Contents

Chapter 1
Introduction

Computer programs are formulated in a programming language and specify classes of computing processes. Computers, however, interpret sequences of particular instructions, but not program texts. Therefore, the program text must be translated into a suitable instruction sequence before it can be processed by a computer. This translation can be automated, which implies that it can be formulated as a program itself. The translation program is called a *compiler*, and the text to be translated is called *source text* (or sometimes *source code*).

It is not difficult to see that this translation process from source text to instruction sequence requires considerable effort and follows complex rules. The construction of the first compiler for the language *Fortran* (formula translator) around 1956 was a daring enterprise, whose success was not at all assured. It involved about 18 man-years of effort, and therefore figured among the largest programming projects of the time.

The intricacy and complexity of the translation process could be reduced only by choosing a clearly defined, well-structured source language. This occurred for the first time in 1960 with the advent of the language *Algol 60,* which established the technical foundations of compiler design that still are valid today. For the first time, a formal notation was also used for the definition of the language's structure (Naur, 1960).

The translation process is now guided by the structure of the analysed text. The text is decomposed, *parsed* into its components according to the given *syntax.* For the most elementary components, their semantics is recognized, and the meaning (semantics) of the composite parts is the result of the semantics of their components. Naturally, the meaning of the source text must be preserved by the translation.

The translation process essentially consists of the following parts:

(1) The sequence of characters of a source text is translated into a corresponding sequence of *symbols* of the vocabulary of the language. For instance, identifiers consisting of letters and digits, numbers consisting of digits, delimiters and operators consisting of special characters are recognized in this phase, which is called *lexical analysis.*

(2) The sequence of symbols is transformed into a representation that directly mirrors the syntactic structure of the source text and lets this structure easily be recognized. This phase is called *syntax analysis* (parsing).

(3) High-level languages are characterized by the fact that objects of programs, for example variables and functions, are classified according to their type. Therefore, in addition to syntactic rules, the language is also defined by compatibility rules among types of operators and operands. Hence, verification of whether these compatibility rules are observed by a program is an additional duty of a compiler. This verification is called *type checking.*

(4) On the basis of the representation resulting from step 2, a sequence of instructions taken from the instruction set of the target computer is generated. This phase is called *code generation.* In general it is the most involved part, not least because the instruction sets of many computers lack the desirable regularity. Often, the code generation part is therefore subdivided further.

A partitioning of the compilation process into as many parts as possible was the predominant technique until about 1980, because until then the available store was too small to accommodate the entire compiler. Only individual compiler parts would fit, and they could be loaded one after the other in sequence. The parts were called *passes,* and the whole was called a *multipass compiler.* The number of passes was typically 4–6, but reached 70 in a particular case (for PL/I) known to the author. Typically, the output of pass k served as input of pass k+1, and the disk served as intermediate storage (Figure 1.1). The very frequent access to disk storage resulted in long compilation times.

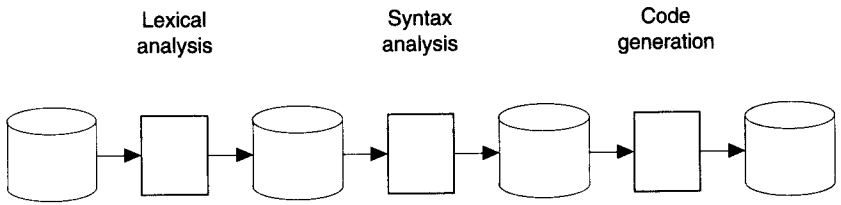


Figure 1.1 Multipass compilation.

Modern computers with their apparently unlimited stores make it feasible to avoid intermediate storage on disk. And with it, the complicated process of serializing a data structure for output, and its reconstruction on input, can be discarded as well. With *single-pass compilers*, increases in speed by factors of several thousands are therefore possible. Instead of being tackled one after another in strictly sequential fashion, the various parts (tasks) are interleaved. For example, code generation is not delayed until all preparatory tasks are completed, but starts immediately after the recognition of the first sentential structure of the source text.

A wise compromise exists in the form of a compiler with two parts, namely a *front end* and a *back end.* The first part comprises lexical and syntax analyses and type checking, and it generates a tree representing the syntactic structure of the source text. This tree is held in main store and constitutes the interface to the second part which handles code generation. The main advantage of this solution lies in the independence of the front end of the target computer and its instruction set. This advantage is inestimable if compilers for the same language and for various computers must be constructed, because the same front end serves them all.

The idea of decoupling source language and target architecture has also led to projects creating several front ends for different languages generating trees for a single back end. Whereas for the implementation of m languages for n computers $m * n$ compilers had been necessary, now m front ends and n back ends suffice (Figure 1.2).

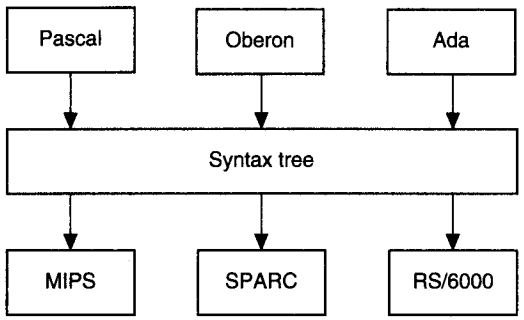


Figure 1.2 Front ends and back ends.

This modern solution to the problem of porting a compiler reminds us of the technique which played a significant role in the propagation of Pascal around 1975 (Wirth, 1971). The role of the structural tree was assumed by a linearized form, a sequence of commands of an abstract computer. The back end consisted of an interpreter program which was implementable with little effort, and the linear instruction sequence was called P-code. The drawback of this solution was the inherent loss of efficiency common to interpreters.

Frequently, one encounters compilers which do not directly generate binary code, but rather assembler text. For a complete translation an assembler is also involved, after the compiler. Hence, longer translation times are inevitable. Since this scheme hardly offers any advantages, we do not recommend this approach.

Increasingly, high-level languages are also employed for the programming of microcontrollers used in embedded applications. Such systems are primarily used for data acquisition and automatic control of machinery. In these cases, the store is typically small and is insufficient to carry a compiler. Instead, software is generated with the aid of other computers capable of compiling. A compiler which generates code for a computer different from the one executing the

compiler is called a *cross compiler*. The generated code is then transferred – *downloaded* – via a data transmission line.

In the following chapters we shall concentrate on the theoretical foundations of compiler design, and thereafter on the development of an actual single-pass compiler.

Chapter 2
Language and Syntax

Every language displays a structure called its grammar or *syntax*. For example, a correct sentence always consists of a subject followed by a predicate, 'correct' here meaning *well formed*. This fact can be described by the following formula:

```
sentence = subject predicate.
```

If we add to this formula the two further formulas

```
subject = "John" | "Mary".
predicate = "eats" | "talks".
```

then we define herewith exactly four possible sentences, namely

```
John eats
John talks
Mary eats
Mary talks
```

where the symbol '|' is to be pronounced as 'or'. We call these formulas *syntax rules, productions,* or simply syntactic *equations*. Subject and predicate are syntactic classes. A shorter notation for the above omits meaningful identifiers:

```
S = AB.          L = {ac, ad, bc, bd}
A = "a" | "b".
B = "c" | "d".
```

We will use this shorthand notation in the subsequent, short examples. The set L of sentences which can be generated in this way, that is, by repeated substitution of the left-hand sides by the right-hand sides of the equations, is called the *language*.

The example above evidently defines a language consisting of only four sentences. Typically, however, a language contains infinitely many sentences.

The following example shows that an infinite set may very well be defined with a finite number of equations. The symbol Ø stands for the empty sequence.

```
S = A.                L = {Ø, a, aa, aaa, aaaa, ... }
A = "a" A | Ø.
```

The means to do so is *recursion* which allows a substitution (here of A by "a"A) to be repeated arbitrarily often.

Our third example is again based on the use of recursion. But it generates not only sentences consisting of an arbitrary sequence of the same symbol, but also nested sentences:

```
S = A.                L = {b, abc, aabcc, aaabccc, ... }
A = "a" A "c" | "b".
```

It is clear that arbitrarily deep nestings (here of As) can be expressed, a property particularly important in the definition of structured languages.

Our fourth and last example exhibits the structure of expressions. The symbols E, T, F, and V stand for expression, term, factor, and variable.

```
E = T | A "+" T.
T = F | T "*" F.
F = V | "(" E ")".
V = "a" | "b" | "c" | "d".
```

From this example it is evident that a syntax does not only define the set of sentences of a language, but also provides them with a structure. The syntax decomposes sentences into their constituents as shown in the example of Figure 2.1. The graphical representations are called *structural trees* or *syntax trees.*

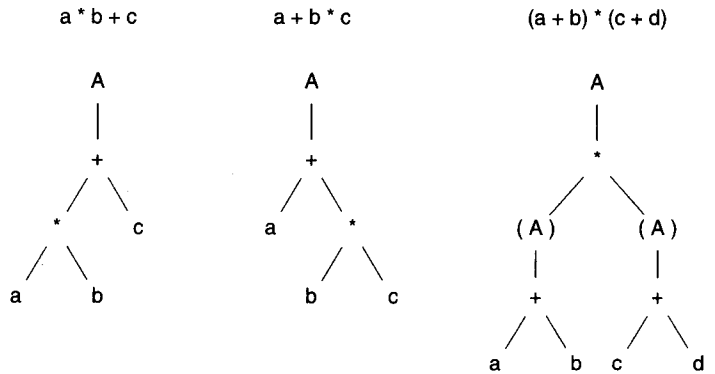


Figure 2.1 Structure of expressions.

Let us now formulate the concepts presented above more rigorously. A language is defined by the following:

(1) The set of *terminal symbols.* These are the symbols that occur in its sentences. They are said to be terminal, because they cannot be substituted by any other symbols. The substitution process stops with terminal symbols. In our first example this set consists of the elements a, b, c and d. The set is also called *vocabulary.*

(2) The set of *nonterminal symbols.* They denote syntactic classes and can be substituted. In our first example this set consists of the elements S, A and B.

(3) The set of syntactic *equations* (also called *productions*). These define the possible substitutions of nonterminal symbols. An equation is specified for each nonterminal symbol.

(4) The *start symbol.* It is a nonterminal symbol, in the examples above denoted by S.

A *language* is, therefore, the set of sequences of terminal symbols which, starting with the start symbol, can be generated by repeated application of syntactic equations, that is, substitutions.

We also wish to define rigorously and precisely the notation in which syntactic equations are specified. Let nonterminal symbols be identifiers as we know them from programming languages, that is, as sequences of letters (and possibly digits), for example, expression, term. Let terminal symbols be character sequences enclosed in quotes (strings), for example, "=", "|". For the definition of the structure of these equations it is convenient to use the tool just being defined itself:

```
syntax      = production syntax | Ø.
production  = identifier "=" expression ".".
expression  = term | expression "|" term.
term        = factor | term factor.
factor      = identifier | string.

identifier  = letter | identifier letter | identifier digit.
string      = stringhead """.
stringhead  = """ | stringhead character.
letter      = "A" | ... | "Z".
digit       = "0" | ... | "9".
```

This notation was introduced in 1960 by J. Backus and P. Naur in almost identical form for the formal description of the syntax of the language Algol 60. It is therefore called *Backus Naur Form* (BNF) (Naur, 1960). As our example shows, using recursion to express simple repetitions is rather detrimental to readability. Therefore, we extend this notation by two constructs to express repetition and optionality. Furthermore, we allow expressions to be enclosed within parentheses.

Thereby an extension of BNF called EBNF (Wirth, 1977) is postulated, which again we immediately use for its own, precise definition:

```
syntax      = {production}.
production  = identifier "=" expression ".".
expression  = term {"|" term}.
term        = factor {factor}.
factor      = identifier | string | "(" expression ")" | "[" expression "]"
              | "{" expression "}".

identifier  = letter {letter | digit}.
string      = """ {character} """.
letter      = "A" | ... | "Z".
digit       = "0" | ... | "9".
```

A factor of the form {x} is equivalent to an arbitrarily long sequence of x, including the empty sequence. A production of the form

```
A = AB | Ø.
```

is now formulated more briefly as A = {B}. A factor of the form [x] is equivalent to 'x or nothing', that is, it expresses optionality. Hence, the need for the special symbol Ø for the empty sequence vanishes.

The idea of defining languages and their grammar with mathematical precision goes back to N. Chomsky. It became clear, however, that the presented, simple scheme of substitution rules was insufficient to represent the complexity of spoken languages. This remained true even after the formalisms were considerably expanded. In contrast, this work proved extremely fruitful for the theory of programming languages and mathematical formalisms. With it, Algol 60 became the first programming language to be defined formally and precisely. In passing, we emphasize that this rigour applied to the syntax only, not to the semantics.

The use of the Chomsky formalism is also responsible for the term *programming language*, because programming languages seemed to exhibit a structure similar to spoken languages. We believe that this term is rather unfortunate on the whole, because a programming language is not spoken, and therefore is not a language in the true sense of the word. Formalism or formal notation would have been more appropriate terms.

One wonders why an exact definition of the sentences belonging to a language should be of any great importance. In fact, it is not really. However, it is important to know whether or not a sentence is well formed. But even here one may ask for a justification. Ultimately, the structure of a (well-formed) sentence is relevant, because it is instrumental in establishing the sentence's meaning. Owing to the syntactic structure, the individual parts of the sentence and their meaning

can be recognized independently, and together they yield the meaning of the whole.

Let us illustrate this point using the following, trivial example of an expression with the addition symbol. Let E stand for expression, and N for number:

```
E = N | E "+" E.
N = "1" | "2" | "3" | "4".
```

Evidently, '4 + 2 + 1' is a well-formed expression. It may even be derived in several ways, each corresponding to a different structure, as shown in Figure 2.2.

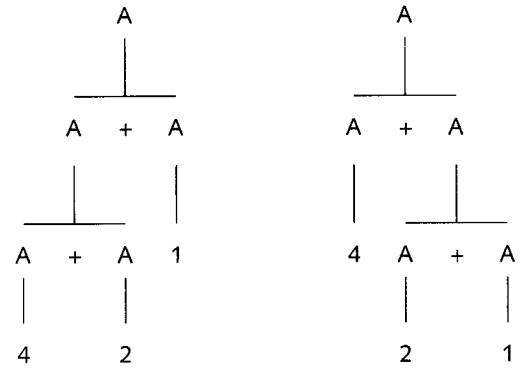


Figure 2.2 Differing structural trees for the same expression.

The two differing structures may also be expressed with appropriate parentheses, namely as (4 + 2) + 1 and as 4 + (2 + 1), respectively. Fortunately, thanks to the associativity of addition both yield the same value 7. But this need not always be the case. The mere use of subtraction in place of addition yields a counter example which shows that the two differing structures also yield a different interpretation and result: (4 – 2) – 1 = 1, 4 – (2 – 1) = 3. The example illustrates two facts:

(1) Interpretation of sentences always rests on the recognition of their syntactic structure.

(2) Every sentence must have a single structure in order to be unambiguous.

If the second requirement is not satisfied, ambiguous sentences arise. These may enrich spoken languages; ambiguous programming languages, however, are simply useless.

We call a syntactic class ambiguous if it can be attributed several structures. A language is ambiguous if it contains at least one ambiguous syntactic class (construct).

EXERCISES

2.1 The Algol 60 Report contains the following syntax (translated into EBNF):

```
primary = unsignedNumber | variable |
  "(" arithmeticExpression ")" | ... .
factor = primary | factor "↑" primary.
term = factor | term ("×" | "/" | "÷") factor.
simpleArithmeticExpression = term | ("+" | "–") term |
  simpleArithmeticExpression ("+" | "–") term.
arithmeticExpression = simpleArithmeticExpression |
  "IF" BooleanExpression "THEN" simpleArithmeticExpression
  "ELSE" arithmeticExpression.
relationalOperator = "<" | "≤" | "=" | "≥" | ">" | "≠".
relation = arithmeticExpression relationalOperator
  arithmeticExpression.
BooleanPrimary = logicalValue | variable | relation |
  "(" BooleanExpression ")" | ... .
BooleanSecondary = BooleanPrimary | "¬" BooleanPrimary.
BooleanFactor = BooleanSecondary |
  BooleanFactor "∧" BooleanSecondary.
BooleanTerm = BooleanFactor | BooleanTerm "∨" BooleanFactor.
implication = BooleanTerm | implication "⊃" BooleanTerm.
simpleBoolean = implication | simpleBoolean "≡" implication.
BooleanExpression = simpleBoolean |
  "IF" BooleanExpression "THEN" simpleBoolean
  "ELSE" BooleanExpression.
```

Determine the syntax trees of the following expressions, in which letters are to be taken as variables:

```
x + y + z
x × y + z
x + y × z
(x – y) × (x + y)
–x ÷ y
a + b < c + d
a + b < c ∨ d ≠ e ∧ ¬ f ⊃ g > h ≡ i × j = k ↑ l ∨ m – n + p ≤ q
```

2.2 The following productions also are part of the original definition of Algol 60. They contain ambiguities which were eliminated in the Revised Report.

```
forListElement = arithmeticExpression | arithmeticExpression
  "STEP" arithmeticExpression "UNTIL" arithmeticExpression |
  arithmeticExpression "WHILE" BooleanExpression.
forList = forListElement | forList "," forListElement.
```



```
forClause = "FOR" variable ":=" forList "DO".
forStatement = forClause statement.
compoundTail = statement "END" | statement ";" compoundTail.
compoundStatement = "BEGIN" compoundTail.
unconditional Statement = basicStatement | forStatement |
  compoundStatement | ... .
ifStatement = "IF" BooleanExpression "THEN"
  unconditionalStatement.
conditionalStatement = ifStatement | ifStatement "ELSE" statement.
statement = unconditionalStatement | conditionalStatement.
```

Find at least two different structures for the following expressions and statements. Let A and B stand for 'basic statements'.

```
IF a THEN b ELSE c = d
IF a THEN IF b THEN A ELSE B
IF a THEN FOR ... DO IF b THEN A ELSE B
```

Propose an alternative syntax which is unambiguous.

2.3 Consider the following constructs and find out which ones are correct in Algol, and which ones in Oberon (see Appendix A):

```
a + b = c + d
a * –b
a < b & c < d
```

Evaluate the following expressions:

```
5 * 13 DIV 4 =
13 DIV 5 * 4 =
```

Chapter 3
Regular Languages

Syntactic equations of the form defined in EBNF generate *context-free* languages. The term 'context-free' is due to Chomsky and stems from the fact that substitution of the symbol left of '=' by a sequence derived from the expression to the right of '=' is always permitted, regardless of the context in which the symbol is embedded within the sentence. It has turned out that this restriction to context freedom (in the sense of Chomsky) is quite acceptable for programming languages, and that it is even desirable. Context dependence in another sense, however, is indispensable. We will return to this topic in Chapter 8.

Here we wish to investigate a subclass rather than a generalization of context-free languages. This subclass, known as *regular languages*, plays a significant role in the realm of programming languages. In essence, they are the context-free languages whose syntax contains no recursion except for the specification of repetition. Since in EBNF repetition is specified directly and without the use of recursion, the following, simple definition can be given:

> A language is *regular*, if its syntax can be expressed by a single EBNF expression.

