Chapter Seven

Oberon X

7.1 Introduction

Mathematics and scientific calculations often use operators in conjunction with *structured data types*, such as complex numbers, vectors or matrices for example. To allow the construction of expressions with such general data types which deliver structured data types as result, it is necessary to extend the Oberon language by defining new user–defined operators. An immediate consequence of allowing general data types as the result of computations, is that such types should also be allowed to emerge from a function procedure. Then, it becomes possible to integrate such functions as operands in expressions. The use of operators in expressions in place of function procedures offers a decisive advantage of readability, clarity and expressiveness. This is particularly valuable in applications relying on mathematics for their solutions: the transition from the solution's algorithm to its implementation is an easy step and the relation between them remains evident. All these advantages are obtained by a few extensions of the original Oberon language [RW94] which we describe. The extensions are emphasized in bold characters.

7.2 Functions with arbitrary result type

In Oberon [RW94 – A.10.1], the *qualident* which terminates the syntactic production *FormalParameters* must be a *simple type*; that is, array and record structures cannot be the result of function procedures. In Oberon X, this restriction is removed and a function result may be any named structured type or an anonymous open array, as in this example based on a named static array type:

```
TYPE String = ARRAY 64 OF CHAR;
PROCEDURE Upcase (in: String): String;
```

As a corollorary, user–defined operators, described in the next section, may also produce structured types as results. Therefore, RETURN statements, appearing in user–defined operators (see section 7.3) and other functions, may contain expressions composed with structured data types. The assignment compatibility rules apply when records or arrays are returned as results of procedure functions. The modified syntax rules are:

```
FormalParameters = "(" [ FPSection {";" FPSection} ] ")" [ ":"
ResultType ].
ResultType       = {ARRAY OF} qualident.
```

7.3 User–defined operators

For the construction of expressions, the original Oberon language uses the following operators [RW94 – A.8.2]:

```
relation         = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS .
AddOperator      = "+" | "–" | OR .
MulOperator      = "*" | "/" | DIV | MOD | "&" .
```

and the unary operators "+" , "−" , "~"

Oberon X considerably extends this set of operators with *user−defined operators* conceived as procedures, a central concept in Oberon. To define these new operators, the syntax rules of the *ProcedureDeclaration* [RW94 − A.10] have been slightly extended:

```
ProcedureDeclaration  = ProcedureHeading ";" ProcedureBody
                        (ident | operator).
ProcedureHeading      = PROCEDURE ["*"] (ident | operator) ["*"]
                        [FormalParameters].
operator  = """ (
              "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS |
              "+" | "−" | "*" | "/" | DIV | MOD | OR | "&" | "~" |
              "++" | "+−" | "+*" | "+/" | "+~" | "+↑" | "+<" | "+>" |
              "−+" | "−−" | "−*" | "−/" | "−~" | "−↑" | "−<" | "−>" |
              "*+" | "*−" | "**" | "*/" | "*~" | "*↑" | "*<" | "*>" |
              "/+" | "/−" | "/*" | "//" | "/~" | "/↑" | "/<" | "/>" |
              "%" | "\"  | ":=" ) """.
```

The new operators include the standard Oberon ones, which may thus be *redefined* and a provision of additional symbols, to which the assignment symbol ":=" is also added.

An operator is constructed exactly in the same manner as any procedure (or function procedure). Instead of a procedure identifier *ident*, an *operator* symbol is placed in the *ProcedureHeading* and repeated at the end of the *ProcedureDeclaration*. To avoid any misunderstanding with the possible neighbouring asterisks, the operator is enclosed in double quotes (").

Though the syntax rules given in section 7.2 for *FormalParameters* allow any number of formal parameters, their number and the requirement for a result depend on the operator:

Operator	Formal parameters required	Result
Dyadic	two value or variable parameters	result type
Monadic	one value or variable parameter	result type
Assignment	first one must be a variable parameter second one must be a value parameter	no result

The assignment operator is described in the next section.

All these operators have no predefined meaning in Oberon X: they must be used exclusively as user−defined operators. As can be expected, they will be defined in Oberon XSC, but they may equally well be used in other applications. String manipulation algorithms, for example, could make use of them.

