link linear list of formal parameters, which is organized through `Object.link`. The first parameter (i.e. the object this field points to) denotes the receiver.

- form=strArray

  len number of elements in this dimension

  base the struct of an element in this dimension

  link `NIL`

- form=strDynArray

  base the struct of an element in this dimension

  link `NIL`

- form=strRecord

  len extension level, i.e. number of extensions, which lead to this record. 0 means that there is no base record available.

  base the struct of the record which the current one is derived from. If `len=0` holds true, this field is `NIL`. If len>0, then this field points to a record structure, which could hold fields and type bound procedures, which are also visible within the current record structure. These fields are not replicated, though.

  link a binary search tree of objects, in which all fields and type bound procedures belonging to this record are kept.
  If *link.mode=objField* then follow the object's `next` pointers to get hold of all of this record's fields in the order of declaration. This linear list is terminated with `NIL`.
  If *link.mode=objTBProc* then no record fields are declared at this extension level. Typebound procdures are only kept in the binary tree, not in the list of declaration order.

Appendix B.3 Constants

```
TYPE
   Const* = POINTER TO ConstDesc;
   ConstDesc* = RECORD
         string*  :  POINTER TO ARRAY OM.maxSizeString OF CHAR;
         intval*  :  LONGINT;     (* an integer constant *)
         intval2* :  LONGINT;
         set*     :  SET;         (* a set constant *)
         real*    :  LONGREAL;    (* a real constant *)
      END; (* RECORD *)
```

string pointer to a 0X terminated string

intval,intval2 integer constants

set set constant

real (long-)real constant

flags flags set by the front end
`flagExport` is set, if the struct was been written to the symbol file
`flagExternal` is set, if the struct belongs to an external definition
`flagUnion` is set, if the struct (record) denotes a C-union

- form=strUndef
  "undefined" struct which is used for forward declarations as long as the type is not resolved, and to mark types that couldn't be determined due to compilation errors.
  base the struct itself for error recovery reasons

- form={strBool, strChar, strShortInt, strInteger, strLongInt, strReal, strLongReal, strSet}
  predefined types, structure of a predefined type

- form=strNone
  result type of a proper procedure, denoting the absence of a function result

- form=strString
  type used for string constants
  len length of the string (excl. 0X)
  base `NIL`
  link `NIL`

- form=strNil
  the type of identifier "`NIL`"

- form={strSysByte, strSysPtr}
  types from module `SYSTEM`

- form=strPointer
  base type structure of the pointer's base type
  link `NIL`

- form=strProc
  len number of formal parameters
  base the procedure's result type.
  link List of formal parameters, which are hold in an via `Object.link` linked linear list. These formal parameters are put in the procedure's scope when calling `InsertParams` in the front end.

- form=strTBProc
  obj `NIL`
  len number of formal parameters incl. receiver

next `NIL`

type the type bound procedure's struct, which holds the result type and the formal parameter list. (*type.form = strTBProc*).

- mode=objType

  link `NIL`

  type the structure for this type.

- mode=objForwardType

  link `NIL`

  type a dummy struct, which is of `form=strUndef`.

- mode=objForwardProc

  link `NIL`

  type the struct of this procedure.

- mode=objForwardTBProc

  left, right more objects in the current record

  link `NIL`

  next `NIL`

  type the struct of this type bound procedure.

Appendix B.2 Structs

Structs are used for the internal representation of types.

```
TYPE
   Struct* = POINTER TO StructDesc;
   StructDesc* = RECORD
      form* :SHORTINT;
      base* : Struct;   (* base type or function result *)
      obj*  : Object;   (* object, when named type *)
      link* : Object;   (* the parameter list *)
      flags*: SET;
      len*  : LONGINT;
      size* : LONGINT;
   END;  (* RECORD *)
```

As before, we will firstly enlist all fields with their meaning. Exceptions are explicitly mentioned afterwards.

form Type of the struct. This determines if it's a predefined type (e.g. `strInteger`) or a structured type (e.g. `strArray`).

obj an object, if the type is defined as part of a type declaration, NIL otherwise

be called).

sort a linear list of objects, which is linked using the `Object.sort` pointer. This list contains all objects of the scope (excluding imported modules or formal parameters) in a sorted way. The objects are sorted by their names using the Oberon-2 string comparison functions "<" and ">". The list starts with the "smallest" identifier.

- mode=objConst

  link `NIL`

  type the structure of a predefined type. This is the type associated with this constant.

  mark export mark, only `exportNot` or `exportWrite` allowed.

  const a constant that holds the data.

- mode=objVar

  link the next formal parameter, if (`flagParam IN Object.flag`) holds true. `NIL`, if no more formal parameters follow or `flagParam` is not set

  flags `flagParam` is set, if part of a formal parameter list

- mode=objVarPar

  link the next formal parameter, `NIL` if no more formal parameters follow

  flags `flagParam` is always set

- mode=objField

  scope `NIL`

  left,
  right other fields and type bound procedures belonging to the record, organised as a binary tree.

  link `NIL`

  next the next field object declared in the corresponding record. `NIL`, if no field follows

- mode={objExtProc,objIntProc,objLocalProc}

  link the anchor of the procedure's scope, in which all local objects are stored. (*link.mode=objScope*).

  type the struct of this procedure, which holds the result type and the formal parameter list. (*type.form = strProc).*

  mark export mark, `exportNot` or `exportWrite`

- mode=objTBProc

  left,
  right more objects in the record this type bound procedure belongs to. *(left|right).mode IN {objTBProc, objField}*

  link the scope anchor for this type bound procedure. (*link.mode = objScope*).

const `NIL`, except for module and constant objects

name identifier of the object

*extName*external name of the object, if not `NIL`

next the next object in the order of declaration

pos position of the object in the source code

mark export mark. Possible values are `exportNot`, `exportRead` or `exportWrite`.

