Persistent Oberon
Language Specification

Luc Bläser
Institute of Computer Systems
ETH Zurich, Switzerland
blaeser@inf.ethz.ch

The programming language Persistent Oberon is an extension of Active Oberon [Gut97, Reali02] with institutionalized support of data persistence. Modules constitute the intrinsic persistent roots of this language, representing state-full singleton instances with conceptually infinite lifetime. A module is automatically loaded and initialized by the system exactly once, when it is used for the first time, either by the user or another loaded module importing this module. Thereafter, the module stays permanently alive and survives all subsequent system restarts. As a consequence, the entire data state of modules is automatically persistent. Naturally, references also belong to this persistent state and by default, remain valid at system restart. Hence, all objects that are transitively reachable from modules are persistent.

As the persistent state should always be available in a consistent way when the system is resumed after an interruption or failure, the program execution has to necessarily reflect the states of consistency. For this purpose, the language features the concept of transactions, which define statement sequences that change the program from one consistent state to another. In Persistent Oberon, a normal statement sequence can be defined as a transaction. All modifications, which are performed by the execution of a transaction (including the code of directly or indirectly called procedures), are either completely applied or not at all. During the unfinished transactions, these changes are only temporarily valid and are discarded at a system interruption. A transaction may also execute statement sequences which are again defined as transactions, leading to a scenario of nested transactions [Moss85].

The following language specification describes the new or changed concepts of Persistent Oberon, as extensions to the underlying Active Oberon language report [Reali02]. The complete syntax specification of Persistent Oberon is summarized in Appendix A.

1 Data Longevity

1.1 Syntax of Types

Figure 1 shows the modified EBNF-syntax of types with attribute support. *RefAttr*, called *reference attributes*, are the attributes denoting the semantics of a reference to a declared type. The reference semantics of a type are only specified on the outermost level where the type of elements in a syntactic construct must be defined, i.e. the type definition of

variables or fields but also the element-type of an array type and the type of the referenced element of an explicit pointer type. The reference attributes are written in front of the element-type whose reference semantics is defined. Types with reference semantics are object-, pointer, procedure- and delegate-types, where the reference of a delegate points to the object associated with the delegate and a procedure-reference refers to the procedure's module. This syntax of reference attributes prohibits the reference semantic to be directly attached with a declared type and thus, emphasizes orthogonal lifetime-definition of types.

The attributes appearing right after the tokens ARRAY, RECORD, OBJECT or PROCEDURE give semantic information inherent the declared type and are not related with reference semantics.

```
VarDecl      = IdentList ':' [RefAttr] Type.
Type         = Qualident
             | ARRAY [ArrTypeAttr] ConstExpr {',' ConstExpr} OF [RefAttr] Type
             | RECORD [RecTypeAttr] ['(' Qualident ')'] [FieldList] END
             | POINTER [PtrTypeAttr] TO [RefAttr] Type
             | OBJECT [ObjAttr] ['(' Qualident ')'] [IMPLEMENTS Qualident] {DeclSeq} Body]
             | PROCEDURE [ProcTypeAttr] [FormalPars].
AttrSet      = '{' ident {',' ident} '}'.
RefAttr      = AttrSet. // reference attributes, the default is PERSISTENT
ArrTypeAttr  = AttrSet. // array type attributes
RecTypeAttr  = AttrSet. // record type attributes
PtrTypeAttr  = AttrSet. // pointer type attributes
ObjAttr      = AttrSet. // object type attributes
ProcTypeAttr = AttrSet. // DELEGATE is a predefined attribute
FieldList    = FieldDecl {';' FieldList }.
FieldDecl    = [IdentList ':' [RefAttr] Type].
```

Figure 1: Attributes for type and reference semantics

1.2 Reference Attributes

A reference of a type is semantically classified by the annotated reference attributes (*RefAttr*), with the following predefined attribute identifiers:

PERSISTENT	The reference is called *persistent*.
TRANSIENT	The reference is called *transient*.
WEAK	The reference is called *weak*.

As per default, a reference is *persistent* if none of these attributes is annotated with the element-type definition. Rules governing reference attributes are listed below:

(1) At most, one of these attribute identifiers can be used per *RefAttr*. A reference must be either persistent, transient or weak.

(2) *RefAttr* can only be annotated for a type that basically represents a reference type, i.e. object-, pointer-, procedure- or delegate-type. Otherwise the value type would be associated with reference attributes.

