ETH

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Régis Crelier

OP2: A Portable Oberon Compiler

February 1990

125

Authors' address:

Computersysteme
ETH-Zentrum
CH-8092 Zurich, Switzerland

e-mail: crelier@inf.ethz.ch

Abstract

This report describes a portable compiler for the language Oberon [Wirth 89]. The compiler consists of two parts: the so-called front-end and back-end. The front-end builds a machine-independent structure representing the program. This structure is made up of a symbol table and a syntax tree, rather than a linear sequence of pseudo-instructions coded in an "intermediate language". Whereas the front-end can remain unchanged, the back-end has to be reprogrammed, when the compiler is retargeted. The chosen structure allows the insertion of an optional phase, doing different grades of optimization. This technical report provides a detailed description of the intermediate program representation and is first of all directed to the designer of a back-end.

Contents

1. Introduction

Nowadays, portability has become a very important criterion for program quality. Considering the increasing complexity of software products, we cannot imagine rewriting a program each time it has to be implemented on a different computer. If the program is coded in a higher level programming language, it makes the job easier. Indeed, a programming language is (or should be) machine-independent, so that programs can be moved from one machine to another without any changes. Programming difficulties resulting from machine-dependencies (like processor instruction set or input-output devices e.g.) are made invisible to the programmer and are handled by the compiler or the operating system.

A compiler is a program as well, and it may be ported. If it can continue to produce the same code as before, but on a different machine (cross compiler), it is not more difficult to port it than any other program also written in a higher level programming language. But if the produced code must run on the new machine, the compiler has to be rewritten and it is not the same program anymore. By the term *portable compiler*, we actually understand a compiler that needs minimal effort to be adapted to a new machine and/or to be modified to produce new code.

Most related works attempt to reduce this adaptation cost to a minimum, without considering in the first place the influence on the compilation speed and on the code quality. A classification of such automated retargetable code generation techniques and a survey of the works on those techniques is presented in [Ganapathi 82]. The basic idea is to produce code for a virtual machine. This code is then expanded into real machine instructions. The expansion can be done by handwritten translators [Amman 74] or by a machine-independent code generation algorithm, in which case each intermediate language instruction [Tanenbaum 89] or each recognized pattern of these [Glanville 78] is expanded into a fixed pattern of target machine instructions recorded in tables. Trees may replace linear code to feed the pattern matching algorithm [Aho 89, Cattell 79, Fraser 86], but the code quality seldom satisfies, so that a peephole optimization phase is added, which still increase the compilation time.

Portability, code quality, and compilation speed are given the same importance in our approach. Thus, a pattern matching or table-driven code generation has been directly discarded. An elegant way to write a compiler today is to design it as single-pass compiler. This implies that all compilation phases are executed in the same time, and that no intermediate representation of the source text is needed between the different phases, making therefore the compiler very compact and efficient. But this technique has disadvantages too: since machine-independent and machine-dependent phases are mixed up, it is very difficult to modify the compiler for a new machine. If the programmer doesn't understand the whole program very well, he doesn't stand a chance to port it in due time. And if he still succeeds, the danger is very high that the compiler doesn't work correctly anymore.

One solution to the problem is to clearly separate the compilation phases into two groups: the machine-independent ones making up the *front-end* (lexical and syntaxical analysis, including type checking) and the machine-dependent ones making up the *back-end* (storage allocation, code generation). Only the back-end must be modified when the compiler is ported. The front-end is responsible for constructing an intermediate representation of the program and a symbol table which is used to check context-sensitive syntax, such as type compatibility or scope rules for example. If no errors were found, the front-end then passes control to the back-end, which generates code from this intermediate representation. A noteworthy advantage consists in the fact that the back-end receives a structure which is free of errors. Thus, it doesn't need to deal with type checking and error recovery. It must only guarantee that implementation specific restrictions are respected (e.g. maximal code length or maximal jump distance). Furthermore, it is now possible to program the front-end and the back-end separately. Each is implemented as one (or several) independent module(s). An interesting point has to be noticed: this structure makes it possible to add extra passes which may be required to improve the code. This optimization phase cannot easily be embedded in a conventional single-pass compiler.

The intermediate representation could be a stream of instructions for a virtual machine; we have preferred an internal abstract syntax tree. The tree has some non-negligible advantages in comparison with a linear code: no linearization of the program is needed, because the tree is a natural mapping of the Oberon syntax, and it is therefore easy to embed its recursive construction algorithm in the recursive-descent parser. The structure of the program is kept intact, so that a control-flow analysis needed for optimization is already done. The reordering of program pieces is easier to perform in a tree than in an instruction stream. A disadvantage resides in the fact that the tree uses a quite large heap space, but this should not be a real problem nowadays.

In the compiler for the Ceres workstation [Eberle 87], the symbol table and the syntax tree take up 32 times more memory space than the produced code (or about 8 times the source text), one third being occupied by the symbol table only. The garbage collector is called after the compilation of each module. This version of the compiler (6200 program lines, 10 modules, 58 KB code) recompiles itself within 30 seconds.

2. Modularization

The basic version of the portable Oberon compiler consists of nine modules (see Fig.1) written in Oberon. Module names have all the same form: *OPx*. "O" stands for Oberon, "P" for portable, the third letter specifies the function of the module .

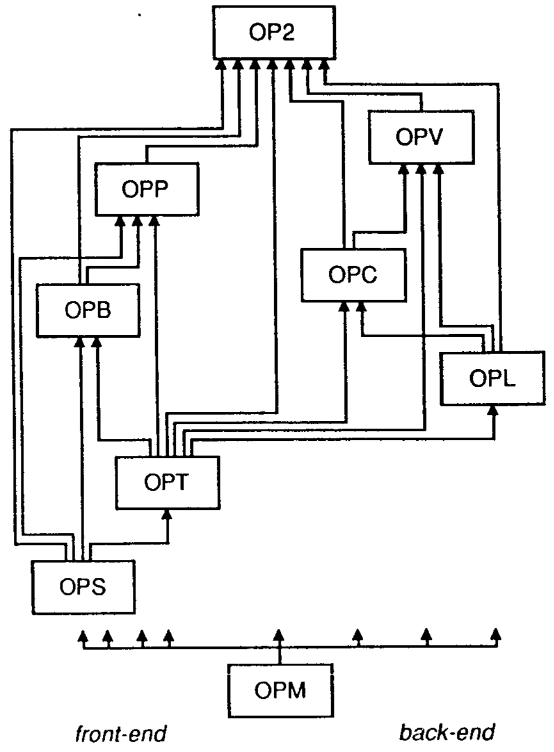


Fig. 1 Module import graph (an arrow from A to B means *B imports A*)

The topmost module (OP2) doesn't do anything but call the front-end with the source text to be compiled as parameter. If no error has been detected, it then calls the back-end with the root of the tree that was returned by the front-end as parameter.

The lowest module of the whole hierarchy is OPM, where "M" stands for machine. This module defines and exports several constants used to parametrize the front-end, so that some

implementation restrictions can already be checked in the front-end modules. Some of these constants reflect target machine characteristics, like basic type sizes (these maximal values are used in the front-end to check the evaluation of constant expressions). Other constants define restrictions, like maximal number of exit statements in a loop statement, or of labels in a case statement.

OPM has a second function. It works as interface between the compiler and the host machine, i.e. the machine on which the compiler is running. This interface includes for example procedures to write messages and errors to the host screen and procedures to create and read symbol files. It would be possible to split OPM into two distinct modules, the first for the target, and the second for the host machine. But it is not worth while, because the module is short enough so that it is no problem to distinguish what is target-specific from what is host-specific. Furthermore we must take care that the number of modules doesn't grow unduly large.

Between the highest and the lowest module, we find the front-end and back-end, which consist of four modules each. There is no interaction during compilation between these two sets of modules. The definition in the front-end of the symbol table and of the syntax tree is known in the back-end too (which explains the presence of arrows from OPT to back-end modules), but there is no transfer of control, such as procedure calls.

The front-end is controlled by the module OPP, a syntax directed top-down recursive-descent parser. Its main task is to check syntax and to call appropriate procedures to construct the symbol table and the syntax tree. The parser requests terminal symbols from the scanner (OPS) and calls procedures of module OPT, the symbol table handler, and of OPB, the syntax tree builder, which also checks type compatibility.

The back-end is controlled by the module OPV, the tree traverser. It first traverses the symbol table to enter machine-dependent data in the table elements (using OPM constants), such as the size of types, the address of variables and the offset of record fields. It then traverses the syntax tree and calls for each node or recognized pattern of nodes, a corresponding procedure of OPC (code emitter), which in turn synthesizes machine instructions using procedures of OPL, the lower level code generator. The module OPC may be split into several parallel modules, in order to maintain a reasonable module size.

3. Part One: Description of the Front-End

3.1 Symbol Table

The symbol table represents all declarations, i.e. constants with their value, types with their structure, variables with their type and procedures with their parameters and local declarations. It is used in the front-end to check whether context-sensitive rules of the language are respected. But the back-end utilizes it too; it stores the addresses of variables in it and uses the type information contained in it.

The symbol table is a dynamically allocated data structure with three different types of elements:

```
TYPE
    Const = POINTER TO ConstDesc;
    Object = POINTER TO ObjDesc;
    Struct = POINTER TO StrDesc;
```

An *Object* is a record (or, more exactly, a pointer to a record), which represents a declared, named object. The name of an object, which is limited to 23 characters plus a terminating mark, is stored in the record itself, and is used as key to retrieve the object in its scope. Each scope is represented by a sorted binary tree of objects, which is anchored to the owner procedure, which in turn belongs as object to the enclosing scope.