The requirement that a single equation suffices also implies that only terminal symbols occur in the expression. Such an expression is called a *regular expression.*

Two brief examples of regular languages may suffice. The first defines identifiers as they are common in most languages; and the second defines integers in decimal notation. We use the nonterminal symbols letter and digit for the sake of brevity. They can be eliminated by substitution, whereby a regular expression results for both identifier and integer.

```
identifier = letter {letter | digit}.
integer    = digit {digit}.
letter     = "A" | "B" | ... | "Z".
digit      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```


The reason for our interest in regular languages lies in the fact that programs for the recognition of regular sentences are particularly simple and efficient. By 'recognition' we mean the determination of the structure of the sentence, and thereby naturally the determination of whether the sentence is well formed, that is, it belongs to the language. Sentence recognition is called *syntax analysis.*

For the recognition of regular sentences a finite automaton, also called a *state machine,* is necessary and sufficient. In each step the state machine reads the next symbol and changes state. The resulting state is solely determined by the previous state and the symbol read. If the resulting state is unique, the state machine is *deterministic*, otherwise *nondeterministic.* If the state machine is formulated as a program, the state is represented by the current point of program execution.

The analysing program can be derived directly from the defining syntax in EBNF. For each EBNF construct K there exists a translation rule which yields a program fragment Pr(K). The translation rules from EBNF to program text are shown below. In these rules sym denotes a global variable always representing the symbol last read from the source text by a call to procedure next. Procedure error terminates program execution, signalling that the symbol sequence read so far does not belong to the language.

K	Pr(K)
"x"	IF sym = "x" THEN next ELSE error END
(exp)	Pr(exp)
[exp]	IF sym IN first(exp) THEN Pr(exp) END
{exp}	WHILE sym IN first(exp) DO Pr(exp) END
fac_0 fac_1 ... fac_n	Pr(fac_0); Pr(fac_1); ... Pr(fac_n)
$term_0$ | $term_1$ | ... | $term_n$	CASE sym OF first($term_0$): Pr($term_0$) | first($term_1$): Pr($term_1$) ... | first($term_n$): Pr($term_n$) END

The set first(K) contains all symbols with which a sentence derived from construct K may start. It is the set of *start symbols* of K. For the two examples of identifiers and integers they are:

```
first(integer) = digits = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}
first(identifier) = letters = {"A", "B", ... , "Z"}
```

The application of these simple translation rules generating a parser from a given syntax is, however, subject to the syntax being deterministic. This precondition may be formulated more concretely as follows:

K	Cond(K)
$term_0$ | $term_1$	The terms must not have any common start symbols.
fac_0 fac_1	If fac_0 contains the empty sequence, then the factors must not have any common start symbols.
[exp] or {exp}	The sets of start symbols of exp and of symbols that may follow K must be disjoint.

These conditions are satisfied trivially in the examples of identifiers and integers, and therefore we obtain the following programs for their recognition:

```
IF s IN letters THEN next ELSE error END;
WHILE sym IN letters + digits DO
  CASE sym OF
    "A" .. "Z": next
  | "0" .. "9": next
  END
END

IF sym IN digits THEN next ELSE error END;
WHILE sym IN digits DO next END
```

Frequently, the program obtained by applying the translation rules can be simplified by eliminating conditions which are evidently established by preceding conditions. The conditions sym IN letters and sym IN digits are typically formulated as follows:

```
("A" <= sym) & (sym <= "Z")          ("0" <= sym) & (sym <= "9")
```

The significance of regular languages in connection with programming languages stems from the fact that the latter are typically defined in two stages. First, their syntax is defined in terms of a vocabulary of *abstract* terminal symbols. Second, these abstract symbols are defined in terms of sequences of *concrete* terminal symbols, such as ASCII characters. This second definition typically has a regular syntax. The separation into two stages offers the advantage that the definition of the abstract symbols, and thereby of the language, is independent of any concrete representation in terms of any particular character sets used by any particular equipment.

This separation also has consequences on the structure of a compiler. The process of syntax analysis is based on a procedure to obtain the next symbol. This procedure in turn is based on the definition of symbols in terms of sequences of one or more characters. This latter procedure is called a *scanner,* and syntax analysis on this second, lower level, *lexical analysis.* The definition of symbols in terms of characters is typically given in terms of a regular language, and therefore the scanner is typically a state machine.

We summarize the differences between the two levels as follows:

Process	*Input element*	*Algorithm*	*Syntax*
Lexical analysis	Character	Scanner	Regular
Syntax analysis	Symbol	Parser	Context free

As an example we show a scanner for a parser of EBNF. Its terminal symbols and their definition in terms of characters are

```
symbol     = {blank} (identifier | string | "(" | ")" | "[" | "]" | "{" | "}" | "|" | "=" | ".") .
identifier = letter {letter | digit}.
string     = """ {character} """.
```

From this we derive the procedure GetSym which, upon each call, assigns a numeric value representing the next symbol read to the global variable sym. If the symbol is an identifier or a string, the actual character sequence is assigned to the further global variable id. It must be noted that typically a scanner also takes into account rules about blanks and ends of lines. Mostly these rules say: blanks and ends of lines separate consecutive symbols, but otherwise are of no significance. Procedure GetSym, formulated in Oberon, makes use of the following declarations.

```
CONST IdLen = 32;
  ident = 0; literal = 2; lparen = 3; lbrak = 4; lbrace = 5; bar = 6; eql = 7;
  rparen = 8; rbrak = 9; rbrace = 10; period = 11; other = 12;

TYPE Identifier = ARRAY IdLen OF CHAR;

VAR ch: CHAR;
  sym: INTEGER;
  id: Identifier;
  R: Texts.Reader;
```

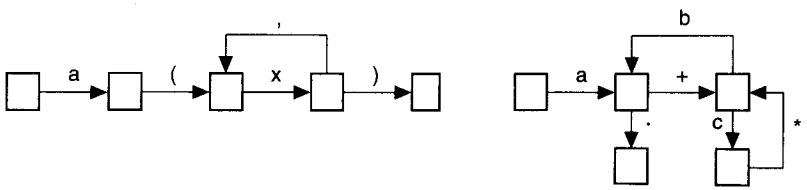
Note that the abstract reading operation is now represented by the concrete call Texts.Read(R, ch). R is a globally declared Reader specifying the source text. Also note that variable ch must be global, because at the end of GetSym it may contain the first character belonging to the next symbol. This must be taken into account upon the subsequent call of GetSym.

```
PROCEDURE GetSym;
  VAR i: INTEGER;
```

```
BEGIN
  WHILE ~R.eot & (ch <= " ") DO Texts.Read(R, ch) END;  (*skip blanks*)
  CASE ch OF
    "A" .. "Z", "a" .. "z": sym := ident; i := 0;
      REPEAT id[i] := ch; INC(i); Texts.Read(R, ch)
      UNTIL (CAP(ch) < "A") OR (CAP(ch) > "Z");
      id[i] := 0X
  | 22X:  (*quote*)
      Texts.Read(R, ch); sym := literal; i := 0;
      WHILE (ch # 22X) & (ch > " ") DO
        id[i] := ch; INC(i); Texts.Read(R, ch)
      END;
      IF ch <= " " THEN error(1) END;
      id[i] := 0X; Texts.Read(R, ch)
  | "=" : sym := eql; Texts.Read(R, ch)
  | "(" : sym := lparen; Texts.Read(R, ch)
  | ")" : sym := rparen; Texts.Read(R, ch)
  | "[" : sym := lbrak; Texts.Read(R, ch)
  | "]" : sym := rbrak; Texts.Read(R, ch)
  | "{" : sym := lbrace; Texts.Read(R, ch)
  | "}" : sym := rbrace; Texts.Read(R, ch)
  | "|" : sym := bar; Texts.Read(R, ch)
  | "." : sym := period; Texts.Read(R, ch)
  ELSE sym := other; Texts.Read(R, ch)
  END
END GetSym
```

EXERCISE

3.1 Sentences of regular languages can be recognized by finite state machines. They are usually described by transition diagrams. Each node represents a state, and each edge a state transition. The edge is labelled by the symbol that is read by the transition. Consider the following diagrams and describe the syntax of the corresponding languages in EBNF.



Chapter 4
Analysis of Context-free Languages

4.1 The method of recursive descent

Regular languages are subject to the restriction that no nested structures can be expressed. Nested structures can be expressed with the aid of recursion only (see Chapter 2).

A finite state machine therefore cannot suffice for the recognition of sentences of context-free languages. We will nevertheless try to derive a parser program for the third example in Chapter 2, by using the methods explained in Chapter 3. Wherever the method will fail – and it must fail – lies the clue for a possible generalization. It is indeed surprising how small the necessary additional programming effort turns out to be.

The construct

```
A = "a" A "c" | "b".
```

leads, after suitable simplification and the use of an IF instead of a CASE statement, to the following piece of program:

```
IF sym = "a" THEN
  next;
  IF sym = A THEN next ELSE error END;
  IF sym = "c" THEN next ELSE error END
ELSIF sym = "b" THEN next
ELSE error
END
```

Here we have blindly treated the nonterminal symbol A in the same fashion as terminal symbols. This is of course not acceptable. The purpose of the third line of the program is to parse a *construct* of the form A (rather than to read a symbol A). However, this is precisely the purpose of our program too. Therefore, the simple solution to our problem is to give the program a name, that is, to give it the form of a procedure, and to substitute the third line of program by a call to this

procedure. Just as in the syntax the *construct* A is recursive, so is the *procedure* A recursive:

```
PROCEDURE A;
BEGIN
  IF sym = "a" THEN
    next; A;
    IF sym = "c" THEN next ELSE error END
  ELSIF sym = "b" THEN next
  ELSE error
  END
END A
```

The necessary extension of the set of translation rules is extremely simple. The only additional rule is:

> A parsing algorithm is derived for each nonterminal symbol, and it is formulated as a procedure carrying the name of the symbol. The occurrence of the symbol in the syntax is translated into a call of the corresponding procedure.

Note: this rule holds regardless of whether the procedure is recursive or not.

It is important to verify that the conditions for a deterministic algorithm are satisfied. This implies among other things that in an expression of the form

$term_0$ | $term_1$

the terms must not feature any common start symbols. This requirement excludes left recursion. If we consider the left recursive production

```
A = A "a" | "b".
```

we recognize that the requirement is violated, simply because b is a start symbol of A (b IN first(A)), and because therefore first(A"a") and first("b") are not disjoint. "b" is the common element.

The simple consequence is: left recursion can and must be replaced by repetition. In the example above A = A "a" | "b" is replaced by A = "b" {"a"}.

Another way to look at our step from the state machine to its generalization is to regard the latter as a *set* of state machines which call upon each other and upon themselves. In principle, the only new condition is that the state of the calling machine is resumed after termination of the called state machine. The state must therefore be preserved. Since state machines are nested, a stack is the appropriate form of store. Our extension of the state machine is therefore called a *pushdown automaton.* Theoretically relevant is the fact that the stack (pushdown

store) must be arbitrarily deep. This is the essential difference between the finite state machine and the infinite pushdown automaton.

The general principle which is suggested here is the following: consider the recognition of the sentential construct which begins with the start symbol of the underlying syntax as the uppermost goal. If during the pursuit of this goal, that is, while the production is being parsed, a nonterminal symbol is encountered, then the recognition of a construct corresponding to this symbol is considered as a subordinate goal to be pursued first, while the higher goal is temporarily suspended. This strategy is therefore also called *goal-oriented parsing.* If we look at the structural tree of the parsed sentence we recognize that goals (symbols) higher in the tree are tackled first, lower goals (symbols) thereafter. The method is therefore called *top-down parsing* (Knuth, 1971; Aho and Ullman, 1977). Moreover, the presented implementation of this strategy based on recursive procedures is known as *recursive descent parsing.*

Finally, we recall that decisions about the steps to be taken are always made on the basis of the single, next input symbol only. The parser looks ahead by one symbol. A *lookahead* of several symbols would complicate the decision process considerably, and thereby also slow it down. For this reason we will restrict our attention to languages which can be parsed with a lookahead of a single symbol.

As a further example to demonstrate the technique of recursive descent parsing, let us consider a parser for EBNF, whose syntax is summarized here once again:

```
syntax     = {production}.
production = identifier "=" expression "." .
expression = term {"|" term}.
term       = factor {factor}.
factor     = identifier | string | "(" expression ")" | "[" expression "]" |
             "{" expression "}".
```

By application of the given translation rules and subsequent simplification the following parser results. It is formulated as an Oberon module:

```
MODULE EBNF;
  IMPORT Viewers, Texts, TextFrames, Oberon;

  CONST IdLen = 32;
    ident = 0; literal = 2; lparen = 3; lbrak = 4; lbrace = 5; bar = 6; eql = 7;
    rparen = 8; rbrak = 9; rbrace = 10; period = 11; other = 12;

  TYPE Identifier = ARRAY IdLen OF CHAR;

  VAR ch: CHAR;
    sym: INTEGER;
    lastpos: LONGINT;
    id: Identifier;
    R: Texts.Reader;
```

```
  W: Texts.Writer;

PROCEDURE error(n: INTEGER);
  VAR pos: LONGINT;
BEGIN pos := Texts.Pos(R);
  IF pos > lastpos+4 THEN  (*avoid spurious error messages*)
    Texts.WriteString(W, "  pos"); Texts.WriteInt(W, pos, 6);
    Texts.WriteString(W, "  err"); Texts.WriteInt(W, n, 4); lastpos := pos;
    Texts.WriteString(W, "  sym "); Texts.WriteInt(W, sym, 4);
    Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
  END
END error;

PROCEDURE GetSym;
BEGIN ... (*see Chapter 3*)
END GetSym;

PROCEDURE expression;

  PROCEDURE term;

    PROCEDURE factor;
    BEGIN
      IF sym = ident THEN record(T0, id, 1); GetSym
      ELSIF sym = literal THEN record(T1, id, 0); GetSym
      ELSIF sym = lparen THEN
        GetSym; expression;
        IF sym = rparen THEN GetSym ELSE error(2) END
      ELSIF sym = lbrak THEN
        GetSym; expression;
        IF sym = rbrak THEN GetSym ELSE error(3) END
      ELSIF sym = lbrace THEN
        GetSym; expression;
        IF sym = rbrace THEN GetSym ELSE error(4) END
      ELSE error(5)
      END
    END factor;

  BEGIN (*term*) factor;
    WHILE sym < bar DO factor END
  END term;

BEGIN (*expression*) term;
  WHILE sym = bar DO GetSym; term END
END expression;

PROCEDURE production;
BEGIN (*sym = ident*) GetSym;
  IF sym = eql THEN GetSym ELSE error(7) END;
  expression;
```



```
  IF sym = period THEN GetSym ELSE error(8) END
END production;

PROCEDURE syntax;
BEGIN
  WHILE sym = ident DO production END
END syntax;

PROCEDURE Compile*;
BEGIN (*set R to the beginning of the text to be compiled*)
  lastpos := 0; Texts.Read(R, ch); GetSym; syntax;
  Texts.Append(Oberon.Log, W.buf)
END Compile;

BEGIN Texts.OpenWriter(W)
END EBNF.
```

4.2 Table-driven top-down parsing

The method of recursive descent is only one of several techniques to realize the top-down parsing principle. Here we shall present another technique: table-driven parsing.

The idea of constructing a general algorithm for top-down parsing for which a specific syntax is supplied as a parameter is hardly far-fetched. The syntax takes the form of a data structure which is typically represented as a graph or table. This data structure is then interpreted by the general parser. If the structure is represented as a graph, we may consider its interpretation as a traversal of the graph, guided by the source text being parsed.

First, we must determine a data representation of the structural graph. We know that EBNF contains two repetitive constructs, namely sequences of factors and sequences of terms. Naturally, they are represented as lists. Every element of the data structure represents a (terminal) symbol. Hence, every element must be capable of denoting two successors represented by pointers. We call them next for the next consecutive factor and alt for the next alternative term. Formulated in the language Oberon, we declare the following data types:

```
Symbol  =  POINTER TO SymDesc;
SymDesc  =  RECORD alt, next: Symbol END
```

Then formulate this abstract data type for terminal and nonterminal symbols by using Oberon's type extension feature (Reiser and Wirth, 1992). Records denoting terminal symbols specify the symbol by the additional attribute sym:

```
Terminal  =  POINTER TO TSDesc;
TSDesc  =  RECORD (SymDesc) sym: INTEGER END
```

Elements representing a nonterminal symbol contain a reference (pointer) to the data structure representing that symbol. Out of practical considerations we introduce an indirect reference: the pointer refers to an additional header element, which in turn refers to the data structure. The header also contains the name of the structure, that is, of the nonterminal symbol. Strictly speaking, this addition is unnecessary; its usefulness will become apparent later.

```
Nonterminal = POINTER TO NTSDesc;
NTSDesc = RECORD (SymDesc) this: Header END
Header = POINTER TO HDesc;
HDesc = RECORD sym: Symbol; name: ARRAY n OF CHAR END
```

As an example we choose the following syntax for simple expressions. Figure 4.1 displays the corresponding data structure as a graph. Horizontal edges are next pointers, vertical edges are alt pointers.

```
expression = term {("+" | "–") term}.
term       = factor {("*" | "/") factor}.
factor     = id | "(" expression ")" .
```

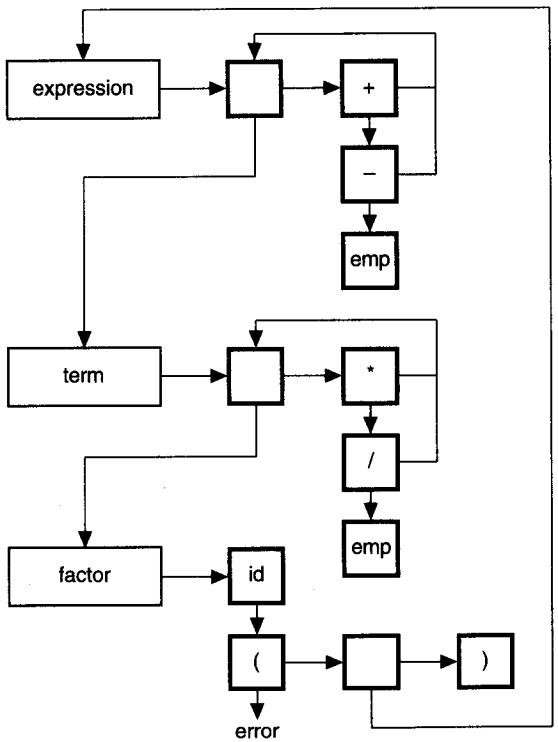


Figure 4.1 Syntax as data structure.

Now we are in a position to formulate the general parsing algorithm in the form of a concrete procedure:

```
PROCEDURE Parsed(hd: Header): BOOLEAN;
  VAR x: Symbol; match: BOOLEAN;
BEGIN x := hd.sym; Texts.WriteString(Wr, hd.name);
  REPEAT
    IF x IS Terminal THEN
      IF x(Terminal).sym = sym THEN match := TRUE; GetSym
      ELSE match := (x = empty)
      END
    ELSE match := Parsed(x(Nonterminal).this)
    END;
    IF match THEN x := x.next ELSE x := x.alt END
  UNTIL x = NIL;
  RETURN match
END Parsed
```

The following remarks must be kept in mind:

(1) We tacitly assume that terms always are of the form

$$T = f_0 \mid f_1 \mid \ldots \mid f_n$$

where all factors except the last start with a *distinct, terminal* symbol. Only the last factor may start with either a terminal or a nonterminal symbol. Under this condition it is possible to traverse the list of alternatives and in each step to make only a single comparison.

(2) The data structure can be derived from the syntax (in EBNF) automatically, that is, by a program which compiles the syntax.

(3) In the procedure above the name of each nonterminal symbol to be recognized is output. The header element serves precisely this purpose.

(4) Empty is a special terminal symbol and element representing the empty sequence. It is needed to mark the exit of repetitions (loops).

4.3 Bottom-up parsing

Both the recursive-descent and table-driven parsing shown here are techniques based on the principle of top-down parsing. The *primary goal* is to show that the text to be analysed is derivable from the *start symbol*. Any nonterminal symbols encountered are considered as subgoals. The parsing process constructs the syntax tree beginning with the start symbol as its root, that is, in the top-down direction.

However, it is also possible to proceed according to a complementary principle in the *bottom-up* direction. The text is read without pursuit of a specific

goal. After each step a test checks whether the just read subsequence corresponds to some sentential construct, that is, the right part of a production. If this is the case, that subsequence is replaced by the corresponding nonterminal symbol. The recognition process again consists of consecutive steps, of which there are two distinct kinds:

(1) Shifting the next input symbol into a stack (shift step),

(2) Reducing a stacked sequence of symbols into a single nonterminal symbol according to a production (reduce step).

Parsing in the bottom-up direction is also called *shift-reduce parsing*. The syntactic constructs are built up and then reduced; the syntax tree grows from the bottom to the top (Knuth, 1965; Aho and Ullman, 1977; Kastens, 1990).

Once again, we demonstrate the process with the example of simple expressions. Let the syntax be as follows:

```
E = T | E "+" T.        expression
T = F | T "*" F.        term
F = id | "(" E ")".     factor
```

and let the sentence to be recognized be x * (y + z). In order to display the process, the remaining source text is shown to the right, whereas to the left the – initially empty – sequence of recognized constructs is listed. At the far left, the letters S and R indicate the kind of step taken.

```
                    x * (y + z)
S       x             * (y + z)
R       F             * (y + z)
R       T             * (y + z)
S       T *             (y + z)
S       T * (           y + z)
S       T * (y            + z)
R       T * (F            + z)
R       T * (T            + z)
R       T * (E            + z)
S       T * (E +            z)
S       T * (E + z          )
R       T * (E + F          )
R       T * (E + T          )
R       T * (E              )
S       T * (E)
R       T * F
R       T
R       E
```


At the end, the initial source text is reduced to the start symbol E, which here would better be called the *stop* symbol. As mentioned earlier, the intermediate store to the left is a stack.

In analogy to this representation, the process of parsing the same input according to the top-down principle is shown below. The two kinds of steps are denoted by M (match) and P (produce, expand). The start symbol is E.

```
        E           x * (y + z)
P       T           x * (y + z)
P       T * F       x * (y + z)
P       F * F       x * (y + z)
P       id * F      x * (y + z)
M         * F         * (y + z)
M           F           (y + z)
P           (E)         (y + z)
M           E)            y + z)
P           E + T)        y + z)
P           T + T)        y + z)
P           F + T)        y + z)
P           id + T)       y + z)
M             + T)          + z)
M               T)            z)
P               F)            z)
P               id)           z)
M               )             )
M
```

Evidently, in the bottom-up method the sequence of symbols read is always reduced at its *right end,* whereas in the top-down method it is always the *leftmost* nonterminal symbol which is expanded. According to Knuth the bottom-up method is therefore called LR-parsing, and the top-down method LL-parsing. The first L expresses the fact that the text is being read from *left* to right. Usually, this denotation is given a parameter k (LL(k), LR(k)). It indicates the extent of the lookahead being used. We will always implicitly assume $k = 1$.

Let us briefly return to the bottom-up principle. The concrete problem lies in determining which kind of step is to be taken next, and, in the case of a reduce step, how many symbols on the stack are to be involved in the step. This question is not easily answered. We merely state that in order to guarantee an efficient parsing process, the information on which the decision is to be based must be present in an appropriately compiled way. Bottom-up parsers always use tables, that is, data structured in an analogous manner to the table-driven top-down parser presented above. In addition to the representation of the syntax as a data structure, further tables are required to allow us to determine the next step in an efficient manner. Bottom-up parsing is therefore in general more intricate and complex than top-down parsing.

There exist various LR parsing algorithms. They impose different boundary conditions on the syntax to be processed. The more lenient these conditions are, the more complex the parsing process. We mention here the SLR (DeRemer, 1971) and LALR (LaLonde et al., 1971) methods without explaining them in any fur- ther detail.

EXERCISES

4.1 Algol 60 contains a multiple assignment of the form v1 := v2 := ... vn := e. It is defined by the following syntax:

```
assignment  = leftpartlist expression.
leftpartlist = leftpart | leftpartlist leftpart.
leftpart    = variable ":=" .
expression  = variable | expression "+" variable.
variable    = ident | ident "[" expression "]" .
```

What is the degree of lookahead necessary to parse this syntax according to the top-down principle? Propose an alternative syntax for multiple assignments requiring a lookahead of one symbol only.

4.2 Determine the symbol sets first and follow of the EBNF constructs production, expression, term, and factor. Using these sets, verify that EBNF is deterministic.