In Figure 2, the use of reference attributes is outlined. A new reference type must be declared without reference attribute yet (*A, B*) and the reference attribute can only be annotated where the type is used for an element (*v, w, B*). If no reference attribute is specified on the element-site, the default reference semantics is persistent (*x, C*).

```
TYPE
    A = OBJECT … END;
    B = POINTER TO ARRAY 8 OF {TRANSIENT} A;
    C = ARRAY 4 OF A;

VAR
    v: {TRANSIENT} A;  (* {TRANSIENT} OBJECT .. END *)
    w: {PERSISTENT} A;  (* {PERSISTENT} OBJECT .. END *)
    x: A;  (* {PERSISTENT} OBJECT .. END *)
    y: {WEAK} B;
       (* {WEAK} POINTER TO ARRAY 8 OF {TRANSIENT} OBJECT .. END *)
    z: C;  (* ARRAY 4 OF {PERSISTENT} OBJECT .. END *)
```

Figure 2: Use of reference attributes

1.3 Assignment Compatibility of Reference Types

The reference semantics are part of the type of variable, field, array-element, or the referenced element of an explicit pointer. It may be either declared explicitly using reference attributes or implicitly to be persistent, by omitting the reference attribute. In the following, we use the term variable as representative for all these element kinds. Each type declaration forms a user distinct type. Variables of a reference type can be assigned to variables of the same type or of a reference type, that differs only in the reference semantics for the first indirection. In case of delegates, the first indirection is the associated object instance of the delegate. Delegates are only assignment-compatible if the signature and the reference attributes of all parameters match exactly. Figure 3 gives examples of valid and incompatible variable assignments.

```
TYPE
A = OBJECT .. END;
VAR
    v: {PERSISTENT} A;
    w: {TRANSIENT} A;
    x:  {TRANSIENT} POINTER TO {PERSISTENT} A;
    y:  {TRANSIENT} POINTER TO {WEAK} A;
    z: {PERSISTENT} POINTER TO {PERSISTENT} A;
v = w; w = v; (* both ok *)
x = y; y = x; (* both invalid because of the second indirection! *)
x = z; z = x; (* both ok *)
```

Figure 3: Examples of valid and incompatible variable assignments

The syntax of formal parameters of procedure declaration is given in Figure 4. Return types have no reference attributes and have always transient semantics. The types of arguments in a procedure call, must match exactly the type of the corresponding formal parameter, if it is a VAR-parameter (call-by-reference). For parameters with call-by-value semantics having a reference type, the type of the argument and the parameter must be equal or differ only in the reference semantics for the first indirection. The rules for type compatibility concerning parameter passing are illustrated in Figure 5.

```
ProcDecl      = PROCEDURE ProcHead ';' {DeclSeq} Body ident.
ProcHead      = [ProcDeclAttr] ['*' | '&'] IdentDef [FormalPars].
ProcDeclAttr  = AttrSet.  // no predefined attributes
FormalPars    = '(' [FPSection {';' FPSection}] ')' [':' Qualident].
FPSection     = [VAR] ident {',' ident} ':' [RefAttr] Type.
```

Figure 4: Syntax of procedure declaration

```
PROCEDURE Foo(a: POINTER TO A; VAR b: POINTER TO A);
(* a and b are implicit persistent pointers *)

VAR
x: {TRANSIENT} POINTER TO A;
y: {PERSISTENT} POINTER TO A;

Foo(x, y);  (* ok *)
Foo(y, y);  (* ok *)
Foo(x, x);  (* invalid because of the second argument *)
Foo(y, x);  (* invalid because of the second argument *)
```

Figure 5: Type compatibility for procedure calls

1.4 Method Overriding

Whenever methods are overridden in sub-types, the reference attributes of all reference-typed parameters must be identical.

1.5 Lifetime of Objects

The reason for the classification of references into *persistent*, *transient* and *weak* is to implicitly denote the lifetime of objects (and other pointer-typed data), by the *principle of strongest reachability*: The reference chain to an object from a root giving the longest lifetime, defines the object's lifetime. The following definitions specify the lifetime of objects in the presence of transactions.

Definition 1 (Object, Root Element)
An *object* is a dynamically created instance of a pointer-type, containing all the value-typed data accessible from the instance without dereferencing an indirection. A *root element* is a module, a stack or a register.