The precedence rules and the associativity rules for operators are the same as those dictated in the original Oberon language definition. For the composite operators (all of them begin with "+", "−", "*" or "/"), the precedence and the associativity is determined by the first character. The symbols "%" and "\" are available only as monadic operators and they have the same precedence as "~". To these, one may add the two other monadic operators "+" and "−".

Finally, user−operators may be defined recursively.

Example:

```
PROCEDURE "+" (head, tail: String): String;
BEGIN
   ...
END "+";
```

defines the concatenation operation *head + tail* for the operands of type String. This result is returned by a RETURN statement, just like in any normal function procedure.

7.4 Operator overloading and identification

Within a module, operators may be *overloaded* any number of times as long as they are distinguishable by their formal parameter types. Operator overloading means using a single symbol to represent operators with different operand types. Because of that however, forward declarations are forbidden.

```
PROCEDURE "+" (a, b: INTEGER): INTEGER;
```

redefines the standard addition of INTEGER values and replaces it in its scope. In this manner, the floating point arithmetic provided in Oberon can be replaced by a user–defined arithmetic.

Overloaded operators may be used in the composition of expressions:

dyadic: a **operator** b —— monadic: **operator** a

At compile time, the operands of an expression are paired with the formal parameters of the operators from left to right. Due to operator overloading, several operators may be found eligible. This conflicting situation is resolved by an *operator identification*. Since this identification takes place at compile time, it takes only into account the static type of the operands, not their dynamic types. The algorithm is:

- *consider* the operators in the current scope and those imported
- *identify* all operator declarations whose formal parameter types are
  - direct base types of (i.e. the same as)
  - indirect base types of
  - compatible, according to the hierarchy of numeric types, with
  - open array types compatible with array types of

  the corresponding actual parameter types. All these conditions are tested for all the formal parameters
- *select* among the candidates the operator having the shortest *type distance* to the actual operand types
- if the selection is undecidable, *select* an operator in the module being compiled according to the standard scope rules
- report an error if no operator can be found.

The type distance is a conceptual measure which can be implemented in different ways. In essence, any concrete measurement scheme must tend to select the only operator for which the formal and the actual types are in perfect match. When such a concordance is not found, the type distance could for instance be expressed by the number of type conversion steps to apply to an operand to match the corresponding formal parameter. For dyadic operators, a weighing of the operands could be envisaged. The practical realisation is part of the Oberon X implementation.

A user–defined assignment operator makes it possible to define an assignment for incompatible operand types, otherwise forbidden by the *assignment compatibility* rules. Its implementation always takes the form:

```
PROCEDURE ":=" (VAR left: FormalType1; right: FormalType2)
BEGIN
   ...
END ":=";
```

It is the programmer's responsability to define how the *right* expression is assigned to the variable *left*. In a way, this provides a type conversion function in operator notation.

Two examples of complete operator definition conclude this section. The first one defines an operator for adding two complex values:

```
TYPE Complex = RECORD
                   Re, Im: LONGREAL
               END;

PROCEDURE "+" (a, b: Complex): Complex;
BEGIN
   a.Re := a.Re + b.Re;
   a.Im := a.Im + b.Im;
   RETURN a
END "+";
```

The second one defines an assignment for real vectors of arbitrary but compatible length with the help of open array parameters (see section 7.6):

```
PROCEDURE ":=" (VAR a: ARRAY OF REAL; b: ARRAY OF REAL);
VAR i: INTEGER;
BEGIN
   IF LEN(a) = LEN(b) THEN
      FOR i:= 0 TO LEN(b) - 1 DO
         a[i] := b[i]
      END
   ELSE error
   END
END ":=";
```

The *error* procedure may contain any thinkable error processing routine or a simple HALT statement forcing a program trap.

7.5 Operators in expressions

The modified syntax for the composition of expression [RW – A.8.2] is:

```
expression       = SimpleExpression [relation SimpleExpression].
relation         = [ident "."]
                   ( "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS ).
SimpleExpression  = [ [ident "."]
                   ("+" | "-") ] term {AddOperator term)}.
AddOperator      = [ident "."]
                   ( "+" | "-" | OR |
                   "++" | "+-" | "+*" | "+/" | "+~" | "+^" | "+<" |
"+>" |
                   "-+" | "--" | "-*" | "-/" | "-~" | "-^" | "-<" |
"->" ).
term             = factor {MulOperator factor}.
MulOperator      = [ident "."]
                   ( "*" | "/" | DIV | MOD | "&" |
                   "*+" | "*-" | "**" | "*/" | "*~" | "*^" | "*<" |
"*>" |
                   "/+"  | "/-" | "/*" | "//" | "/~" | "/^"  | "/<" |
"/>" ).
factor           = number | CharConstant | string | NIL | set |
                   designator [ActualParameters] |
                   "(" expression ")" | Unary Operator factor.
UnaryOperator    = [ident "."]
                   ( "~" | "%" | "\" ).
```