The object declaration is the following:

```
ObjDesc = RECORD
    left, right, link, scope: Object;
    name: OPS.Name;
    marked, leaf: BOOLEAN;
    mode, mnolev: SHORTINT;
    typ: Struct;
    conval: Const;
    adr, linkadr: LONGINT
END;
```

In module OPS:

```
Name = ARRAY 24 OF CHAR;
```

The tag field *mode* specifies the class of the object. The table below gives a survey of existing modes and the meaning of the fields for each mode (the fields *left, right, name, marked, mnolev, typ* and *linkadr* have always the same meaning independently on the mode and are hence not listed in the table):

mode	adr	conval	link	scope	leaf	
Undef						*Resulting from an erroneous decl*
Var	*vadr*		*next*		*regopt*	*Glob or loc var, value parameter*
VarPar	*vadr*		*next*		*regopt*	*Var parameter*
Con		*val*				*Constant*
Fld	*off*		*next*			*Record field*
Typ						*Named type*
LProc		*sizes*	*firstpar*	*scope*	*leaf*	*Local procedure*
XProc	*pno*	*sizes*	*firstpar*	*scope*	*leaf*	*External procedure*
SProc	*fno*	*sizes*				*Standard procedure*
CProc		*code*	*firstpar*	*scope*		*Code procedure*
IProc	*pno*	*sizes*		*scope*	*leaf*	*Interrupt procedure*
Mod	*key*			*scope*		*Module*
Head	*txtpos*		*owner*	*firstvar*		*Scope anchor*

The fields *left* and *right* are part of the binary tree structure and are used to sort the objects in their scope. Parameters of the same procedure, as well as fields of the same record, and variables of the same scope are additionally linked together to maintain the declaration order using field *link*. These chains of objects are traversed in the back-end during storage allocation. The field *scope* of a procedure object points to the anchor of its local scope of objects. Such an anchor (*mode* = *Head*) is anonymous; its *left* field points to the anchor of the enclosing scope and its *right* field points to the first object of the actual scope.

For an external object, *mnolev* indicates the module number from which module the object is imported[1]. For a local object, *mnolev* equals the procedure nesting level of the object declaration[2]. If *marked* is true, the object is exported. The field *adr* is initialized in the back-end module OPV; it gives the address of a variable or of a parameter, the offset of a record field or the entry number of an external procedure. *conval* is a pointer to a constant descriptor, which contains the value of the constant object or, in case of a procedure object, the total sizes of its parameters and of its local variables. These sizes are computed in OPV module and are not involved in front-end modules. The field *linkadr* can be freely used by the back-end, too. The field *leaf* indicates whether a procedure is a leaf procedure or whether a variable is a candidate to be allocated permanently in a register (see *Simple Optimizations*).

[1] *mnolev* < 0, module number = -*mnolev*

[2] *mnolev* ≥ 0, level = *mnolev*, 0 means global

The field *typ* reflects the object type, which type is represented by a pointer to a record called *StrDesc*:

```
StrDesc = RECORD
    form, comp, mno: SHORTINT;
    ref, sysflag: INTEGER;
    n, size, adr, txtpos: LONGINT;
    BaseTyp: Struct;
    link, strobj: Object
END ;
```

The fields *form* and *comp* differentiate kinds of types, e.g. basic types like boolean, long integer or set, and composite types like array or record, as shown in the following table:

form	*comp*	*n*	*BaseTyp*	*link*	*mno*	*sysflag*
Undef	*Undef*					
Byte	*Basic*					
Bool	*Basic*					
Char	*Basic*					
SInt	*Basic*					
Int	*Basic*					
LInt	*Basic*					
Real	*Basic*					
LReal	*Basic*					
Set	*Basic*					
String	*Basic*					
NilTyp	*Basic*					
NoTyp	*Basic*					
Pointer	*Basic*		*PBaseTyp*		*mno*	*sysflag*
ProcTyp	*Basic*		*ResTyp*	*params*	*mno*	
Comp	*Array*	*nofel*	*ElemTyp*		*mno*	*sysflag*
Comp	*DynArr*	*dim*	*ElemTyp*		*mno*	
Comp	*Record*	*exlev*	*RBaseTyp*	*fields*	*mno*	*sysflag*

Instead of two fields *form* and *comp*, only one could suffice. But as there are more than 16 kinds of form, it wouldn't be possible to use sets on 16-bit machines to make the distinction efficiently, and the compiler wouldn't be portable anymore. We emphasize on the fact that this partition doesn't cost any execution time, because normally only one field (*form* or *comp*) is involved in a test.

The field *n* is a general-purpose variable, which represents the extension level of a record (0 if not

an extended record), or it denotes the number of elements of an array, or the number of dimensions of the element type of a not imported dynamic array. Dynamic arrays are also called open arrays. *BaseTyp* is a pointer to another structure, which can be the type that a pointer type is pointing to, the result type of a procedure type[3], the element type of an array or open array, or the base record type from which a record has been extended. The field *link* is the anchor of an object list, which can be the parameter list of the procedure type or the field list of the record type. *mno* is the module number, from which module the type is imported. The field *txtpos* denotes the text position of the type declaration for possible later error reporting. The field *adr* is set to 0 for predefined types and to an OPM constant for other types; it can be freely used in the back-end. *size* is the required memory space expressed in bytes for an element of this type. This field is initialized by the front-end to the correct value with OPM constants for basic types and set to a dummy value (-1) for composite types. The back-end has to correct this value during storage allocation. Each exported structure is given a reference number, which is stored in *ref* and in the symbol file, this field *ref* is reserved until the symbol file has been written. *sysflag* is a special field containing a number copied from the source text[4], whose interpretation is left to the back-end (see *Interfacing with an Existing Operating System*). A named type has a pointer *strobj* to the corresponding object containing the name.

The third record type used in the symbol table is *Const*:

```
ConstDesc = RECORD
    ext: ConstExt;         (* string or code for code proc *)
    intval: LONGINT;       (* constant val or adr, proc par size, text position or lower case label *)
    intval2: LONGINT;      (* string length, proc var size or higher case label *)
    setval: SET;           (* set constant val, "ELSE" present in case or proc body present *)
    realval: LONGREAL      (* real or longreal constant val *)
END ;

ConstExt = POINTER TO OPS.String;
```

In module OPS:

```
    String = ARRAY 256 OF BYTE;
```

Const is used to store the values of declared constants or some attributes of procedure objects (such as parameter and local variable space, which are computed later in the back-end), or the value of an anonymous constant appearing in expressions. In the latter case, the record is not anchored to an object but to a node of the syntax tree (see next paragraph).

[3]*notyp* with form *NoTyp* if the procedure is not a function

[4]*sysflag* = 0 if not explicitly specified in the source text

3.2 Syntax Tree

On the one hand, the front-end constructs a symbol table representing all declarations as explained above, and on the other hand, it builds a syntax tree representing all statements of the program. The Oberon syntax is mapped into a binary tree of elements called *Nodes*, all having the same form:

```
NodeDesc = RECORD
    left, right, link: Node;
    class, subcl: SHORTINT;
    typ: Struct;
    obj: Object;
    conval: Const
END ;
```

A binary tree has been chosen because each Oberon construct can be easily decomposed into a root element identifying the construct and two subtrees representing its components: an operator has a left and a right operand, an assignment has a left and a right side, a while statement has a condition and a sequence of statements, and so on... Some Oberon constructs are organized sequentially: we find lists of parameters in procedure calls and sequences of statements in structured statements. It would be expensive to insert dummy nodes to link the subtrees representing these constructs; an additional *link* field in the node is much cheaper.

Each node has a class, as specified by the tag field *class*, and possibly a subclass according to the field *subcl*. Each node is given a type, represented by the field *typ*, which is a pointer to a *StrDesc* of the symbol table. Similarly, a leaf node representing a declared object has its field *obj* pointing to the corresponding *ObjDesc* of the symbol table. The field *conval* points to a *ConstDesc* containing the value of an anonymous or named constant. Such a *ConstDesc* denoting the position in the source text is anchored to the root node of each statement. This allows locating compilation errors reported by the back-end, like implementation restrictions or machine specific limitations.

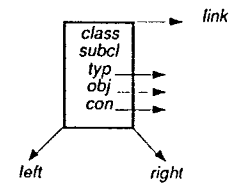
While generating code for a node, we typically have to recursively evaluate left and right subtrees, then the node itself, and finally the linked successor if any. A traversal of the tree may look like this:

```
Traverse(node: Node):
    WHILE node # NIL DO
        Traverse(node^.left)
        Traverse(node^.right)
        Do something with node
        node := node^.link
    END
```

A detailed description of the syntax tree is given in the appendix. Some examples of Oberon statements with their corresponding syntax tree follow:

```
CONST c = 2;
VAR x, y:SHORTINT;
...
x := x*y + c;
INC(y);
...
```

Picture for a *NodeDesc*:



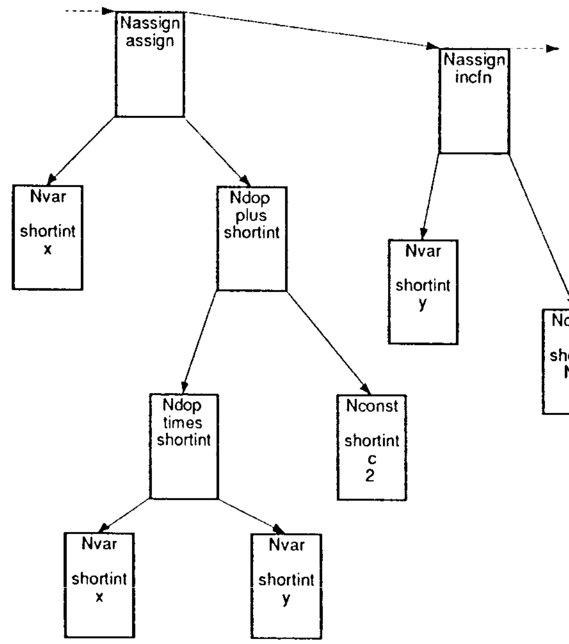


Fig. 2 Assignments

Fig. 2 shows the tree representing an assignment of an expression to a variable followed by a standard procedure call. Note that although there is a reference to the constant c in the symbol table, the constant descriptor is nevertheless copied and directly anchored to the node *Nconst*. Thus the copy can be modified (e.g. by constant expression evaluation at compile-time) without altering the symbol table structure. In this case the object reference is reset to NIL. This reference is used for constants like strings, which need to be allocated; each use of the constant refers to the same address stored in the symbol table, so that the constant be allocated only once.

```
CONST c = 1;
VAR x: INTEGER;
    y: SHORTINT;
    z: REAL;
...
z := x*y + c;
...
```

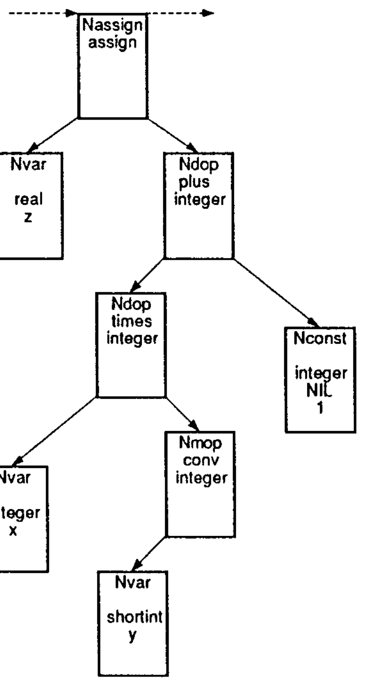


Fig. 3 Conversion nodes in arithmetic expressions

The rules of numeric type compatibility are very flexible in Oberon; it is possible to add for example short integers (SHORTINT) to long integers (LONGINT). Fig. 3, 4 and 5 show how conversion operators are inserted in the syntax tree. The interpretation of these operators is left to the back-end.