type a *struct* which describles the type structure of the object

flags various flags set by the front end to give additional information to the backend
`flagExport` is set, if the corresponding object is written into the symbol file
`flagParam` is set, if the corresponding object is a formal parameter
`flagReceiver` is set, if the corresponding formal parameter is a receiver
`flagForward` is set, if the corresponding object was declared forward (type or proc)
`flagHasLocalProc` is set, if local procedures are declared in the scope
`flagEmptyBody` is set, if the procedure's body is empty
`flagExternal` is set, if the object belongs to an external definition (s.a. extName)
`flagAddressed` is set for an object in Object.flags, if it is passed as a variable parameter to a normal (ie, not predefined) procedure or is used as argument of `SYSTEM.ADR`

- mode=objUndef

  undefined object, signals error conditions

- mode=objModule

  link the scope anchor of the module (*link.mode=objScope*). All globally declared objects are inserted into this scope.

  const constant that holds the symbol file key (checksum). This key is only valid, if the module was imported (`mnolev <= systemMnolev`), otherwise it will be calculated when the symbol file is written.

- mode=objScope

  scope the enclosing scope. This is the way to get from an inner scope to an enclosing one. The scopes are layered through this mechanism. *(scope.mode=objScope*).

  left pointer to the scope's object. (*left.mode IN {obj*Proc, obj*TBProc, objModule}*)

  link pointer to the root of the scope's binary search tree.

  right pointer to the scope's first declared object. To walk through the objects in the order of declaration, use this pointer and follow the object's `next` pointers. Formal parameters are not part of this list. They are only accessible via the procedure's *struct* or can be found in the binary search tree.

  next After a call of `CloseScope` at `mnolev=compileMnolev` a list (organised via `Object.next`) of all imported modules is linked to this pointer. This holds always true for the symbol tree after the symbol file is written (i.e. when the back end should

Appendix B Description of the Symbol Table

The symbol table is build out of three major elements: `objects` for named declarations, `structs` as an abstract type description and `consts` for holding constant values. The management of the symbol table is implemented in module `OTable`.

Appendix B.1 Objects

```
TYPE
   Object = POINTER TO ObjectDesc;
   ObjectDesc* = RECORD
      mode*    : SHORTINT; (* kind of object *)
      scope-   : Object; (* the object's scope *)
      left*,right*: Object;
      link*    : Object; (* internal pointer for objects *)
      next*    : Object; (* the next object in declaration sequence *)
      sort-    : Object; (* the next object in alphabetic order *)
      name*    : ARRAY OMachine.maxSizeIdent OF CHAR;
      extName* : POINTER TO ARRAY OMachine.maxSizeString OF CHAR;
      mark*    : SHORTINT; (* state of export *)
      type*    : Struct;
      const*   : Const;
      flags*   : SET;
      mnolev*  : LONGINT;
      pos      : LONGINT; (* source position of the object *)
   END; (* RECORD *)
```

Objects are organized in a lexicographic sorted binary tree within their scope. They are linked via the fields `Object.left` and `Object.right`.

Here a short description of the fields. Differences from these are enlisted explicitly later.

mode This is the field which identifies the object, i.e. if it's a procedure, a module or a variable declaration.

left an object with a "smaller" identifier

right an object with a "greater" identifier

scope the scope anchor of the scope in which this object was declared *scope.mode=objScope*

sort the next object in the sort order

mnolev definitionlevel of the object
`predeclMnolev`: predefined identifier
`systemMnolev`: identifier of internal module `SYSTEM`
<=`systemMnolev`: identifier from imported module
`compileMnolev`: declared on global module level (i.e. 0) of currently compiled module
>`compileMnolev`: nesting depth of the procedure holding this object. For every procedure this value is increased by1

Appendix A.7 Simplifications of the Syntax Tree

During compilation the parser executes the following simplifications:

- Constant Folding
  Calls to predefined functions and operators with constant operands are evaluated during compilation and replaced by a constant.
  Note: There are no checks for overflow, expressions like `MAX(LONGINT)+1` will be accepted without a warning.

- Boolean Simplifications
  The shortcut evaluation of `&` and `OR` allows to reduce some expressions if one of the operands is a constant:

Tabelle 1:

expression	will be replaced by
`FALSE OR b`	`b`
`TRUE OR b`	`TRUE`
`FALSE & b`	`FALSE`
`TRUE & b`	`b`
`b OR FALSE`	`b`
`b & TRUE`	`b`

- Folding of Conversion Nodes
  Type changes of constants are done during compilation. For all other expressions type conversion nodes are joined as far as possible.

  Example:

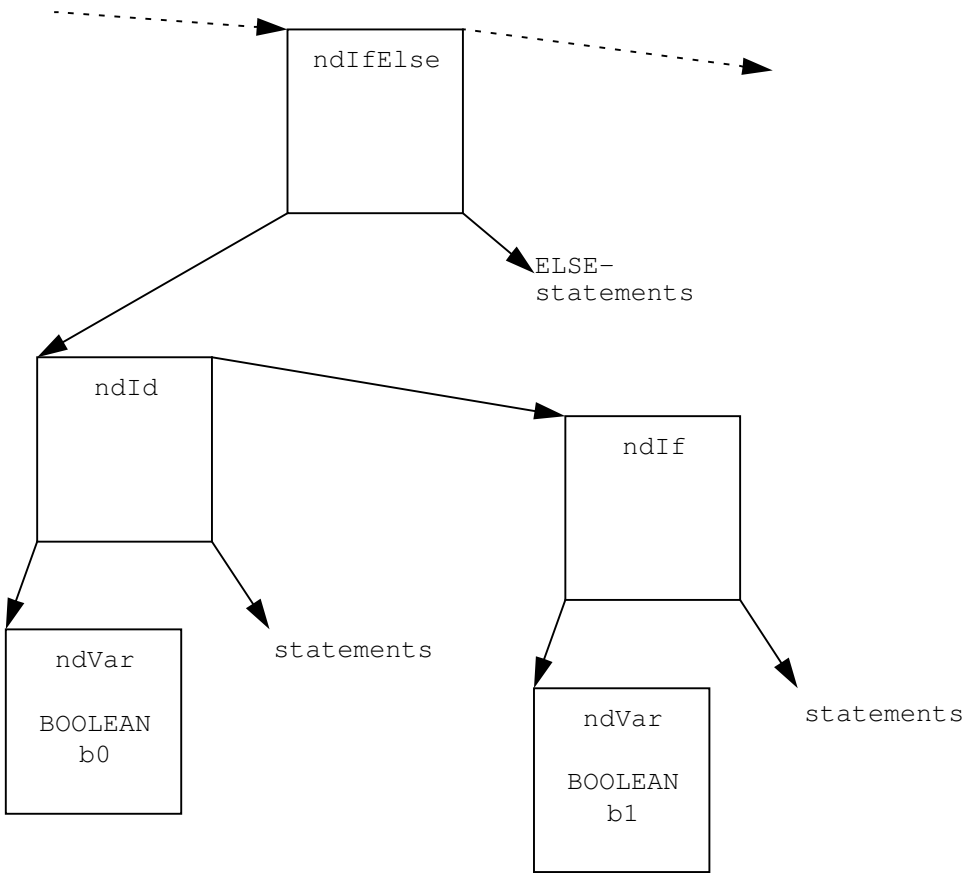
```
VAR si : SHORTINT; i : INTEGER; li : LONGINT; ch : CHAR;
i := LONG (SHORT (i));
li := LONG (ORD (ch));
li := LONG (LONG (si));
si := SHORT (SHORT (li));
li := LONG (ORD ("a"));
```

  The first expression needs two conversion node, the following three just one and the last expression will be translated into a single constant.

- Unreachable `IF` and `WITH` Branches
  Branches in `IF` statements are eliminated if they can't be reached due to constant guards. If a type test in a `WITH` statement can never be evaluated to `TRUE`, because a preceding type test tests the same variable against a base type, then the whole block will be removed. Additionally a warning is emitted.

IF-Statement

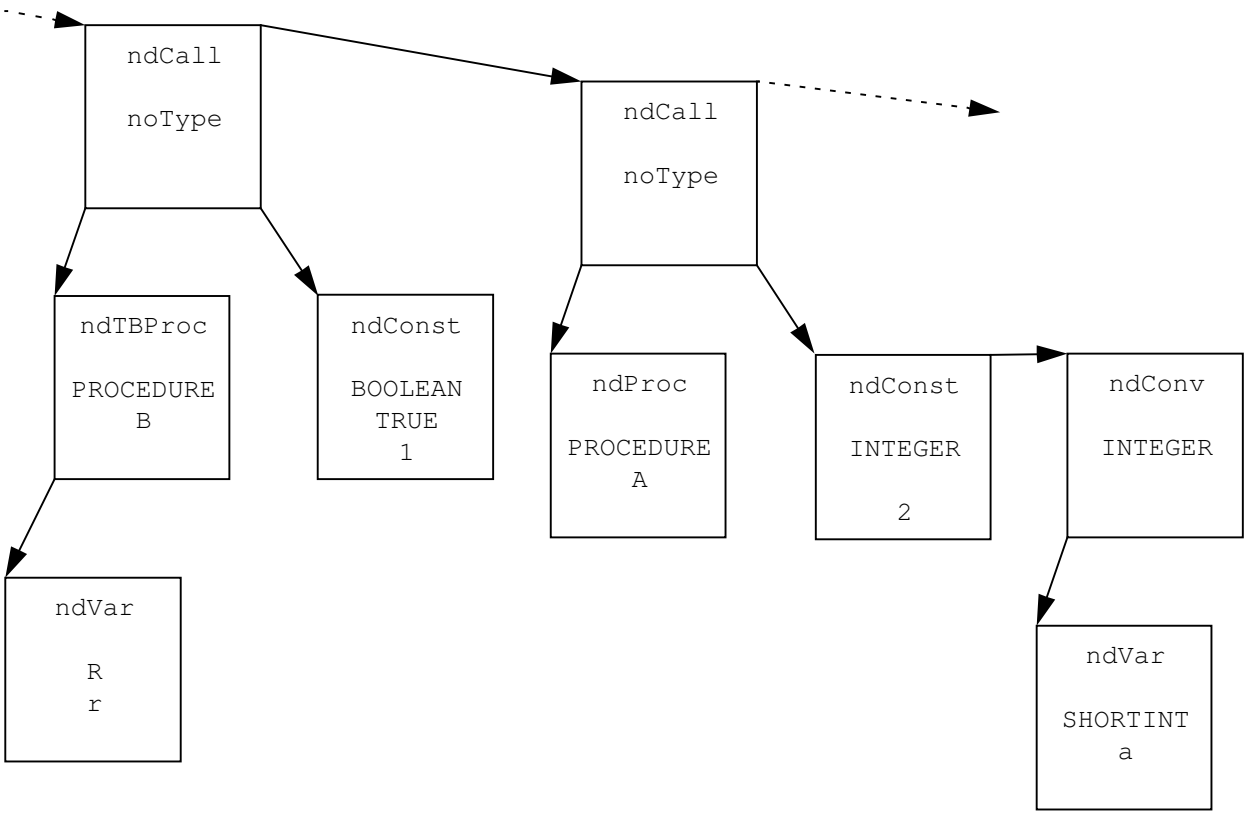
```
VAR

   b0, b1, b2: BOOLEAN;
...
IF b0 THEN
   ...
ELSIF b1 THEN
   ...
ELSIF FALSE & b2 THEN

   ...
ELSE
   ...
END
```



The parser recognizes that `FALSE & b2` can be simplified to `FALSE`. This causes the associated `IF` branch to become unreachable and the whole branch is eliminated.