Definition 2 (Current State, Latest Irreversible State)
The *current state* of an object or root element is all data values contained in it, inclusive pointer values, at a given point in time. The *latest irreversible state* of an object or root element at time *t* is the latest state, which has been irreversibly committed by a transaction[1], before or at time *t*. The latest irreversible state for objects, which have just been created in a running transaction, is not defined.

Definition 3 (Persistent Objects and Modules)
A module is always persistent. An object is persistent at time *t* if and only if, it is reachable by a chain of persistent references from a module, by only considering the latest irreversible states of objects and root elements at time *t*.

Definition 4 (Transient Objects)
A root element, except a module, is always transient. An object is transient at time *t*, if and only if:

(1) the object is not persistent at time *t* and

(2) is reachable over a chain of non-weak references from a root element or an object having a running activity at time *t*, by considering the current states at time *t* and also the latest irreversible states at time *t*, together.

Definition 5 (Garbage Objects)
An object, which is neither persistent nor transient at time *t*, is garbage at time *t*.

An important requirement is that persistent references cannot point to transient roots. This is guaranteed, since references to the stack are impossible in Oberon.

The previous definitions are illustrated in Figure 6 with an example of exactly one running transaction and the absence of activities contained in objects. Persistent objects and modules are only determined using the latest irreversible states. Hence, objects are transient, if they are not persistent and are reachable in the union of the latest irreversible state set with the current state set. Note, that objects, which exist in the set of current states but have no latest irreversible state, have just been created by a running transaction.

1.6 Weak References

A weak reference does not force its referenced object to stay alive during program execution. Weak references are reset to *NIL* as soon as the referenced object is deleted as garbage. Hence, weak references can be reset at any point in time during execution. Furthermore, weak references of persistent objects or modules are set to *NIL* when the program or system is restarted.

[1] In the presence of nested transactions, only the top-level transactions perform an irreversible commit.

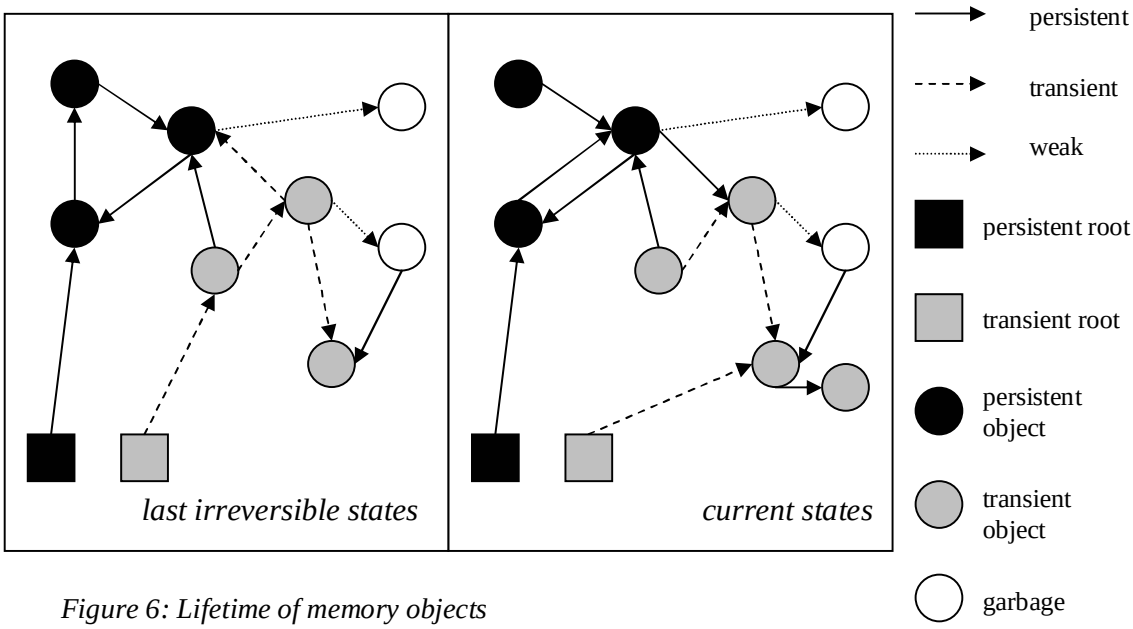

Figure 6: Lifetime of memory objects

1.7 Persistence of Modules

Modules are persistent and form the implicit roots for persistence. The latest irreversibly committed state of a module, including all reachable value-typed data, persistent references and persistent delegates, is permanent. This state will still be present when a module is reloaded after it has been unloaded or after a system restart or system crash. The statement block of a module will only be executed when a module is loaded for the first time and is not reincarnated from the persistent object system.