Note that the type of the constant is automatically changed from shortint to integer (doing this, the reference to *c* is reset to NIL) and that no conversion operator is inserted between the assignment node and its right operand, because some target machines (like NS32000) can convert and move data in one instruction. A conversion operator would always decompose this instruction into two much more expensive steps: conversion and assignment. Thus, the back-end is responsible for checking the type of the left and the right side of an assignment, and performing a conversion, if necessary.

```
VAR x: LONGINT;
    y: INTEGER;
    ch: CHAR;
...
y := SHORT(x)+ORD(ch);
...
```

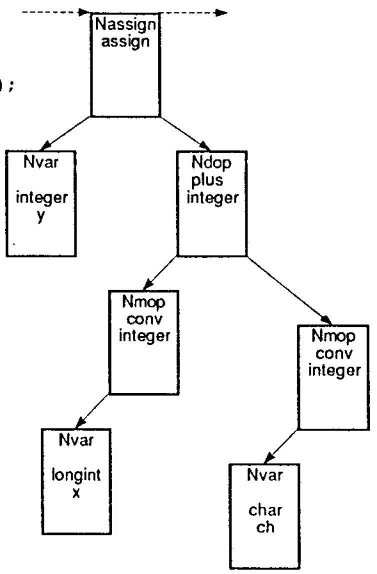


Fig. 4 Conversion nodes for standard functions

```
CONST c = {0..2, 4};
VAR s: SET;
    x, y: INTEGER;
...
s := c*{2..6}+{x..3}+{y};
EXCL(s, x);
...
```

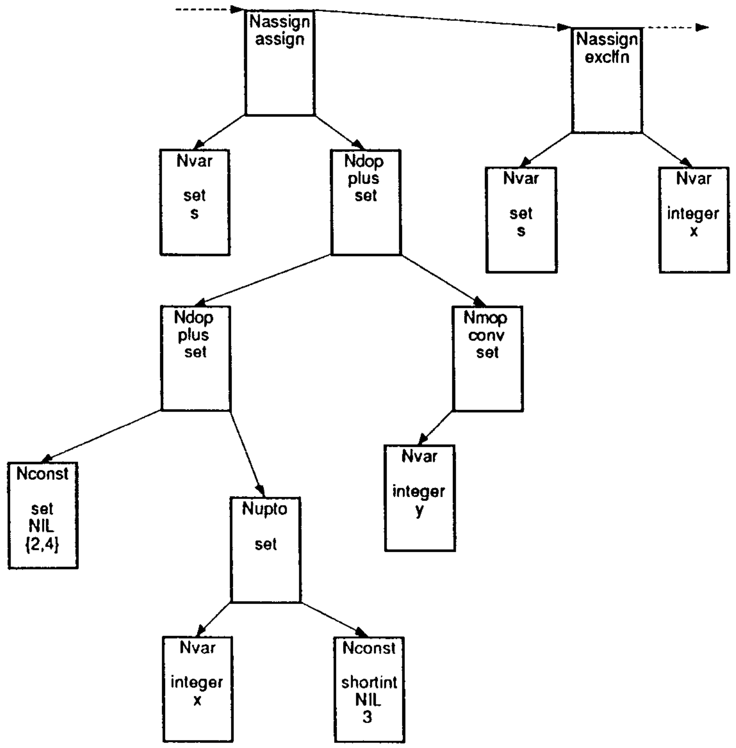


Fig. 5 Conversion nodes and sets

Note that the set intersection operation has been executed and that only one node has been created, because the operands were constants.

Examples of structured statements like *if* or *case* are found in Fig. 6 and 7.

```
IF expr1 THEN stat1
ELSIF expr2 THEN stat2
ELSIF expr3 THEN stat3
ELSE stat4
END
```

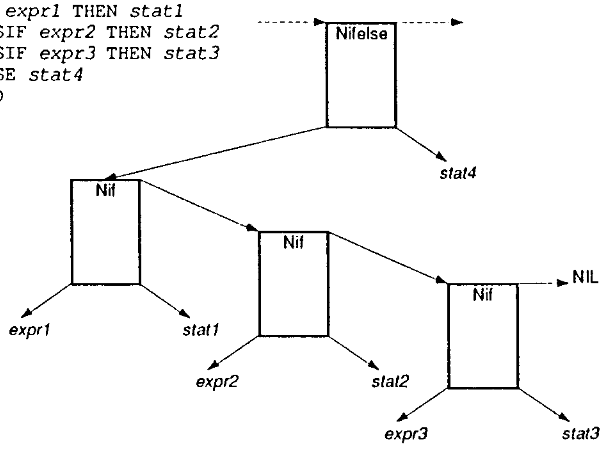


Fig. 6 IF statement

```
CASE expr OF
  1       : stat1
| 2..4, 6: stat2
ELSE stat3
END
```

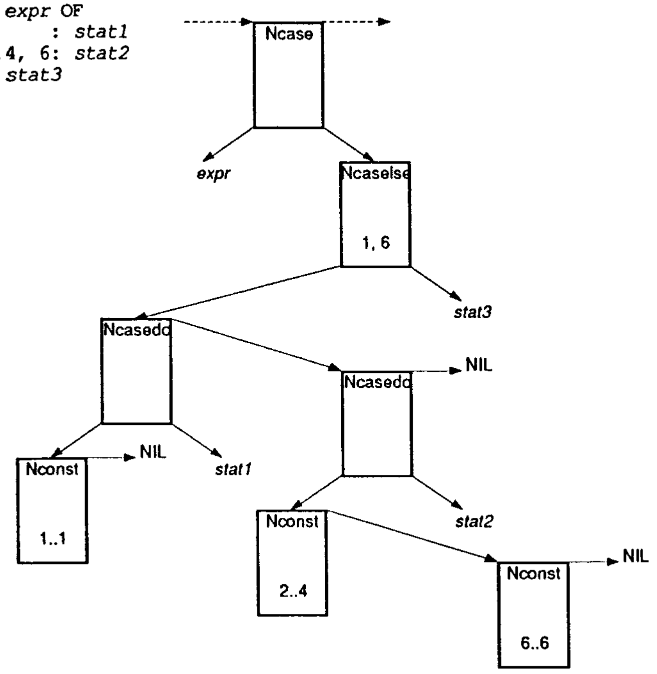


Fig. 7 CASE statement

The lower bound of a range is stored in *conval^.intval* and the upper bound in *conval^.intval2* of the *Nconst* node.

conval^.intval of the *Ncaselse* node denotes the least label of the whole statement, while *conval^.intval2* the largest one (1 and respectively 6 in the example above).

conval^.setval of the *Ncaselse* node is not empty if the keyword "ELSE" appeared in the source text. If it is empty, then the back-end has to generate a trap.

Fig. 8 shows the syntax tree for a module with its procedures and their statement sequences. *Ninittd* nodes initialize type descriptors of declared records or arrays.

```
MODULE M;

 TYPE
  Rec = RECORD
   ...
  END;

 PROCEDURE P;
 BEGIN
  statP
 END P;

 PROCEDURE Q;

  PROCEDURE R;
   BEGIN
    statR
   END R;

  BEGIN
   statQ
  END Q;

BEGIN
 statM
END M.
```

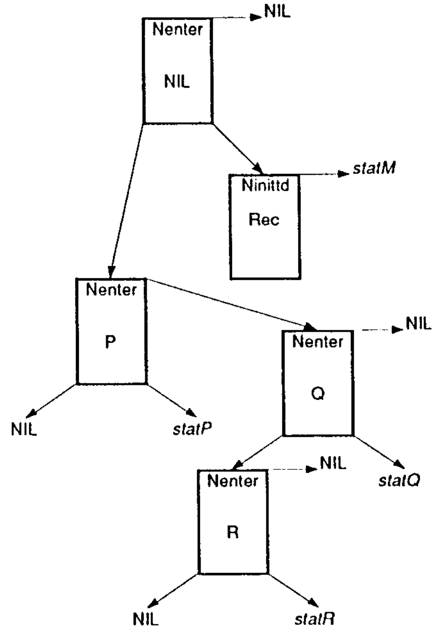


Fig. 8 Module and procedures

Ninittd nodes corresponding to locally declared records or arrays do not appear in the procedure body but in the module initialization part, similarly to global ones (type descriptors need to be initialized only once).

The whole syntax tree representing the module (all local procedures included) and the symbol table (all local scopes included) remain stored in memory during the compilation of a module.

3.3 Simple Optimizations

The front-end performs some simple optimizations:

• evaluation of constant expressions (constant folding)
Every expression or subexpression exclusively containing constant operands is evaluated and replaced by the result.

• boolean expression with one constant operand
If the left operand of a boolean operator (*OR*/*&*) is a constant, the expression is replaced by the result, i.e. by the constant operand or by the right operand. If the right operand is constant, folding is only performed in the following cases:

x & TRUE	is replaced by *x*
x OR FALSE	is replaced by *x*

• transformation of IF statements containing constant conditions
Unreachable code due to a constant condition in an *IF-ELSIF-ELSE* statement is removed.

