, 1998.

[Fra94] M. Franz. Extensible Programming: ein neues Paradigma für die Softwareentwicklung. In *Informatik-Fachtagung Moderne Programmierparadigmen*. FH Braunschweig/Wolfenbüttel, Fachbereich Informatik, 6.-7. Oktober 1994.

[Gut96] Jürg Gutknecht. Oberon, gadgets, and some archetypal aspects of persistent objects. *Information Sciences*, 93(01):65–86, 1996.

[Mö93] H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer, 1993.

[RW92] M Reiser and N. Wirth. *Programming in Oberon. Steps beyond Pascal and Modula-2*. Addison-Wesley, 1992.

[Ste97] Chr. Steindl. Reflections in Oberon. In *JMLC'97*, volume LNCS 1024. Springer, 1997.

[Szy96] Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[Wir86] N. Wirth. *Algorithmen und Datenstrukturen mit Modula-2*. Teubner, Stuttgart, 1986.