```
syntax      = {production}.
production  = id "=" expression "." .
expression  = term {"|" term}.
term        = factor {factor}.
factor      = id | string | "(" expression ")" | "[" expression "]" |
              "{" expression "}".
id          = letter {letter | digit}.
string      = """ {character} """.
```

4.3 Write a parser for EBNF and extend it with statements generating the data structure (for table-driven parsing) corresponding to the syntax read.

Chapter 5
Attributed Grammars and Semantics

In attributed grammars certain attributes are associated with individual constructs, that is, with nonterminal symbols. The symbols are parametrized and represent whole classes of variants. This serves to simplify the syntax, but is, in practice, indispensable for extending a parser into a genuine translator (Rechenberg and Mössenböck, 1985). The translation process is characterized by the association of a (possibly empty) output with every recognition of a sentential construct. Each syntactic equation (production) is accompanied by additional rules defining the relationship between the attribute values of the symbols which are reduced, the attribute values for the resulting nonterminal symbol, and the issued output. We present three applications for attributes and attribute rules.

5.1 Type rules

As a simple example we shall consider a language featuring several data types. Instead of specifying separate syntax rules for expressions of each type (as was done in Algol 60), we define expressions exactly once, and associate the data type T as attribute with every construct involved. For example, an expression of type T is denoted as exp(T), that is, as exp with attribute value T. Rules about type compatibility are then regarded as additions to the individual syntactic equations. For instance, the requirements that both operands of addition and subtraction must be of the same type, and that the result type is the same as that of the operands, are specified by such additional attribute rules:

Syntax		*Attribute rule*	*Context condition*
$exp(T_0)$ =	$term(T_1)$ |	$T_0 := T_1$	
	$exp(T_1)$ "+" $term(T_2)$ |	$T_0 := T_1$	$T_1 = T_2$
	$exp(T_1)$ "–" $term(T_2)$.	$T_0 := T_1$	$T_1 = T_2$

If the requirements are relaxed to allow operands of the types INTEGER and REAL to be admissible in mixed expressions, the rules become more complicated:

T_0 := if (T_1 = INTEGER) & (T_2 = INTEGER) then INTEGER else REAL,

T_1 IN {INTEGER, REAL}
T_2 IN {INTEGER, REAL}

Rules about type compatibility are indeed also static in the sense that they can be verified without execution of the program. Hence, their separation from purely syntactic rules appears quite arbitrary, and their integration into the syntax in the form of attribute rules is entirely appropriate. However, we note that attributed grammars obtain a new dimension, if the possible attribute values (here, types) and their number are not known a priori.

If a syntactic equation contains a repetition, then it is appropriate with regard to attribute rules to express it with the aid of recursion. In the case of an option, it is best to express the two cases separately. This is shown by the following example where the two expressions

exp(T_0) = term(T_1) {"+" term(T_2)}. exp(T_0) = ["–"] term(T_1).

are split into pairs of terms, namely

exp(T_0) = term(T_1) | exp(T_1) "+" term(T_2).

exp(T_0) = term(T_1) | "–" term(T_1).

The type rules associated with a production come into effect whenever a construct corresponding to the production is recognized. This association is simple to implement in the case of a recursive descent parser: program statements implementing the attribute rules are simply interspersed within the parsing statements, and the attributes occur as parameters to the parser procedures standing for the syntactic constructs (nonterminal symbols). The procedure for recognizing expressions may serve as a first example to demonstrate this extension process, where the original parsing procedure serves as the scaffolding:

```
PROCEDURE expression;
BEGIN term;
  WHILE (sym = "+") OR (sym = "–") DO
    GetSym; term
  END
END expression
```

is extended to implement its attribute (type) rules:

```
PROCEDURE expression(VAR typ0: Type);
  VAR typ1, typ2: Type;
BEGIN term(typ1);
  WHILE (sym = "+") OR (sym = "–") DO
    GetSym; term(typ2);
    typ1 := ResType(typ1, typ2)
  END;
  typ0 := typ1
END expression
```


5.2 Evaluation rules

As our second example we consider a language consisting of expressions whose factors are numbers only. It is a short step to extend the parser into a program not only recognizing, but at the same time also evaluating expressions. We associate with each construct its value through an attribute called val. In analogy to the type compatibility rules in our previous example, we now must process evaluation rules while parsing. Thereby we have implicitly introduced the notion of *semantics*.

Syntax	*Attribute rule (semantics)*
exp(v_0) = term(v_1) |	v_0 := v_1
exp(v_1) "+" term(v_2) |	v_0 := v_1 + v_2
exp(v_1) "–" term(v_2).	v_0 := v_1 – v_2
term(v_0) = factor(v_1) |	v_0 := v_1
term(v_1) "*" factor(v_2) |	v_0 := v_1 * v_2
term(v_1) "/" factor(v_2).	v_0 := v_1 / v_2
factor(v_0) = number(v_1) |	v_0 := v_1
"(" exp(v_1) ")".	v_0 := v_1

Here, the attribute is the computed, numeric value of the recognized construct. The necessary extension of the corresponding parsing procedure leads to the following procedure for expressions:

```
PROCEDURE expression(VAR val0: INTEGER);
  VAR val1, val2: INTEGER; op: CHAR;
BEGIN term(val1);
  WHILE (sym = "+") OR (sym = "–") DO
    op := sym; GetSym; term(val2);
    IF op = "+" THEN val1 := val1 + val2  ELSE val1 := val1 – val2 END
  END;
  val0 := val1
END expression
```

5.3 Translation rules

A third example of the application of attributed grammars exhibits the basic structure of a compiler. The additional rules associated with a production here do not govern attributes of symbols, but specify the output (code) issued when the production is applied in the parsing process. The generation of output may be considered as a side-effect of parsing. Typically, the output is a sequence of instructions. In this example, the instructions are replaced by abstract symbols, and their output is specified by the operator put.

Syntax		*Output rule (semantics)*
exp	= term	
	exp "+" term	put("+")
	exp "–" term.	put("–")
term	= factor	
	term "*" factor	put("*")
	term "/" factor.	put("/")
factor	= number	put(number)
	"(" exp ")".	

As can easily be verified, the sequence of output symbols corresponds to the parsed expression in postfix notation. The parser has been extended into a translator.

Infix notation	*Postfix notation*
2 + 3	2 3 +
2 * 3 + 4	2 3 * 4 +
2 + 3 * 4	2 3 4 * +
(5 – 4) * (3 + 2)	5 4 – 3 2 + *

The procedure for parsing and translating expressions is as follows:

```
PROCEDURE expression;
  VAR op: CHAR;
BEGIN term;
  WHILE (sym = "+") OR (sym = "–") DO
    op := sym; GetSym; term; put(op)
  END
END expression
```

When using a table-driven parser, the tables expressing the syntax may easily be extended also to represent the attribute rules. If the evaluation and translation

rules are also contained in associated tables, one is tempted to speak about a formal definition of the language. The general, table-driven parser grows into a general, table-driven compiler. This, however, has so far remained a pipe dream, but the idea goes back to the 1960s. It is represented schematically by Figure 5.1.

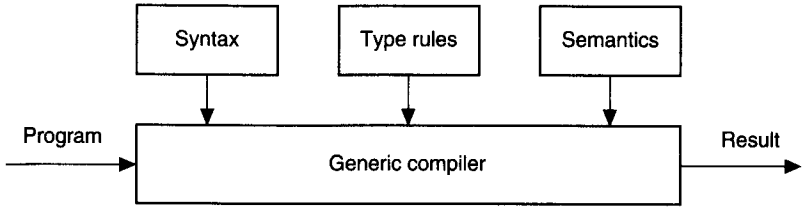


Figure 5.1 Schema of a general, parametrized compiler.

Ultimately, the basic idea behind every language is that it should serve as a means for communication. This means that partners must use and understand the *same* language. Promoting the ease by which a language can be modified and extended may therefore be rather counterproductive. Nevertheless, it has become customary to build compilers using table-driven parsers, and to derive these tables automatically from the syntax with the help of tools. The semantics are expressed by procedures whose calls are also integrated automatically into the parser. Compilers thereby not only become bulkier and less efficient than is warranted, but also much less transparent. The latter property remains one of our principal concerns, and therefore we shall not pursue this course any further.

EXERCISE

5.1 Extend the program for syntactic analysis of EBNF texts in such a way that it generates (1) a list of terminal symbols, (2) a list of nonterminal symbols, and (3) for each nonterminal symbol the sets of its start and follow symbols. Based on these sets, the program is then to determine whether the given syntax can be parsed top-down with a lookahead of a single symbol. If this is not so, the program displays the conflicting productions in a suitable way. Hint: Use Warshall's algorithm (R. W. Floyd, Algorithm 96, *Comm. ACM*, June 1962).

```
TYPE matrix = ARRAY [1..n],[1..n] OF BOOLEAN;
PROCEDURE ancestor(VAR m: matrix; n: INTEGER);
(* Initially m[i,j] is TRUE, if individual i is a parent of individual j.
  At completion, m[i, j] is TRUE, if i is an ancestor of j *)
  VAR i, j, k: INTEGER;
```

```
BEGIN
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO
      IF m[j, i] THEN
        FOR k := 1 TO n DO
          IF m[i, k] THEN m[j, k] := TRUE END
        END
      END
    END
  END
END ancestor
```

It may be assumed that the numbers of terminal and nonterminal symbols of the analysed languages do not exceed a given limit (for example, 32).

Chapter 6
The Programming Language Oberon-0

In order to avoid getting lost in generalities and abstract theories, we shall build a specific, concrete compiler, and explain the various problems that arise during the project. In order to do this, we must postulate a specific source language.

Of course we must keep this compiler, and therefore also the language, sufficiently simple in order to remain within the scope of an introductory tutorial. On the other hand, we wish to explain as many of the fundamental constructs of languages and compilation techniques as possible. Out of these considerations have grown the boundary conditions for the choice of the language: it must be simple, yet representative. We have chosen a subset of the language *Oberon* (Reiser and Wirth, 1992), which is a condensation of its ancestors *Modula-2* (Wirth, 1982) and *Pascal* (Wirth, 1971) into their essential features. Oberon may be said to be the latest offspring in the tradition of *Algol 60* (Naur, 1960). Our subset is called *Oberon-0,* and it is sufficiently powerful to teach and exercise the foundations of modern programming methods.

Concerning program structures, Oberon-0 is reasonably well developed. The elementary statement is the assignment. Composite statements incorporate the concepts of the statement sequence and conditional and repetitive execution, the latter in the form of the conventional if- and while-statements. Oberon-0 also contains the important concept of the subprogram, represented by the procedure declaration and the procedure call. Its power mainly rests on the possibility of parametrizing procedures. In Oberon, we distinguish between value and variable parameters.

With respect to data types, however, Oberon-0 is rather frugal. The only elementary data types are integers and the logical values, denoted by INTEGER and BOOLEAN. It is thus possible to declare integer-valued constants and variables, and to construct expressions with arithmetic operators. Comparisons of expressions yield Boolean values, which can be subjected to logical operations.

The available data structures are the array and the record. They can be nested arbitrarily. Pointers, however, are omitted.

Procedures represent functional units of statements. It is therefore appropriate to associate the concept of locality of names with the notion of the procedure. Oberon-0 offers the possibility of declaring identifiers local to a procedure, that is, in such a way that the identifiers are valid (visible) only within the procedure itself.

This very brief overview of Oberon-0 serves to provide the reader with the context necessary to understand the subsequent syntax, defined in terms of EBNF.

```
ident = letter {letter | digit}.
integer = digit {digit}.

selector = {"." ident | "[" expression "]"}.
number = integer.
factor = ident selector | number | "(" expression ")" | "~" factor.
term = factor {("*" | "DIV" | "MOD" | "&") factor}.
SimpleExpression = ["+"|"–"] term {("+"|"–" | "OR") term}.
expression = SimpleExpression
  [("=" | "#" | "<" | "<=" | ">" | ">=") SimpleExpression].
assignment = ident selector ":=" expression.
ActualParameters = "(" [expression {"," expression}] ")".
ProcedureCall = ident selector [ActualParameters].
IfStatement = "IF" expression "THEN" StatementSequence
  {"ELSIF" expression "THEN" StatementSequence}
  ["ELSE" StatementSequence] "END".
WhileStatement = "WHILE" expression "DO" StatementSequence "END".
statement = [assignment | ProcedureCall | IfStatement | WhileStatement].
StatementSequence = statement {";" statement}.

IdentList = ident {"," ident}.
ArrayType = "ARRAY" expression "OF" type.
FieldList = [IdentList ":" type].
RecordType = "RECORD" FieldList {";" FieldList} "END".
type = ident | ArrayType | RecordType.
FPSection = ["VAR"] IdentList ":" type.
FormalParameters = "(" [FPSection {";" FPSection}] ")".
ProcedureHeading = "PROCEDURE" ident [FormalParameters].
ProcedureBody = declarations ["BEGIN" StatementSequence]
  "END" ident.
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody.
declarations = ["CONST" {ident "=" expression ";"}]
  ["TYPE" {ident "=" type ";"}]
  ["VAR" {IdentList ":" type ";"}]
  {ProcedureDeclaration ";"}.
module = "MODULE" ident ";" declarations
  ["BEGIN" StatementSequence] "END" ident ".".
```

The following example of a module may help the reader to appreciate the character of the language. The module contains various, well-known sample procedures whose names are self-explanatory.


```
MODULE Sample;

PROCEDURE Multiply;
  VAR x, y, z: INTEGER;
BEGIN Read(x); Read(y); z := 0;
  WHILE x > 0 DO
    IF x MOD 2 = 1 THEN z := z + y END;
    y := 2*y; x := x DIV 2
  END;
  Write(x); Write(y); Write(z); WriteLn
END Multiply;

PROCEDURE Divide;
  VAR x, y, r, q, w: INTEGER;
BEGIN Read(x); Read(y); r := x; q := 0; w := y;
  WHILE w <= r DO w := 2*w END;
  WHILE w > y DO
    q := 2*q; w := w DIV 2;
    IF w <= r THEN r := r – w; q := q + 1 END
  END;
  Write(x); Write(y); Write(q); Write(r); WriteLn
END Divide;

PROCEDURE BinSearch;
  VAR i, j, k, n, x: INTEGER;
    a: ARRAY 32 OF INTEGER;
BEGIN Read(n); k := 0;
  WHILE k < n DO Read(a[k]); k := k + 1 END;
  Read(x); i := 0; j := n;
  WHILE i < j DO
    k := (i+j) DIV 2;
    IF x < a[k] THEN j := k ELSE i := k+1 END
  END;
  Write(i); Write(j); Write(a[j]); WriteLn
END BinSearch;

END Sample.
```

EXERCISE

6.1 Determine the code for the computer defined in Chapter 9, generated from the program listed at the end of this chapter.

Chapter 7
A Parser for Oberon-0

7.1 The scanner

Before starting to develop a parser, we first turn our attention to the design of its scanner. The scanner has to recognize terminal symbols in the source text. First, we list its vocabulary:

```
* DIV MOD & + - OR
= # < <= > >= . , : ) ]
OF THEN DO ( [ ~ := ;
END ELSE ELSIF IF WHILE
ARRAY RECORD CONST TYPE VAR
PROCEDURE BEGIN MODULE
```

The words written in upper-case letters represent single, terminal symbols, and they are called *reserved words.* They must be recognized by the scanner, and therefore cannot be used as identifiers. In addition to the symbols listed, identifiers and numbers are also treated as terminal symbols. Therefore the scanner is also responsible for recognizing identifers and numbers.

It is appropriate to formulate the scanner as a module. In fact, scanners are a classic example of the use of the module concept. It allows certain details to be hidden from the client, the parser, and to make accessible (to export) only those features which are relevant to the client. The exported facilities are summarized in terms of the module's interface definition:

```
DEFINITION OSS;  (*Oberon subset scanner*)
  IMPORT Texts;
  CONST IdLen = 16;
    (*symbols*) null = 0;
    times = 1; div = 3; mod = 4; and = 5; plus = 6; minus = 7; or = 8;
    eql = 9; neq = 10; lss = 11; geq = 12; leq = 13; gtr = 14;
    period = 18; comma = 19; colon = 20; rparen = 22; rbrak = 23;
    of = 25; then = 26; do = 27;
    lparen = 29; lbrak = 30; not = 32; becomes = 33;
```



```
    number = 34; ident = 37;
    semicolon = 38; end = 40; else = 41; elsif = 42;
    if = 44; while = 46;
    array = 54; record = 55;
    const = 57; type = 58; var = 59;
    procedure = 60; begin = 61; module = 63; eof = 64;

  TYPE Ident = ARRAY IdLen OF CHAR;

  VAR val: LONGINT;
    id: Ident;
    error: BOOLEAN;

