Type declarations

A data type determines the set of values that variables of that type may assume, and the operators that are applicable. A type declaration is used to associate an identifier with the type. Such association may be with unstructured (basic) types, or it may be with structured types, in which case it defines the structure of variables of this type and, by implication, the operators that are applicable to the components. There are two different structures, namely arrays and records, with different component selectors.

```
$  TypeDeclaration  =  identdef "=" type.
$  type  =  qualident | ArrayType | RecordType | PointerType |
         ProcedureType.
```

Examples:

```
 Table      =  ARRAY N OF REAL
 Tree       =  POINTER TO Node
 Node       =  RECORD key: INTEGER;
               left, right: Tree
              END

 CenterNode  =  RECORD (Node)
                name: ARRAY 32 OF CHAR;
                subnode: Tree
               END

 Function*   =  PROCEDURE (x: INTEGER): INTEGER
```

Record types

A record type is a structure consisting of a fixed number of elements of possibly different types. The record type declaration specifies for each element, called a field, its type and an identifier that denotes the field. The scope of these field identifiers is the record definition itself, but they are also visible within field designators (see Section 8.1) referring to elements of record variables.

```
$  RecordType  =  RECORD ["(" BaseType ")"] FieldListSequence END.
$  BaseType  =  qualident.
$  FieldListSequence  =  FieldList {";" FieldList}.
$  FieldList  =  [IdentList ":" type].
$  IdentList  =  identdef {"," identdef}.
```

If a record type is exported, field identifiers that are to be visible outside the declaring module must be marked. They are called
public fields; unmarked fields are called private fields.

Record types are extensible, that is, a record type can be defined as an extension of another record type. In the examples
above, CenterNode (directly) extends Node, which is the (direct) base type of CenterNode. More specifically, CenterNode
extends Node with the fields name and subnode.

Definition: A type T0 extends a type T, if it equals T or if it directly extends an extension of T. Conversely, a type T is a base
type of T0, if it equals T0, or if it is the direct base type of a base type of T0.

Examples:

```
 RECORD day, month, year: INTEGER
 END

 RECORD
   name, firstname: ARRAY 32 OF CHAR;
   age: INTEGER;
   salary: REAL
 END
```

Pointer types

Variables of a pointer type P assume as values pointers to variables of some type T. The pointer type P is said to be bound to
T, and T is the pointer base type of P. T must be a record or array type. Pointer types inherit the extension relation of their
base types. If a type T0 is an extension of T and P0 is a pointer type bound to T0, then P0 is also an extension of P.
$ PointerType = POINTER TO type.

If p is a variable of type P = POINTER TO T, then a call of the predefined procedure NEW(p) has the following effect (see
10.2): A variable of type T is allocated in free storage, and a pointer to it is assigned to p. This pointer p is of type P; the
referenced variable p^ is of type T. Failure of allocation results in p obtaining the value NIL. Any pointer variable may be
assigned the value NIL, which points to no variable at all.

- type−bound procedures (methods)
- one initializer
- one body

The design of this language extension has been driven by unification and symmetry. We limited changes to the absolute minimun and reused known concepts as scopes and locality.

Type−bound Procedures

The type−bound procedures are declared locally to the record scope. A type−bound procedure can override another type−bound procedure declared in any base type of the record, but it must have the same name and the same signature. The call of an overridden procedure can only be done in a type−bound procedure by using an arrow after the procedure name.

```
TYPE
   ObjectDesc = RECORD
      VAR
         local, local2: LocalData;

      PROCEDURE P;
      BEGIN
         (* do something *)
      END P;

   END ObjectDesc;
```

Record fields and other methods of the same record can be accessed by naming them (without qualifier) as they are part of the scope containing the procedure. The reserved keyword SELF can be used access the fields and methods when they are shadowed by a locally declared symbol.

```
TYPE
   ObjectDesc = RECORD
      VAR
         local, local2: LocalData;

      PROCEDURE P;
      BEGIN  local := local2              (*copy the fields*)
      END P;
      PROCEDURE Q(local: LocalData);
      BEGIN  SELF.local := local       (*assign the parameter to the field local*)
   END ObjectDesc;
```

Initializers

A record can have an initialization procedure. This procedure allows the parametrisation of the records during their allocation. This procedure is called automatically by the NEW command. If it has parameters, they must be passed to NEW. The initializer procedure is always automatically exported.

```
TYPE
   Object = POINTER TO ObjectDesc;
   ObjectDesc = RECORD
      VAR
         local: LocalData;

      PROCEDURE & Init(ext: LocalData);    (*initializer*)
      BEGIN
         (* initialize the local fields *)
         local := F(ext)
      END Init;
   END ObjectDesc;

VAR x: Object;
BEGIN ... NEW(x, SomeData); ...
END
```