Procedure Call

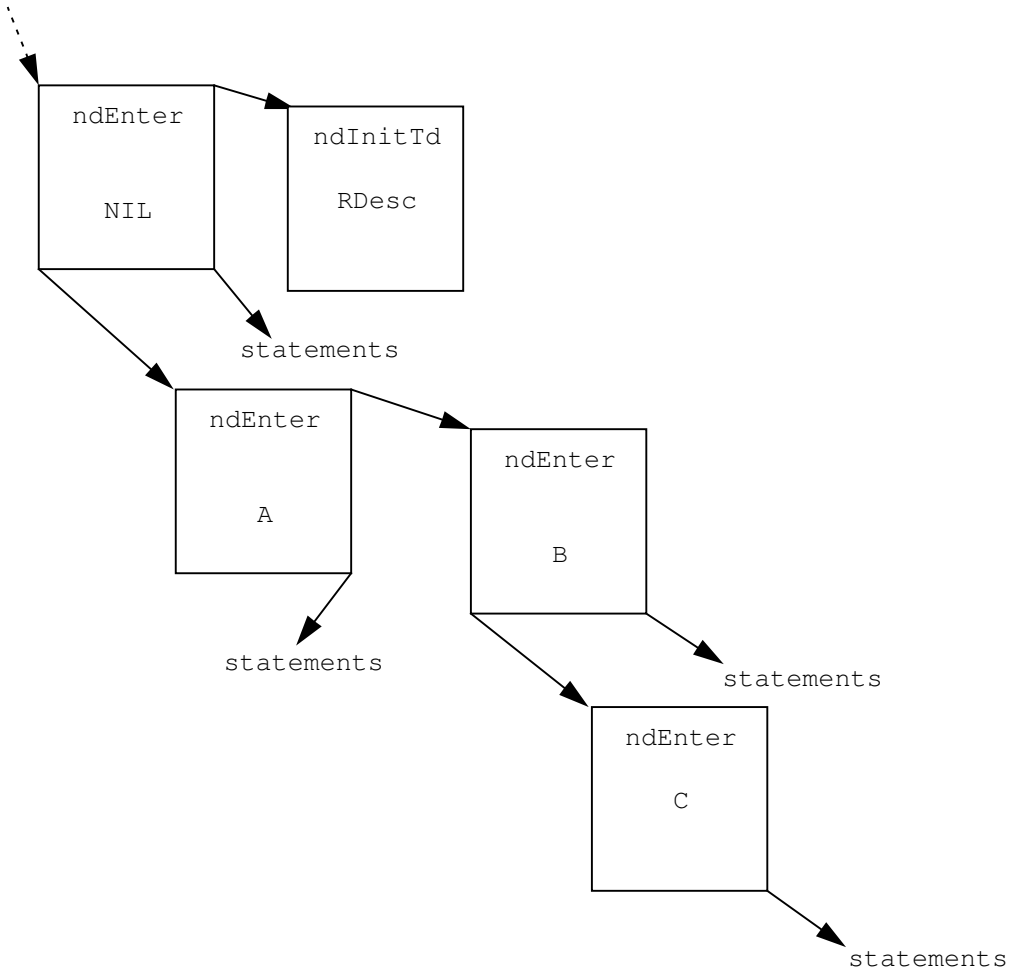
```
VAR
   a : SHORTINT;

   r : R;
   ...
r. B (TRUE);

A(2, a);
```



In the nodes `ndProc` and `ndTBProc` the field *type* (in the table above it is labeled as `PROCEDURE`) contains a link to the data type that is associated with each procedure. It describes the procedure's formal parameter list and it's result type. When calling a type bound procedure the receiver is encoded as part of the procedure designator.