• convert nodes
A convert node (class = Nmop, subcl = conv) is never applied to a constant, but the constant is immediately converted. An application of a convert node to another convert node may be optimized in some cases, examples:

VAR si: SHORTINT; i: INTEGER; li: LONGINT;

li := LONG(LONG(si))	only one convert node is needed
si := SHORT(SHORT(li))	only one convert node is needed
i := SHORT(LONG(i))	no convert node is needed
i := LONG(SHORT(i))	two convert nodes are needed (to enable range check)

• integer multiplication by a power of two
An integer multiplication (class = Ndop, subcl = times) by 2^n is replaced by an arithmetic shift of n places (subcl = ash).

• integer division by a power of two
An integer division (class = Ndop, subcl = div) by 2^n is replaced by an arithmetic shift of -n places (subcl = ash).

• modulus by a power of two
A modulus (class = Ndop, subcl = mod) by 2^n is replaced by a bit mask operation with m (subcl = msk), where m is a longint constant with all bit set to one, excepted the n least significant set to zero.

• standard function LEN
LEN(a[$expr_1$, expr2, ..., exprm], n) is replaced by *LEN(a, m+n)*
(Possible side effects in *expri* are ignored).

• leaf procedures and register variables
The field *leaf* of a procedure object (mode = LProc, XProc or IProc) indicates whether the procedure is a leaf of the procedure hierarchy tree, i.e. whether it doesn't call any other procedure; this information may be used to generate more efficient procedure calling code. In the case of a variable object (mode = Var or VarPar), *leaf* is true if the variable is a candidate to be allocated permanently in a register.

A variable or value-parameter (mode = Var) is not candidate if its address is used somewhere in the program.or if it is accessed from an intermediate level (or exported). The address of a variable is needed when the variable is passed as actual parameter to a procedure with formal var-parameter or to the standard procedures SYSTEM.ADR, NEW, SYSTEM.NEW or SYSTEM.MOVE.

A var-parameter (mode = VarPar) is not candidate if accessed from an intermediate level (the address of a var-parameter descriptor is never used).

Some implementations of NEW and SYSTEM.NEW might not be able to use the address of the pointer variable to directly write the returned value in the variable. These procedures might instead return the value in a register. In this case, the pointer variable remains a candidate (see *Parametrization of the Front-End*).

3.4 Module SYSTEM

The module SYSTEM contains low-level procedures and functions that are specific to a given target machine. The following table gives a survey of the operations already implemented in the front-end. If some of them are useless in a given implementation, they can be removed, or new ones can be inserted (see *Parametrization of the Front-End*). *v* stands for a variable, *a*, *n* and *x* for expressions, and *T* for a type.

Function procedures:

Name	Argument types	Result type	Function
ADR(v)	any	LONGINT	address of variable v
BIT(a, n)	a: LONGINT n: integer type	BOOLEAN	Mem[a][n]
CC(n)	integer constant	BOOLEAN	Condition n
LSH(x, n)	x, n: integer type	LONGINT	logical shift
ROT(x, n)	x: integer type, BYTE, CHAR n: integer type	type of x	rotation
VAL(T, x)	T, x: any type	T	x interpreted as of type T

Proper procedures:

Name	Argument types	Function
GET(a, v)	a: LONGINT v: any basic type	v := Mem[a]
GETREG(n, v)	n: integer type v: any basic type	assign content of register specified by n to v
PUT(a, x)	a: LONGINT x: any basic type	Mem[a] := x
PUTREG(n, x)	n: integer type x: any basic type	assign x to register specified by n
MOVE(v0, v1, n)	v0, v1: any type n: integer type	assign first n bytes of v0 to v1
NEW(v, n)	v: any pointer type n: integer type	allocate storage block of n bytes assign its address to v

4. Part Two: How to Write a Back-End

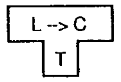
We will now explain how to proceed to write a back-end for a specific machine. Although an extensive experience in writing compilers is not needed, knowledge of the subject is nevertheless an advantage.

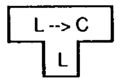
4.1 Porting the Compiler

Considering the starting situation, we have at our disposal the OP2 Compiler written in Oberon consisting of a machine-independent front-end and an empty back-end that we must complete. This programming work being done, we are in possession of an Oberon source program, that has to be compiled by an Oberon compiler to be executed. The problem is that we haven't yet an executable version of the compiler.

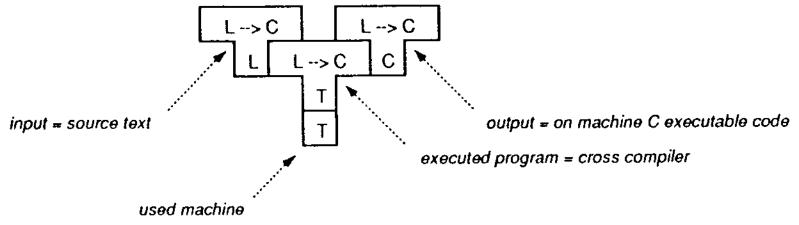
It would be unreasonable to start from scratch, i.e. to iteratively bootstrap the compiler, the initial version being hand coded in assembler. Therefore we need an already working compiler. It may be a Modula-2 compiler (a Modula-2 version of the front-end is available) or, much better, an Oberon compiler running on a different machine as the new compiler will do.

In any case, we have to bootstrap the compiler to obtain an executable version of it. Some possible bootstrap sequences are illustrated with the help of T-diagrams:

A compiler compiling language L into code C and running on machine T: 

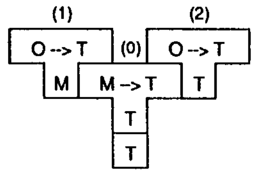
The source text of the compiler written in L: 

Example of a cross-compilation executed on machine T:


L --> C
L
L --> C
L --> C
C
T
T
input = source text
output = on machine C executable code
executed program = cross compiler
used machine

For more details about T-diagrams, please refer to [Wirth 86].

Figure 9 shows how we can obtain an executable version of OP2 if we have at our disposal on the one hand a machine T and a Modula-2 compiler (0) producing T code and running on T, on the other hand a version of OP2 (1) written in Modula-2 producing T code:



O: Oberon

M: Modula-2

T: Target machine (or code)

Fig. 9 Bootstrap step using a Modula-2 compiler

The result is an Oberon compiler (2) producing code for and running on T. The goal is already reached, but the code quality is that of the Modula-2 compiler and not of OP2. Furthermore, it is preferable to have a source version of the compiler written in Oberon to facilitate the maintenance of the compiler. For these reasons, we also write OP2 (3) in Oberon (its better to do this first and then translate it to Modula-2), and we add a compilation step as follows:

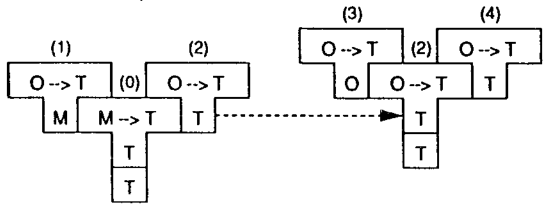


Fig. 10 Self-compilation step

If the Modula-2 version of the Oberon compiler (1) produces exactly the same code as the Oberon version (3), and it is usually the case, because (1) is the translation of (3) in Modula-2, then we must obtain a rigorously identical copy of the used compiler after a further self-compilation step ((5) = (4) in fig. 11). In any case, even if (1) and (3) don't produce the same code , i.e. (5) ≠ (4), a fixpoint must be reached after an additional similar bootstrap step ((6) = (5)), if not, then we have to look for a bug in the Modula-2 or Oberon version (the original Modula-2 compiler (0) is supposed to be error-free(?)).

After testing, we can leave aside the Modula-2 version of the Oberon compiler, and work exclusively with the new bootstrapped compiler ((5) or (6)).

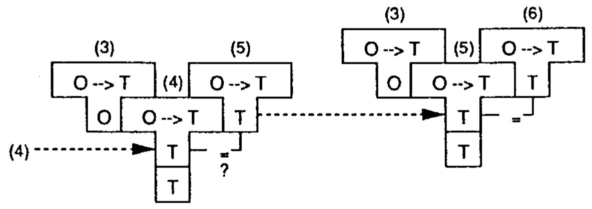


Fig. 11 Fixpoint test

Remark:

If we already have an Oberon compiler (other than OP2) for the machine T and we want to port OP2, the procedure to follow is the same as described above. We only need to replace the Modula-2 version source text (1) with the OP2 Oberon source text (3) and to compile it using the existing Oberon compiler instead of using the Modula-2 compiler (0). The next bootstrap steps are identical and a fixpoint must be already reached after the third compilation ((5) = (4)).

Let's now consider another starting configuration: we haven't yet a working compiler on the target machine T or we prefer not to use it for some reason. But we have at our disposal another machine H (for host) and a Modula-2 compiler (0) running on it and producing H code. Our goal is to obtain an Oberon compiler running on T and producing T code. We have at hand the source text of OP2 written in Oberon (3) and in Modula-2 (1), both producing code for the target machine T. Note that (1) is no more a one-to-one Modula-2 translation of (3), because the compiled version of (1) is running on H, but the compiled version of (3) is running on T. Modules OP2 and OPM have to be tuned differently in each version (see *Host Interface*). The compilation steps are illustrated in fig. 12.

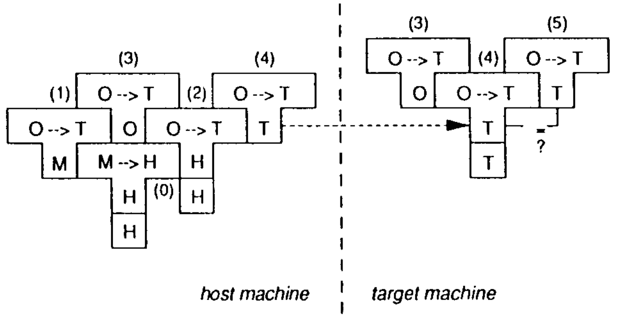


Fig. 12 Bootstrap with two machines

The same test procedure as explained above can be applied. If a further bootstrap step is necessary to reach the fixpoint, that means that (1) doesn't produce the same code as (3).

