A Portable Oberon Compiler

Erich Schikuta
Franz Posch
Institute of Statistics and Computer Science
University of Vienna

ABSTRACT

At the University of Vienna a compiler for the Oberon programming language ([Wirt89]) was developed. The goal of this project was to reach a high degree of portability to run the compiler on different systems available at the campus. Therefore the UNIX*) operating system with the C programming language and the tools LEX and YACC were chosen. The syntax definition of Oberon prohibits an easy one-symbol look--ahead analysis with YACC. This problem was solved by a special approach during the lexical analysis. An interpreter is part of the system. It simulates a virtual 0-address machine, which is the basis of the generated target code.

THE OBERON LANGUAGE

The Oberon programming language was developed by N. Wirth as a successor to MODULA-2. Oberon is a small but powerful language. Several features well known from Pascal and MODULA-2 were omitted, like variant records, opaque types, enumeration types, subrange types, local modules, main program - procedure distinction, WITH-statement, FOR-statement and coroutines. At the opposite new features introduced in Oberon are type extension and type inclusion.

The type extension allows the construction of new data types on the basis of existing ones. Values of an extended data type are compatible to the original type and can be assigned to variables of the basis type. This is an advantage to MODULA-2, because now it is possible to create abstract data structures administrating elements of different data types. For example the type

Y = RECORD length, width: REAL END;

can be extended by a further attribute 'height',

X = RECORD (Y) height: REAL END;

X is a direct extension of Y. This extension can be extended again.

A type X is an extension of type Y, if X = Y or X is a direct extension of an extension of Y. Conversely Y is a base type of X, if Y = X or Y is the direct base type of a base type of X.

The type inclusion allows to assign a value to variables of a different type, if one type includes the other type. A type X includes another type Y, if the values of Y are values of X too. That means

LONGREAL > REAL > LONGINT > INTEGER > SHORTINT

where '>' stands for 'includes'.

A complete description of the Oberon language can be found in ([Wirt89]).

THE PORTABLE OBERON COMPILER

The goal of the project was to develop a portable Oberon compiler ([ScPo89]). This approach is different in comparison with the existing implementation at the ETH Zrich. There is only one processor type (a NS32000) supported and the language is embedded into the proprietary Oberon operating system ([GuWi89]).

Basis for the presented implementation at the University of Vienna was the UNIX operating system with the C programming language and the tools LEX and YACC. LEX was used for the development of the lexical analyzer and the YACC for the parser and the code generator.

The compiler was constructed as a one--pass compiler ([AhSe86]). All phases of the

*) UNIX is a registered trademark of Bell Laboratories

translation, from the analysis to the code generation, are performed by one pass through the Oberon program. The only exception is the calculation and insertion of addresses of forward jumps into the code.

THE PARSER

The Lexical Analysis

An Oberon program consists of a stream of characters. The smallest unit of characters having a collective meaning is called a token. A token can be an identifier, a constant, an operator, a reserved word etc. A complete Oberon program is built up of tokens. The lexical analysis translates the input character stream into a list of tokens.

Syntax Analysis

The syntax analysis groups the list of tokens into grammatical phrases, checks the correctness and produces a parse tree for the code generation.

The Ambiguity of Oberon

The syntax of Oberon is defined by an extended Backus-Naur form, called EBNF. Thereby an ambiguity is apparent, which prevents a simple context-free language recognition by a one- symbol look-ahead.

A procedure is called by a designator and an eventually following actual parameter list. Further a designator can be accompanied by a type guard in brackets.

Respective EBNF definition:

```
ProcedureCall = designator [ActualParameters]
    designator = qualident { "." ident ¦ "[" ExpList "]" ¦ "("
    qualident ")" ¦ "^" }
    qualident = [ ident "." ] ident
    ident = letter { letter ¦ digit }
```

These definitions allow the following syntactically correct constructs:

```
tree(CenterNode).subnode -> designator followed by a type guard
Tree(root) -> procedure call with actual parameters
```

Therefore it is not possible to decide by a one-symbol look- ahead, if a qualified identifier in brackets is a type guard or an actual parameter.

For YACC this situation leads to the commonly known shift/reduce conflict. Automatically the parser created by YACC would shift and try to accept the longest possible input stream. A possibly following qualifier in brackets would be erroneously accepted as type guard, and the actual parameter list would not be identified.

A possibility could be to terminate the type constructor list by a unique symbol (e.g. '{' and '}'), which is not allowed to appear in a designator. This would be a clean solution, but would lead to a change of the Oberon syntax.

Therefore this problem is solved using a little "trick", but keeping the original Oberon definition.

LEX gives the possibility to do a look-ahead over more than one symbol. Because of this an expression in brackets can be analyzed before the symbol is passed to YACC. A opening bracket followed by an identifier is recognized as a (only syntactical) designator. This information is passed to YACC by the special meta symbol "LP" and marks the indefinite input. In all other situations the token for the opening bracket is transferred. Therefore The symbol "LP" allows to process the input syntactically correctly. The following semantic analysis has to check, if the qualifier in brackets stands for type guard or for an actual parameter list.