  PROCEDURE Mark(msg: ARRAY OF CHAR);
  PROCEDURE Get(VAR sym: INTEGER);
  PROCEDURE Init(T: Texts.Text; pos: LONGINT);
END OSS.
```

The symbols are mapped onto integers. The mapping is defined by a set of constant definitions. Procedure Mark serves to output diagnostics about errors discovered in the source text. Typically, a short explanation is written into a log text together with the position of the discovered error. Procedure Get represents the actual scanner. It delivers for each call the next symbol recognized. The procedure performs the following tasks (the complete listing is shown in Appendix C):

(1) Blanks and line ends are skipped.

(2) Reserved words, such as BEGIN and END, are recognized.

(3) Sequences of letters and digits starting with a letter, which are not reserved words, are recognized as identifiers. The parameter sym is given the value ident, and the character sequence itself is assigned to the global variable id.

(4) Sequences of digits are recognized as numbers. The parameter sym is given the value number, and the number itself is assigned to the global variable val.

(5) Combinations of special characters, such as := and <=, are recognized as a symbol.

(6) Comments, represented by sequences of arbitrary characters beginning with (* and ending with *), are skipped.

(7) The symbol null is returned, if the scanner reads an illegal character (such as $ or %). The symbol eof is returned if the end of the text is reached. Neither of these symbols occurs in a well-formed program text.

7.2 The parser

The construction of the parser follows strictly the rules explained in Chapters 3 and 4. However, before the construction is undertaken, it is necessary to check

whether the syntax satisfies the restricting rules guaranteeing determinism with a lookahead of one symbol. For this purpose, we first construct the sets of start and follow symbols. They are listed in the following tables.

S	*First(S)*	*Possibly empty*
selector	. [	*
factor	(~ integer ident	
term	(~ integer ident	
SimpleExpression	+ − (~ integer ident	
expression	+ − (~ integer ident	
assignment	ident	
ProcedureCall	ident	
statement	ident IF WHILE	*
StatementSequence	ident IF WHILE	*
FieldList	ident	*
type	ident ARRAY RECORD	
FPSection	ident VAR	
FormalParameters	(	
ProcedureHeading	PROCEDURE	
ProcedureBody	END CONST TYPE VAR PROCEDURE BEGIN	
ProcedureDeclaration	PROCEDURE	
declarations	CONST TYPE VAR PROCEDURE	*
module	MODULE	

S	*Follow(S)*
selector	* DIV MOD + − = # < <= > >= ,)] OF THEN DO ; END ELSE ELSIF
factor	* DIV MOD + − = # < <= > >= ,)] OF THEN DO ; END ELSE ELSIF
term	+ − = # < <= > >= ,)] OF THEN DO ; END ELSE ELSIF
SimpleExpression	= # < <= > >= ,)] OF THEN DO ; END ELSE ELSIF
expression	,)] OF THEN DO ; END ELSE ELSIF
assignment	; END ELSE ELSIF
ProcedureCall	; END ELSE ELSIF
statement	; END ELSE ELSIF
StatementSequence	END ELSE ELSIF
FieldList	; END
type	) ;
FPSection	) ;
FormalParameters	) ;

ProcedureHeading	;
ProcedureBody	ident
ProcedureDeclaration	;
declarations	END BEGIN

The subsequent checks of the rules for determinism show that this syntax of Oberon-0 may indeed be handled by the method of recursive descent using a lookahead of one symbol. A procedure is constructed corresponding to each nonterminal symbol. Before the procedures are formulated, it is useful to investigate how they depend on each other. For this purpose we design a *dependence graph* (Figure 7.1). Every procedure is represented as a node, and an edge is drawn to all nodes on which the procedure depends, that is, calls directly or indirectly. Note that some nonterminal symbols do not occur in this graph, because they are included in other symbols in a trivial way. For example, ArrayType and RecordType are contained in type only and are therefore not explicitly drawn. Furthermore we recall that the symbols ident and integer occur as terminal symbols, because they are treated as such by the scanner.

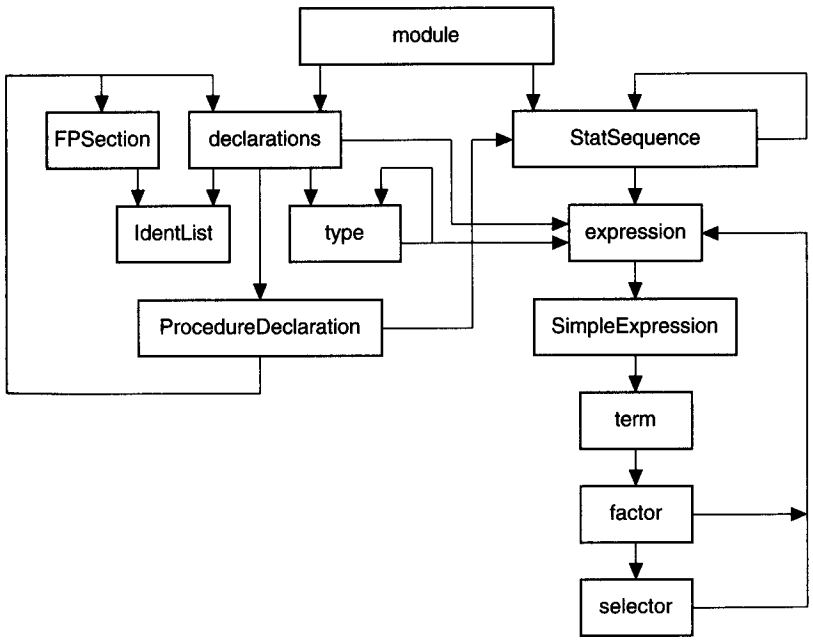


Figure 7.1 Dependence diagram of parsing procedures.

Every loop in the diagram corresponds to a recursion. It is evident that the parser must be formulated in a language that allows recursive procedures. Furthermore, the diagram reveals how procedures may possibly be nested. The only procedure

which is not called by another procedure is Module. The structure of the program mirrors this diagram. The program is listed in Appendix C in an already augmented form. The parser, like the scanner, is also formulated as a module.

7.3 Coping with syntactic errors

So far we have considered only the rather simple task of determining whether or not a source text is well formed according to the underlying syntax. As a side-effect, the parser also recognizes the structure of the text read. As soon as an unacceptable symbol turns up, the task of the parser is completed, and the process of syntax analysis is terminated. For practical applications, however, this proposition is unacceptable. A genuine compiler must indicate an error diagnostic message and thereafter proceed with the analysis. It is then quite likely that further errors will be detected. Continuation of parsing after an error detection is, however, possible only under the assumption of certain hypotheses about the nature of the error. Depending on this assumption, a part of the subsequent text must be skipped, or certain symbols must be inserted. Such measures are necessary even when there is no intention of correcting or executing the erroneous source program. Without an at least partially correct hypothesis, continuation of the parsing process is futile (Graham and Rhodes, 1975; Rechenberg and Mössenböck, 1985).

The technique of choosing good hypotheses is complicated. It ultimately rests upon heuristics, as the problem has so far eluded formal treatment. The principal reason for this is that the formal syntax ignores factors which are essential for the human recognition of a sentence. For instance, a missing punctuation symbol is a frequent mistake, not only in program texts, but an operator symbol is seldom omitted in an arithmetic expression. To a parser, however, both kinds of symbols are syntactic symbols without distinction, whereas to the programmer the semicolon appears as almost redundant, and a plus symbol as the essence of the expression. This kind of difference must be taken into account if errors are to be treated sensibly. To summarize, we postulate the following quality criteria for error handling:

(1) As many errors as possible should be detected in a single scan through the text.

(2) As few additional assumptions as possible about the language should be made.

(3) Error handling features should not slow down the parser appreciably.

(4) The parser program should not grow in size significantly.

We can conclude that error handling strongly depends on a concrete case, and that it can be described by general rules only with limited success. Nevertheless, there are a few heuristic rules which seem to have relevance beyond our specific

language, Oberon. Notably, they concern the design of a language just as much as the technique of error treatment. Without doubt, a simple language structure significantly simplifies error diagnostics, or, in other words, a complicated syntax complicates error handling unnecessarily.

Let us differentiate between two cases of incorrect text. The first case is where symbols are *missing*. This is relatively easy to handle. The parser, recognizing the situation, proceeds by omitting one or several calls to the scanner. An example is the statement at the end of factor, where a closing parenthesis is expected. If it is missing, parsing is resumed after emitting an error message:

```
IF sym = rparen THEN Get(sym) ELSE Mark(" ) missing") END
```

Virtually without exception, only *weak symbols* are omitted, symbols which are primarily of a syntactic nature, such as the comma, semicolon and closing symbols. A case of *wrong* usage is an equality sign instead of an assignment operator, which is also easily handled.

The second case is where wrong symbols are present. Here it is unavoidable to skip them and to resume parsing at a later point in the text. In order to facilitate resumption, Oberon features certain constructs beginning with distinguished symbols which, by their nature, are rarely misused. For example, a declaration sequence always begins with the symbol CONST, TYPE, VAR, or PROCEDURE, and a structured statement always begins with IF, WHILE, REPEAT, CASE, and so on. Such *strong symbols* are therefore never skipped. They serve as *synchronization points* in the text, where parsing can be resumed with a high probability of success. In Oberon's syntax, we establish four synchronization points, namely in factor, statement, declarations and type. At the beginning of the corresponding parser procedures symbols might be skipped. The process is resumed when either a correct start symbol or a strong symbol is read.

```
PROCEDURE factor;
BEGIN (*sync*)
  IF sym < lparen THEN Mark("ident?");
    REPEAT Get(sym) UNTIL sym >= lparen
  END;
  ...
END factor;

PROCEDURE StatSequence;
BEGIN (*sync*)
  IF sym < ident THEN Mark("Statement?");
    REPEAT Get(sym) UNTIL sym >= ident
  END;
  ...
END StatSequence;
```

```
PROCEDURE Type;
BEGIN (*sync*)
  IF (sym # ident) & (sym >= const) THEN Mark("type?");
    REPEAT Get(sym) UNTIL (sym = ident) OR (sym >= array)
  END;
  ...
END Type;

PROCEDURE declarations;
BEGIN (*sync*)
  IF sym < const THEN Mark("declaration?");
    REPEAT Get(sym) UNTIL sym >= const
  END;
  ...
END declarations;
```

Evidently, a certain ordering among symbols is assumed at this point. This ordering has been chosen such that the symbols are grouped to allow simple and efficient range tests. Strong symbols not to be skipped are assigned a high ranking (ordinal number) as shown in the definition of the scanner's interface.

In general, the rule holds that the parser program is derived from the syntax according to the recursive descent method and the explained translation rules. If a symbol read does not meet expectations, an error is indicated by a call of procedure Mark, and analysis is resumed at the next synchronization point. Frequently, follow-up errors are diagnosed, whose indication may be omitted, because they are merely consequences of a formerly indicated error. The statement which results for every synchronization point can be formulated generally as follows:

```
IF ~(sym IN follow(SYNC)) THEN Mark(msg);
  REPEAT Get(sym) UNTIL sym IN follow(SYNC)
END
```

where follow(SYNC) denotes the set of symbols which may correctly occur at this point.

In certain cases it is advantageous to depart from the statement derived by this method. An example is the construct of statement sequence. Instead of

```
Statement;
WHILE sym = semicolon DO Get(sym); Statement END
```

we use the formulation

```
LOOP (*sync*)
  IF sym < ident THEN Mark("ident?"); ... END;
  Statement;
  IF sym = semicolon THEN Get(sym)
  ELSIF sym IN follow(StatSequence) THEN EXIT
  ELSE Mark("semicolon?")
  END
END
```


This replaces the two calls of Statement by a single call, whereby this call may be replaced by the procedure body itself, making it unnecessary to declare an explicit procedure. The two tests after Statement correspond to the legal cases where, after reading the semicolon, either the next statement is analysed or the sequence terminates. Instead of the condition sym IN follow(StatSequence) we use a Boolean expression which again makes use of the specifically chosen ordering of symbols:

```
(sym >= semicolon) & (sym < if) OR (sym >= array)
```

The construct above is an example of the general case where a sequence of identical subconstructs which may be empty (in this case, statements) are separated by a weak symbol (here, semicolon). A second, similar case is manifest in the parameter list of procedure calls. The statement

```
IF sym = lparen THEN
  Get(sym); expression;
  WHILE sym = comma DO Get(sym); expression END;
  IF sym = rparen THEN Get(sym) ELSE Mark(") ?") END
END
```

is replaced by

```
IF sym = lparen THEN Get(sym);
  LOOP expression;
    IF sym = comma THEN Get(sym)
    ELSIF (sym = rparen) OR (sym >= semicolon) THEN EXIT
    ELSE Mark(") or , ?")
    END
  END;
  IF sym = rparen THEN Get(sym) ELSE Mark(") ?") END
END
```

A further case of this kind is the declaration sequence. Instead of

```
IF sym = const THEN ... END;
IF sym = type THEN ... END;
IF sym = var THEN ... END;
```

we employ the more liberal formulation

```
LOOP
  IF sym = const THEN ... END;
  IF sym = type THEN ... END;
  IF sym = var THEN ... END;
  IF (sym >= const) & (sym <= var) THEN
    Mark("bad declaration sequence")
  ELSE EXIT
  END
END
```

The reason for deviating from the previously given method is that declarations in a wrong order (for example variables before constants) must provoke an error message, but at the same time can be parsed individually without difficulty. A further, similar case can be found in Type. In all these cases, it is absolutely mandatory to ensure that the parser can never get caught in the loop. The easiest way to achieve this is to make sure that in each repetition at least one symbol is being read, that is, that each path contains at least one call of Get. Thereby, in the worst case, the parser reaches the end of the source text and stops. We refer to the listing in Appendix C for further details.

It should now have become clear that there is no such thing as a perfect error-handling strategy which will translate all correct sentences with great efficiency and also sensibly diagnose all errors in ill-formed texts. Every strategy will handle certain abstruse sentences in a way that appears unexpected to its author. The essential characteristics of a good compiler, regardless of details, are that (1) no sequence of symbols leads to its crash, and (2) frequently encountered errors are correctly diagnosed and subsequently generate no, or few additional, spurious error messages. The strategy presented here operates satisfactorily, although it is possible to improve it. The strategy is remarkable in the sense that the error-handling parser is derived according to a few, simple rules from the straight parser. The rules are augmented by the judicious choice of a few parameters which are determined by ample experience in the use of the langauge.

EXERCISES

7.1 The scanner of the compiler listed in Appendix C uses a linear search of array KeyTab to determine whether or not a sequence of letters is a key word. As this search occurs very frequently, an improved search method would certainly result in increased efficiency. Replace the linear search in the array by

(1) A binary search in an ordered array.

(2) A search in a binary tree.

(3) A search of a hash table. Choose the hash function so that at most two comparisons are necessary to find out whether or not the letter sequence is a key word.

Determine the overall gain in compilation speed for the three solutions.

7.2 Where is the Oberon syntax not LL(1), that is, where is a lookahead of more than one symbol necessary? Change the syntax in such a way that it satisfies the LL(1) property.

7.3 Extend the scanner in such a way that it accepts real numbers as specified by the Oberon syntax (see Appendix A).

Chapter 8
Consideration of Context Specified by Declarations

8.1 Declarations

Although programming languages are based on context-free languages in the sense of Chomsky, they are by no means context free in the ordinary sense of the term. The context sensitivity is manifest in the fact that every identifier in a program must be declared. Thereby it is associated with an object of the computing process which carries certain permanent properties. For example, an identifier is associated with a variable, and this variable has a specific data type as specified in the identifier's declaration. An identifier occurring in a statement refers to the object specified in its declaration, and this declaration lies outside the statement. We say that the declaration lies in the *context* of the statement.

Consideration of context evidently lies beyond the capability of context-free parsing. In spite of this, it is easily handled. The context is represented by a data structure which contains an entry for every declared identifier. This entry associates the identifier with the denoted object and its properties. The data structure is known by the name *symbol table.* This term dates back to the time of assemblers, when identifiers were called *symbols.* However, the structure is typically more complex than a simple array.

The parser will now be extended in such a way that, when parsing a declaration, the symbol table is suitably augmented. An entry is inserted for every declared identifier. To summarize:

- Every declaration results in a new symbol table entry.
- Every occurrence of an identifier in a statement requires a search of the symbol table in order to determine the attributes (properties) of the object denoted by the identifier.

A typical attribute is the object's *class.* It indicates whether the identifier denotes a constant, a variable, a type or a procedure. A further attribute in all languages with data types is the object's *type.*

The simplest form of data structure for representing a set of items is the list. Its major disadvantage is a relatively slow search process, because it has to be traversed from its root to the desired element. For the sake of simplicity – data

structures are not the topic of this text – we declare the following data types representing linear lists:

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD
  name: Ident;  class: INTEGER;
  type: Type;  next: Object;
  val: LONGINT
END;
```

The following declarations are, for example, represented by the list shown in Figure 8.1.

```
CONST N = 10;
TYPE T = ARRAY N OF INTEGER;
VAR x, y: T
```

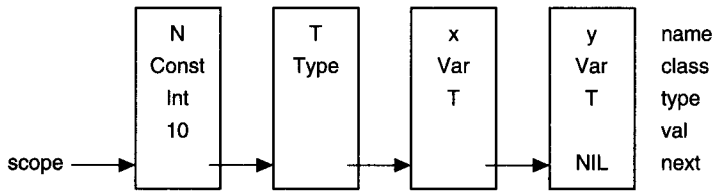


Figure 8.1 Symbol table representing objects with names and attributes.

For the generation of new entries we introduce the procedure NewObj with the explicit parameter class, the implied parameter id and the result obj. The procedure checks whether the new identifier (id) is already present in the list. This would signify a multiple definition and constitute a programming error. The new entry is appended at the end of the list, so that the list mirrors the order of the declarations in the source text. The end of the list is marked by a guard element (sentinel) to which the new identifier is assigned before the list traversal starts. This measure simplifies the termination condition of the while-statement.

```
PROCEDURE NewObj(VAR obj: Object; class: INTEGER);
  VAR new, x: Object;
BEGIN x := origin; guard.name := id;
  WHILE x.next.name # id DO x := x.next END;
  IF x.next = guard THEN
    NEW(new); new.name := id; new.class := class; new.next := guard;
    x.next := new; obj := new
  ELSE obj := x.next; Mark("multiple declaration")
  END
END NewObj
```

In order to speed up the search process, the list is often replaced by a tree structure. Its advantage becomes noticeable only with a fairly large number of entries. For structured languages with local scopes, that is, ranges of visibility of identifiers, the symbol table must be structured accordingly, and the number of entries in each scope becomes relatively small. Experience shows that as a result the tree structure yields no substantial benefit over the list, although it requires a more complicated search process and the presence of three successor pointers per entry instead of one. Note that the linear ordering of entries must also be recorded, because it is significant in the case of procedure parameters.

8.2 Entries for data types

In languages featuring data types, their consistency checking is one of the most important tasks of a compiler. The checks are based on the type attribute recorded in every symbol table entry. Since data types themselves can be declared, a pointer to the respective type entry appears to be the obvious solution. However, types may also be specified anonymously, as exemplified by the following declaration:

```
VAR a: ARRAY 10 OF INTEGER
```

The type of variable a has no name. An easy solution to the problem is to introduce a proper data type in the compiler to represent types as such. Named types then are represented in the symbol table by an entry of type Object, which in turn refers to an element of type Type.

```
Type = POINTER TO TypDesc;
TypDesc = RECORD
  form, len: INTEGER;
  fields: Object;
  base: Type
END
```

The attribute form differentiates between elementary types (INTEGER, BOOLEAN) and structured types (arrays, records). Further attributes are added according to the individual forms. Characteristic for arrays are their length (number of elements) and the element type (base). For records, a list representing the fields must be provided. Its elements are of the class Field. As an example, Figure 8.2. shows the symbol table resulting from the following declarations:

```
TYPE R = RECORD f, g: INTEGER END;
VAR x: INTEGER;
  a: ARRAY 10 OF INTEGER;
  r, s: R;
```

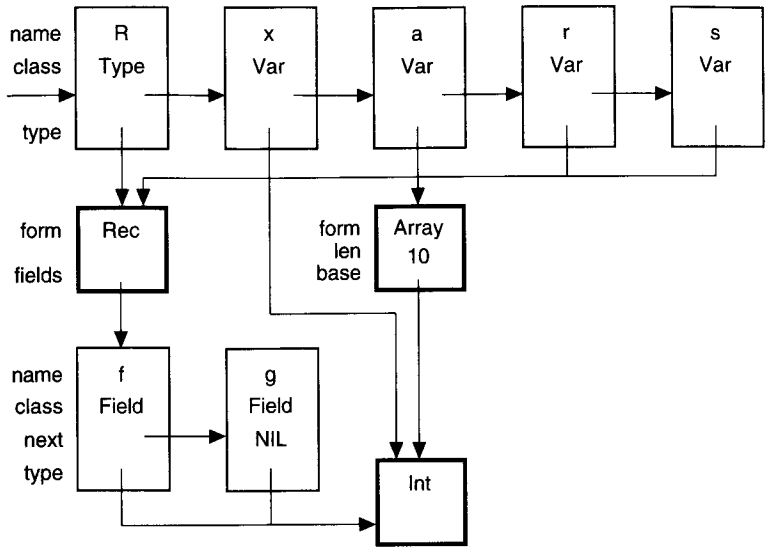



Figure 8.2 Symbol table representing declared objects.

As far as programming methodology is concerned, it would be preferable to introduce an extended data type for each class of objects, using a base type with the fields id, type and next only. We refrain from doing so, not least because all such types would be declared within the same module, and because the use of a numeric discrimination value (class) instead of individual types avoids the need for numerous, redundant type guards and thereby increases efficiency. After all, we do not wish to promote an undue proliferation of data types.

8.3 Data representation at run time

So far, all aspects of the target computer and its architecture, that is, of the computer for which code is to be generated, have been ignored, because our sole task was to recognize source text and to check its compliance with the syntax. However, as soon as the parser is extended into a compiler, knowledge about the target computer becomes mandatory.

First, we must determine the format in which data are to be represented at run time in the store. The choice inherently depends on the target architecture, although this fact is less apparent because of the similarity of virtually all computers in this respect. Here, we refer to the generally accepted form of the store as a sequence of individually addressable byte cells, that is, of *byte-oriented* memories. Consecutively declared variables are then allocated with monotonically increasing or decreasing addresses. This is called *sequential allocation.*

Every computer features certain elementary data types together with corresponding instructions, such as integer addition and floating-point addition. These types are invariably scalar types, and they occupy a small number of consecutive memory locations (bytes). An example of an architecture with a fairly rich set of types is National Semiconductor's family of NS32000 processors:

Data type	*Number of bytes*	*Data type*	*Number of bytes*
INTEGER	2	LONGREAL	8
LONGINT	4	CHAR	1
SHORTINT	1	BOOLEAN	1
REAL	4	SET	4

From the foregoing we conclude the following:

- Every type has a *size.*
- Every variable has an *address.*

These attributes, type.size and obj.adr, are determined when the compiler processes declarations. The sizes of the elementary types are given by the machine architecture, and corresponding entries are generated when the compiler is loaded and initialized. For structured, declared types, their size has to be computed.

The size of an array is its element size multiplied by the number of its elements. The address of an element is the sum of the array's address and the element's index multiplied by the element size. Let the following general declarations be given:

TYPE T = ARRAY n OF T_0
VAR a: T

Then type size and element address are obtained from the following equations:

$$size(T) = n * size(T_0)$$
$$adr(a[x]) = adr(a) + x * size(T_0)$$

For multidimensional arrays, the corresponding formulas (see Figure 8.3) are:

TYPE T = ARRAY $n_{k-1}, \ldots, n_1, n_0$ OF T_0

$$size(T) = n_{k-1} * \ldots * n_1 * n_0 * size(T_0)$$
$$adr(a[x_{k-1}, \ldots, x_1, x_0]) = adr(a) + x_{k-1} * n_{k-2} * \ldots * n_0 * size(T_0) + \ldots + x_2 * n_1 * n_0 * size(T_0) + x_1 * n_0 * size(T_0) + x_0 * size(T_0) = adr(a) + (((\ldots x_{k-1} * n_{k-2} + \ldots + x_2) * n_1 + x_1) * n_0 + x_0) * size(T_0)$$

(Horner schema)

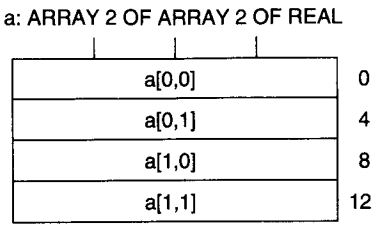


Figure 8.3 Representation of a matrix.

Note that for the computation of the size the array's lengths in all dimensions are known, because they occur as constants in the program text. However, the index values needed for the computation of an element's address are typically not known before program execution.

In contrast, for record structures, both type size and field address are known at compile time. Let us consider the following declarations:

TYPE T = RECORD f_0: T_0; f_1: T_1; ... ; f_{k-1}: T_{k-1} END
VAR r: T

Then the type's size and the field addresses are computed according to the following formulas:

$$size(T) = size(T_0) + \ldots + size(T_{k-1})$$
$$adr(r.f_i) = adr(r) + offset(f_i)$$
$$offset(f_i) = size(T_0) + \ldots + size(T_{i-1})$$

Absolute addresses of variables are usually unknown at the time of compilation. All generated addresses must be considered as *relative* to a common *base address* which is given at run time. The effective address is then the sum of this base address and the address determined by the compiler.