If an Oberon compiler (maybe OP2) exists on H, the Modula-2 translation is superfluous, and only one compilation of (3) by this compiler is needed before the obtained executable version can be transmitted to the target machine, on which the bootstrap process remains unchanged.

We must warn that the bootstrapping sequence might be more difficult to perform than it may seem at the first look, because of the following possible complications:

- The host machine has not necessarily the same byte ordering as the target machine (little-endian versus big-endian). In this case, the output to files has to be treated differently in versions (1) and (3).

- The target machine may have a larger data path width than the host machine (32-bit versus 16-bit). In that case, some target dependent constants may not be representable in the host machine, like basic type sizes for example. A solution to this problem is to temporarily set the maximum values of basic types (in module OPM) to the maximum common value. The bootstrap being done, the constants can be reset to their correct value. The host machine compiler should allow at least 16-bit SETs (or BITSETs in the Modula-2 translation) to produce the first executable version of the compiler (IN-tests are used with up to 16 elements). If it is not foreseen to compile the compiler on the target machine, but to keep a cross-compiler on the less powerful host machine, this temporary reduction of basic type size cannot be applied. We must modify the front-end, so that it can store and manipulate larger target constants. For example, the constant descriptor *OPT.ConstDesc* may contain several sets to represent one constant set on the target machine, and constant arithmetic routines of module OPB must be adapted.

- If the host machine has larger basic types than the target machine, there is no problem: all constants are representable in the host machine. But in any case, the host compiler must write constants in the object file using the target format, e.g. it must extend 16-bit set constants to 32-bit by padding with zeros, or truncate 32-bit ones to 16-bit.

4.2 Host Interface

Like in the preceding paragraph, we must here distinguish between the host machine on which the OP2 compiler is running, and the target machine for which the compiler is generating code. Perhaps the two machine are the same, but this doesn't change anything in the following considerations. We will concentrate here on the compiler dependencies on the host machine. The reader who doesn't want to write a back-end can skip this paragraph, which deals primarily with implementation details.

The compiler has to communicate with its environment, i.e. it takes a text to be compiled as input and it produces code as output. These input and output streams are normally files (sequences of bytes). Furthermore, it must be possible to specify the text to be compiled, using its file name or taking it directly from an already displayed viewer on the screen [Wirth 88]. During the compilation, errors (if any) have to be displayed or listed into a file. All this input and output operations are strongly dependent on the operating system, so that the compiler has to be tuned to the host machine.

The front-end modules (OPS, OPT, OPB and OPP) are host machine-independent, only the topmost module (OP2) and the bottommost one (OPM) have to be modified. The back-end is almost host independent, only the output of the object file (combined with the reference file, see *Debugging*) rely on the file system. It would be possible to implement these host dependent parts in module OPM, but a large number of new procedures would be necessary, whereas they are now all grouped in only one procedure of module OPL (*OPL.OutCode*). Furthermore, the procedure is highly target dependent, so that it has always to be replaced, when the compiler is ported. In any way it should be no problem for the back-end programmer to adapt this procedure, because he has written it himself.

Here follows a description of each constant or procedure of the modules OPM and OP2 that must be tuned to a new host machine. Module OPM, the lowest in the hierarchy, is both target and host dependent. Here we discuss only host dependent parts (which can be also target dependent):

- target machine bound values of basic types expressed in host machine format
  Minimum and maximum values of integer types and maximum value of set type are exported as constants. Real types bounds are exported in the form of variables, so that the exact value can be initialized using binary pattern and low-level operations. If these bounds are not representable in the host format, see *Porting the Compiler*.

- procedure *Init*
  This procedure is called by OP2 with some parameters such as a specifier of the text to be compiled, maybe an open file descriptor or a text *Reader* [Wirth 88]. All necessary initializations have to be done, so that later calls to the procedure *Get* and to the log output procedures (see below) do the right thing as expected.

• procedure *Get*

Each call of *Get* must return the next character of the source text. If no more character is available, a special character *Eot* is returned. This *Eot* character is the one returned by the text system of the host machine (normally *0X*); it has to be exported, so that the scanner can recognize it, when returned by the procedure *Get*.

• function *NewKey*

This function has to return a different long integer each time it is called, i.e. once during the compilation of one module. The current date and time read from the computer clock can be used for example.

• procedure *MakeFileName*

This procedure receives a module name, an extension and possibly a path specifier as parameters and it returns a concatenation of these parameters that represents a legal file name according to the file name syntax of the host machine. The parameter list can be modified if necessary. The procedure is only called in *OP2.CompilationUnit* and *OPL.OutCode*, and these two procedures are host dependent too.

• log output procedures

This is a set of little procedures, that write characters, strings, line breaks or integers on the terminal or log viewer. They are used to give some feedback during the compilation process. When a compiling error is found by the front-end or the back-end, the procedure *Mark* is called and this one displays the error in the log viewer or writes it into a file previously opened by *Init*. *Mark* takes the error number and the text position as parameter. The procedure *err* takes only the error number and calls *Mark* with the current text position, which must be initialized by the procedure *Init* and incremented by *Get*.

• procedures to read symbol files

The procedure *OldSym* opens for reading the existing symbol file of the module whose name *modName* is given as parameter. But this is not the file name, which has instead to be built concatenating a global path, the module name *modName* and a standard extension specifying symbol files, by the procedure *MakeFileName*. If the file is not found, (in this case *done* parameter must be set to FALSE) an error must be reported, but only if it is not the symbol file of the module being currently compiled (*self* parameter is TRUE). The state of the opened file has to be known globally, so that procedures that implicitly read items from this symbol file can refer to it. *CloseOldSym* is called to close a symbol file. The function *eofSF* returns TRUE if the end of the opened file is reached.

• procedures to write symbol files

NewSym creates a new temporary symbol file for writing. The file name has to be built from the module name in the same way as *OldSym* does. If a file with same name already exists, this one must not be removed, it will be compared later with the new one. *RegisterNewSym* makes this temporary file permanent and possibly destroys the already existing one with the same name.

DeleteNewSym deletes the temporary file. *EqualSym* compares an already existing file opened using *OldSym* with the new created one. The key of the old file is different in any way and must not be compared but returned as VAR-parameter. If the files are identical, except the key, *EqualSym* returns TRUE.

The module OP2 is at the top of the module hierarchy, it controls the front-end and the back-end. It is responsible for reading the user input, i.e. the name of a file to be compiled and the desired options, for checking whether the file exists, for initializing all modules and starting compilation.

4.3 Parametrization of the Front-End

As written before, the front-end is machine-independent so that modules OPS, OPT, OPB and OPP don't need any modifications (excepted if new procedures have to be inserted in the module SYSTEM, or useless ones to be removed), when the compiler has to generate code for a different target machine. Only some implementation dependent parameters must be changed in module OPM:

• basic type sizes

These constants give the number of bytes required to store an element of the basic type.

• target machine bound values of basic type expressed in host machine format

see *Host Interface*.

• value of constant NIL

nilval is zero on most machines.

• maximal number of elements per dimension in the declaration of an array

MaxIndex is an integer constant.

• parametrization of the numeric scanner

These values are used while scanning numbers to report errors when limits are exceeded. *MaxExp* is the maximal real number exponent. *MaxDig* is the maximal length of a real number fractional part. The provided values are usually those of the IEEE Standard. *MaxHexDig* is the maximal number of digits in an hexadecimal integer number.

• inclusive range of parameter of standard procedure HALT

If the halt number doesn't lie between *MinHaltNr* and *MaxHaltNr*, the front-end reports an error.

• inclusive range of register number used as parameter in the procedure SYSTEM.GETREG and

SYSTEM.PUTREG
The interpretation of the number is left to the back-end, but the front-end reports an error if this number doesn't lie between *MinRegNr* and *MaxRegNr*.

• maximal value of the parameter of the procedure SYSTEM.CC
If the integer constant parameter doesn't lie between *0* and *MaxCC*, the front-end reports an error. The interpretation of the number (mapping to a condition of the condition code register) is left to the back-end.

• maximal value of the structure system flag
The flag used to mark interface structures (see *Interfacing with an Existing Operating System*) must lie between *0* and *MaxSysFlag*.

• initialization of *linkadr* field in *ObjDesc*
The *linkadr* object field of imported procedures is initialized with *LANotAlloc*. This integer constant must be different from any valid link address, so that the back-end can recognize the already linked procedures.

• initialization of constant allocation address
Strings or real constants may be allocated in a constant frame (the "load immediate" instruction for real constants may be unavailable or more expensive than a "load memory" instruction on RISC processors). The object field *adr* of strings or real constants is initialized with *ConstNotAlloc*, which is a integer constant that must be different from any valid address.

• initialization of *adr* field in *StrDesc*
The *adr* field of each structure representing a user defined type is initialized with *TypAdrUndef*. In case of a predefined basic type, *adr* is always set to 0.

• maximal number of cases
MaxCases limits the number of "CaseLabels" in a case statement (a "CaseLabel" is a pair of values defining a range or a unique value).

• maximal range of a case statement
MaxCaseRange limits the interval between the least and largest label in a case statement (this corresponds to the maximal length of the jump table).

• maximal number of exit statements
MaxExit limits the number of exit statements in a loop statement, including all exits of nested loops.

• hidden pointer field export
The offset of a not exported (opaque) field of an exported record type is generally not present in the symbol file. But the offset of an opaque pointer field may be nevertheless written in the

symbol file, according to the boolean constant *ExpHdPtrFld*. This information is required to build a correct type descriptor of an extension of this type at compile-time (the garbage collector must trace this opaque pointer field). But if the type descriptors are constructed at load-time, this information is not necessary.

• hidden procedure field export
The offset of an opaque procedure variable field of an exported record type may be nevertheless written in the symbol file, according to the boolean constant *ExpHdProcFld*. This allows a safe implementation of a command for module unloading (e.g. System.Free [Wirth 88]). As a matter of fact, a module can only be safely unloaded, if no procedure of this module is installed in an other module remaining loaded.