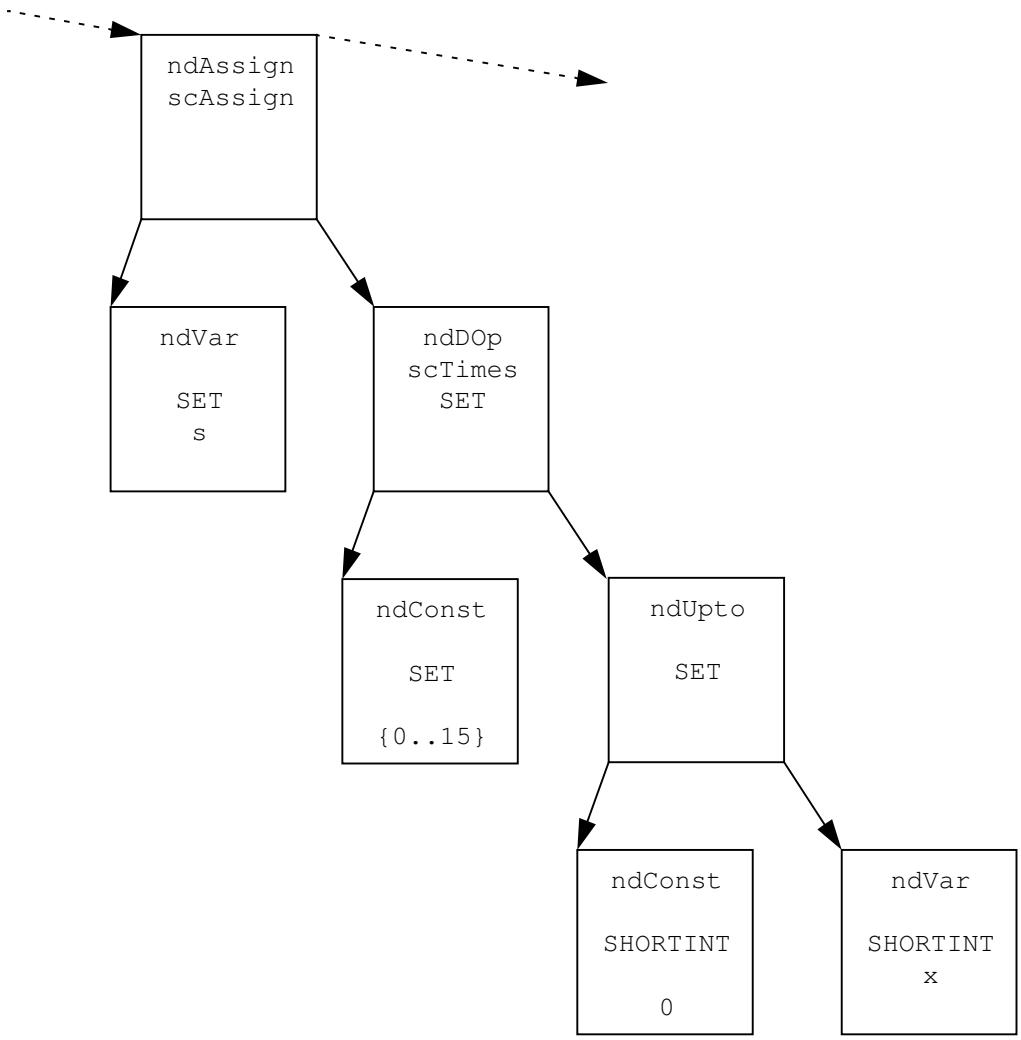
Module

```
MODULE M;

TYPE
   R = POINTER TO RDesc;
   RDesc = RECORD END;

PROCEDURE A (x, y : INTEGER);
   BEGIN ...
   END A;
PROCEDURE (r : R) B (b : BOOLEAN);
   PROCEDURE C;
      BEGIN ...
      END C;
   BEGIN ...
   END B;
BEGIN ...
END M.
```



Set Arithmetics

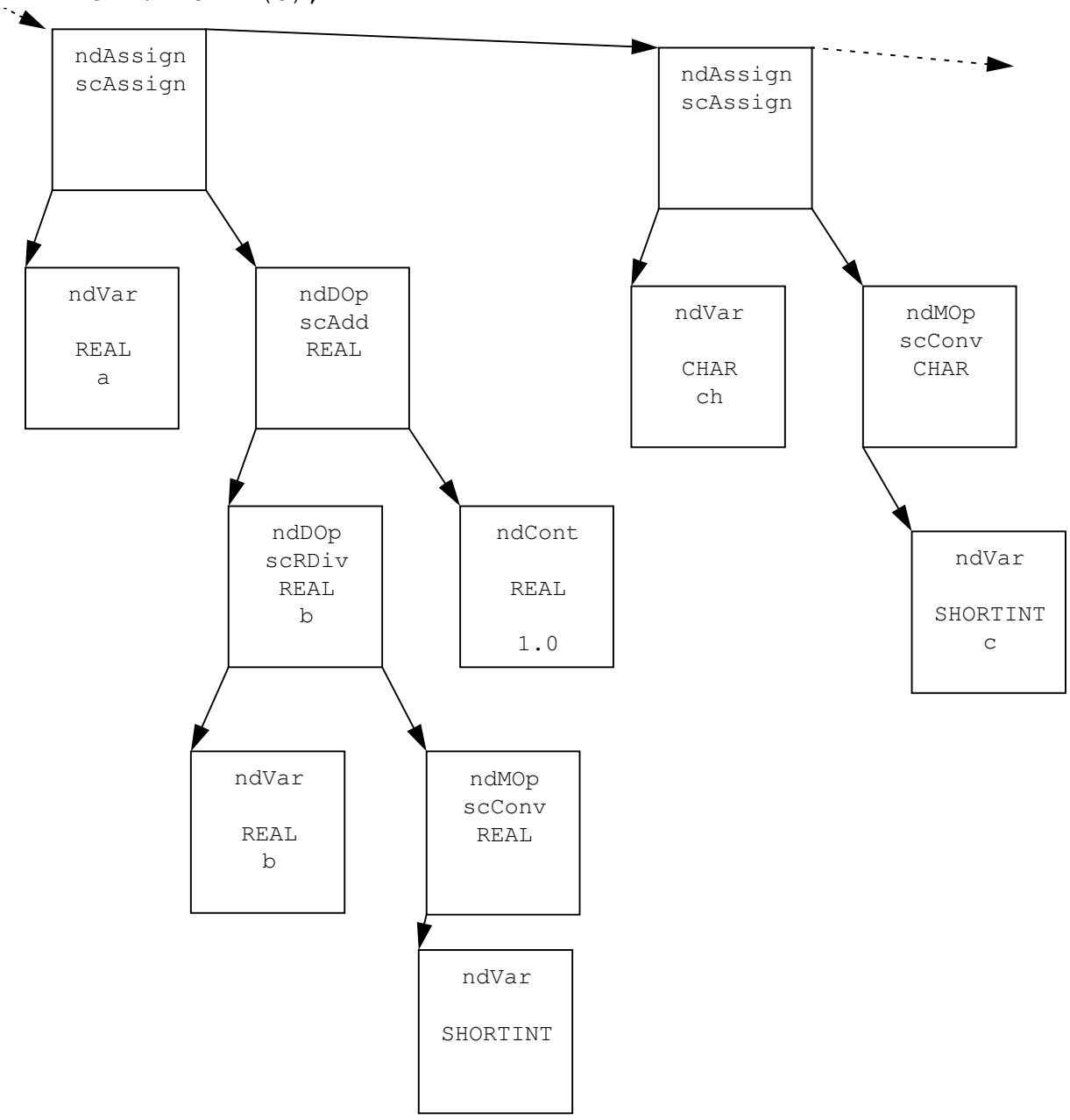
```
CONST
   c = {0..7};
VAR
   x : SHORTINT;
   s : SET;
   ...
s := (c + {8..15}) * {0..x};
```



The constant expression `c+{8..15}` will be translated into `{0..15}` during parsing.