If a computer's store is byte-addressed, as is fairly common, a further point must be considered. Although bytes can be accessed individually, typically a small number of bytes (say 4 or 8) are transferred from or to memory as a packet, a so-called *word.* If allocation occurs strictly in sequential order it is possible that a variable may occupy (parts of) several words (see Figure 8.4). But this should definitely be avoided, because otherwise a variable access would involve several memory accesses, resulting in an appreciable slowdown. A simple method of overcoming this problem is to round up (or down) each variable's address to the next multiple of its size. This process is called *alignment.* The rule holds for elementary data types. For arrays, the size of their element type is relevant, and for records we simply round up to the computer's word size. The price of alignment is the loss of some bytes in memory, which is quite negligible.

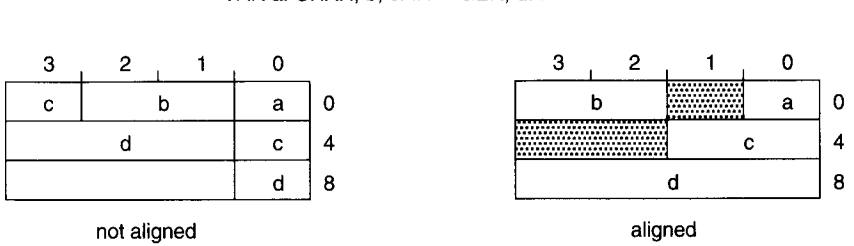


Figure 8.4 Alignment in address computation.

The following additions to the parsing procedure for declarations are necessary to generate the required symbol table entries:

```
IF sym = type THEN (* "TYPE" ident "=" type *)
  Get(sym);
  WHILE sym = ident DO
    NewObj(obj, Typ); Get(sym);
    IF sym = eql THEN Get(sym) ELSE Mark("= ?") END;
    Type1(obj.type);
    IF sym = semicolon THEN Get(sym) ELSE Mark("; ?") END
    END
END;

IF sym = var THEN  (* "VAR" ident {"," ident} ":" type *)
  Get(sym);
  WHILE sym = ident DO
    IdentList(Var, first); Type1(tp); obj := first;
    WHILE obj # guard DO
      obj.type := tp; INC(adr, obj.type.size); obj.val := –adr; obj := obj.next
    END;
    IF sym = semicolon THEN Get(sym) ELSE Mark("; ?") END
  END
END;
```

Here, procedure IdentList is used to process an identifier list, and the recursive procedure Type1 serves to compile a type declaration.

```
PROCEDURE IdentList(class: INTEGER; VAR first: Object);
  VAR obj: Object;
BEGIN
  IF sym = ident THEN
    NewObj(first, class); Get(sym);
    WHILE sym = comma DO
      Get(sym);
```



```
    IF sym = ident THEN NewObj(obj, class); Get(sym)
    ELSE Mark("ident?")
    END
  END;
  IF sym = colon THEN Get(sym) ELSE Mark(":?") END
END
END IdentList;

PROCEDURE Type1(VAR type: Type);
  VAR n: INTEGER;
    obj, first: Object; tp: Type;
BEGIN type := intType; (*sync*)
  IF (sym # ident) & (sym < array) THEN Mark("ident?");
    REPEAT Get(sym) UNTIL (sym = ident) OR (sym >= array)
  END;
  IF sym = ident THEN
    find(obj); Get(sym);
    IF obj.class = Typ THEN type := obj.type ELSE Mark("type?") END
  ELSIF sym = array THEN
    Get(sym);
    IF sym = number THEN n := val; Get(sym)
    ELSE Mark("number?"); n := 1
    END;
    IF sym = of THEN Get(sym) ELSE Mark("OF?") END;
    Type1(tp); NEW(type); type.form := Array; type.base := tp;
    type.len := n; type.size := type.len * tp.size
  ELSIF sym = record THEN
    Get(sym); NEW(type); type.form := Record; type.size := 0; OpenScope;
    LOOP
      IF sym = ident THEN
        IdentList(Fld, first); Type1(tp); obj := first;
        WHILE obj # guard DO
          obj.type := tp; obj.val := type.size; INC(type.size, obj.type.size);
          obj := obj.next
        END
      END;
      IF sym = semicolon THEN Get(sym)
      ELSIF sym = ident THEN Mark("; ?")
      ELSE EXIT
      END
    END;
    type.fields := topScope.next; CloseScope;
    IF sym = end THEN Get(sym) ELSE Mark("END?") END
  ELSE Mark("ident?")
  END
END Type1;
```

Following a longstanding tradition, addresses of variables are assigned negative values, that is, negative offsets to the common base address determined during program execution. The auxiliary procedures OpenScope and CloseScope ensure that the list of record fields is not intermixed with the list of variables. Every record declaration establishes a new scope of visibility of field identifiers, as required by the definition of the language Oberon. Note that the list into which new entries are inserted is rooted in the global variable topScope.

EXERCISES

8.1 The scope of identifiers is defined to extend from the place of declaration to the end of the procedure in which the declaration occurs. What would be necessary to let this range extend from the beginning to the end of the procedure?

8.2 Consider pointer declarations as defined in Oberon. They specify a type to which the declared pointer is bound, and this type may occur later in the text. What is necessary to accommodate this relaxation of the rule that all referenced entities must be declared prior to their use?

Chapter 9
A RISC Architecture as Target

It is worth noting that our compiler, up to this point, could largely be developed without reference to the target computer for which it is to generate code. But why indeed should the target machine's structure influence syntactic analysis and error handling? On the contrary, such an influence should consciously be avoided. As a result, code generation for an arbitrary computer may be added according to the principle of stepwise refinement to the existing, machine independent parser, which serves like a scaffolding. Before undertaking this task, however, a specific target architecture must be selected.

To keep both the resulting compiler reasonably simple and the development clear of details that are of relevance only for a specific machine and its idiosyncrasies, we postulate an architecture according to our own choice. Thereby we gain the considerable advantage that it can be tailored to the needs of the source language. This architecture does not exist as a real machine; it is therefore *virtual.* But since every computer executes instructions according to a fixed algorithm, it can easily be specified by a program. A real computer may then be used to execute this program which interprets the generated code. The program is called an *interpreter,* and it *emulates* the virtual machine, which therefore can be said to have a semi-real existence.

It is not the aim of this text to present the motivations for the choice of the specific virtual architecture with all its details. This chapter is rather intended to serve as a descriptive manual consisting of an informal introduction and a formal definition of the computer in the form of the interpretive program. This formalization may even be considered as an example of an exact, algorithmic specification of a processor.

In the definition of this computer we intentionally follow closely the line of RISC architectures. The acronym RISC stands for *reduced instruction set computer,* where 'reduced' is to be understood as relative to architectures with large sets of complex instructions, as these were dominant until about 1980. This is obviously not the place to explain the essence of the RISC architecture, nor to expound on its various advantages. Here it is obviously attractive because of its simplicity and clarity of concepts, which simplify the description of the instruction set and the choice of instruction sequences corresponding to specific language constructs. The architecture chosen here is almost identical to the one presented by Hennessy and Patterson (1990) under the name DLX. The small deviations are

due to our desire for increased regularity. Among commercial products, the MIPS architecture is closest to our virtual machine.

Resources and registers

From the viewpoints of the programmer and the compiler designer the computer consists of an arithmetic unit, a control unit and a store. The arithmetic unit contains 32 registers R0–R31, with 32 bits each. R0 is considered as always holding the value 0. The control unit consists of the instruction register, IR, holding the instruction currently being executed, and the program counter, PC, holding the address of the instruction to be fetched next (Figure 9.1). Branch instructions to subroutines implicitly use register R31 to store the return address. The memory consists of 32-bit words, and it is byte-addressed, that is, word addresses are multiples of 4.

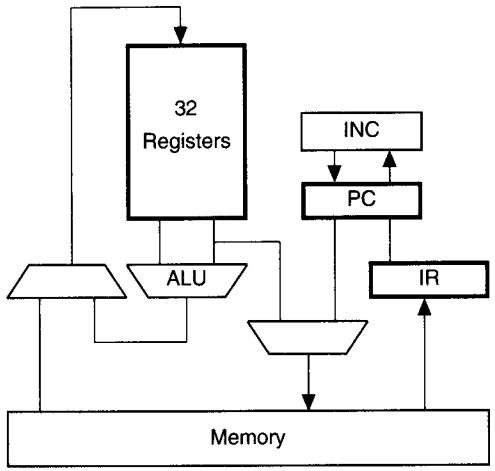


Figure 9.1 Schematic of the RISC structure.

Instruction formats

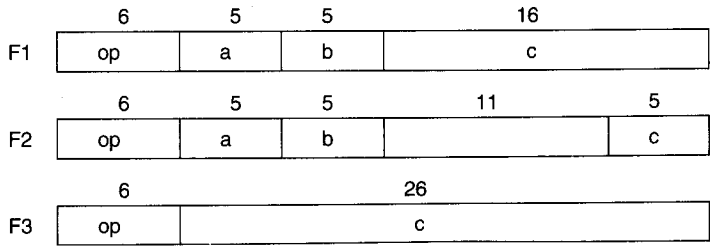


Figure 9.2 Instruction formats.

Register instructions (formats F1 and F2)

ADD	a, b, c	R.a := R.b + R.c	ADDI	a, b, c	R.a := R.b + c
SUB	a, b, c	R.a := R.b – R.c	SUBI	a, b, c	R.a := R.b – c
MUL	a, b, c	R.a := R.b * R.c	MULI	a, b, c	R.a := R.b * c
DIV	a, b, c	R.a := R.b DIV R.c	DIVI	a, b, c	R.a := R.b DIV c
MOD	a, b, c	R.a := R.b MOD R.c	MODI	a, b, c	R.a := R.b MOD c
CMP	a, b, c	R.a := R.b – R.c	CMPI	a, b, c	R.a := R.b – c
CHK	a, c	0 <= R.a < R.c	CHKI	a, c	0 <= R.a < c
AND	a, b, c	R.a := R.b & R.c	ANDI	a, b, c	R.a := R.b & c
BIC	a, b, c	R.a := R.b & ~R.c	BICI	a, b, c	R.a := R.b & ~c
OR	a, b, c	R.a := R.b OR R.c	ORI	a, b, c	R.a := R.b OR c
XOR	a, b, c	R.a := R.b XOR R.c	XORI	a, b, c	R.a := R.b XOR c
LSH	a, b, c	R.a := LSH(R.b, R.c)	LSHI	a, b, c	R.a := LSH(R.b, c)
ASH	a, b, c	R.a := ASH(R.b, R.c)	ASHI	a, b, c	R.a := ASH(R.b, c)

The 16-bit field c is sign extended to 32 bits. In the case of LSH (logical shift) and ASH (arithmetic shift) a positive count (R.c or c) denotes a shift to the left, a negative count a shift to the right. The SUB and CMP instructions both form a difference. They behave differently when the result is outside the range $-2^{31} \leq x-y < 2^{31}$. Then SUB indicates arithmetic overflow, whereas CMP yields a result with the wrong magnitude but with the correct sign.

Load/store instructions (format F1)

LDW	a, b, c	R.a := Mem[R.b + c]	load word
LDB	a, b, c	R.a := Mem[R.b + c]	load byte
POP	a, b, c	R.a := Mem[R.b]; R.b := R.b + c	pop stack
STW	a, b, c	Mem[R.b + c] := R.a	store word
STB	a, b, c	Mem[R.b + c] := R.a	store byte
PSH	a, b, c	R.b := R.b – c; Mem[R.b] := R.a	push stack

Control instructions (Format F1, address PC-relative)

BEQ	a, c	Branch to c if R.a = 0
BNE	a, c	Branch to c if R.a # 0
BLT	a, c	Branch to c if R.a < 0
BGE	a, c	Branch to c if R.a >= 0
BGT	a, c	Branch to c if R.a > 0
BLE	a, c	Branch to c if R.a <= 0
BSR	c	Save PC in R31, then branch to c (format F1, address PC-relative)

Additional notes

(1) Instructions RD, WRD, WRH and WRL are not typical computer instructions. We have added them here to provide a simple and effective way for input and output. Compiled and interpreted programs can thus be tested and obtain a certain reality.

(2) Instructions LDB and STB load and store a single byte. Without them, it would not make sense to speak about a byte-oriented computer. However, we refrain from specifying them here, because such program statements would hardly mirror their implementation in hardware truthfully.

(3) Instructions PSH and POP behave like STW and LDW, whereby the value of the base register R.b is incremented or decremented by the amount c. They will allow the handling of procedure parameters (see Chapter 12).

(4) Instructions CHK and CHKI allow the validity of array indices to be tested. In case of failure they generate a trap.

Chapter 10
Expressions and Assignments

10.1 Straight code generation according to the stack principle

The third example in Chapter 5 showed how to convert an expression from conventional infix form into its equivalent postfix form. Our ideal computer would be capable of directly interpreting postfix notation. As also shown, such an ideal computer requires a stack for holding intermediate results. Such a computer architecture is called a *stack architecture.*

Computers based on a stack architecture are not in common use. Sets of explicitly addressable registers are preferred to a stack. Of course, a set of registers can easily be used to emulate a stack. Its top element is indicated by a global variable representing the stack pointer (SP) in the compiler. This is feasible, since the number of intermediate results is known at compile time, and the use of a global variable is justified because the stack constitutes a global resource.

To derive the program for generating the code corresponding to specific constructs, we first postulate the desired code patterns. This method will also be successfully employed later for other constructs beyond expressions and assignments. Let the code for a given construct K be given by the following table:

K	Code(K)	*Side-effect*
ident	LDW i, 0, adr(ident)	INC(SP)
number	ADDI i, 0, value	INC(SP)
(exp)	code(exp)	
fac0 * fac1	code(fac0) code(fac1) MUL i, i, i+1	DEC(SP)
term0 + term1	code(term0) code(term1) ADD i, i, i+1	DEC(SP)
ident := exp	code(exp) STW 1, 0, adr(ident)	DEC(SP)

To begin, we restrict our attention to simple variables as operands, and we omit selectors for structured variables. First, consider the assignment u := x + y*z:

```
LDW   1, 0, x      R1 := x          x          SP = 1
LDW   2, 0, y      R2 := y          x, y           2
LDW   3, 0, z      R3 := z          x, y, z        3
MUL   2, 2, 3      R2 := R2 * R3    x, y*z         2
ADD   1, 1, 2      R1 := R1 + R2    x + y*z        1
STW   1, 0, u      u := R1          –              0
```

From this it is quite evident how the corresponding parser procedures must be extended:

```
PROCEDURE factor;
  VAR obj: Object;
BEGIN
  IF sym = ident THEN
    find(obj); Get(sym); INC(RX); Put(LDW, RX, 0, –obj.val)
  ELSIF sym = number THEN
    INC(RX); Put(ADDI, RX, 0, val); Get(sym)
  ELSIF sym = lparen THEN
    Get(sym); expression;
    IF sym = rparen THEN Get(sym) ELSE Mark(" ) missing") END
  ELSIF ...
  END
END factor;

PROCEDURE term;
  VAR op: INTEGER;
BEGIN factor;
  WHILE (sym = times) OR (sym = div) DO
    op := sym; Get(sym); factor; DEC(RX);
    IF op = times THEN Put(MUL, RX, RX, RX+1)
    ELSIF op = div THEN Put(DIV, RX, RX, RX+1)
    END
  END
END term;

PROCEDURE SimpleExpression;
  VAR op: INTEGER;
BEGIN
  IF sym = plus THEN Get(sym); term
  ELSIF sym = minus THEN Get(sym); term;
    Put(SUB, RX, 0, RX)
  ELSE term
  END;
```



```
  WHILE (sym = plus) OR (sym = minus) DO
    op := sym; Get(sym); term; DEC(RX);
    IF op = plus THEN Put(ADD, RX, RX, RX+1)
    ELSIF op = minus THEN Put(SUB, RX, RX, RX+1)
    END
  END
END SimpleExpression;

PROCEDURE Statement;
  VAR obj: Object;
BEGIN
  IF sym = ident THEN
    find(obj); Get(sym);
    IF sym = becomes THEN
      Get(sym); expression;
      Put(STW, RX, 0, obj.val); DEC(RX)
    ELSIF ...
    END
  ELSIF ...
  END
END Statement;
```

Here we have introduced the generator procedure Put. It can be regarded as the counterpart of the scanner procedure Get. We assume that it deposits an instruction in a global array, using the variable pc as an index denoting the next free location in the array. With this assumption, procedure Put is formulated in Oberon as follows, whereby LSH(x, n) is a function yielding the value x left shifted by n bit positions:

```
PROCEDURE Put(op, a, b, d: INTEGER);
BEGIN
  code[pc] := LSH(LSH(LSH(op, 5) + a, 5) + b, 16) + (d MOD 10000H);
  INC(pc)
END Put
```

Addresses of variables are indicated here by simply using their identifier. In reality, the address values obtained from the symbol table stand in place of the identifiers. They are offsets to a base address computed at run time, that is, the offsets are added to the base address to yield the effective address. This holds not only for our RISC machine, but also for virtually all common computers. We take this fact into account by specifying addresses using pairs consisting of the offset a and the base (register) r. At present we simply assume r = 0, and hence the effective address is equal to the offset (R[0] + a = a, because R[0] = 0 by definition).

10.2 Delayed code generation

Consider as a second example the expression x + 1. According to the scheme presented in Section 10.1, we obtain the corresponding code

```
LDW   1, 0, x        R1 := x
ADDI  2, 0, 1        R2 := 1
ADD   1, 1, 2        R1 := R1 + R2
```

This shows that the generated code is correct, but certainly not optimal. The flaw lies in the fact that the constant 1 is loaded into a register, although this is unnecessary, because our computer features an instruction which lets constants be added immediately to a register (*immediate addressing mode*). Evidently some code has been emitted prematurely. The remedy must be to delay code emission in certain cases until it is definitely known that there is no better solution. How is such a delayed code generation to be implemented?

In general, the method consists in associating the information which would have been used for the selection of the emitted code with the resulting syntactic construct. From the principle of attributed grammars presented in Chapter 5, such information is retained in the form of attributes. Code generation therefore depends not only on the syntactically reduced symbols, but also on the values of their attributes. This conceptual extension is reflected by the fact that parser procedures are extended to include a result parameter which represents these attributes. Because there are usually several attributes, a record structure is chosen for these parameters; we call their type Item (Wirth and Gutknecht, 1992).

In the case of our second example, it is necessary to indicate whether the value of a factor, term or expression is held (at run time) in a register, as has been the case so far, or if the value is simply a known constant. The latter case will quite likely lead to a later instruction with immediate mode. It now becomes clear that the attribute must specify a mode for each factor, term or expression, that is, where the value is stored and how it is to be accessed. This attribute mode corresponds to the addressing mode of computer instructions, and its range of possible values depends on the set of addressing modes which the target computer features. For each addressing mode offered, there is a corresponding item mode. A mode is also implicitly introduced by object classes. Object classes and item modes partially overlap. In the case of our RISC architecture, there are only three modes:

Item mode	*Object class*	*Addressing mode*	*Additional attributes*
Var	Var	Direct	a Value in memory at address a
Const	Const	Immediate	a Value is the constant a
Reg	–	Register	r Value held in register R[r]

With this in mind, we declare the data type Item as a record structure with fields mode, type, a and r. Evidently, the type of the item is also an attribute. It will not be mentioned any further below, because we shall consider only the single type Integer.

The parser procedures now emerge as functions with result type Item. Programming considerations, however, suggest that we should use proper procedures with a result parameter instead of function procedures.

```
Item = RECORD
  mode: INTEGER;
  type: Type;
  a, r: LONGINT;
END
```

Let us now return to our example to demonstrate the generation of code for the expression x+1. The process is shown in Figure 10.1. The transformation of a Var-Item into a Reg-Item is accompanied by the emission of an LDW instruction, and the transformation of a Reg-Item and a Const-Item into a Reg-Item is accompanied by emitting an ADDI instruction.

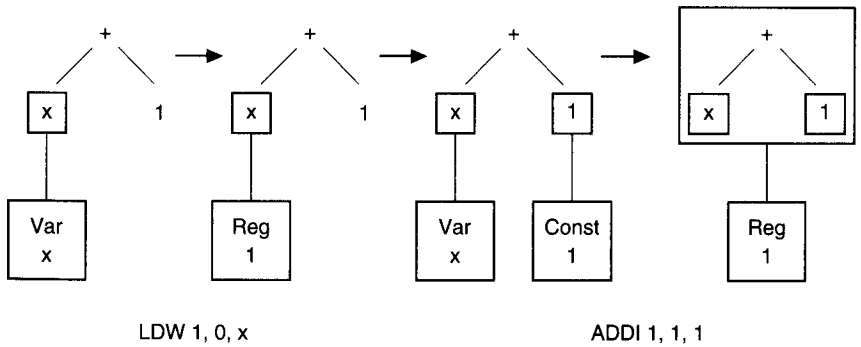


Figure 10.1 Generating items and instructions for the expression x+1.