Record Body

This extension is implemented only in systems supporting concurrency.
A Record can have a body, which represents the record activity. The body is executed asynchronously after the record has been allocated and the initializer called. A body doesn't override the bodies of the base types of a record. All bodies of the record hierarchy are executed concurrently.

```
TYPE
   Object = POINTER TO ObjectDesc;
   ObjectDesc = RECORD
      VAR
         local: LocalData;
   BEGIN
      (*object activity*)
   END ObjectDesc;

VAR x: Object;
BEGIN
   ...
   NEW(x); (*the record body is executed concurrently*)
   ...
END
```

Syntax

```
  Scope    = DeclSeq Body.
  DeclSeq  = {CONST {ConstDecl";"} | TYPE {TypeDecl} | VAR {VarDecl";"} |
ProcDecl";"}.
  TypeDecl = IdentDef "=" Type.
  Type     = ...| RECORD ["("Qualident")"] (Scope | FieldList) | ....
```

We changed the syntax of the Record declaration: a record can be defined by a FieldList (Oberon) or a Scope (Active Oberon).

The Active Oberon Language / Access Protection

An object can be seen as a resource and the various active objects may potentially compete for the use of the object, so that some kind of access protection is indispensable. Our protection model is similar to a monitor. Every single object instance is protected. If a type−bound procedure cannot be accessed, because the object is currently protected, the calling object is preempted by the system until the called object is accessible again. We implement a Reader−Writer access protection model. Type−bound procedures can either be unprotected, shared or exclusive.

- unprotected: no access control is done. Usually these procedures don’t read or write the data of the object, the just call other procedures.
- shared: many shared entries are allowed. The system ensures that no exclusive procedure is executed in the current object as long as a shared one is executing.
- exclusive: the system ensures that no other protected procedures (exclusive or shared) are executed in the current object as long as a this one is executing.

The initializer and the body of a record can be protected. The protection type is specified by an annotation after the BEGIN of the procedure body.

```
TYPE
   Object = POINTER TO ObjectDesc;
   ObjectDesc = RECORD
      VAR t: T;

      PROCEDURE P(t: T);
      BEGIN {EXCLUSIVE}    (* mutually exclusive access *)
      END P;

      PROCEDURE Q(t: T);
      BEGIN {SHARED}    (* shared access *)
      END Q;

      PROCEDURE R;
      BEGIN    (* unprotected*)
      END R;
   END ObjectDesc;
```

The same access protection is implemented for modules. Every module can be used as a monitor.

The Active Oberon Language / Passivate

PASSIVATE

The PASSIVATE command is used to synchronize a thread with a state of the system. As long as the condition describing the state is FALSE, the thread remains passivated. If the procedure containing the PASSIVATE instruction is protected, the protection of the object containing PASSIVATE is released while the thread is preempted. The PASSIVATE condition can be any boolean expression. The evaluation of the expression is not allowed to block, i.e. the evaluation must not call protected procedures.

local conditions vs. global conditions

Local conditions are conditions based upon data local to an object. This implies that the data can only be changed by the object methods. This contrasts to the global data, which can be changed by any process in the system.
Reevaluation stategy:

- local conditions are reevaluated every time a protected (EXCLUSIVE or SHARED) method in the object is called. Unprotected entries don't cause a reevaluation.
- global conditions are polled. The polling strategy is system dependent.

EXAMPLE 1:

```
MODULE Buffers; (* bounded buffer *)

CONST
	BufLen = 256;

TYPE
	Buffer = POINTER TO BufferDesc;
	BufferDesc = RECORD (OBJECT)
		VAR
			data: ARRAY BufLen OF CHAR;
			in, out: LONGINT;

		PROCEDURE Put* (ch: CHAR);
		BEGIN {EXCLUSIVE}
			PASSIVATE ((in + 1) MOD BufLen # out);
			data[in] := ch; in := (in + 1) MOD BufLen
		END Put;

		PROCEDURE Get* (VAR ch: CHAR);
		BEGIN {EXCLUSIVE}
			PASSIVATE (in # out);
			ch := data[out]; out := (out + 1) MOD BufLen
		END Get;

		PROCEDURE & Init;
		BEGIN
			in := 0; out := 0;
		END Init;

	END BufferDesc;

END Buffers
```