• reset of field *leaf* if NEW procedure used
The boolean constant *NEWusingAdr* indicates whether the field leaf of a pointer variable p (mode = Var) has to be set to FALSE, when NEW(p) or SYSTEM.NEW(p, n) is called (see *Simple Optimizations*).

Because of its very nature, the module SYSTEM, which is handled in a special way by the compiler, cannot be portable. It is however possible to parametrize some of its procedures (GETREG, PUTREG) like shown here above, without modifying front-end modules. The implementation of these SYSTEM procedures being done in the back-end, their interpretation can be changed without affecting the syntax and hence the front-end. However, if some of these procedures become meaningless for a specific target machine (e.g. function CC on a machine without condition code), they should be removed, or if new ones are needed, the front-end must be modified. Due to the well defined structure of the compiler, this correction is easy to do and we will now briefly explain how to proceed.

The SYSTEM procedures are predefined in the symbol table and coded in the syntax tree. The function procedures and proper procedures of module SYSTEM are handled exactly in the same way as standard procedures and functions of the language: to each procedure or function to be defined corresponds a call to the procedure *EnterProc(name, fn)* in the initialization part of the module OPT, where *name* is a string representing the procedure name (e.g. "LSH") and where *fn* is a number identifying this procedure in the compiler. In the syntax tree, three kinds of nodes are involved: *Nmop* and *Ndop* to represent functions and *Nassign* for procedures. The subclass of the *Nassign* node is the function number itself, but subclasses of *Nmop* and *Ndop* nodes are operator numbers, that identify the operation to perform (addition, type conversion, absolute value, address, rotation, ...).

To insert a new procedure, we must first choose a new function number, which must be greater than all existing ones and make a call to the procedure *OPT.EnterProc* with the new name and this new number. Additional code to check the parameter types and number, and build the nodes must be inserted in the procedures *OPB.StPar0*, *OPB.StPar1*, *OPB.StParN* and *OPB.StFct*. Then, in case of a function procedure, a new operator number has to be chosen and code must be inserted in *OPB.MOp* and *OPB.Op*.

The module OPV must be modified to be able to recognize these new node subclasses corresponding to the new SYSTEM procedures and to make the appropriate calls to new OPC procedures to implement them.

If a SYSTEM procedure has to be removed, the corresponding call to *EnterProc* must be suppressed in module OPT, as well as superfluous code in modules OPB, OPV and OPC.

4.4 Object File Format

Compiled Oberon modules have the form of an object file and are not stand-alone applications. They can only be executed in an Oberon specific environment [Wirth 88]. This run-time infrastructure consists among other things of a command interpreter, a module loader/linker, a memory allocation system, a garbage collector, and a trap handler. Before presenting a possible object file format, we must first pay attention to some run-time requirements specific to the language Oberon.

Dynamic memory is allocated in the heap by the procedure NEW, but no explicit deallocation procedure is provided in Oberon for safety and convenience. A garbage collector must be therefore programmed and embedded in the run-time system. The conventional mark-scan principle is used in the Ceres implementation. The garbage collector operates only between commands. At these moments, the stack is guaranteed to be void, since no local variables exist. In the mark phase, the entire dynamically allocated structure must be traversed and marked, without requiring further storage space.

The roots of this structure are anchored in global variables in form of pointers, whose offsets must be present in the object file, so that the garbage collector knows where to start its mark phase. But more information is needed in order to traverse the structure, namely, the offsets of pointer fields of each traced block (record or array). During the scan phase, the size of each block must be known as well. For this purpose, each block receives a tag, which is a hidden pointer (offset -4) that points to a type descriptor containing these pieces of needed information.

Each declared type gets, in addition to such a descriptor, a tag allocated in constant frame that is pointing to the type descriptor. This tag serves as anchor to retrieve the descriptor of a given type; at compile-time, only the relative address of the tag in the constant frame is known but not the address of the descriptor itself.

Instead of reserving space in each block for a counter that remembers during the mark phase which pointer comes next to be traversed (in this case, the counter length limits the number of pointers in a record), it is also possible to shift the tag over the pointer offsets in the type descriptor, the last offset being a negative one resetting the tag to its initial value.

Type descriptors are also used for another purpose: run-time type-tests and type-guards. In addition to the block size and to a list of pointer offsets, the descriptors contains the whole "ancestry" of the described type, i.e. a pointer table, where the i-th pointer points to the type descriptor of the i-th generation type. This table allows type checks to perform in constant time and its fix length makes an index check unnecessary: in the test *v IS T* or guard *v(T)*, only the i-th pointer in the type descriptor of variable *v* will be compared with the tag of type *T*, where *i* is the generation level of *T*, which is known at compile-time. The pointer to the "generation zero" type, which is not an extension of any other type, is not stored in the type descriptor because no run-time type checks referring to this generation will be emitted by the compiler. Hence a table length of n suffices for an extension hierarchy of height n+1 (n equals 7 in the Ceres implementation).

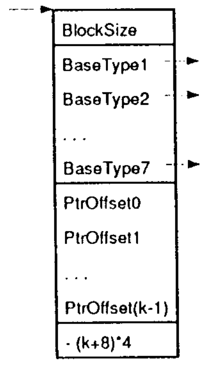


Fig. 13 Possible implementation of a type descriptor

Type descriptors have to be listed in the object file. At load-time, they are allocated in the heap (or in the constant frame) and their "ancestry" table is initialized, normally by explicit code inserted by the compiler in the module initialization part (if descriptors are allocated in the constant frame, this initialization can be done by the absolute linker, if any).

A possible object file format is presented here. A file consists of ten logical blocks:

- The header block contains the module name and key, the different sizes of the following blocks, and the total size of global variables.

- The entry block is a list of offsets denoting procedure entry points in the code.

- The command block gives the name and entry point offset of command procedures.

- The pointer block lists the offsets of global pointers allowing the garbage collector to start its mark phase.

- The import block lists the name and key of imported modules.

- The link block contains a list of pairs consisting of a module number and of an entry number that are used to build the link table. If an absolute linker is used, an offset denoting the beginning of a fix-up chain is provided with each pair. In this case, no link table is used at run-time but the code is patched along the fix-up chain by the linker.

- The data block contains all constant data like strings, case jump tables, or type descriptor anchors (these last ones have to be patched, when allocating the type descriptors).

- The code block contains the actual executable code.

- The type block lists all type descriptors, which must be allocated and initialized.

- The reference block contains debugging information (see *Debugging*).

4.5 Storage Allocation and Code Generation

The first compilation phase, including syntax analysis, type checking and construction of the intermediate representation, is controlled by the parser module OPP. OP2 gives control to the module OPV in the second phase, i.e. storage allocation and code generation.

This module first traverses the symbol table and distributes addresses to variables, offsets to record fields, sizes to types, entry numbers to external procedures, calculates procedure frame sizes (parameters and local variables), and allocates type descriptors and/or constant type tags. All this is done in three steps by the procedure *AdrAndSize*, which is exported from OPV; the first step treats only exported types and procedures in alphabetical order, as well as their parameters and whose types. Then, in a second step, the global variables are treated in the declaration order. Lastly, the third step handles remaining types and procedures, i.e. those who are not exported, as well as local variables, types and procedures of local scopes. This handling in three steps guarantees that no new symbol file will be created if not necessary, even if new (not exported, of course) procedures or types are inserted or if the declaration order of existing ones (maybe exported) is changed.

The programmer must only take care that exported variables be declared before local ones, so that a modification of not exported variables doesn't imply the creation of a new symbol file.

The procedure *AdrAndSize* and its satellite procedures can be freely modified, if a different storage allocation scheme is desired. One of these satellite procedures, called *TypSize*, calculates the size of the type, whose representing structure node is given as parameter. This procedure is exported and installed by OP2 in the module OPB, so that the front-end can call it to immediately evaluate the standard function SIZE(T).
After storage allocation, code generation takes place. The procedure *CompilationUnit*, another exported procedure of OPV, recursively traverses the syntax tree and calls procedures of underlying modules, that take and/or return "items" as parameters, and that produce code as side effect. An item is a record representing an operand of an operation. It indicates where the operand can be found. Items make it possible to delay emission of code, so that processor addressing modes can be optimally used. The addressing mode is specified by an item tag field; other attributes like type, address, value of constant are stored in item fields as well. Some of these fields are common for every processor architecture and can already be initialized in OPV.

The predefined fields and predefined modes as listed below:

```
TYPE
    ItemBase = RECORD
    mode, mnolev: SHORTINT;
    typ: OPT.Struct
END ;

CONST
    (* item base modes (=object modes) *)
    Var = 1; VarPar = 2; Con = 3; LProc = 6; XProc = 7; CProc = 9; IProc = 10;
```

But it is necessary to extend this base record by extra fields and modes according to the specific target machine architecture. It is possible to do this using the Oberon type extension construct and creating a new record type named *Item* that extends *ItemBase*, but new fields may be directly inserted in the base item as well. In the latter case, the *ItemBase* record has to be renamed to *Item*. New modes simply correspond to new declared constants.

The initialization of mandatory fields are done in OPV, when an item is newly created, i.e. when a leaf node of class *Nvar*, *Nvarpar*, *Nconst* or *Nproc* is encountered while traversing the tree. Module OPV then calls the procedure *OPC.CompleteItem* with the incomplete item and the leaf node as parameter, so that extra fields can be initialized too (for efficiency reasons, this procedure can be "dissolved" into OPV).

If structured statements such as if, while, repeat, loop or case have to be translated using conventional schemes, *OPV.CompilationUnit* and its satellite procedures don't need any modification. One doesn't even need to know how the syntax tree looks like; OPV traverses it and calls the appropriate procedures of OPC and OPL modules. But this is perhaps not always possible, and sometimes, OPV procedures must be modified; for example, if the branch instruction delay slot of a RISC processor must be optimally used.

While module OPV is almost machine-independent, module OPC is not. Most procedures of this module have to be rewritten, but the interface to OPV, i.e. procedures signatures, usually remains unchanged. These procedures select target machine instructions to be emitted and then call procedures of module OPL to synthesize them. Thus the interface of OPL to OPC is machine-dependent. While addressing modes are mirrored in item modes, instruction formats are reflected by procedure signatures of OPL. The back-end programmer is free to organize module OPL (e.g. constant frame, code frame, register allocation, ...) and to write these procedures and name them as he likes.