Note the similarity of the two types Item and Object. Both describe objects, but whereas Objects represent declared, named objects, whose visibility reaches beyond the construct of their declaration, Items describe objects which are always strictly bound to their syntactic construct. Therefore, it is strongly recommended not to allocate Items dynamically (in a heap), but rather to declare them as local parameters and variables.

```
PROCEDURE factor(VAR x: Item);
BEGIN
  IF sym = ident THEN find(obj); Get(sym);
```

```
    x.mode := obj.class; x.a := obj.adr; x.r := 0
  ELSIF sym = number THEN x.mode := Const; x.a := val; Get(sym)
  ELSIF sym = lparen THEN Get(sym); expression(x);
    IF sym = rparen THEN Get(sym) ELSE Mark(" ) missing") END
  ELSIF ...
  END
END factor;

PROCEDURE term(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN factor(x);
  WHILE (sym = times) OR (sym = div) DO
    op := sym; Get(sym); factor(y); Op2(op, x, y)
  END
END term;

PROCEDURE SimpleExpression(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN
  IF sym = plus THEN Get(sym); term(x)
  ELSIF sym = minus THEN Get(sym); term(x); Op1(minus, x)
  ELSE term(x)
  END;
  WHILE (sym = plus) OR (sym = minus) DO
    op := sym; Get(sym); term(y); Op2(op, x, y)
  END
END SimpleExpression;

PROCEDURE Statement;
  VAR obj: Object; x, y: Item;
BEGIN
  IF sym = ident THEN
    find(obj); Get(sym); x.mode := obj.class; x.a := obj.adr; x.r := 0;
    IF sym = becomes THEN Get(sym); expression(y);
      IF y.mode # Reg THEN load(y) END;
      Put(STW, y.r, 0, x.a)
    ELSIF ...
    END
  ELSIF ...
  END
END Statement;
```

The code generating statements are now merged into two procedures, Op1 and Op2. The principle of delayed code emission is also used here to avoid the emission of arithmetic instructions if the compiler can perform the operation itself. This is the case when both operands are constants. The technique is known as *constant folding*.


```
PROCEDURE Op1(op: INTEGER; VAR x: Item);  (* x := op x *)
  VAR t: LONGINT;
BEGIN
  IF op = minus THEN
    IF x.mode = Const THEN x.a := –x.a
    ELSE
      IF x.mode = Var THEN load(x) END;
      Put(SUB, x.r, 0, x.r)
    END
  ...
  END
END Op1;

PROCEDURE Op2(op: INTEGER; VAR x, y: Item);  (* x := x op y *)
BEGIN
  IF (x.mode = Const) & (y.mode = Const) THEN
    IF op = plus THEN x.a := x.a + y.a
    ELSIF op = minus THEN x.a := x.a – y.a
    ...
    END
  ELSE
    IF op = plus THEN PutOp(ADD, x, y)
    ELSIF op = minus THEN PutOp(SUB, x, y)
    ...
    END
  END
END Op2;

PROCEDURE PutOp(cd: LONGINT; VAR x, y: Item);
  VAR r: LONGINT;
BEGIN
  IF x.mode # Reg THEN load(x) END;
  IF x.r = 0 THEN GetReg(x.r); r := 0 ELSE r := x.r END;
  IF y.mode = Const THEN Put(cd+16, r, x.r, y.a)
  ELSE
    IF y.mode # Reg THEN load(y) END;
    Put(cd, x.r, r, y.r); EXCL(regs, y.r)
  END
END PutOp;

PROCEDURE load(VAR x: Item);
BEGIN (*x.mode # Reg*)
  IF x.mode = Var THEN GetReg(x.r); Put(LDW, x.r, 0, x.a)
  ELSIF x.mode = Const THEN
    IF x.a = 0 THEN x.r := 0 ELSE GetReg(x.r); Put(ADDI, x.r, 0, x.a) END
  END;
  x.mode := Reg
END load;
```

Whenever arithmetic expressions are evaluated, the inherent danger of *overflow* exists. The evaluating statements should therefore be suitably guarded. In the case of addition guards can be formulated as follows:

```
IF x.a >= 0 THEN
  IF y.a <= MAX(INTEGER) – x.a THEN x.a := x.a + y.a
  ELSE Mark("overflow")
  END
ELSE
  IF y.a >= MIN(INTEGER) – x.a THEN x.a := x.a + y.a
  ELSE Mark("underflow")
  END
END
```

The essence of delayed code generation is that code is not emitted before it is clear that no better solution exists. For example, an operand is not loaded into a register before this is known to be unavoidable.

We also abandon allocation of registers according to the rigid stack principle. This is advantageous in certain cases which will be explained later. Procedure GetReg delivers and reserves any one of the free registers. The set of free registers is suitably represented by a global variable regs. Of course, care has to be taken to release registers whenever their value is no longer relevant.

```
PROCEDURE GetReg(VAR r: LONGINT);
  VAR i: INTEGER;
BEGIN i := 1;
  WHILE (i < 32) & (i IN regs) DO INC(i) END;
  INCL(regs, i); r := i
END GetReg;
```

The principle of delayed code generation is also useful in many other cases, but it becomes indispensable when considering computers with complex addressing modes, for which reasonably efficient code has to be generated by making good use of the available complex modes. As an example we consider code emission for a CISC architecture, which typically offers instructions with two operands, one of them also representing the result. Let us consider the expression u := x + y * z once more and obtain the following instruction sequence:

```
MOV   y, R1        R1 := y
MUL   z, R1        R1 := R1 * z
ADD   x, R1        R1 := R1 + x
MOV   R1, u        u := R1
```

This is obtained by delaying the loading of variables until they are to be joined with another operand. Because the instruction replaces the first operand with the

operation's result, the operation cannot be performed on the variable's original location, but only on an intermediate location, typically a register. The copy instruction is not issued until it becomes absolutely necessary. A side-effect of this measure is that, for example, the simple assignment x := y does not transfer via a register at all, but occurs directly through a copy instruction, which both increases efficiency and decreases code length:

```
MOV   y, x         x := y
```

10.3 Indexed variables and record fields

So far we have considered *simple* variables only in expressions and assignments. Access to elements of structured variables, arrays and records necessitates the selection of the element according to a computed index or a field identifier, respectively. Syntactically, the variable's identifier is followed by one or several selectors. This is mirrored in the parser by a call of the procedure selector within factor and also in statement:

```
find(obj); Get(sym); x.mode := obj.class; x.a := obj.adr; x.r := 0; selector(x)
```

Procedure selector processes not only a single selection, but if needed an entire sequence of selections. The following formulation shows that the attribute type of the operand x is also relevant.

```
PROCEDURE selector(VAR x: Item);
  VAR y: Item; obj: Object;
BEGIN
  WHILE (sym = lbrak) OR (sym = period) DO
    IF sym = lbrak THEN
      Get(sym); expression(y);
      IF x.type.form = Array THEN Index(x, y)
      ELSE Mark("not an array")
      END;
      IF sym = rbrak THEN Get(sym) ELSE Mark("]?") END
    ELSE Get(sym);
      IF sym = ident THEN
        IF x.type.form = Record THEN
          FindField(obj, x.type.fields); Get(sym);
          IF obj # guard THEN Field(x, obj) ELSE Mark("undef") END
        ELSE Mark("not a record")
        END
      ELSE Mark("ident?")
      END
    END
  END
END selector;
```

The address of the selected element is given by the formulas derived in Section 8.3. In the case of a field identifier the address computation is performed by the compiler. The address is the sum of the variable's address and the field's offset.

```
PROCEDURE Field(VAR x: Item; y: Object);  (* x := x.y *)
BEGIN INC(x.a, y.val); x.type := y.type
END Field;
```

In the case of an indexed variable, code is emitted according to the formula

```
adr(a[k])  =  adr(a) + k * size(T)
```

Here a denotes the array variable, k the index, and T the type of the array's elements. Index computation requires two instructions; the scaled index is added to the register component of the address. Let the index be stored in register R.j, and let the array address be stored in register R.i.

```
MULI    j, j, size(T)
ADD     i, i, j
```

Procedure Index emits the above index code, checks whether the indexed variable is indeed an array, and computes the element's address directly if the index is a constant.

```
PROCEDURE Index(VAR x, y: Item);  (* x := x[y] *)
  VAR z: Item;
BEGIN
  IF y.type # intType THEN Mark("index not integer") END;
  IF y.mode = Const THEN
    IF (y.a < 0) OR (y.a >= x.type.len) THEN
      Mark("index out of range")
    END;
    x.a := x.a + y.a * x.type.base.size
  ELSE
    IF y.mode # Reg THEN load(y) END;
    Put(MULI, y.r, y.r, x.type.base.size);
    Put(ADD, y.r, x.r, y.r);
    EXCL(regs, x.r); x.r := y.r
  END;
  x.type := x.type.base
END Index;
```

We can now show the code resulting from the following program fragment containing one- and two-dimensional arrays:


```
MODULE T1;
  VAR i, j: INTEGER;                              adr –4, –8
    a: ARRAY 4 OF INTEGER;                        adr –24
    b: ARRAY 3 OF ARRAY 5 OF INTEGER;             adr –84
BEGIN i := a[j]; i := a[2]; i := a[i+j]; i := b[i][j]; i := b[2][4]
END T1.
```

```
LDW     1, 0, –8          i := a[j]
MULI    1, 1, 4
LDW     2, 1, –24         a
STW     2, 0, –4          i

LDW     1, 0, –16         i := a[2]
STW     1, 0, –4

LDW     1, 0, –4          i := a[i+j];
LDW     2, 0, –8
ADD     1, 1, 2           i+j
MULI    1, 1, 4
LDW     2, 1, –24
STW     2, 0, –4          i

LDW     1, 0, –4          i := b[i][j]
MULI    1, 1, 20
LDW     2, 0, –8          j
MULI    2, 2, 4
ADD     2, 1, 2
LDW     1, 2, –84         b
STW     1, 0, –4          i

LDW     1, 0, –28         i := b[2][4]
STW     1, 0, –4
```

Note that the validity of the index can be checked only if the index is a constant, that is, it is of known value. Otherwise the index cannot be checked until run time. Although the test is of course redundant in correct programs, its omission is not recommended. In order to safeguard the abstraction of the array structure the test is wholly justified. However, the compiler designer should attempt to achieve utmost efficiency. The test takes the form of the statement

```
IF (k < 0) OR (k >= n) THEN HALT END
```

where k is the index and n the array's length. For our virtual computer, we simply postulate a corresponding instruction. In other cases a suitable instruction sequence must be found, whereby the following must be considered: since in Oberon the lower array bound is always 0, a single comparison suffices if the index value is considered as an unsigned integer. This is so because negative values

in complement representation appear with a sign bit value 1, yielding an unsigned value larger than the highest (signed) integer value.

Procedure Index is extended accordingly, generating a CHK instruction, which results in termination of the computation in case of an invalid index.

```
IF y.mode # Reg THEN load(y) END;
Put(CHKI, y.r, 0, x.type.base.len);
Put(MULI, y.r, y.r, x.type.base.size);
```

Finally, an example of a program is shown with nested data structures. It demonstrates clearly how the special treatment of constants in selectors simplifies the code for address computations. Compare the code resulting for variables indexed by expressions with those indexed by constants. CHK instructions have been omitted for the sake of brevity.

```
MODULE T2a;
  TYPE R0 = RECORD x, y: INTEGER END;
    R1 = RECORD u: INTEGER;              offset 0
      v: ARRAY 4 OF R0;                  offset 4
      w: INTEGER                         offset 36
    END;
  VAR i, j, k: INTEGER;                  adr –4, –8, –12
    s: ARRAY 2 OF R1;                    adr –92
BEGIN k := s[i].u; k := s[1].w;
  k := s[i].v[j].x; k := s[1].v[2].y;
  s[0].v[i].y := k
END T2a.

LDW   1, 0, –4          i
MULI  1, 1, 40
LDW   2, 1, –92         s[i].u
STW   2, 0, –12         k

LDW   1, 0, –16         s[1].w
STW   1, 0, –12

LDW   1, 0, –4          i
MULI  1, 1, 40
LDW   2, 0, –8          j
MULI  2, 2, 8
ADD   2, 1, 2
LDW   1, 2, –88         s[i].v[j].x
STW   1, 0, –12

LDW   1, 0, –28         s[1].v[2].y
STW   1, 0, –12
```



```
LDW   1, 0, –4          i
MULI  1, 1, 8
LDW   2, 0, –12         k
STW   2, 1, –84         s[0].v[i].y
```

A desire to keep target-dependent parts of the compiler separated from target-independent parts suggests that code generating statements should be collected in the form of procedures in a separate module. We shall call this module OSG and present its interface. It contains several of the generator procedures encountered so far. The others will be explained in Chapters 11 and 12.

```
DEFINITION OSG;
  IMPORT OSS, Texts, Fonts, Display;

  CONST Head = 0; Var = 1; Par = 2; Const = 3; Fld = 4; Typ = 5; Proc = 6;
    SProc = 7; Boolean = 0; Integer = 1; Array = 2; Record = 3;

  TYPE Object = POINTER TO ObjDesc;
    ObjDesc = RECORD
      class, lev: INTEGER;
      next, dsc: Object;
      type: Type;
      name: OSS.Ident;
      val: LONGINT;
    END;

    Type = POINTER TO TypeDesc;
    TypeDesc = RECORD
      form: INTEGER;
      fields: Object;
      base: Type;
      size, len: INTEGER;
    END;

    Item = RECORD
      mode, lev: INTEGER;
      type: Type;
      a: LONGINT;
    END;

  VAR boolType, intType: Type;
    curlev, pc: INTEGER;

  PROCEDURE FixLink (L: LONGINT);
  PROCEDURE IncLevel (n: INTEGER);
  PROCEDURE MakeConstItem (VAR x: Item; typ: Type; val: LONGINT);
  PROCEDURE MakeItem (VAR x: Item; y: Object);
  PROCEDURE Field (VAR x: Item; y: Object);
```

The use of subtraction ($x - y \neq 0$ standing for $x \neq y$) has an implicit pitfall: subtraction may lead to overflow, resulting in either program termination or a wrong result. Therefore a specific comparison instruction CMP is used in place of subtraction, which avoids overflow, but correctly indicates whether the difference is either zero, positive or negative. The result is typically stored in a special register called *condition code,* consisting of the two bits denoted by N and Z, indicating whether the difference is negative or zero respectively. All conditional branch instructions then implicitly refer to this register as argument.

```
IF x = y                          CMP  x, y
                                  BNE  L
  THEN StatSequence               code(StatSequence)
END                           L   ...
```

In our virtual RISC computer we use a CMP instruction which behaves like subtraction. We ignore overflow gracefully.

11.2 Conditional and repeated statements

The question of how a Boolean value is to be represented by an item now arises. In the case of stack architecture the answer is easy: since the result of a comparison lies on the stack like any other result, no special item mode is necessary. A CMP instruction, however, requires further thought. We shall first restrict our consideration to the simple cases of pure comparisons without further Boolean operators.

In the case of an architecture with a CMP scheme, it is necessary to indicate in the resulting item which register holds the computed difference, and which relation is represented by the comparison. For the latter a new attribute is required; we call the new mode Cond and its new attribute (record field) c. The mapping of relations to values of c is defined by

```
=    0        #    1
<    2        >=   3
<=   4        >    5
```

The construct containing comparisons is the expression. Its syntax is

```
expression = SimpleExpression [("=" | "#" | "<" | "<=" | ">" | ">=")
   SimpleExpression].
```

The corresponding, extended parser procedure is easily derived:

```
PROCEDURE expression(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN  SimpleExpression(x);
  IF (sym >= eql) & (sym <= gtr) THEN
    op := sym; Get(sym); SimpleExpression(y); Relation(op, x, y)
  END
END expression;

PROCEDURE Relation(op: INTEGER; VAR x, y: Item);
BEGIN
  IF (x.type.form # Integer) OR (y.type.form # Integer) THEN
    Mark("bad type")
  ELSE
    IF (y.mode = Const) & (y.a = 0) THEN load(x)
    ELSE PutOp(CMP, x, y)
    END;
    x.c := op - eql; EXCL(regs, x.r); EXCL(regs, y.r)
  END;
  x.mode := Cond; x.type := boolType
END Relation;
```


The code scheme presented at the beginning of this chapter yields the corresponding parser program for handling the IF construct in StatSequence:

```
ELSIF sym = if THEN
  Get(sym); expression(x); CJump(x);
  IF sym = then THEN Get(sym) ELSE Mark("THEN?") END;
  StatSequence; Fixup(x.a)
  IF sym = end THEN Get(sym) ELSE Mark("END?") END
```

Procedure CJump(x) generates the necessary branch instruction according to its parameter x.c in such a way that the jump is taken if the specified condition is *not* satisfied.

Here a difficulty becomes apparent which is inherent in all single-pass compilers. The destination location of branches is still unknown when the instruction is to be emitted. This problem is solved by adding the location of the branch instruction as an attribute to the item generated. This attribute is used later when the destination of the jump becomes known in order to complete the branch with its true address. This is called a *fixup.* The simple solution is possible only if code is deposited in a global array where elements are accessible at any time. It is not applicable if the emitted code is immediately stored on disk. To represent the address of the incomplete branch instruction we use the item field a.

```
PROCEDURE CJump(VAR x: Item);
BEGIN
  IF x.type.form = Boolean THEN
    Put(BEQ + negated(x.c), x.r, 0, 0); EXCL(regs, x.r); x.a := pc-1
```

```
    ELSE OSS.Mark("Boolean?"); x.a := pc
    END
  END CJump;

  PROCEDURE negated(cond: LONGINT): LONGINT;
  BEGIN
    IF ODD(cond) THEN RETURN cond-1 ELSE RETURN cond+1 END
  END negated;

  PROCEDURE Fixup(L: LONGINT);
  BEGIN code[L] := code[L] DIV 10000H * 10000H + pc - L
  END Fixup;
```

Procedure CJump issues an error message if x is not of type BOOLEAN. Note that branch instructions use addresses relative to the instruction's location (PC-relative); therefore the value pc–L is used.

Finally, we have to show how conditional statements in their general form are compiled; the syntax is

```
"IF" expression "THEN" StatSequence
{"ELSIF" expression "THEN" StatSequence}
["ELSE" StatSequence]
"END"
```

and the corresponding code pattern is

```
IF expression THEN                code(expression)
                                  Bcond L0
  StatSequence                    code(StatSequence)
                                  BR L
ELSIF expression THEN         L0  code(expression)
                                  Bcond L1
  StatSequence                    code(StatSequence)
                                  BR L
ELSIF expression THEN         L1  code(expression)
                                  Bcond L2
  StatSequence                    code(StatSequence)
                                  BR L
ELSIF ...                         ...
ELSE StatSequence             Ln  code(StatSequence)
END                           L   ...
```

from which the parser statements can be derived as part of procedure StatSequence. Although an arbitrary number of ELSIF constructs can occur and thereby also an arbitrary number of jump destinations L1, L2, ... may result, a single item variable x suffices. It is assigned a new value for every ELSIF instance.


```
  ELSIF sym = if THEN
    Get(sym); expression(x); CJump(x);
    IF sym = then THEN Get(sym) ELSE Mark("THEN ?") END;
    StatSequence; L := 0;
    WHILE sym = elsif DO
      Get(sym); FJump(L); Fixup(x.a); expression(x); CJump(x);
      IF sym = then THEN Get(sym) ELSE Mark("THEN ?") END;
      StatSequence
    END;
    IF sym = else THEN
      Get(sym); FJump(L); Fixup(x.a); StatSequence
    ELSE Fixup(x.a)
    END;
    FixLink(L);
    IF sym = end THEN Get(sym) ELSE Mark("END ?") END
    ...

  PROCEDURE FJump(VAR L: LONGINT);
  BEGIN
    Put(BEQ, 0, 0, L); L := pc-1
  END FJump;
```

However, a new situation arises in which not only a single branch refers to the destination label L at the end, but an entire set, namely as many as there are IF and ELSIF branches in the statement. The problem is elegantly solved by storing the links of the list of incomplete branch instructions in these instructions themselves, and by letting variable L represent the root of this list. The links are established by the parameter of the Put operation called in FJump. It suffices to replace procedure Fixup by FixLink, in which the entire list of instructions to be fixed up is traversed. It is essential that variable L is declared local to the parser procedure StatSequence, because statements may be nested, which leads to recursive activation. In this case, several instances of variable L will coexist, representing different lists.

```
  PROCEDURE FixLink(L: LONGINT);
    VAR L1: LONGINT;
  BEGIN
    WHILE L # 0 DO
      L1 := code[L] MOD 10000H; Fixup(L); L := L1
    END
  END FixLink;
```

Compilation of the WHILE statement is very similar to that of the simple IF statement. In addition to the conditional forward jump, an unconditional backward jump is necessary. The syntax and the corresponding code pattern are:

```
WHILE expression DO          L0  code(expression)
                                 Bcond  L1
  StatSequence                   code(StatSequence)
END                              BR  L0
                             L1  ...
```

From this we derive the corresponding, extended parser procedure:

```
ELSIF sym = while THEN
  Get(sym); L := pc; expression(x); CJump(x);
  IF sym = do THEN Get(sym) ELSE Mark("DO ?") END;
  StatSequence; BJump(L); Fixup(x.a);
  IF sym = end THEN Get(sym) ELSE Mark("END ?") END

PROCEDURE BJump(L: LONGINT);
BEGIN Put(BEQ, 0, 0, L–pc)
END BJump;
```

To summarize, we display two statements using variables i and j, together with the generated code:

```
IF i < j THEN i := 0 ELSIF i = j THEN i := 1 ELSE i := 2 END;
WHILE i > 0 DO i := i – 1 END

 4  LDW   1, 0, –4          i
 8  LDW   2, 0, –8          j
12  CMP   1, 1, 2
16  BGE   1, 0, 3           (jump over 3 instructions to 28)
20  STW   0, 0, –4          i := 0
24  BEQ   0, 0, 10          (jump over 10 instructions to 64)
28  LDW   1, 0, –4
32  LDW   2, 0, –8
36  CMP   1, 1, 2
40  BNE   1, 0, 4           (jump over 4 instructions to 56)
44  ADDI  1, 0, 1
48  STW   1, 0, –4          i := 1
52  BEQ   0, 0, 3           (jump over 3 instructions to 64)
56  ADDI  1, 0, 2
60  STW   1, 0, –4          i := 2
64  LDW   1, 0, –4
68  BLE   1, 0, 5           (jump over 5 instructions to 88)
72  LDW   1, 0, –4
76  SUBI  1, 1, 1
80  STW   1, 0, –4          i := i–1
84  BEQ   0, 0, –5          (jump back over 5 instructions to 64)
88  ...
```


11.3 Boolean operations

It is of course tempting to treat Boolean expressions in the same way as arithmetic expressions. Unfortunately, however, this would in many cases lead not only to inefficient, but even to wrong code. The reason lies in the definition of Boolean operators, namely

```
p OR q  = if p then TRUE else q
p & q   = if p then q else FALSE
```

This definition specifies that the second operand q *need not* be evaluated if the result is uniquely given by the value of the first operand p. Programming languages definitions even go a step further by specifying that in these cases the second operand *must not* be evaluated. This rule is postulated in order that the second operand may be left undefined without causing program execution to be terminated. A typical example, involving a pointer x, is

```
(x # NIL) & (x↑.size > 4)
```

Boolean expressions with Boolean operators therefore assume the form of conditional statements (more precisely, conditional expressions), and it is appropriate to use the same compilation techniques as for conditional statements. Boolean expressions and conditional statements merge, as the following example shows. The statement

```
IF (x <= y) & (y < z) THEN S END
```

is compiled in the same way as its equivalent formulation

```
IF x <= y THEN IF y < z THEN S END END
```

With the intention of deriving a suitable code pattern, let us first consider the following expression containing three relations connected by the & operator. We postulate the desired code pattern as shown in Figure 11.1, considering only the pattern to the left for the moment. a, b, ... , f denote numeric values.