Since each module has only one instance per system, problems of root synchronizations arising in other systems, where multiple instances of a program are possible, do not occur.

1.8 Persistent Objects

Persistent objects survive program or system restarts and system crashes. The latest irreversibly committed state of a persistent object and module is available at the next program reincarnation, except the transient or weak references contained in the object or module. Transient or weak references are namely reset to *NIL* at program reincarnation.

1.9 Transient Objects

Transient objects or transient root elements, stay alive during program execution and are discarded on program termination or system crash. Transient objects correspond to the instances with reference type in non-persistent programming languages.

1.10 Garbage Objects

Garbage objects can be deleted by the system at any point in time.

1.11 Lifetime of Activities

The Active Oberon programming language features objects that can have an active statement block. In fact, for each instance of an object having an active statement block, a thread is launched, which executes the active statement block. During the execution of an activity, the containing object cannot become garbage. As each persistent object can have an active statement block, it must be defined whether the corresponding thread is also persistent. Each thread has an associated stack that potentially holds transient references to objects. If a thread and its stack would be made persistent in our model, reincarnating a persistent thread on system restart makes no sense, since transient references on the stack would not be valid anymore and, hence, reset to *NIL*. Thus, the threads of a persistent object are defined to be not persistent. Instead, it is the user's decision whether the thread of a persistent object is manually relaunched at program reincarnation.

1.12 Lifetime of Delegates

Persistent delegates in persistent objects survive system restarts and failures and can still be called during future program reincarnations. Transient delegates are reset to *NIL* on system restart. Weak delegates may be reset to *NIL* at any point in time, when the associated invocation target is disposed by the runtime system.

1.13 Initializers

The semantics of object initializers remains unchanged. The initializer of an object is invoked after an object has been created and before the object can be used. The initializer of a persistent object is not executed when the object is reincarnated.

1.14 Transactions

The concept of transactions is integrated into Persistent Oberon. Each modification that affects persistence is executed within a transaction, either as an explicit ACID-transaction or an implicit mini-transaction. Furthermore, the support of nested transactions is described.

1.15 Syntax of Statement Blocks

The EBNF-syntax for attributing statement blocks, shown in Figure 7, corresponds mainly to the original Active Oberon syntax. Furthermore, a new statement *ABORT* is provided to permit the abort and rollback of a running transaction. *ABORT* is a new reserved keyword.

```
StatBlock       = BEGIN [StatBlockAttr] [StatSeq] END.
StatBlockAttr   = AttrSet.
AttrSet         = '{' ident {',' ident} '}'.
Statement       = [Designator ':= Expr
                | Designator ['(' ExprList ')']
                | IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq} [ELSE StatSeq] END
                | CASE Expr THEN Case {'|' Case} [ELSE StatSeq] END
                | WHILE Expr DO StatSeq END
                | REPEAT StatSeq UNTIL Expr
                | FOR ident ':=' Expr TO Expr [BY ConstExpr] DO StatSeq END
                | LOOP StatSeq END
                | WITH Qualident ':' Qualident DO StatSeq END
                | EXIT
                | RETURN [Expr]
                | AWAIT '(' Expr ')'
                | ABORT
                | StatBlock
                ].
```

Figure 7: Attributes for statement blocks, ABORT statement

Additionally to *ACTIVE* and *EXCLUSIVE*, there exists a predefined attribute identifier *TRANSACTION*, for statement block attributes (*StatBlockAttr*) to declare an explicit ACID-transaction. The *TRANSACTION*-attribute cannot be specified together with *EXCLUSIVE* because a transaction is the more general concept than an exclusive lock. The use and semantics of the statement block modifiers, *ACTIVE* and *EXCLUSIVE*, remain unchanged.

The statement *ABORT* can only be used within a statement block with the attribute *TRANSACTION*.

1.16 Exit of a Transaction Statement Block

A statement block with the attribute *TRANSACTION* is successfully committed, when the statement block is exited as follows:

(1) The end of the *TRANSACTION*-statement block is reached during program execution.

(2) The RETURN-statement is called within the statement block.

(3) The EXIT-statement leaves a loop that encloses the *TRANSACTION*-statement block.