7.6 Overloading of standard mathematical functions

Oberon X allows overloading of some mathematical functions of a *single argument* which produce a result of the *same type* as the argument, as in the following case:

```
PROCEDURE SIN (x: Complex): Complex;
```

Mathematically, this is expressed by the definition: *f*: T –> T
where T is any data type. The function name must be spelled entirely in capital letters.

The syntax rules for the overloading of functions are the same as described earlier. The standard functions which may be overloaded are:

Function name	Meaning	Inverse function name
COS	cosine	ARCCOS
COT	cotangent	ARCCOT
SIN	sine	ARCSIN
TAN	tangent	ARCTAN
COSH	hyperbolic cosine	ARCCOSH
COTH	hyperbolic cotangent	ARCCOTH
SINH	hyperbolic sine	ARSINH
TANH	hyperbolic tangent	ARTANH
EXP	power function, base e	LN
EXP2	power function, base 2	LOG2
EXP10	power function, base 10	LOG10
SQRT	square root	SQR

7.7 Extended arrays

Unless they appear as open array parameters in a procedure header, arrays in Oberon need a specified size at compile time.

In Oberon X, the syntax rules for the *ArrayType* [RW94 – A.6.2] make it possible to construct three array variants:

```
ArrayType = ARRAY length { "," length } OF type.
length    = expression | "*".
```

First, the length of a dimension may take the shape of an asterisk, corresponding to the already familiar concept of dynamic array. Second, the length may take the shape of an arbitrary expression, introducing the new concept of semidynamic array. Finally, the original concept of static array is naturally integrated in the new syntax. The array type is thus determined by the length specification.

By definition, all these array variants may be *multidimensional*, that is, the array elements may themselves be arrays, and mixing the different length specification forms is in principle acceptable. But this possibility is restricted by the implementation. In any array type, the array elements must be of the same type as the base type or of the static type. All these array types are homogenous. An example and a counterexample are:

```
TYPE acceptable = ARRAY * OF ARRAY 42 OF T;    (*   ARRAY *, 42 OF T;   *)
TYPE invalid = ARRAY 42 OF ARRAY * OF T;
```

For multidimensional arrays, the shorthand notation should be preferred but it cannot be used for open arrays.

Static arrays

The number of elements in any dimension is fixed and expressed by a *constant expression*. This is the classical Oberon construction.

Open arrays

The number of elements is variable and left open. This type is used as formal type in what is known as "open array parameter". In Oberon X, it may also be used as result type for procedures and function procedures (see section 7.2.2).

Semidynamic arrays

The number of elements in any dimension is variable and expressed by an *arbitrary but, at the same time, restricted expression* of integer type. The expressions may only be composed of variables *local to the containing procedure* and function calls may not be used as operands. Because the value of the length expressions must be well–defined at scope activation time, semidynamic array variables may only be defined as local procedure variables or as result type in procedures and procedure functions.

These length expressions are evaluated at run–time when entering the body of the surrounding procedure. All the computed values must represent positive integers, else a run–time error will be detected. For *named* array types, the lengths remain constant during the procedure execution, even if the values of the variables from which they were computed change. The computed lengths apply to all the variables of this array type over its entire scope. This limitation allows to allocate the array storage space from the stack, thereby making it very efficient.