Type Conversions

```
VAR
   a, b : REAL;
   c : SHORTINT;
   ch : CHAR;
   ...
a := b / c + 1;
ch := CHR (c);
```

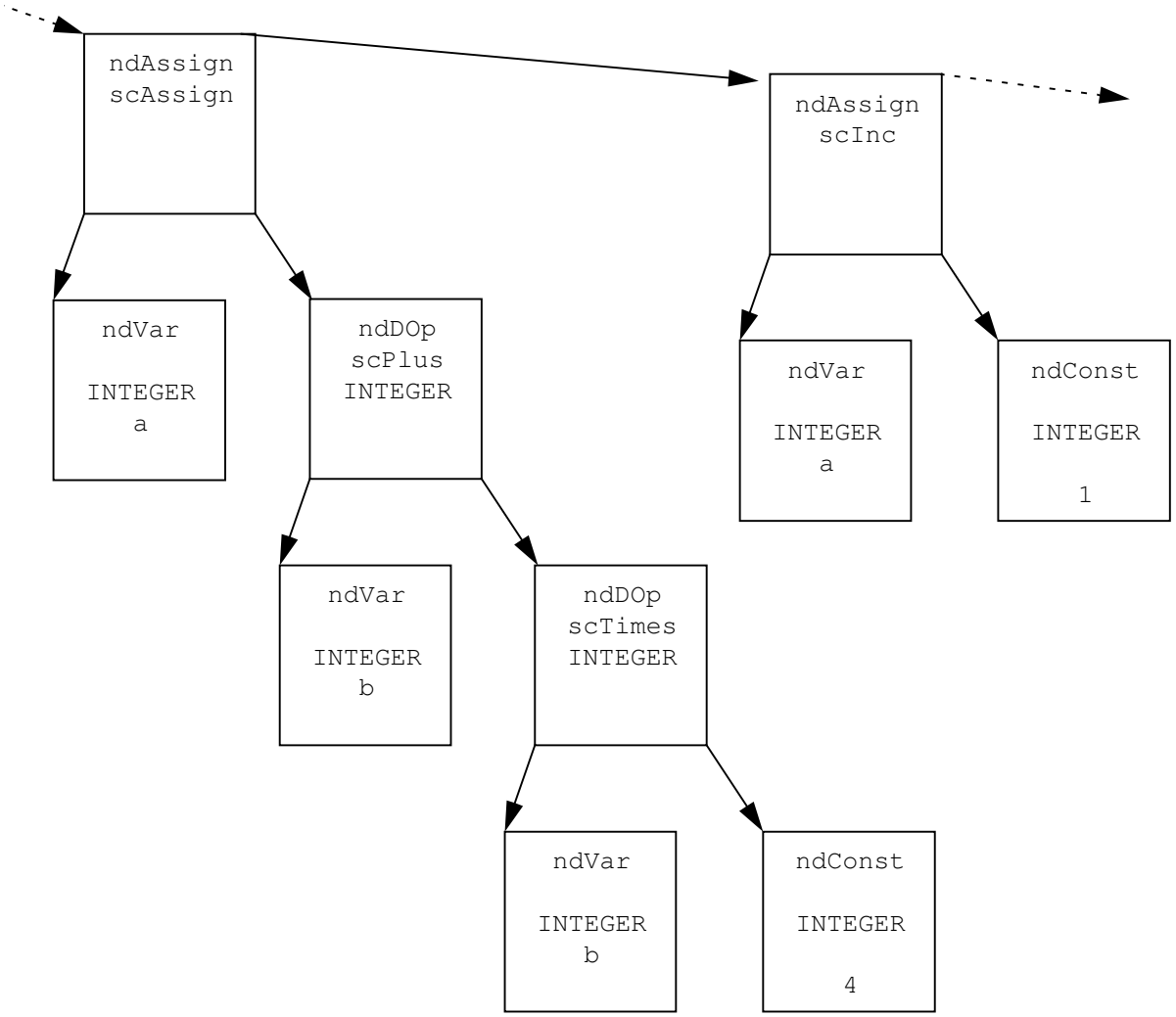


Type conversions can occur implicitly as part of expressions, or as the result of a call to a predefined function like `LONG`, `CHR`, or `ENTIER`. Type conversions of constants are evaluated by the front end and won't show up in the syntax tree.

Appendix A.6 Examples

Assignments

```
VAR
   a, b, c : INTEGER;
   ...
a := b+c*4;
INC (a);
```



Predefined functions like `INC`, `DEC`, `INCL`, `EXCL` are translated into nodes of class `ndAssign`. The exact node type is determined by the field *subcl*.

- class=ndAssert (Assertion)
  left trap number (integer constant)
  right guard (*boolean expression*)

Appendix A.4 Module Structure

The root of the syntax tree, and of each procedure, is a `ndEnter` node. `ndEnter` and `ndFoward` nodes on the same level are connected with *link*.

- class=ndEnter (module or procedure)
  obj contains the procedure declaration. *obj*=`NIL` marks the module entry.
  left list of local procedures (`ndEnter` or `ndForward`, linked with *link*)
  right procedure resp. module body (*statements*)
  link next procedure resp. list of `ndInitTd` nodes (if the node represents the module)
- class=ndForward (marker for forward declared procedures)
  obj contains the procedure declaration
- class=ndInitTd (type descriptor)
  type `RECORD` data type
  link next `ndInitTd` node

Appendix A.5 Initialisation of Type Descriptors

The parser creates a `ndInitTd` node for each record data type defined in the module or in a procedure. The list of those nodes can be found in *mod.link*, where *mod* is the root of the syntax tree. All further nodes are linked with *link* and are appended to the list in the order of definition in the source program. This ensures that `ndInitTd` nodes of base types appear before the ones for records extending them. A declaration of the kind '`TYPE RecordA = RecordB;`' does not define a new record. Therefor no new entry is created for `RecordA`.