Generally, he will declare two global arrays of bytes, one for the code frame and one for the constant frame. The index in the code frame must be named *pc* and be exported, because it is used in OPV. An other mandatory variable is *level*, which is incremented by OPV when entering a procedure and decreased when leaving it. This variable is needed by OPL to follow the static link, when generating code for accessing intermediate level variables. Two further variables must be declared in OPL: *entno* (number of entries, maximum is *MaxEntry*) and *dsize* (total size of global variables). They are set by OPV during storage allocation and their value will be written in the object file.

In addition to items, OPL must define and export two other opaque types: first *RegSet*, which reflects the occupancy of the register set (*RegSet* is normally a record type; a variable of this type is returned from the procedure *OPL.SaveRegisters* to OPV and passed later to the procedure *OPL.RestoreRegister*), and then *Label*, which designates an address in the code frame (*Label* is normally an INTEGER or LONGINT; a variable of this type is returned by branch procedures when the target address is not yet known and must be patched later; the variable is given as parameter to a fix-up procedure.

Some special procedures that are not synthesizing code are mandatory, because there are called from OPV or OP2. Such a procedure is *OPL.OutCode*, which must write out an object file containing the code frame, constant frame and all needed pieces of information to load, link and execute the module (see *Object File Format*). For example, this procedure has to collect all commands and global pointer offsets. It can do that by traversing the global scope. The type descriptors have to be also listed, if they are not already allocated in the constant frame. Basically, it would be necessary to traverse all local scopes to also collect the local declared types. But this can be avoided, if the procedure *OPL.AllocTypDesc*, which is called for each record or array type, stores the type anchors in a global table.

Contrary to the module OPL, the module OPC has no global state (i.e. no global variables, except the compiler options) and consists only of a collection of procedures. This module usually becomes very large; in our back-end for the Ceres workstation, we have split it into two modules (OPC and OPCa).

In addition to exported procedures, whose signatures are known in OPV, the back-end programmer may write some auxiliary procedures called at many locations in the module, e.g. a procedure to load an item or its effective address into a register.

Some procedures are not easy to implement. *OPC.TypTest* must generate code for three different type-test forms according to the boolean parameters *guard* and *equal*. In the "IS-test" (*guard* = FALSE), the i-th base type tag (see *Object File Format*) in the type descriptor of the item must be compared with the tag of the given type *testtyp*, which has extension level *i*. The test is always emitted and the result is described by the item var-parameter. Another form is the "guard-test" (*guard* = TRUE, *equal* = FALSE); the same test as before is generated (only if the type check compiler option was chosen), but code for checking the result is also emitted, so that a trap is called, if the type doesn't match. The item remains unchanged. The last form is the "implied guard-test" (*guard* = TRUE, *equal* = TRUE), which is only generated if the compiler option allows it. Here, the type tag of the item must be equal to the tag of the type *testtyp*, else a trap is called. The item remains unchanged. This last form of type-test is not visible in the source text, but is explicitly performed at the time of each record assignment. It make sure that the dynamic and static types of the destination record are equal. Note that for each form, if *testtyp* is a pointer type, it must be first dereferenced.

Oberon supports multidimensional open arrays as procedure formal parameters. Since the size of the corresponding actual parameter is not known at compile-time, but only the number of dimensions is, it is not possible to reserve space in the procedure parameter frame. We can instead reserve space for an open array descriptor consisting of the effective address of the array and a dimension length vector. Each element of that vector indicates (at run-time) the number of elements in the considered dimension. When an array is passed as parameter, a corresponding descriptor is built and pushed onto the stack. If the formal parameter is a value-parameter, the array has to be copied onto the stack in the called procedure and the new address is written in the descriptor (procedure *OPC.CopyDynArr*).

Special care must be taken, while records are passed as actual parameters to formal var-parameters. In addition to the effective address of the actual parameter, a type tag pointing to the type descriptor of the actual parameter type must be passed too, so that type-tests can be implemented. Three different cases must be treated separately; first, if the actual parameter is a var-parameter, then its tag is copied ; if the actual parameter is a dereferenced pointer (parameter *deref* of *OPC.Param* is TRUE in this case), then the type tag of the dynamically allocated block is copied; lastly, if the record is a local or global variable, then the type tag of its type (known at compile-time) is passed.

4.6 Interfacing with an Existing Operating System

We will not explain here the interface between the Oberon run-time system and the underlying operating system, but the possibilities offered to the Oberon programmer of writing modules that are able to communicate directly with the operating system, in so far as these possibilities have been implemented in the compiler. We emphasize that these facilities are not part of the language Oberon, but are local extensions specific to this compiler.

First of all, a mechanism for supervisor calls or traps to the operating system routines must be provided. The chosen approach is very flexible; the programmer can define "code procedures" using the following syntax (to be inserted in Oberon syntax [Wirth 89]):

CodeProcedureDeclaration = PROCEDURE "-" identdef [FormalParameters] n {"," n} ";".

n is an integer constant ($0 \leq n < 256$) and this procedure has no body.

When a code procedure is called, actual parameters are first pushed onto the stack like parameters of normal procedures, and the constants are then directly copied as in-line code in place of the usual subroutine call instruction. These constants represent normally code for a supervisor call. But this scheme may be freely modified by the back-end programmer, so that actual parameters are pushed using a different alignment for example, or/and so that constant code is always emitted and so that these numbers consequently becomes only parameters (e.g. trap number) of a fix code sequence that must not be always repeated in the number list anymore.

The module SYSTEM provides some low-level procedures to read/write from/to registers (parameters of operating system routines) or absolute memory locations (memory-mapped hardware controllers) or to directly write in-line code. Another low-level facility is offered by interrupt procedures: a "+" preceding the name of a (normally parameterless) procedure specifies it to be an interrupt procedure (mode = IProc), that can be installed as device handler. The back-end programmer can foresee special entry code (e.g. registers saving) and exit code (e.g. registers restoring and "return from interrupt" op-codes).

Some declared types used for parameters of operating system routines need perhaps a different alignment, or must not be traced during garbage collection, or needs double dereferencing, or ... For these reasons and other, the record, array or pointer types may be marked by the programmer at the time of their declaration by inserting a "sysflag" immediately after the keywords RECORD, ARRAY or POINTER. The syntax is:

sysflag = "[" n "]".

where n is an integer constant between 0 and *OPM.MaxSysFlag*.

This number is copied in the field *sysflag* of the *Struct* record; if sysflag is absent in a record declaration, the sysflag value of the base record type (if any, else 0) is copied in the extended record type; if sysflag is absent in an array or pointer declaration, the field *sysflag* is set to 0. The interpretation of this flag is left to the back-end.

4.7 Debugging

Experience has shown that efficient programming work can be also performed without a powerful and hence expensive run-time or post-mortem debugger, let alone a symbolic tracer. A simple trap-handler displaying the state of the procedure stack suffices in most cases (else the programmer should insert halt statements and run again the program to find the bug).

We will explain here how necessary information has to be generated by the compiler, so that the trap-handler can decode and write the contents of the procedure stack in a readable form for the programmer. These pieces of information are normally written in a special file called "reference file", that may be appended to the object file.

When a trap occurs, the handler must first recognize the procedure in which the trap has been called. It knows only the program counter value at the location of the trap call. Thus, the compiler must generate a list of pairs consisting of procedure name and pc value. This value is always relative to the beginning of the code frame.

Using this list, the handler can find the name of the procedure in which the trap occurred. It can then follow the dynamic link in the procedure stack, read each time the return address and write the corresponding procedure name of the caller.

If the compiler also generates information to describe local variables, i.e. their name, offset, type and whether value or var-parameter, then, their value can be read from the stack and be displayed with their name.

Three procedures of OPL write the needed information in the (temporary) reference file, that can be previously opened in *OPL.Init*. These procedures are called from OPV after each generated procedure code block. *OPL.OutRefPoint* writes the pc value, *OPL.OutRefName* writes the name given as parameter and *OPL.OutRefs* traverses the procedure scope given as parameter and writes all local variable names (using *OPL.OutRefName*), offset, object mode and type form (only if basic).

If a more powerful debugger is nevertheless desired, then the compiler can generate more information using the same principle, so that dynamically allocated structures can be traced and displayed too.

References and Further Reading

[Aho 86]
Aho A. V., Sethi R., Ullman J. D.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley

[Aho 89]
Aho A. V., Ganapathi M., Tjiang S. W. K.
Code Generation Using Tree Matching and Dynamic Programming.
ACM Transactions on Programming Languages and Systems 11:4, 491-516

[Amman 74]
Amman U., Jensen K., Nägeli H., Nori K.
The Pascal 'P' Compiler: Implementation Notes.
Departement Informatik, ETH Zürich

[Cattell 79]
Cattell R. G. G., Newcomer J. M., Leverett B. W.
Code Generation in a Machine-Independent Compiler.
ACM SIGPLAN Notices 14:8, 65-75

[Eberle 87]
Eberle H.
Development and Analysis of a Workstation Computer.
Ph.D. Thesis, ETH Zürich

[Fraser 86]
Fraser C. W., Wendt A.
Integrating Code Generation and Optimization.
ACM SIGPLAN Notices 21:6, 242-248

[Ganapathi 82]
Ganapathi M., Fischer C. N., Hennessy J. L.
Retargetable Compiler Code Generation.
Computing Surveys 14:4, 573-592

[Glanville 78]
Glanville R. S., Graham S. L.
A New Method for Compiler Code Generation.
Fifth ACM Symposium on Principles of Programming Languages, 231-240

[Gutknecht 89]
Gutknecht J.
The Oberon Guide.
Report 119, Departement Informatik, ETH Zürich

[Heiz 88]
Heiz W.
Modula-2 auf einem RISC: Realisierung und Vergleich.
Ph.D. Thesis, ETH Zürich

[Powell 84]
Powell M. L.
A Portable Optimizing Compiler for Modula-2
ACM SIGPLAN Notices 19:6, 310-314

[Tanenbaum 89]
Tanenbaum A. S., Kaashoek M. F., Langendoen K. G., Jacobs C. J. H.
The Design of Very Fast Portable Compilers.
ACM SIGPLAN Notices 24:11, 125-131

[Wirth 86]
Wirth N.
Compilerbau, Eine Einführung.
B. G. Teubner Stuttgart

[Wirth 88]
Wirth N., Gutknecht J.
The Oberon System.
Report 88, Departement Informatik, ETH Zürich

[Wirth 89]
Wirth N.
The Programming Language Oberon (Revised Report).
Report 111, Departement Informatik, ETH Zürich

Appendix: Syntax Tree Definition

The syntax tree generated by the front-end conforms to a syntax described in this appendix. The back-end relies on this syntax when traversing the tree.