The following example should demonstrate how this works. x1 and x2 are both 10 x 10 square matrices since they are both of type Mat. In contrast, the semidynamic arrays x2 and y2 of anonymous type differ in sizes.

```
PROCEDURE p (n: INTEGER);
TYPE Mat: ARRAY n, n OF REAL;
VAR x1: Mat;    x2: ARRAY n, n OF REAL;

    PROCEDURE q (n: INTEGER);
    VAR y1: Mat;    y2: ARRAY n, n OF REAL;
    . . .
    END q;

BEGIN
    n := n + 5;
    q(n);
END p;

p(10);
```

The next function procedure initializes a two–dimensional n x n matrix, passing the result in an open array:

```
PROCEDURE zero (n: INTEGER): ARRAY OF ARRAY OF REAL;
VAR i, j: INTEGER; Mat: ARRAY n, n OF REAL;
BEGIN
    FOR i := 0 TO LEN(Mat, 0) – 1 DO
        FOR j := 0 TO LEN(Mat, 1) – 1 DO
            Mat[i, j] := 0.0
        END
    END;
    RETURN Mat
END zero;
```

Dynamic arrays

The number of elements in any dimension is variable and is denoted by an asterisk. It is the programmer's responsability to allocate storage space on the heap for a dynamic array by calling the standard procedure NEW:

```
NEW ( arrayvariable, length0, length1, . . . );
```

The length values may be expressed by *arbitrary expressions* of integer type and the number of length entries must correspond to the number of dimensions of the variable. At execution time, the evaluation of each expression must result in a positive integer, else a run–time error will be detected. To prevent accessing array elements in an array variable which has not yet been allocated, all the dimension lengths are initialized to 0.

```
MODULE DynVector;
IMPORT In;
TYPE field = ARRAY * OF INTEGER;

PROCEDURE ReadField (VAR a: field);
VAR i, n: INTEGER;
BEGIN
   In.Int(n);
   NEW(a, n);
   FOR i := 0 TO LEN(a) – 1 DO
      In.Int(a[i])
   END
END ReadField;
```

Dynamic array pointers

Oberon X slightly extends the Oberon pointer concept without need for changing the syntax rules. The already available pointers to *static array* are now complemented by *dynamic array pointers*.

```
POINTER TO ARRAY * OF T;
```

the basic dynamic array pointer, is conceptually different from a pure dynamic array in that it implicitely defines a nested dynamic structure. First, a pointer always references an object which must be instantiated on the heap with a NEW call, and second, a dynamic array is also a structure which must be allocated at run–time. Whatever the implementation, the programmer may assume that the allocation of a pointer–based dynamic array will require two allocation steps. However, these two steps will be implicitely performed by a single standard procedure call

```
NEW(arraypointervariable, length0, length1, . . . );
```

The specifications for the length expressions are exactly the same as those described for dynamic arrays.

Accessing the elements of such a structure will accordingly require two dereferencing steps, resulting in a run–time performance penalty compared to pure dynamic arrays. A noticeable advantage is that *inhomogenous* array structures can be constructed with the help of such pointers. Implementation restrictions will not allow that using pure dynamic arrays.

Assignment compatibility rules

The existing assignment compatibility rules [RW94 – 4.3.1] have been slightly modified to suit the requirements of arrays.

- an array assignment is always allowed when the participating operand types have the same name

- assigning a dynamic array to another one of the same type, leads to the implicit re–allocation of the target array. The implementation may avoid a re–allocation when all the dimension lengths of the participating arrays are matching
- any array type (static, dynamic, semidynamic or open) may be assigned to an open array, provided the participating arrays have the same structure.
  This applies equally to passing a value to an open array parameter and a function result with a RETURN statement.

For example, two array variables based on these different types would be isomorphous yet incompatible:

```
TYPE Mat1 = ARRAY *, * OF T;
TYPE Mat2 = ARRAY * OF ARRAY * OF T;
```

All other possible combinations of array types as operand types in assignments are not supported by the compiler in favor of type safety. To work around this, the user may construct user–defined assignment operators performing array type conversions. For example, declaring

```
PROCEDURE "+" (a, b: ARRAY OF T): ARRAY OF T;
PROCEDURE ":=" (VAR a: ARRAY OF T; b: ARRAY OF T);
```

makes it possible to write `w := (x + y) + z;`
for operands of any array type.

Assignments of array pointer variables behave naturally and array pointer variables may be explicitely dereferenced:

```
VAR p, q: POINTER TO ARRAY ...;

p  := q;     (* copies the pointer *)
p↑ := q↑;    (* Reallocates p↑ and copies q↑ *)
```

The standard notation is used to access the elements of array variables. If index checking is enabled for the compilation, an access to an array element out of specified bounds will be detected.