As the left-hand pattern shows, a conditional branch instruction is emitted for every & operator. The jump is executed if the preceding condition is *not* satisfied (F-jump). This results in the instructions BGE to represent the < relation, BNE for the = relation, and so on. We obtain the postcondition P and a list of jumps, executed if ~P.

If we consider the problem of generating the required code, we can see that the parser procedure term, as it is known for processing arithmetic terms, must be extended slightly. In particular, a branch instruction must be emitted *before* the second operand is processed, whereas at the end this instruction's address must be fixed up. The former task is performed by procedure Op1, the latter by Op2.

P = (a < b) & (c < d) & (e < f)

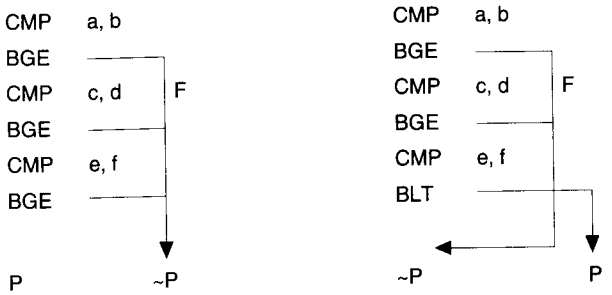


Figure 11.1 Code pattern for AND operator.

```
PROCEDURE term(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN factor(x);
  WHILE (sym >= times) & (sym <= and) DO
    op := sym; Get(sym);
    IF op = and THEN Op1(op, x) END;
    factor(y); Op2(op, x, y)
  END
END term;

PROCEDURE Op1(op: INTEGER; VAR x: Item);   (* x := op x *)
  VAR t: LONGINT;
BEGIN
  IF op = minus THEN ...
  ELSIF op = and THEN
    IF x.mode # Cond THEN loadBool(x) END;
    Put(BEQ + negated(x.c), x.r, 0, x.a);
    EXCL(regs, x.r); x.a := pc–1
  END
END Op1;
```

If the first Boolean factor is represented by item x in mode Cond, then at the present position x is TRUE and the instructions for the evaluation of the second operand must follow. However, if it is not in mode Cond, it must be transferred into this mode. This task is executed by procedure loadBool. We assume that the value FALSE is represented by 0. The attribute value c = 1 therefore causes the instruction BEQ to become active, if x equals 0.

```
PROCEDURE loadBool(VAR x: Item);
BEGIN
  IF x.type.form # Boolean THEN OSS.Mark("Boolean?") END;
  load(x); x.mode := Cond; x.c := 1
END loadBool;
```


The OR operator is treated analogously, with the difference that jumps are taken if their respective conditions are satisfied (T-jump). The instructions are listed in the dual list with links in the item field b. The postcondition of a sequence of terms connected with OR operators is FALSE. Consider again the left-hand code pattern only in Figure 11.2.

P = (a < b) OR (c < d) OR (e < f)

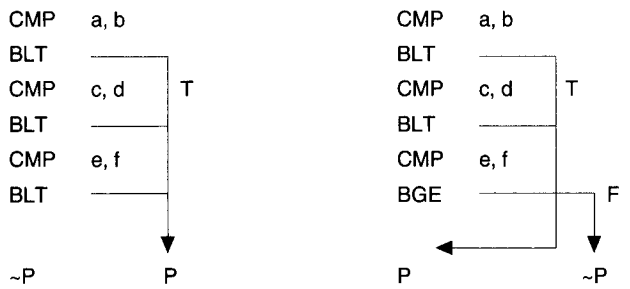


Figure 11.2 Code pattern for OR operator.

Next, we consider the implementation of negation. Here it turns out that under the scheme presented no instructions need be emitted whatsoever. Only the condition value represented by the item field c has to be negated, and the lists of F-jumps and T-jumps need to be exchanged. The result of negation is shown in the code patterns in Figures 11.1 and 11.2 on the right-hand side for both expressions with & and OR operators. The affected procedures are extended as shown below:

```
PROCEDURE SimpleExpression(VAR x: Item);
  VAR y: Item; op: INTEGER;
BEGIN term(x);
  WHILE (sym >= plus) & (sym <= or) DO
    op := sym; Get(sym);
    IF op = or THEN Op1(op, x) END;
    term(y); Op2(op, x, y)
  END
END SimpleExpression;

PROCEDURE Op1(op: INTEGER; VAR x: Item);   (* x := op x *)
  VAR t: LONGINT;
BEGIN
  IF op = minus THEN ...
```

```
    ELSIF op = not THEN
      IF x.mode # Cond THEN loadBool(x) END;
      x.c := negated(x.c); t := x.a; x.a := x.b; x.b := t
    ELSIF op = and THEN
      IF x.mode # Cond THEN loadBool(x) END;
      Put(BEQ + negated(x.c), x.r, 0, x.a); EXCL(regs, x.r);
      x.a := pc–1; FixLink(x.b); x.b := 0
    ELSIF op = or THEN
      IF x.mode # Cond THEN loadBool(x) END;
      Put(BEQ + x.c, x.r, 0, x.b); EXCL(regs, x.r);
      x.b := pc–1; FixLink(x.a); x.a := 0
    END
  END Op1;
```

When compiling expressions with & and OR operators, care must be taken that in front of every & condition P, and in front of every OR condition ~P, must hold. The respective lists of jump instructions must be traversed (the T-list for &, the F-list for OR), and the designated instructions must be fixed up appropriately. This occurs through procedure calls of FixLink in Op1. As examples, we consider the expressions

```
P = ((a < b) & (c < d)) OR ((e < f) & (g < h))
Q = ((a < b) OR (c < d)) & ((e < f) OR (g < h))
```

and the resulting code shown in Figure 11.3.

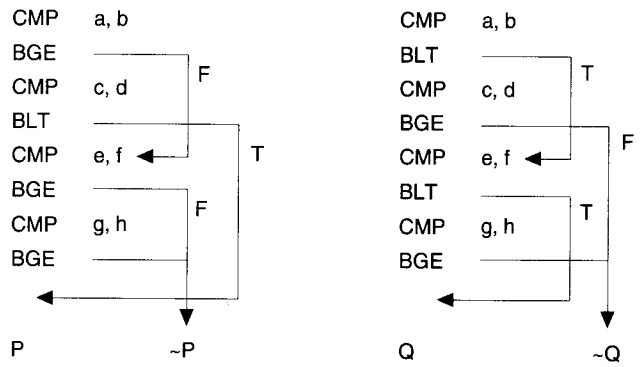


Figure 11.3 Code patterns for Boolean expressions with AND and OR operators.

It may also happen that a list of a subordinate expression may merge with the list of its containing expression (see F-link in the pattern for Q in Figure 11.3). This merger is accomplished by procedure merged(a, b), yielding as its value the concatenation of its argument lists. It is called from within procedure Op2.


```
PROCEDURE Op2(op: INTEGER; VAR x, y: Item);  (* x := x op y *)
BEGIN
  IF (x.type.form = Integer) & (y.type.form = Integer) THEN
    ...
  ELSIF (x.type.form = Boolean) & (y.type.form = Boolean) THEN
    IF y.mode # Cond THEN loadBool(y) END;
    IF op = or THEN x.a := y.a; x.b := merged(y.b, x.b); x.c := y.c
    ELSIF op = and THEN x.a := merged(y.a, x.a); x.b := y.b; x.c := y.c
    END
  ELSE ...
  END;
END Op2;
```

11.4 Assignments to Boolean variables

Compilation of assignments to Boolean variables turns out to be more complicated than might be expected. The reason is the item mode Cond, which must be converted into an assignable value 0 or 1. This is achieved by the code pattern shown in Figure 11.4.

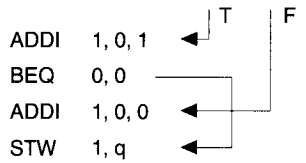


Figure 11.4 Code pattern for assignment to Boolean variable.

This causes the simple assignment q := x < y to appear as a disappointingly long code sequence. We should, however, be aware that Boolean variables (commonly called *flags*) occur (should occur) infrequently, although the notion of the type Boolean is indeed fundamental. It is inappropriate to strive for optimal implementation of rarely occurring constructs at the price of an intricate process. However, it is essential that the frequent cases are handled optimally.

Nevertheless, we handle assignments to a Boolean item *not* in the Cond mode as a special case, namely as a conventional assignment avoiding the involvement of jumps. Hence, the assignment p := q results in the expected code sequence

```
LDW 1, 0, q
STW 1, 0, p
```

As a consequence, the affected procedure Store turns out as follows:

```
PROCEDURE Store(VAR x, y: Item); (* x := y *)
BEGIN ...
  IF y.mode = Cond THEN
    FixLink(y.b); GetReg(y.r); Put(ADDI, y.r, 0, 1); Put(BEQ, 0, 0, 2);
    FixLink(y.a); Put(ADDI, y.r, 0, 0)
  ELSIF y.mode # Reg THEN load(y)
  END;
  IF x.mode = Var THEN Put(STW, y.r, x.r, x.a)
  ELSE Mark("illegal assignment")
  END;
  EXCL(regs, x.r); EXCL(regs, y.r)
END Store;
```

EXERCISES

11.1 Mutate the language Oberon-0 into a variant Oberon-D by redefining the conditional and the repeated statement as follows:

```
statement = ...
  "IF" guardedStatements {"|" guardedStatements} "FI" |
  "DO" guardedStatements {"|" guardedStatements} "OD" .
guardedStatements = condition "–>" statement {";" statement}.
```

The new form of statement

IF B_0 –> S_0 | B_1 –> S_1 | ... | B_n –> S_n FI

shall mean that of all conditions (Boolean expressions) B_i that are true, one is selected arbitrarily and its corresponding statement sequence S_i is executed. If none is true, program execution is aborted. Any statement sequence S_i will be executed only when the corresponding condition B_i is true. B_i is therefore said to be the *guard* of S_i.

The statement

DO B_0 –> S_0 | B_1 –> S_1 | ... | B_n –> S_n OD

shall mean that as long as any of the conditions B_i is true, one of them is chosen arbitrarily, and its corresponding statement sequence S_i is executed. The process terminates as soon as all B_i are false. Here too, the B_i function as guards. The DO–OD construct is a repetitive, nondeterministic construct. Adjust the compiler accordingly.

11.2 Extend Oberon-0 and its compiler by a FOR statement:

```
statement = [assignment | ProcedureCall |
  IfStatement | WhileStatement | ForStatement].
ForStatement = "FOR" identifier ":=" expression "TO" expression
  ["BY" expression] "DO" StatementSequence "END" .
```

The expression preceding the symbol TO specifies the starting value, the one thereafter the ending value of the control variable denoted by the identifier. The expression after BY indicates the increment. If missing, let 1 be its default value.

11.3 Consider the implementation of the case statement of Oberon (see Appendix A). Its essential property is that it uses a table of jump addresses for the various cases, and an indexed jump instruction.

Chapter 12
Procedures and the Concept of Locality

12.1 Run-time organization of the store

Procedures, which are also known as subroutines, are perhaps the most important tool for structuring programs. Because of their frequency of occurrence, it is mandatory that their implementation is efficient. Implementation is based on the BSR instruction which saves the current PC value, and a return is simply achieved by reloading the saved value into the PC register.

The question as to where the return address should be saved arises immediatel y. In many computers it is deposited in a register, and we have adopted this solution in our RISC. This guarantees the utmost efficiency, because no additional memory access is involved. But having to save the register's value into memory before the next procedure call is unavoidable, because otherwise the old return address would be overwritten. Thereby the return address of the first call would be lost. In the implementation of a compiler this *link register* value must be saved at the beginning of each procedure call.

To store the link, a stack is the obvious solution. The reason is that procedure activations occur in a nested fashion; procedures terminate in the reverse order of their calls. The store for the return addresses must therefore operate according to the *first-in last-out* principle. This results in the following, fixed code sequences at the beginning and end of every procedure. They are called the procedure's *prologue* and *epilogue*. We use R30 for the stack pointer SP and R31 as link register LNK.

```
Call                BSR  P                 branch to subroutine
Prologue      P     PSH  LNK, SP, 4        push link
Epilogue            POP  LNK, SP, 4        pop link
                    RET  0, 0, LNK         return jump
```

This code pattern is valid under the assumption that the BSR instruction deposits the return address in R31. Note that this is specified as a hardware feature (Chapter 9), whereas the use of R30 as stack pointer is merely a software convention determined by the compiler design or by the underlying operating system. Whenever the system is started, R30 must be initialized to point to an area of memory reserved for the stack.

Algol 60 introduced the very fundamental concept of *local* variables. It implied that every identifier declared had a limited range of visibility and validity. In Pascal (and also in Oberon) this range is the procedure body. In concrete terms, variables may be declared local to a procedure such that they are visible and valid within this procedure only. The intended consequence is that upon entry to the procedure memory is allocated automatically for these local variables, and it is released upon the procedure's termination. Local variables of different procedures may therefore share the same storage area – but never simultaneously, of course.

At first sight this scheme seems to inflict a certain loss of efficiency upon the procedure call mechanism. Fortunately, however, this need not be so, because the storage blocks for the sets of local variables can be allocated, like return addresses, according to the stack principle. The return address may indeed also be considered as a (hidden) local variable, and it is only natural to use the same stack for variables and return addresses. The storage blocks are called *procedure activation records* or *activation frames.* Release of a block upon procedure termination is achieved by simply resetting the stack pointer to its value before the procedure call. Hence, allocation and release of local storage is optimally efficient.

Addresses of local variables generated by the compiler are always relative to the base address of the respective activation frame. Since in programs most variables are local, their addressing also must be highly efficient. This is achieved by reserving a register to hold the base address, and to make use of the fact that the effective address is the sum of a register value and the instruction's address field (register relative addressing mode). The reserved register is called the *frame pointer* (FP). These considerations are taken into account by the following prologue and epilogue, where R29 assumes the role of the frame pointer:

```
Prologue      P     PSH  LNK, SP, 4        push link
                    PSH  FP, SP, 4         push FP
                    ADD  FP, 0, SP         FP := SP
                    SUBI SP, SP, n         SP := SP–n  (n = frame size)

Epilogue            ADD  SP, 0, FP         SP := FP
                    POP  FP, SP, 4         pop FP
                    POP  LNK, SP, 4        pop link
                    RET  0, 0, LNK         return jump
```

The activation frames resulting from consecutive procedure calls are linked by a list of their base addresses. The list is called the *dynamic link,* because it denotes the dynamic sequence of procedure activations. Its root lies in the frame pointer register FP (see Figure 12.1).

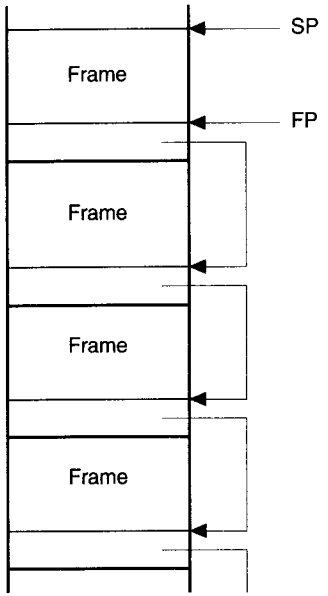


Figure 12.1 List of activation frames in the stack.

The state of the stack before and after a procedure call is shown in Figure 12.2. Note that the epilogue reverts the stack to its original state by removing return address and dynamic link entry.

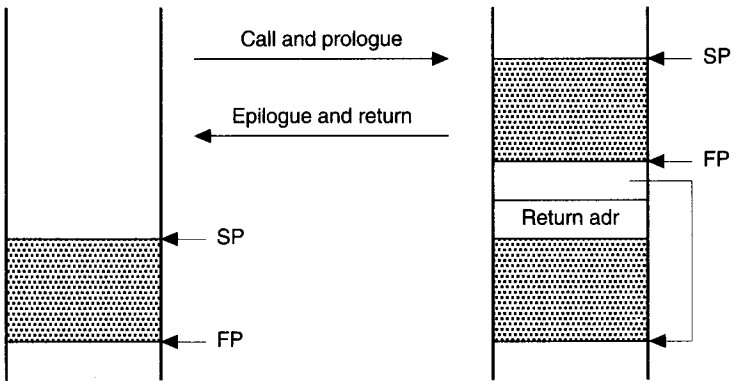


Figure 12.2 States of the stack before and after procedure call.

If we carefully consider the necessity of the two pointers SP and FP, we may come to the conclusion that FP is actually superfluous, because the variables' offset addresses could be made relative to SP instead of FP. This proposition,

however, is valid only if the sizes of all variables are known at compile time. This is not so in the case of open (dynamic) arrays, as will become apparent later. But obviously the retention of a second pointer (FP) requires additional storage accesses upon every procedure call and return, which are undesirable.

In order to improve efficiency and in particular to reduce the length of the instruction sequences for both prologue and epilogue, computers with more complex instructions feature special instructions corresponding to prologue and epilogue. Two examples may help at this point; the second features special, dedicated registers for the pointers SP and FP. The number of required instructions, however, remains the same.

	Motorola 680x0	*National Semiconductor 32x32*
Call	BSR P	BSR P
Prologue	LINK D14, n	ENTER n
Epilogue	UNLNK D14	EXIT
Return jump	RTD	RET

12.2 Addressing of variables

We recall that the address of a local variable is relative to the base address of the activation frame containing the variable, and that this base address is held in register FP. The latter, however, holds only for the record activated last, and thereby only for variables which belong to the procedure in which they are referenced. In many programming languages procedure declarations may be nested, giving rise to references to variables which are local to some procedure, but not to the procedure referencing them. The following example demonstrates the situation, with R being local to Q, and Q and S local to P:

```
                                  Object   Level
PROCEDURE P;                        P        0
  VAR x: INTEGER;                   x        1

  PROCEDURE Q;                      Q        1
    VAR y: INTEGER;                 y        2

    PROCEDURE R;                    R        2
      VAR z: INTEGER;               z        3
    BEGIN x := y + z
    END R;

  BEGIN R
  END Q ;
```

```
PROCEDURE S;                    S     1
BEGIN Q
END S;

BEGIN Q; S
END P;
```

Let us trace the chain of calls P → Q → R. It is tempting to believe that, when accessing variables x, y, or z in R, their base address could be obtained by traversing the dynamic link list. The number of steps would be the difference between the levels of the call and of the declaration. This difference is 2 for x, 1 for y, and 0 for z. But this assumption is wrong. R could also be reached through the call sequence P → S → Q → R as shown in Figure 12.3. Access to x would then lead in two steps to the activation frame of S instead of P.

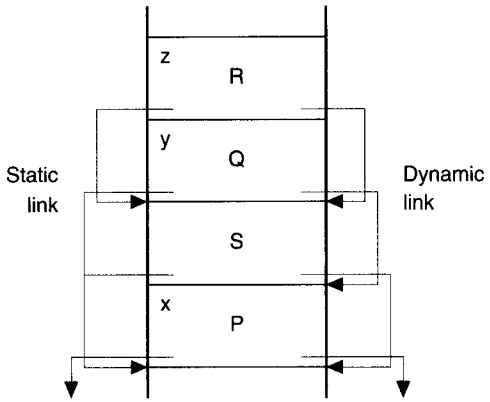


Figure 12.3 Dynamic and static links in the stack.