The *TRANSACTION*-statement block can also be prematurely exited and aborted by the ABORT-statement or when an unhandled exception occurs during the execution of the statement block.

1.17 Explicit Transactions

A statement block with the attribute *TRANSACTION*, declares all the code within this statement block to be executed as an transaction, including directly or indirectly called methods or procedures. The concept of nested transactions [Moss81] allows it to execute transactions within other enclosing transactions. Hence, we must distinguish between explicit top-level transactions and nested transactions, also called subtransactions.

1.18 Explicit Top-Level Transactions

An explicit top-level transaction is an explicit ACID-transaction, which is not executed within another transaction and offers the following features:

(1) *Isolation*: All data that is accessed during the execution of an explicit transaction statement block, including directly or indirectly called methods or procedures, are isolated from other processes, until the transaction is terminated. Transactions are only defined to be isolated on the level of read- and write-operations on data contained in objects or modules but not for semantic higher operations. We define transactions to be executed isolated, if the effects of read- and write-operations are the same as it would be in a serial execution order of the transactions, known as serializability of transactions.

(2) *Atomic Commit*: When an explicit transaction statement block is successfully exited, an atomic commit is performed. First of all, all objects that would be persistent after the commit are determined. Then, the states of all these objects, which have been modified or created by the committing transaction, are atomically made permanent.

(3) *Atomic Rollback*: The statement *ABORT* or an unhandled exception within an explicit transaction block, initiates the atomic abort of the running transaction and the exit from the enclosing transaction block. On transaction abort, the backup-states of all objects and modules modified by the running transaction are reconstructed. The objects, created during the aborted transaction, are discarded.

(4) *No External Abort*: Apart from system traps or program terminations, a transaction can only be aborted, when the *ABORT*-statement is executed. Each transaction, whose execution end is reached, must be able to commit. This implies that optimistic concurrency models and strict two phase locking cannot be used [BHG87]; because they would lead to deadlocks, if that would not be resolved using unexpected external transaction rollbacks, initiated by the transaction scheduler.

1.19 Nested Transactions

Nested transactions [Moss81], or subtransactions, are explicit ACID-transactions that are executed within other (sub)transactions. This can happen, if nested statement blocks with *TRANSACTION*-attributes are used or if a procedure or method with a transaction statement block is called from a running transaction. An abort of a subtransaction does not imply the containing transaction to be aborted. The updates of subtransactions only become permanent, if the enclosing top-level transaction commits. The beginning of a nested transaction defines a kind of fix-point of the state in a transaction, which can be restored by an abort. The semantics of isolation (1) and the exclusion of external abort (4), remain the same as for explicit top-level transactions.

(2) *Atomic Commit*: At the successful exit of the transaction statement block, the modifications are taken over for the enclosing transaction and the information for a potential rollback of the current transaction is discarded. The states of the modified objects and modules are only stored irreversibly, when the enclosing top-level transaction is committed.

(3) *Atomic Rollback*: The statement *ABORT* restores atomically the states of all objects modified by the aborted transaction as they were at the beginning of the transaction. Additionally, the transaction statement block is exited. The same is true for an unhandled exception within a subtransaction block.

It is not possible to concurrently execute multiple subtransactions within the same containing transaction because nested active statement-blocks are not supported.

1.20 Implicit Mini-Transaction

A write-operation on data of a persistent object or module, which is not performed during a running explicit transaction, is executed as an implicit mini-transaction. The write-operation of a mini-transaction is atomically committed with the same semantics like an explicit top-level transaction. This implies that other objects may become atomically persistent by such a write-operation.