- class=ndWhile (`WHILE` statement)
  left guard (*boolean expression*)
  right loop body (*statements*)

- class=ndRepeat (`REPEAT` statement)
  left loop body (*statements*)
  right termination condition (*boolean expression*)

- class=ndFor (`FOR` statement)
  left.link control variable (*designator*)
  left.left starting value (*expression*)
  left.right ending value (*expression*)
  left.conval.intval step (the parser sets it to 1 if no `BY` is given)
  left.type type of step constant
  right loop body (*statements*)

- class=ndLoop (`LOOP` statement)
  left loop body (*statements*)

- class=ndWithElse (`WITH` statement)
  left list of guards (`ndWithGuard` linked with *link*)
  right `ELSE` part (*statements*)
  conval.set = { } iff no `ELSE` part exists

- class=ndWithGuard (sub structure of `WITH` statement)
  left variable (`ndVar` or `ndVarPar`)
  right body of regional type guard (*statements*)
  obj data type against which *left* is tested (`obj.mode=objType`)
  link next guard resp. `NIL`

- class=ndExit (`EXIT` statement)
  left `LOOP` construct that is left (`left.mode=ndLoop`)

- class=ndReturn (`RETURN` statement)
  obj the procedure which is left with `RETURN`, or `NIL` if it is the module
  left result value (*expression*), `NIL` for proper procedures

- class=ndTrap (`HALT` statement)
  left trap number (integer constant)

- `SYSTEM.DISPOSE` on pointer variable (`subclass=scDispose`)
  left: pointer (*designator*)
- `SYSTEM.COLLECT` (`subclass=scCollect`)
  has no parameters

- class=ndCall (procedure or function call)

  left procedure designator (*designator*), for type bound procedures this also contains the actual receiver.
  Note: This can also be a procedure variable.

  type result type (`noType` for proper procedures)

  right parameter list (`NIL` when empty; otherwise list of *expressions* linked with *link*).
  Note: When calling type bound procedures the formal parameter list in *left.type* contains the formal receiver as first parameter; the actual receiver is stored as part of the procedure designator in *left.left* and not in the parameter list.

- class=ndIfElse (`IF` statement)

  right `ELSE` part (*statements*)

  left list of guarded commands (`ndIf`) linked with *link*

- class=ndIf (`IF` statement: guarded command)

  left guard (*boolean expression*)

  right *statements*

  link next guarded command (`ndIf`), or `NIL`

- class=ndCase (`CASE` statement)

  left selection expression (integer or `CHAR` *expression*)

  right.left list of `CASE` branches (`ndCaseDo` linked with *left*)

  right.right `ELSE part` (*statements*)

  right.conval constant record with the following values:

  - *intval*: minimal `CASE` label
  - *intval2*: maximal `CASE` label
  - *set*={} iff no `ELSE` part exists

- class=ndCaseDo (sub structure of `CASE` statement)

  link list of `CASE` labels for this branch (`ndConst` linked with *link*). These `ndConst` have a special format: it's an intervall [`conval.intval`..`conval.intval2`] and the field *type* holds a copy of the `CASE` expression's type.

  right *statements*

  left next `ndCaseDo` branch, or `NIL`

- class=ndConst (constant)

  obj constant declaration (`obj.mode=objConst`) or `NIL`, if the constant wasn't referenced with it's name

  conval constant value. Which field actually holds the value is determined by *type*.

- class=ndCall: See Statements.

Appendix A.3 Statements

Most statements (like `LOOP`, `WHILE`, etc.) can be translated into a single node with all substructures appended to this node. Other statements, e.g. `FOR`, have so many attributes attached to them that the informations have to be attached to more than one node.

A sequence of statements is linked with field *link. type* holds the pseudo type `OTable.noType`.

- class=ndAssign (assignment and predefined procedures)

Standard procedures like `INC`, `DEC`, `INCL`, `EXCL` etc. will be translated into nodes of this class. They are identified by the field *subclass:*

- normal assignment (`scAssign`)
- `EXCL` (`scExcl`), `INCL` (`scIncl`)
- `INC` (`scInc`), `DEC` (`scDesc`)
- `COPY` (`scCopy`)
- `SYSTEM.GET` (`scGet`), `SYSTEM.PUT` (`scPut`)
- `SYSTEM.GETREG` (`scGetReg`), `SYSTEM.PUTREG` (`scPutReg`)
- `NEW (scNewFix, scNewDyn) and SYSTEM.NEW` (`scNewSys`)
- `SYSTEM.DISPOSE` (`scDispose`)
- `SYSTEM.COLLECT` (`scCollect`)

The first parameter is stored in *left*, the second in *right* (*expressions*).

Special cases:

- `SYSTEM.MOVE` (`subclass=scMove`)
  The (three) parameters (*expressions*) are stored as list (chained with *link*) in *right.*
- `NEW` on fixed size data type (`subclass=scNewFix`)
  left: pointer (*designator*)
- `NEW` on (multiple) open array type (`subclass=scNewDyn`)
  left: pointer (*designator*)
  right: list of dimension lengths (*expressions*), linked with *link*
- `COPY` (`subclass=scCopy`)
  left: string constant or variable (*designator* or *constexpression*)
  right: string variable, destination of copy (*designator*)

- class=ndConst (constant): See Expression.

Appendix A.2 Expressions

All Oberon-2 constructs denoting a value are part of the category *expression*. If in the following section a substructure's value is marked as *expression*, then this includes *designators* (see preceding chapter).