EXAMPLE 2

```
MODULE Barriers;
```

```
(*
        Demo with barriers: every Accumulator is synchronized to the
        next 100000 multiple: an accumulator can continue only when all
        the accumulators have reached the limit.
*)
  IMPORT Kernel, Streams, Randoms;

  TYPE
    Barrier = POINTER TO BarrierDesc;
    BarrierDesc = RECORD (OBJECT)
      VAR n, N: LONGINT;

      PROCEDURE Enter*;
        VAR i: LONGINT;
      BEGIN { EXCLUSIVE } i := n DIV N; INC(n); PASSIVATE (i < n DIV N)
      END Enter;

      PROCEDURE & Init (nofProcs: LONGINT);
      BEGIN N := nofProcs; n := 0
      END Init;

    END BarrierDesc;

  VAR B: Barrier;

  TYPE
    Accumulator = POINTER TO AccuDesc;
    AccuDesc = RECORD (Kernel.ObjDesc)
      VAR S, L: LONGINT; M: REAL; log: Streams.Stream;
    BEGIN { PARALLEL }
      NEW(log, "Accumulator", 100);
      S := 0; L := 0; M := Randoms.UniRand()*10000;
      REPEAT
        S := S + ENTIER(Randoms.UniRand()*M);
        log.Int(S); log.Ln;
        IF S < L THEN Hold(10) ELSE B.Enter; L := L + 100000 END
      UNTIL S > 1000000
    END AccuDesc;

  PROCEDURE Start*;
    VAR A: Accumulator;
  BEGIN NEW(B, 5); NEW(A); NEW(A); NEW(A); NEW(A); NEW(A)
  END Start;
END Barriers
```

EXAMPLE 3

```
(* Parallel tree insertion *)
MODULE Trees;
```

```
IMPORT Kernel, Randoms, Streams, SYSTEM;

TYPE
  Lock = RECORD (OBJECT)
    VAR free: BOOLEAN;

    PROCEDURE Seize;
    BEGIN {EXCLUSIVE} PASSIVATE(free); free := FALSE
    END Seize;

    PROCEDURE Free;
    BEGIN {EXCLUSIVE} free := TRUE
    END Free;

  END Lock;

  Node = POINTER TO NodeDesc;
  NodeDesc = RECORD
    VAR left, right: Node; key: REAL; lock: Lock;

    PROCEDURE Insert (new: Node);
      VAR i: INTEGER;
    BEGIN lock.Seize;
      IF new.key <= key THEN
        IF left = NIL THEN
          FOR i := 0 TO 9999 DO END;
          left := new; lock.Free
        ELSE lock.Free; left.Insert(new)
        END
      ELSE
        IF right = NIL THEN
          FOR i := 0 TO 9999 DO END;
          right := new; lock.Free
        ELSE lock.Free; right.Insert(new)
        END
      END
    END Insert;

    PROCEDURE& Init;
    BEGIN lock.Free
    END Init;

  END NodeDesc;

VAR root: Node; log: Streams.Stream;

TYPE
```

```
   Agent = POINTER TO AgentDesc;
   AgentDesc = RECORD

VAR i: INTEGER; x: Node;
   BEGIN { PARALLEL, TIMESLICED} i := 0;
    REPEAT
      NEW(x); x.key := Randoms.UniRand(); root.Insert(x); INC(i)
    UNTIL i = 2000;
    log.Int(Kernel.GetTimer()); log.Ln
   END AgentDesc;

  PROCEDURE Start*;
   VAR i: INTEGER; a: Agent;
  BEGIN
   log.Int(Kernel.GetTimer()); log.Ln;
   NEW(root); root.key := Randoms.UniRand();
   FOR i := 0 TO 49 DO NEW(a) END
  END Start;

  PROCEDURE CheckSubtree (x: Node; VAR n: LONGINT);
   VAR nL, nR: LONGINT;
  BEGIN nL := 0; nR := 0;
   IF (x.left # NIL) & (x.left.key <= x.key) THEN CheckSubtree(x.left, nL) END;
   IF (x.right # NIL) & (x.right.key > x.key) THEN CheckSubtree(x.right, nR) END;
   n := nL + 1 + nR
  END CheckSubtree;

  PROCEDURE Check*;
   VAR n: LONGINT;
  BEGIN CheckSubtree(root, n);
   log.Int(n); log.Ln
  END Check;

BEGIN NEW(log, "Trees", 150)
END Trees.
```

EXAMPLE 4:

Consider the standard synchronisation problem consisting of 5 philosophers who alternatively think and eat. To eat, a philosopher needs 2 forks, but unfortunately there are only 5 forks on the circular table and each philosopher is only allowed to use 2 forks

nearest to him. Obviously 2 neighbours cannot eat at the same time.

```
MODULE Philosophers;
(*
        Dining Philosophers implementation
*)

        IMPORT Kernel, Streams, Randoms, Oberon, SYSTEM;

        CONST N = 5;
                TimeUnit = 300; (*sec*)
                MaxThinkTime = 5*TimeUnit;
                MaxEatTime = 2*TimeUnit;

        TYPE
                Fork = POINTER TO ForkDesc;
                Philosopher = POINTER TO PhiloDesc;

        VAR
                log: Streams.Stream;
                Forks: ARRAY N OF Fork;
                Philosophers: ARRAY N OF Philosopher;
                exit: BOOLEAN;

                PROCEDURE^ Out;

                TYPE ForkDesc = RECORD (OBJECT)
                                        VAR state: CHAR;

                                        PROCEDURE Take;
                                        BEGIN {EXCLUSIVE} PASSIVATE (state =
"−");                                 state:=”+”; Out
                                        END Take;

                                        PROCEDURE Release;
                                        BEGIN {EXCLUSIVE} state := "−"; Out
                                        END Release;

                BEGIN state := "−"
                END ForkDesc;



                PROCEDURE Out;
                VAR i: INTEGER;
                BEGIN
                        log.Clear;
```

```
                FOR i := 0 TO N−1 DO log.Char(Forks[i].state) END
        END Out;

        PROCEDURE TakeForks (i: INTEGER);
        BEGIN {EXCLUSIVE} Forks[i].Take; Forks[(i+1)MOD N].Take
        END TakeForks;

        TYPE PhiloDesc = RECORD (Kernel.ObjDesc)
                        VAR n: INTEGER; log: Streams.Stream;

                        PROCEDURE Think;
                        BEGIN
                                log.Clear;
                                log.String("−> Thinking"); log.Ln;
                                Hold(ENTIER(1 +
Randoms.UniRand()*MaxThinkTime));
                        END Think;

                        PROCEDURE Eat;
                        BEGIN
                                log.String("−> Take Forks"); log.Ln;
                                TakeForks(n);
                                log.String("−> Eating"); log.Ln;
                                Hold(ENTIER(1+Randoms.UniRand() *
MaxEatTime));
                                Forks[n].Release; Forks[(n+1)MOD
N].Release;
                        END Eat;

                        PROCEDURE& Init (i: INTEGER);
                        BEGIN n := i; NEW(log, "", 100)
                        END Init;

        BEGIN {PARALLEL(2)}
                REPEAT Think; Eat UNTIL exit
        END PhiloDesc;



        PROCEDURE Start*;
        VAR i: INTEGER;
```

```
        BEGIN
                exit := FALSE;
                FOR i := 0 TO N−1 DO NEW (Forks[i]) END;
                FOR i := 0 TO N−1 DO NEW (Philosophers[i], i) END
        END Start;

        PROCEDURE Stop*;
        VAR i: INTEGER;
        BEGIN exit := TRUE
        END Stop;

BEGIN NEW(log, "Log", 50)
END Philosophers.
```

EXAMPLE 5:

```
MODULE Lorenz;
(*
        Visualisation of the Lorenz attractor using the Dynamics module and
        the operator feature of the language
```

```
*)

  IMPORT Dynamics;

  TYPE
    Object = POINTER TO ObjDesc;
    ObjDesc = RECORD (Dynamics.ObjDesc)
      VAR sigma, r, b: REAL;

      PROCEDURE f (x: Dynamics.Vector): Dynamics.Vector;
            VAR y: Dynamics.Vector;
      BEGIN y[0] := sigma*(x[1]−x[0]); y[1] := −x[0]*x[2] + r*x[0] − x[1]; y[2] :=
x[0]*x[1] − b*x[2];
            RETURN y
            END f;

      PROCEDURE& Init (col0: INTEGER; sigma0, r0, b0, x0, y0, z0, dt0: REAL);
      BEGIN sigma := sigma0; r := r0; b := b0; InitState (col0, x0, y0, z0, dt0)
      END Init;

    END ObjDesc;

  PROCEDURE Start*;
    VAR x: Object;
  BEGIN Dynamics.Start;
    NEW(x, 1, 10, 28, 8/3, 1.0, 1.0, −10.0, 0.00025);
    NEW(x, 2, 10, 28, 8/3, 1.5, 0.5, −15.0, 0.00025);
    NEW(x, 3, 10, 28, 8/3, 0.5, 1.5, −20.0, 0.00025)
  END Start;

  PROCEDURE Stop*;
  BEGIN Dynamics.Stop
  END Stop;

END Lorenz.
```