Each node of the syntax tree is identified by its field *class* which is a short integer constant whose name begins with a "N". Nodes of the classes *Nmop* (unary operator), *Ndop* (binary operator) and *Nassign* (assignment) have also a subclass denoted by the field *subcl*. There are different class groups, each one consisting of one or more classes.

Every syntactically correct Oberon construct is mapped into a subtree. We say that a subtree belongs to a group if the class of the root node of this subtree belongs to this group. Subtrees representing Oberon constructs of the same nature belongs to the same group. For example, the group *stat* consists of the classes that may appear in the root node of a subtree representing an Oberon statement. There are three major groups: *design* for designators, *expr* for expressions and *stat* for statements.

NIL stands for the empty node (not allocated). The following productions show how classes belong to groups (bold face) and how groups are nested:

design	= Nvar | Nvarpar | Nfield | Nderef | Nindex | Nguard | Neguard | Ntype | Nproc.
expr	= **design** | Nconst | Nupto | Nmop | Ndop | Ncall.
nextexpr	= *NIL* | **expr**.
ifstat	= *NIL* | Nif.
casestat	= Ncaselse.
sglcase	= *NIL* | Ncasedo.
stat	= *NIL* | Ninittd | Nenter | Nassign | Ncall | Nifelse | Ncase | Nwhile | Nrepeat | Noop | Nexit | Nreturn | Nwith | Ntrap.

According to its class, a node can have a left and a right subtree and possibly a linked successor. But the subtrees of a given node must belong to one group as defined in the table on the next page.

The field *typ* is not listed in the table, but is nevertheless always valid. The field *conval* is only valid if specified by a comment between parenthesis. If nothing appears for a given field in a table line, then this field is not defined and should not be read.

<u>group</u>	<u>class</u>	<u>subcl</u>	<u>obj</u>	<u>left</u>	<u>right</u>	<u>link</u>	
design	*Nvar*		*var*			*nextexpr*	
	Nvarpar		*varpar*			*nextexpr*	
	Nfield		*field*	*design*		*nextexpr*	
	Nderef			*design*		*nextexpr*	
	Nindex			*design*	*expr*	*nextexpr*	
	Nguard		*gdtype*	*design*		*nextexpr*	
	Neguard		*gdtype*	*design*		*nextexpr*	
	Ntype		*type*			*nextexpr*	
	Nproc		*proc*			*nextexpr*	
expr	*design*						
	Nconst		*const*				*(conval=value)*
	Nupto			*expr*	*expr*	*nextexpr*	
	Nmop	*not*		*expr*		*nextexpr*	
		minus		*expr*		*nextexpr*	
		is	*tsttype*	*expr*		*nextexpr*	
		conv		*expr*		*nextexpr*	
		abs		*expr*		*nextexpr*	
		cap		*expr*		*nextexpr*	
		odd		*expr*		*nextexpr*	
		adr		*expr*		*nextexpr*	*SYSTEM.ADR*
		cc		*Nconst*		*nextexpr*	*SYSTEM.CC*
	Ndop	*times*		*expr*	*expr*	*nextexpr*	
		div		*expr*	*expr*	*nextexpr*	
		mod		*expr*	*expr*	*nextexpr*	
		and		*expr*	*expr*	*nextexpr*	
		plus		*expr*	*expr*	*nextexpr*	
		minus		*expr*	*expr*	*nextexpr*	
		or		*expr*	*expr*	*nextexpr*	
		eql		*expr*	*expr*	*nextexpr*	
		neq		*expr*	*expr*	*nextexpr*	
		lss		*expr*	*expr*	*nextexpr*	
		leq		*expr*	*expr*	*nextexpr*	
		grt		*expr*	*expr*	*nextexpr*	
		geq		*expr*	*expr*	*nextexpr*	
		in		*expr*	*expr*	*nextexpr*	
		ash		*expr*	*expr*	*nextexpr*	
		msk		*expr*	*Nconst*	*nextexpr*	
		len		*design*	*Nconst*	*nextexpr*	
		bit		*expr*	*expr*	*nextexpr*	*SYSTEM.BIT*
		lsh		*expr*	*expr*	*nextexpr*	*SYSTEM.LSH*
		rot		*expr*	*expr*	*nextexpr*	*SYSTEM.ROT*
	Ncall		*fpar*	*design*	*nextexpr*	*nextexpr*	
nextexpr	*NIL*						
	expr						

group	class	subcl	obj	left	right	link	(continued)
ifstat	*NIL*						
	Nif			*expr*	*stat*	*ifstat*	
casestat	*Ncaselse*				*sglcase*	*stat*	*(conval=min,max)*
sglcase	*NIL*						
	Ncasedo			*Nconst*	*stat*	*sglcase*	
stat	*NIL*						
	Ninittd					*stat*	*(typ=descr)*
	Nenter		*proc*	*stat*	*stat*	*stat*	*(obj=NIL if module)*
	Nassign	*assign*		*design*	*expr*	*stat*	
		newfn		*design*		*stat*	
		incfn		*design*	*expr*	*stat*	
		decfn		*design*	*expr*	*stat*	
		inclfn		*design*	*expr*	*stat*	
		exclfn		*design*	*expr*	*stat*	
		copyfn		*design*	*expr*	*stat*	
		getfn		*design*	*expr*	*stat*	*SYSTEM.GET*
		putfn		*expr*	*expr*	*stat*	*SYSTEM.PUT*
		getrfn		*design*	*Nconst*	*stat*	*SYSTEM.GETREG*
		putrfn		*Nconst*	*expr*	*stat*	*SYSTEM.PUTREG*
		sysnewfn		*design*	*expr*	*stat*	*SYSTEM.NEW*
		movefn		*design*	*expr*	*stat*	*SYSTEM.MOVE*
	Ncall		*fpar*	*design*	*nextexpr*	*stat*	
	Nifelse			*ifstat*	*stat*	*stat*	
	Ncase			*expr*	*casestat*	*stat*	
	Nwhile			*expr*	*stat*	*stat*	
	Nrepeat			*stat*	*expr*	*stat*	
	Nloop			*stat*		*stat*	
	Nexit					*stat*	
	Nreturn		*proc*	*nextexpr*		*stat*	*(obj=NIL if module)*
	Nwith			*Nguard*	*stat*	*stat*	
	Ntrap				*Nconst*	*stat*	

The field *conval^.intval* of each node belonging to the *stat* group gives the end position of the statement in the source text.

The tree is traversed and evaluated (evaluation means here code generation) by the module OPV. To each group corresponds a procedure having the same name as the group itself. This procedure takes a *Node* as parameter (the root node of the subtree to be traversed and evaluated) and possibly returns an *Item* (denoting the evaluation result of this subtree). First, the procedure calls procedures to traverse (and evaluate) the left and right subtrees according to the class of the *Node* parameter and then evaluate the *Node* parameter itself.

An excerpt of the module OPV is listed below:

```
PROCEDURE^ expr(n: OPT.Node; VAR x: OPL.Item);     (* forward declaration *)


PROCEDURE design(n: OPT.Node; VAR x: OPL.Item);
   VAR y: OPL.Item;
BEGIN
   CASE n^.class OF
   ...
   | Nderef:   ...
   | Nindex:   design(n^.left, x); expr(n^.right, y); OPC.Index(x, y)    (* x := x[y] *)
   | Nguard:   ...
   ...
   END ;
   x.typ := n^.typ
END design;


PROCEDURE expr(n: OPT.Node; VAR x: OPL.Item);
   VAR y: OPL.Item;
BEGIN
   CASE n^.class OF
   ...
   | Ndop:     expr(n^.left, x); ...  expr(n^.right, y);
               CASE n^.subcl OF
               ...
               | mod:  OPC.Mod(x, y)    (* x := x MOD y *)
               ...
               END
   ...
   END
END expr;


PROCEDURE stat(n: OPT.Node);
   VAR x: OPL.Item;
      L0, L1: OPL.Label;
BEGIN
   WHILE n # NIL DO ...
      CASE n^.class OF
      ...
      | Nwhile:   L0 := OPL.pc; expr(n^.left, x);
                  OPC.CFJ(x, L1);              (* if not x then jump to L1*)
                  stat(n^.right); OPC.BJ(L0);  (* backwards jump to L0*)
                  OPL.FixLink(L1)              (* fix-up L1 with current pc *)
      ...
      END ;
      n := n^.link
   END
END stat;
```

Gelbe Berichte des Departements Informatik

114	M. Müller	CeNet: AppleTalk - kompatible Kommunikations-Software für CERES
115	W. Gander G.H. Golub D. Gruntz	Solving Linear Equations by Extrapolation
116	B. Sanders	Stepwise Refinement of Mixed Specifications of Concurrent Programs
117	N. Wirth	Modula-2 and Object-Oriented Programming. Drawing Lines, Circles, and Ellipsis in a Raster. Flintstone.
118	H.-J. Schek H.-B Paul M.H. Scholl G. Weikum	The DASDBS Project: Objectives, Experiences, and Future Prospects
119	J. Gutknecht	The Oberon Guide
120	D. Mey	A Predicate Calculus with Control of Derivations
121	H.P. Frei P. Schäuble M.F. Wyle	The Assessment of Information Retrieval Algorithms
122	P. Läuchli	An Elementary Theory for Planar Graphs
123	B. Wüthrich	Detecting Inconsistencies in Deductive Databases
124	C. Pfister	The Graphics Editor Condor The Layout System Pedro
125	R. Crelier	OP2: A Portable Oberon Compiler