- class=ndUpto (set range)

  left lower border (*expression*)

  right upper border (*expression)*. Note: Single set elements are stored as set ranges with equal lower and upper border. A `SET` of the form `{a..b,c,d..e}` will be translated like `{a..b}+{c..c}+{d..e}`.

- class=ndMOp (unary operator or predefined function)

  left operand resp. parameter (*expression*)

  subclass: operator code, can be divided into

  - Expressions
    `scNot`, `scMinus` (includes `SET` and numerical data types)
  - Predefined functions
    `scAbs`, `scCap`, `scOdd`, `scAdr`, `scCc`, or `scSize`
  - Type Conversion (`subclass=scConv`)
    type is the destination type. The original type and the destination type are either numerical or `CHAR`.
  - `SYSTEM.VAL` (`subclass=scVal`)
    type holds the destination type

- class=ndDOp (binary operator or predefined function)

  left left operand resp. first parameter (*expression*)

  right right operand resp. second parameter (*expression*)

  subclass operator code, can be divided into

  - Expressions
    Arithmetic operators (`scMinus`, `scPlus`, `scTimes`, `scRDiv`, `scIDiv`, `scMod`), boolean operators (`scAnd`, `scOr`), `IN` (`scIn`),
    type test (`scIs`), and relations (`scEql`, `scNeq`, `scLss`, `scLeq`, `scGrt`, `scGeq`)
  - Predefined Functions
    `scAsh`, `scBit`, `scLsh`, `scRot`, `scLen`
    Note: For the function `LEN` a second parameter is added if necessary.

ectors. Is therefor `node` part of a designator then `node` has to be a (qualified) identifier with `node.left=NIL`, or it is a selector of a kind determined by `node.class` and whose prefix is stored in `node.left`. *type* holds the data type of the expression that results from applying the selector to it's prefix.

For all nodes of this class holds:

left points to the prefix of the selector (*designator*). The field is `NIL`, if the node denotes a variable, constant, procedure, or type identifier.

type is the type of the current node (when applied to the prefix in *left*).

Here a list of the single classes:

- class=ndVar (variable)

  obj variable or value parameter parameter resp. receiver (`obj.mode=objVar`)

- class=ndVarPar (`VAR` parameter)

  obj `VAR` parameter resp. receiver (`obj.mode=objVarPar`)

- class=ndField (record field)

  obj record field (`obj.mode=objField`)

- class=ndTBProc, class=ndTBSuper (type bound procedure)

  obj type bound procedure (`obj.mode=objTBProc`)

  type the procedure's formal parameter list (`struct.form=strTBProc`).
  Note: The first parameter in the list is actually the receiver.

- class=ndDeref (pointer dereference)

- class=ndIndex (array index)

  right index (*expression*)

- class=ndGuard (type guard)

  obj type against which the variable is checked (`obj.mode=objType`)

- class=ndType (data type)

  obj data type (`obj.mode=objType`). Used in conjunction with type tests, `SIZE` and `SYSTEM.VAL`.

- class=ndProc (procedure)

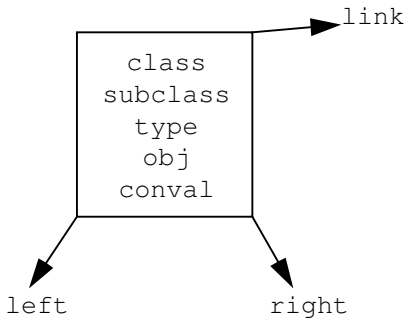
  obj procedure (`mode=objExtProc, objIntProc, or objLocalProc`)
  Note: This can't be a predefined procedure since these are converted by the parser in expressions of the class `ndMOp`, `ndDOp`, or `ndAssign`.

  type the procedure's formal parameter list (`struct.form=strProc`)

Appendix A Syntax Tree

This data structure contains all the elements of the source program that execute actions (i.e. statements and expressions), or that encapsulate actions (module or procedures). Constant, type or variable declarations can't be found here. They are part of the symbol table and are only referenced from nodes of the syntax tree. Running the parser will result in a single node, the root of the syntax tree, describing the compiled module.

```
TYPE
   Node = POINTER TO NodeDesc;
   NodeDesc = RECORD
      class, subclass : SHORTINT;
      left, right, link : Node;
      type : OTable.Struct;
      obj : OTable.Object;
      conval : OTable.Const;
      pos : LONGINT
   END;
```



The function of a single node is determined by the field *class*. The possible values for this field are defined as symbolic constants in module `OEParse` (their prefix is `nd`), e.g. the value `ndWhile` denotes a `WHILE` loop, and `ndVar` stands for an arbitrary variable that isn't declared as a `VAR` parameter.

The classes `ndDOp` (dyadic operator), `ndMOp` (monadic operator) and `ndAssign` (assignment) are further subdivided by the field *subclass* (legal values: `OEParse.scXXX`). E.g. the expression `a-b` will be translated into a node of `class=ndDOp, subclass=scMinus`.

Each expression evaluates into a value of a determined type (see [Mös93b], Appendix A). Each operator expects operands of determined types. This type information is stored in the field *type*. The compiler depends on this field for type checks and when inserting conversion nodes (`class=ndConv`).

obj holds references to declared objects (e.g. for a procedures call or as reference to a variable).

pos is the character position in the source file where the expression resp. the statement was defined. This allows for 'to the point' error messages.

Appendix A.1 Designators

A designator designates a constant, a variable, a procedure or a data type. It has the following (simplified) form:

```
Designator = Qualident {Selector},
```

i.e., an (possibly qualified) identifier is followed by a sequence of selectors. This structure will be translated into a linear list (the elements are linked with *left*) in reversed order with regard to the source text: The list's last element is the qualified identifier, preceded (in reverse order) by the sel-