The Semantic Analysis

The semantic analysis checks that the components of the program fit together, for example, that a variable has to be defined before usage, the data type of an expression is included by the data type of the assignment variable, etc..

Oberon contains many type constructors

to create complex data structures. For the semantic analysis of these type extensions a symbol table, a type tree and a type table of the type constructs was necessary.

The symbol table stores the user defined constants, variables, procedure names and module names. An entry comprises Name, category (constant, variable, ...), type, static nesting level, address and value of the object. The symbol table is organized as a stack. With the entrance into a procedure the local objects are put on the top of the stack and with the exit they are poped (scope).

A user defined data type is represented by the construction of a binary type tree. The structure of the type tree shows the synthesis of the type out of type constructors, basis types and type names. A tree node contains the type information (basis type, type constructor and for arrays the number of elements), the size of the type and a pointer to the left and the right successor, which describe further type components. It is easy to describe recursive data structures by a type tree.

The type table stores the predefined and user defined program types. An entry consists of the pointer to a type tree, the static nesting level of the type declaration (scope), the memory size and an optional type name. New type trees are only inserted into the type table, if no structural equivalent data structure exist. The type table is realized as a stack too.

For typed programming languages the semantic analysis is the most complex part of the program translation. No general formalisms exist for the semantics notation. The description is mostly very informal.

CODE GENERATION

The rest of the paper concentrates on the synthesis of a target language program. Therefore a virtual machine architecture was defined. This hypothetical machine is realized by an interpreter program. It performs the instructions of the target language. This guarantees independence from physical computer architectures.

Memory Management

The memory management phase of the compilation process defines the representation of named objects (variables, procedures, ...). The memory contains the target program, the data objects and the memory allocated at run time.

The static program code is placed at the beginning of the memory, followed by the global data objects. The rest of the memory is divided into two parts, the stack and the heap. The stack holds the procedure segments in last-in-first-out order. The heap stores data objects dynamically allocated at run time.

A procedure segment stores all information necessary to execute the respective procedure. These are a static link to the preceding procedure segment in the stack above, a dynamic link to the procedure segment of the caller procedure, the state of the machine (register values, instruction counter, ...) and the memory for the local objects.

Two kinds of objects with non local names exist: global objects and objects of calling procedures.

The access to global objects is simple; the addresses are known at compile time.

The addresses of objects of calling procedures have to be calculated by following the dynamic links to the respective procedure segments. To speed up this operation an array of pointers to the procedure segments is used, called display. The element i of the array contains the pointer to the procedure segment with the non local names of nesting level i.

Address Calculation

The address of the data object belonging to a name consists of the nesting level of the object and the relative position in the segment. For global names the relative address is the global address too. For structured variables the address calculation process needs several steps. For example, calculation of the address of Variable x.a[14]:

process x"." is recognized as record	address calculation nesting level (2) and relative address (7) are

identifier	extracted from the symbol table
a is recognized as record component	the relative starting address of a (5) is extracted from the type tree and added to the relative address
[14] is recognized as array index	the position of the 14. element in the array is calculated (size of the element is 5), (14 - 1) * 5 -> 65
the name is completed	the object can be found in nesting level 2, with the relative address 77 (7 + 5 + 65)

Certain address calculations have to be performed at run time, e.g., the address of an array element with a variable index.

The Target Language

The target l guage is a 0-address machine language. The data is stored in a separate data memory and all calculations are performed on values on the stack. The instruction set is small and can be separated into three classes:

- arithmetic instructions
- stack and data memory manipulation instructions
- control flow instructions

Arithmetic instructions

ALU arg

the operands are on the stack

arg is one of the following arithmetic or logic operators

ADD	add
SUB	subtract
MUL	multiplicat
DVD	divide
LST	less than
GRT	greater than
LSE	less equal
GRE	greater equal
EQU	equal
NEQ	not equal
AND	logical and
ORL	logical or
NOT	logical not
ANB	bitwise and
ORB	bitwise or
NEG	bitwise not
STB	set Bits from op1 to op 2
DVI	integer divide
MDD	op1 modulo op2
XOR	logical exclusive or

Stack and data memory manipulation instructions

CPY	copy top element of the stack
DLC	free memory on heap
ENT	allocate memory for local variables
LOC	allocate memory on heap
LDA	load of an address
LDC	load of the contents of an address
LDI	load of a constant
POP	pop top element of the stack
STI	store a memory range
STO	store a value

Control flow instructions

CAL	subroutine call
JMP	unconditional jump
JPZ	conditional jump
RET	return from a subroutine

GENERATION OF THE TARGET PROGRAM

Each object of the program has a calculated address. Now it must be decided that the address or value referenced by the address has to be loaded. In an expression the value of an identifier is used. The identifier on the left side of an assignment is an address. The parameters of a procedure are called by reference.