7.8 FOR statement

Oberon does not define a counting loop which is very convenient for the systematic processing of arrays. This construct was introduced in Oberon–2 [RW94 – 14.2] and adopted by Oberon X.

7.9 Examples

Definition of an assignment for initializing a m x n matrix:

```
TYPE MatType: ARRAY m, n OF REAL;
VAR Mat: MatType;

PROCEDURE ":=" (VAR a: MatType; r: REAL);
VAR i, j: INTEGER;
BEGIN
   FOR i := 0 TO LEN(a, 0) – 1 DO
      FOR j := 0 TO LEN(a, 1) – 1 DO
         a[i, j] := r
```

```
        END
    END
END ":=";

Mat := 0;
```

Definition of a matrix assignment:

```
PROCEDURE ":=" (VAR r: ARRAY OF ARRAY OF REAL;
                                e: ARRAY OF ARRAY OF REAL);
VAR i, j: INTEGER;
BEGIN
    IF (LEN(r, 0) = LEN(e, 0)) & (LEN(r, 1) = LEN(e, 1)) THEN
        FOR i:= 0 TO LEN(r, 0) – 1 DO
            FOR j:= 0 TO LEN(r, 1) – 1 DO
                r[i, j] := e[i, j]
            END
        END
    ELSE error
    END
END ":=";
```

Definition of a row vector [1 x cols] by a column vector [rows x 1] product operator (corresponds to the familiar inner vector product). To be conformable for multiplication, the two vectors must have the same length (cols = rows). The result is a scalar [1 x 1].

```
TYPE AType: ARRAY m OF REAL;
VAR A, B: AType; Res: REAL;

PROCEDURE "*" (a, b: ARRAY OF REAL): REAL;
VAR i: INTEGER; scalar: REAL;
BEGIN
    IF LEN(a) = LEN(b) THEN
        (* This test can be omitted when the procedure header is:
              PROCEDURE "*" (a, b: AType): REAL;
           but the operator will loose its generality.                *)
        scalar := 0;
        FOR i := 0 TO LEN(a) – 1 DO
            scalar := scalar + a[i] * b[i]
        END;
        RETURN scalar
    ELSE error
    END
END "*";

Res := A * B;     (* Uses the user–defined "*" and the standard ":=" *)
```

Definition of a column vector [cols x 1] by a row vector [1 x rows] product operator (also known as outer vector product). The two vectors are necessarily conformable for multiplication. The result is a matrix [cols x rows].

```
TYPE CType: ARRAY n OF REAL;
VAR C: CType;

PROCEDURE "&" (a, b: ARRAY OF REAL): ARRAY OF ARRAY OF REAL;
VAR i, j: INTEGER; M: MatType;
BEGIN
    FOR i := 0 TO LEN(a) – 1 DO
        FOR j := 0 TO LEN(b) – 1 DO
            M[i, j] := a[i] * b[j]
        END
    END;
    RETURN M
END "&";

Mat := A & C;     (* Uses the user–defined ":=" and "&" *)
```

The operand symbol "*" may not be used again if it is in conflict with the previous "*" in the same context. The type distance will not help discriminating the two operator definitions.

7.10 Discussion and summary

The concept of operator identification is already present in the original Oberon where several different operations are designated by the same operator symbol. In these case, the actual operation is identified by the type of the operand. A good example is given by the operator "+" which may be combined with operators of any arithmetic type but also with SET type operators.

A user–defined assignment makes most often sense when other operators are defined for the data types involved in the assignment.

In comparison with static arrays, the dimensions of dynamic arrays can be adapted at run–time to the requirements of the applications. In that manner, algorithms operating in vector spaces can be used without modification to operate on vectors and matrices of different size.

7.x Bibliography

[Jan98] Peter Januschke
Oberon XSC
Dissertation, Universität Karlsruhe, 1998.

[Mor97] R. Morelli
Integration of Oberon X in Oberon
Diplomarbeit, Institut für Computersysteme, ETH Zürich, 1997

[RW94] Martin Reiser and Niklaus Wirth
Programming in Oberon – Steps beyond Pascal and Modula
Addison–Wesley Publishing Company, 1992. ISBN 0–201–56543–9

[Tem90] Josef Templ
SPARC–Oberon – User's Guide and Implementation
Departement Informatik ETHZ, Heft 133, 1990.