A Persistent Oberon Syntax

The original syntax of the Active Oberon is described in [Reali02a].

```
Module        = MODULE ident ';' [ImportList] {Definition} {DedclSeq} Body ident '.' .
ImportList    = IMPORT ident [':=' ident] {',' ident [':=' ident]} ';'.
Definition    = DEFINITION ident [REFINES Qualident] {PROCEDURE ident [FormalPars] ';'} END ident.
DeclSeq       = CONST {ConstDecl ';'} | TYPE {TypeDecl ';'} | VAR {VarDecl ';'} | {ProcDecl ';'}.
ConstDecl     = IdentDef '=' ConstExpr.
TypeDecl      = IdentDef '=' Type.
VarDecl       = IdentList ':' [RefAttr] Type.
ProcDecl      = PROCEDURE ProcHead ';' {DeclSeq} Body ident.
ProcHead      = [ProcDeclAttr] ['*' | '&'] IdentDef [FormalPars].
ProcDeclAttr  = AttrSet.  // no predefined attributes
```

```
FormalPars      = ‘(‘ [FPSection {‘;’ FPSection}] ‘)’ [‘:’ Qualident].
FPSection       = [VAR] ident {‘,’ ident} ‘:’ [RefAttr] Type.
Type            = Qualident
                | ARRAY [ArrTypeAttr] ConstExpr {‘,’ ConstExpr} OF [RefAttr] Type
                | RECORD [RecTypeAttr] [‘(‘ Qualident ‘)’] [FieldList] END
                | POINTER [PtrTypeAttr] TO [RefAttr] Type
                | OBJECT [ObjAttr] [‘(‘ Qualident ‘)’] [IMPLEMENTS Qualident] {DeclSeq} Body]
                | PROCEDURE [ProcTypeAttr] [FormalPars].
RefAttr         = AttrSet. // PERSISTENT, TRANSIENT, WEAK are predefined attributes,
                           the default is PERSISTENT
ArrTypeAttr     = AttrSet. // no predefined attributes
RecTypeAttr     = AttrSet. // no predefined attributes
PtrTypeAttr     = AttrSet. // no predefined attributes
ObjAttr         = AttrSet. // no predefined attributes
ProcTypeAttr    = AttrSet. // DELEGATE is a predefined attribute
AttrSet         = ‘{‘ ident {‘,’ ident} ‘}’.
FieldList       = FieldDecl {‘;’ FieldList }.
FieldDecl       = [IdentList ‘:’ [RefAttr] Type].
Body            = StatBlock | END.
StatBlock       = BEGIN [StatBlockAttr] [StatSeq] END.
StatBlockAttr   = AttrSet. // TRANSACTION, ACTIVE, EXCLUSIVE are predefined attributes
StatSeq         = Statement {‘;’ Statement}.
Statement       = [Designator ‘:= Expr
                | Designator [‘(‘ ExprList ‘)’]
                | IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq} [ELSE StatSeq] END
                | CASE Expr THEN Case {‘|’ Case} [ELSE StatSeq] END
                | WHILE Expr DO StatSeq END
                | REPEAT StatSeq UNTIL Expr
                | FOR ident ‘:=’ Expr TO Expr [BY ConstExpr] DO StatSeq END
                | LOOP StatSeq END
                | WITH Qualident ‘:’ Qualident DO StatSeq END
                | EXIT
                | RETURN [Expr]
                | AWAIT ‘(‘ Expr ‘)’
                | ABORT
                | StatBlock
                ].
Case            = [CaseLabels { ‘,’ CaseLabels } ‘:’ StatSeq]
CaseLabels      = ConstExpr [‘..’ ConstExpr].
ConstExpr       = Expr.
Expr            = SimpleExpr [Relation SimpleExpr].
SimpleExpr      = Term {MulOp Term}
Term            = [‘+’ | ‘-‘] Factor {AddOp Factor}.
Factor          = Designator[‘(‘ExprList’)’] | number | character | string | NIL | Set | ‘(‘Expr’)’ | ‘ ‘Factor.
Set             = ‘{‘ [Element {‘,’ Element}] ‘}’.
Element         = Expr [‘..’ Expr].
Relation        = ‘=’ | ‘#’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | IN | IS.
MulOp           = ‘*’ | DIV | MOD | ‘/’ | ‘&’.
AddOp           = ‘+’ | ‘-‘ | OR.
Designator      = Qualident { ‘.’ ident | ‘[‘ExprList‘]’ | ‘^’ | ‘(‘ Qualident ‘)’ }.
ExprList        = Expr {‘,’ Expr}.
IdentList       = IdentDef {‘,’ IdentDef}.
Qualident       = [ident ‘.’] ident.
IdentDef        = ident [‘*’ | ‘-‘].
```

ABORT is a new reserved keyword.

References

[Gut97] J. Gutknecht. Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon. In Proc. of the Joint Modular Languages Conference JMLC'97, Hagenberg, March 1997.

[Moss85] J. E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, Cambridge Mass, 1985.

[Reali02] P. Reali. Active Oberon Language Report. Institute of Computer Systems, ETH Zurich, March 2002. At http://www.bluebottle.ethz.ch/languagereport/ActiveReport.pdf