EXPRESSION

The correct expression evaluation sequence is defined by YACC. The elements have to be placed into postfix order.

Constant expressions are evaluated during the compilation. The result is placed into the produced code. This leads to a small code optimization.

STATEMENTS

The translation of control flow statements produces jump labels in the target code. The statements and their translation will be shown. Labels are represented by numbers. Terms in brackets are statement sequence. Their implementation is not shown completely.

IF-STATEMENT

```
IF expression                 (expression)
                                      JPZ 1
THEN StatementSequence1       (StatementSequence1)
ELSIF expression                      JMP 2
                              1:      (expression)
                                      JPZ 3
THEN StatementSequence2       (StatementSequence2)
ELSE StatementSequence3               JMP 2
                              3:      (StatementSequence3)
END                           2:
```

CASE-STATEMENT

```
CASE expression OF            (expression)
                                      CPY
ConstExpression               (ConstExpression)
                                      ALU EQU
                                      JPZ 1
":" StatementSequence                 (StatementSequence)
                                      JMP 2
                              1:      CPY
"|" ConstExpression ".."      (ConstExpression)
                                      ALU GRE
                                      JPZ 3
                                      CPY
ConstExpression               (ConstExpression)
                                      ALU LSE
                                      JPZ 3
":" StatementSequence         (StatementSequence)
                                      JMP 2
                              3:      CPY
ELSE StatementSequence        (StatementSequence)
                                      POP
END                           2:      POP
```

WHILE-STATEMENT

The WHILE-statement in Oberon is more general than in Modula-2. Multiple conditional expressions can be specified. All expressions are evaluated and the statement sequences of the TRUE expressions are processed. The loop is iterated until none of the expressions is TRUE.

```
WHILE expression              1:      (expression)
                                      JPZ 2
DO StatementSequence                  (StatementSequence)
ELSIF expression                      JMP 3
                              2:      (expression)
                                      JPZ 4
DO StatementSequence                  (StatementSequence)
```

```
    END                     3:      JMP 1
                            4:

REPEAT-STATEMENT
    REPEAT                  1:
    StatementSequence       (StatementSequence)
    UNTIL expression                (expression)
                                    JPZ 1
```

LOOP-STATEMENT

The LOOP-statement is repeated until an EXIT-statement. There can be more than one EXIT, which is implemented by an unconditional jump to the end of the loop.

```
    LOOP                    1:
    IF expression THEN
        EXIT                        JMP 2
    END                             JMP 1
                            2:
```

FUNCTIONS

At the begin of a function declaration the local variables and therefore the amount of memory are not known. The operand of the ENT instruction is a label, which is defined at the end of the function code together with the size of the needed memory.

IMPORT OF MODULES

The objects of the imported module are visible regarding to the definition module specification. So the code of the imported module is included into the target program.

THE CODE INTERPRETER

The compiler produces optional readable assembler code or machine code. The machine code is directly executable by the interpreter program.

The code interpreter is written in the C language, which guarantees unrestricted portability.

The virtual 0-address machine represented by the interpreter consists of a set of registers and separated data and program memory. The data memory is partitioned into a stack and a heap region.

The registers include an instruction counter, an instruction register, a stack pointer, a heap pointer, a counter for the procedural nesting level and the display.

The expressions are in postfix notation. All numeric data types are internally translated into floating point representation. The calculated results are transformed to the correct output format. This produces a lot of execution overhead, but simplifies the development of the interpreter.

PORTABILITY

The developed Oberon language system consists of the compiler and the interpreter program. It reaches a high level of portability. Without effort the whole system is easily transferable to different UNIX-based computer systems. Only a C- compiler and the LEX and YACC programms have to be available to compile the sources of the system. Therefore the portable Oberon system is not only restricted to the UNIX environment

The tradeoff of the portability is a lack in performance. The quality of the code produced by the compiler is comparable to commercial systems. The drawback lies in the interpretation of the code by a program. But this is acceptable for the educational usage of the system.

In the near future it is planned to realize an additional assembler program, which compiles the 0-address code produced by the Oberon compiler into real machine code. This will solve the performance problem of the system.

REFERENCES

[AhSe86]
Aho A., Sethi R., Ullman J.
Compilers - principles, techniques, and tools
Addison Wesley, (1986)

[GuWi89]
Gutknecht J., Wirth N.
The Oberon System
Software - Practice and Experience, 19, (1989)

[ScPo89]
Schikuta E., Posch F.
Entwicklung eines Compilers für die Programmiersprache Oberon
Techn. Report, AfIS, Inst.f.Stat.Inf., (1989), in german

[Wirt88]
Wirth N.
The programming language Oberon
Software - Practice and Experience, 18, (1989)

[Wirt89]
Wirth N.
From Modula to Oberon
The Programming Language Oberon
Tech. report, ETH Zrich, (1989)