Evidently, a second list of activation records is necessary which mirrors the static order of nesting rather than the dynamic order of calls. Hence a second link must be established upon every procedure call. The so-called *procedure mark* now contains, in addition to the return address and the dynamic link, a *static link* element. The static link of a procedure P points to the activation record of the procedure which contains P, that is, in which P is declared locally. It should be noted that this pointer is superfluous for procedures declared globally, if global variables are addressed directly, that is, without base address. Since this is typically the case, and since most procedures are declared globally, the additional complexity caused by the static chain is acceptable. With some justification the absolute addressing of global variables can be considered as a special case of local variable addressing leading to an increase in efficiency.

Finally, note that access to variables via the static link list (intermediate level variables) is less efficient than access to strictly local variables, because every step through the list requires an additional memory access. Several solutions have been proposed and implemented to eliminate this loss of efficiency. They ultimately always rely on the mapping of the static list onto a set of base registers. We consider this as an optimization at the wrong place. First, registers are scarce resources which should not be given away too easily. And second, the copying of link elements into registers upon every call and return may easily cost more than it saves, in particular because references to intermediate-level variables occur quite rarely in practice. The optimization may therefore turn out to be quite the reverse.

In order not to reduce the readability of the Oberon-0 compiler listed in Appendix C, the handling of intermediate-level variables has not been implemented.

Global variables have fixed addresses which must also be considered relative to a frame address. Their absolute values are determined upon loading the code, that is, *after* compilation but *before* program execution. The emitted object code is therefore accompanied by a list of addresses of instructions referring to global variables. The loader must then add to these addresses the base address of the respective frame of global variables. This fixup operation can be omitted if the computer features a PC-relative addressing mode. Our RISC computer does not do so, and in general computers offer the PC-relative mode for branch instructions only.

12.3 Parameters

Parameters constitute the interface between the calling and the called procedures. Parameters on the calling side are said to be *actual parameters,* and those on the called side *formal parameters.* The latter are in fact only place holders for which the actual parameters are substituted. Basically, a substitution is an assignment of the actual value to the formal variable. This implies that every formal parameter must be regarded as a variable bound to the procedure, and that every call be accompanied by a number of assignments called *parameter substitutions.*

Most programming languages distinguish between at least two kinds of parameters. The first is the *value parameter* where, as its name suggests, the *value* of the actual parameter is assigned to the formal variable. The actual parameter is syntactically an expression. The second kind of parameter is the *reference parameter,* where, also as suggested by its name, a *reference* to the actual parameter is assigned to the formal variable. Evidently, the actual parameter must in this case be a variable, because an assignment to the formal parameter is permissible, and this assignment must refer to the actual *variable.* (In Pascal, Modula, and Oberon, the reference parameter is therefore called the *variable parameter.*) The value of the formal variable is in this case a hidden pointer, that is, an address.

Of course, the actual parameter must be evaluated before the substitution takes place. In the case of variable parameters, the evaluation takes the form of an identification of the variable, implying, for example, the evaluation of the index in the case of indexed variables. But how is the destination of this substitution to be determined? Here the stack organization of the store comes into play. The

actual values are simply deposited in sequence on the top of the stack; no explicit destination addresses are required. Figure 12.4 shows the state of the stack after the deposition of the parameters, and after the call and the prologue.

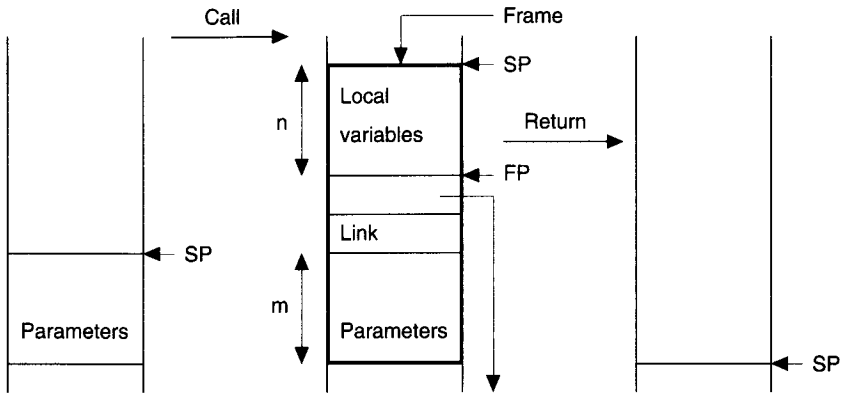


Figure 12.4 Parameter substitution.

It now becomes evident that parameters can be addressed relative to the frame address FP, like local variables. If local variables have negative offsets, parameters have positive offsets. It is particularly worth noting that the called procedure references parameters exactly where they were deposited by the calling procedure. The space allocated for the parameters is regained by the epilogue simply by increasing the value of SP.

```
Epilogue    ADD  SP, 0, FP        SP := FP
            POP  FP, SP, 4        pop FP
            POP  LNK, SP, m+4     pop link and parameters
            RET  0, 0, LNK        return jump
```

In the case of CISC computers with prologue and epilogue represented by special instructions, the required increment of SP is included in the return instruction by specifying the size of the parameter block as parameter (RET m).

12.4 Procedure declarations and calls

The procedure for processing procedure declarations is easily derived from the syntax with the aid of the parser construction rules. The new entry in the symbol table generated for a procedure declaration obtains the class attribute Proc, and its attribute a is given the current value of pc, which is the entry address of the procedure's prologue. Thereafter, a new scope is opened in the symbol table in such a way that (1) new entries for local objects are automatically inserted in the

new scope, and (2) at the end of the procedure the local objects are easily discarded and the previous scope reappears. Here too, the two procedures OpenScope and CloseScope embody the stack principle, and the linkage is established by a header element (class Head, field dsc). Objects are given an additional attribute lev denoting the nesting level of the declared object. Consider the following declarations:

```
CONST N = 10;
VAR x: T;
PROCEDURE P(x, y: INTEGER); ...
```

The resulting symbol table is shown in Figure 12.5. The dsc pointer refers to P's parameters x and y.

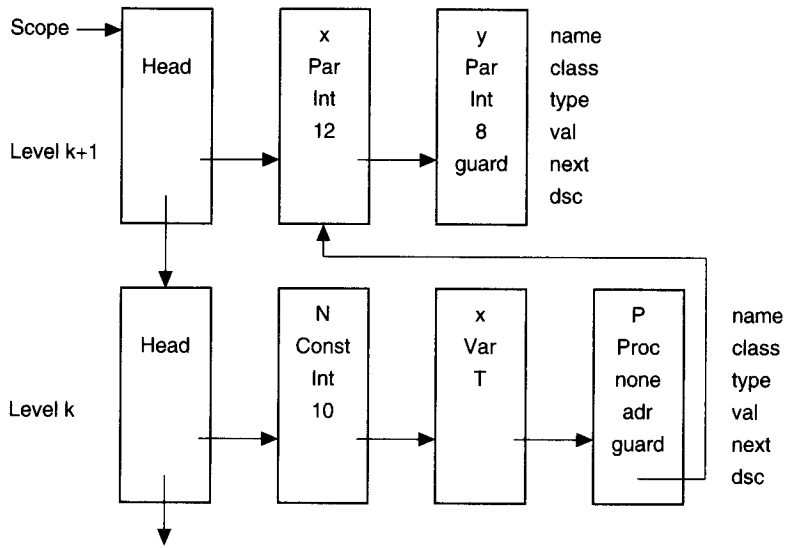


Figure 12.5 Symbol table representing two scopes.

```
PROCEDURE ProcedureDecl;
  VAR proc, obj: Object;
    procid: Ident;
    locblksize, parblksize: LONGINT;

  PROCEDURE FPSection;
    VAR obj, first: Object; tp: Type; parsize: LONGINT;
  BEGIN
    IF sym = var THEN Get(sym); IdentList(Par, first)
    ELSE IdentList(Var, first)
    END;
```

```
    IF sym = ident THEN
      find(obj); Get(sym);
      IF obj.class = Typ THEN tp := obj.type
      ELSE Mark("type?"); tp := intType
      END
    ELSE Mark("ident?"); tp := intType
    END;
    IF first.class = Var THEN parsize := tp.size ELSE parsize := 4 END;
    obj := first;
    WHILE obj # guard DO
      obj.type := tp; INC(parblksize, parsize); obj := obj.next
    END
  END FPSection;

BEGIN (* ProcedureDecl *) Get(sym);
  IF sym = ident THEN
    procid := id;
    NewObj(proc, Proc); Get(sym); parblksize := 8;
    INC(level); OpenScope; proc.val := -1;
    IF sym = lparen THEN Get(sym);
      IF sym = rparen THEN Get(sym)
      ELSE FPSection;
        WHILE sym = semicolon DO Get(sym); FPSection END;
        IF sym = rparen THEN Get(sym) ELSE Mark(")?") END
      END
    END;
    obj := topScope.next; locblksize := parblksize;
    WHILE obj # guard DO
      obj.lev := curlev;
      IF obj.class = Par THEN DEC(locblksize, 4)
      ELSE locblksize := locblksize - obj.type.size
      END;
      obj.val := locblksize; obj := obj.next
    END;
    proc.dsc := topScope.next;
    IF sym = semicolon THEN Get(sym) ELSE Mark(";?") END;
    locblksize := 0; declarations(locblksize);
    WHILE sym = procedure DO
      ProcedureDecl;
      IF sym = semicolon THEN Get(sym) ELSE Mark(";?") END
    END;
    proc.val := pc; Enter(locblksize);
    IF sym = begin THEN Get(sym); StatSequence END;
    IF sym = end THEN Get(sym) ELSE Mark("END?") END;
    IF sym = ident THEN
      IF procid # id THEN Mark("no match") END;
      Get(sym)
    ELSE Mark("ident?")
    END;
    Return(parblksize - 8); CloseScope; DEC(level)
  END
END ProcedureDecl;
```


Within the procedure body, value parameters are treated exactly like local variables. Their entries in the symbol table are of class Var. A new class Par is introduced to represent reference parameters. The addresses (offsets) of formal parameters are derived according to the following formula, whereby the last parameter p_n obtains the least offset, namely the size of the procedure mark (8). The size of variable parameters is always 4, which is the size of an address.

$$adr(p_i) = size(p_{i+1}) + ... + size(p_n) + 8$$

Unfortunately, this implies that the offsets cannot be determined before the entire parameter list has been recognized. In the case of byte-addressed stores it is moreover advantageous always to increment or decrement the stack pointer by multiples of 4, such that parameters are always aligned to word boundaries. In the case of Oberon-0 special attention to this rule is unnecessary, because all data types feature a size of multiples of 4 anyway.

Local declarations are processed by the parser procedure *declarations*. The code for the prologue is emitted by procedure Enter after the processing of local declarations. Emission of the epilogue is performed by procedure Return at the end of ProcedureDecl.

```
PROCEDURE Enter(size: LONGINT);
BEGIN
  Put(PSH, LNK, SP, 4);
  Put(PSH, FP, SP, 4);
  Put(ADD, FP, 0, SP);
  Put(SUBI, SP, SP, size)
END Enter;

PROCEDURE Return(size: LONGINT);
BEGIN
  Put(ADD, SP, 0, FP);
  Put(POP, FP, SP, 4);
  Put(POP, LNK, SP, size+4);
  Put(RET, 0, 0, LNK)
END Return;
```

Procedure MakeItem generates an Item corresponding to a given Object. At this point, the difference between the addressing of local and global variables must be taken into account. (As already mentioned, the handling of intermediate-level

variables is not treated here.) Note, however, that reference parameters (class = Par) require indirect addressing. Since the RISC architecture does not explicitly feature an indirect addressing mode, the value of the formal parameter, which is the address of the actual parameter, is loaded into a register. The actual parameter is then accessed via this register, with offset 0.

```
PROCEDURE MakeItem(VAR x: Item; y: Object);
  VAR r: LONGINT;
BEGIN x.mode := y.class; x.type := y.type; x.a := y.val;
  IF y.lev = 0 THEN x.r := 0 ELSIF y.lev = curlev THEN x.r := FP
  ELSE Mark("level!"); x.r := 0
  END;
  IF y.class = Par THEN
    GetReg(r); Put(LDW, r, x.r, x.a); x.mode := Var; x.r := r; x.a := 0
  END
END MakeItem;
```

Procedure calls are generated within the already encountered procedure StatSequence with the aid of auxiliary procedures Parameter and Call:

```
IF sym = ident THEN
  find(obj); Get(sym); MakeItem(x, obj); selector(x);
  IF sym = becomes THEN ...
  ELSIF x.mode = Proc THEN
    par := obj.dsc;
    IF sym = lparen THEN Get(sym);
      IF sym = rparen THEN Get(sym)
      ELSE
        LOOP expression(y);
          IF IsParam(par) THEN
            Parameter(y, par.type, par.class); par := par.next
          ELSE Mark("too many parameters")
          END;
          IF sym = comma THEN Get(sym)
          ELSIF sym = rparen THEN Get(sym); EXIT
          ELSIF sym >= semicolon THEN Mark(") ?"); EXIT
          ELSE Mark(") or , ?")
          END
        END
      END
    END;
    IF obj.val < 0 THEN Mark("forward call")
    ELSIF ~IsParam(par) THEN Call(x)
    ELSE Mark("too few parameters")
    END
```



```
  ...
PROCEDURE Parameter(VAR x: Item; ftyp: Type; class: INTEGER);
  VAR r: LONGINT;
BEGIN (*SP = 30*)
  IF x.type = ftyp THEN
    IF class = Par THEN (*Var param*)
      IF x.mode = Var THEN
        IF x.a # 0 THEN GetReg(r); Put(ADDI, r, x.r, x.a) ELSE r := x.r END
      ELSE Mark("illegal parameter mode")
      END;
      Put(PSH, r, SP, 4); EXCL(regs, r)  (*push*)
    ELSE (*value param*)
      IF x.mode # Reg THEN load(x) END;
      Put(PSH, x.r, SP, 4); EXCL(regs, x.r)
    END
  ELSE Mark("bad parameter type")
  END
END Parameter;

PROCEDURE IsParam(obj: Object): BOOLEAN;
BEGIN RETURN (obj.class = Par) OR (obj.class = Var) & (obj.val > 0)
END IsParam;

PROCEDURE Call(VAR x: Item);
BEGIN Put(BSR, 0, 0, x.a – pc)
END Call;
```

Here we tacitly assume that the entry addresses of procedures are known when a call is to be compiled. Thereby we exclude forward references which may, for example, arise in the case of mutual, recursive referencing. If this restriction is to be lifted, the locations of forward calls must be retained in order that the branch instructions may be fixed up when their destinations become known. This case is similar to the fixups required for forward jumps in conditional and repeated statements.

In conclusion, we show the code generated for the following, simple procedure:

```
PROCEDURE P(x: INTEGER; VAR y: INTEGER);
BEGIN x := y; y := x; P(x, y); P(y, x)
END P

 0   PSH   LNK, SP, 4          prologue
 4   PSH   FP, SP, 4
 8   ADD   FP, 0, SP
12   SUBI  SP, SP, 0           no local variables
```

```
16  LDW   1, FP, 8
20  LDW   2, 1, 0
24  STW   2, FP, 12          x := y

28  LDW   1, FP, 8
32  LDW   2, FP, 12
36  STW   2, 1, 0            y := x

40  LDW   1, FP, 12          x
44  PSH   1, SP, 4
48  LDW   1, FP, 8           adr(y)
52  PSH   1, SP, 4
56  BSR   0, 0, –14          P(x, y)

60  LDW   1, FP, 8
64  LDW   2, 1, 0            y
68  PSH   2, SP, 4
72  ADDI  1, FP, 12          adr(x)
76  PSH   1, SP, 4
80  BSR   0, 0, –20          P(y, x)

84  ADD   SP, 0, FP          epilogue
88  POP   FP, SP, 4
92  POP   LNK, SP, 12        pop link and parameters
96  RET   0, 0, 31
```

12.5 Standard procedures

Most programming languages feature certain procedures and functions which do not need to be declared in a program. They are said to be *predeclared* and they can be called from anywhere, as they are *pervasive*. These are well-known functions, for example the absolute value of a number (ABS), type conversions (ENTIER, ORD), or frequently encountered statements which merit an abbreviation and are available on many computers as single instructions (INC, DEC). The property common to all these so-called *standard* procedures is that they correspond directly either to a single instruction or to a short sequence of instructions. Therefore, these procedures are handled quite differently by compilers; no call is generated. Instead, the necessary instructions are emitted directly into the code. These procedures are therefore also called *in-line* procedures, a term that makes sense only if the underlying implementation technique is understood.

As a consequence it is advantageous to consider standard procedures as an object class of their own. Thereby the need for a special treatment of calls becomes immediately apparent. For Oberon-0 we postulate procedures Read, Write, WriteHex, and WriteLn, which on the one hand introduce elementary input and output facilities, and on the other hand serve to demonstrate the proposed handling of predeclared procedures. In this case, the term *standard* is admittedly misleading, whereas *predeclared* and *in-line* refer to the core of the subject matter.

The corresponding entries in the symbol table are made when the compiler is initialized, namely in an outermost scope called universe which always remains open (see Appendix C). The new class attribute is denoted by SProc, and attribute val (a in the case of Items) identifies the procedure concerned.

```
IF sym = ident THEN
  find(obj); Get(sym); MakeItem(x, obj); selector(x);
  IF sym = becomes THEN ...
  ELSIF x.mode = Proc THEN ...
  ELSIF x.mode = SProc THEN
    IF obj.val <= 3 THEN param(y); TestInt(y) END;
    IOCall(x, y)
  ...

PROCEDURE IOCall(VAR x, y: Item);
  VAR z: Item;
BEGIN (*x.mode = SProc*)
  IF x.a = 1 THEN (*Read*)
    GetReg(z.r); z.mode := Reg; z.type := intType;
    Put(RD, z.r, 0, 0); Store(y, z)
  ELSIF x.a = 2 THEN (*Write*)
    load(y); Put(WRD, 0, 0, y.r); EXCL(regs, y.r)
  ELSIF x.a = 3 THEN (*WriteHex*)
    load(y); Put(WRH, 0, 0, y.r); EXCL(regs, y.r)
  ELSE (*WriteLn*)
    Put(WRL, 0, 0, 0)
  END
END IOCall;
```

The final example shows a sequence of three statements and the resulting code:

```
Read(x); Write(x); WriteLn

 4  READ  1, 0, 0
 8  STW   1, 0, –4           x
12  LDW   1, 0, –4           x
16  WRD   0, 0, 1
32  WRL   0, 0, 0
```

12.6 Function procedures

A function procedure is a procedure whose identifier simultaneously denotes both an algorithm and its result. It is activated not by a call statement but by a factor of an expression. The call of a function procedure must therefore also take care of returning the function's result. The question therefore arises of which resources should be used.

If our primary goal is the generation of efficient code with the minimal number of memory accesses, then a register is the prime candidate for temporarily holding the function's result. If this solution is adopted, we must renounce the capability of defining functions with a structured result, because structured values cannot be held in a register.

If this restriction is considered as unacceptable, a place in the stack must be reserved to hold the structured result. Typically, it is added to the parameter area of the activation record. The function result is considered as an implicit result (variable) parameter. Correspondingly, the stack pointer is incremented before code for the first parameter is emitted.

At this point, all the concepts contained in the language Oberon-0 and implemented in its compiler have been presented. The compiler's listing is contained in full in Appendix C.

EXERCISES

12.1 Improve the Oberon-0 compiler in such a way that the restriction that variables must be strictly local or entirely global can be lifted.

12.2 Add standard functions to the Oberon-0 compiler, generating in-line code. Consider ABS, INC, DEC.

12.3 Replace the VAR parameter concept by the notion of an OUT parameter. An OUT parameter represents a local variable whose value is assigned to its corresponding actual parameter upon termination of the procedure. It constitutes the inverse of the value parameter, where the value of the actual parameter is assigned to the formal variable upon the start of the procedure.

Chapter 13
Elementary Data Types

13.1 The types REAL and LONGREAL

As early as 1957 integers and real numbers were treated as distinct data types in *Fortran.* This was not only because different, internal representations were necessary, but because it was recognized that the programmer must be aware of when computations could be expected to be exact (namely for integers), and when only approximate. The fact that with real numbers only approximate results can be obtained, may be understood by considering that real numbers are represented by scaled integers with a fixed, finite number of digits. Their type is called REAL, and a real value x is represented by the pair of integers e and m as defined by the equation

$$x = B^{e-w} * m \qquad 1 \le m < B$$

This form is called *floating-point* representation; e is said to be the *exponent,* m the *mantissa.* The base B and the bias w are fixed values for all REAL values, characterizing the chosen number representation. The two IEEE standards of floating-point representations feature the following values for B and w, and to the components e and m a bit s is added for the sign:

Type	B	w	*Number of bits for* e	*Number of bits for* m	*Total*
REAL	2	127	8	23	32
LONGREAL	2	1023	11	52	64

The exact forms of the two types, called REAL and LONGREAL in Oberon, are specified by the following formulas:

$$x = (-1)^s * 2^{e-127} * 1.m \qquad x = (-1)^s * 2^{e-1023} * 1.m$$

The following examples show the floating-point representation of some selected numbers:

Decimal	s	e	1.m	*Binary*	*Hexadecimal*
1.0	0	127	1.0	0 01111111 00000000000000000000000	3F80 0000
0.5	0	126	1.0	0 01111110 00000000000000000000000	3F00 0000
2.0	0	128	1.0	0 10000000 00000000000000000000000	4000 0000
10.0	0	130	1.25	0 10000010 01000000000000000000000	4120 0000
0.1	0	123	1.6	0 01111011 10011001100110011001101	3DC CCCD
−1.5	1	127	1.5	1 01111111 10000000000000000000000	BFC0 0000

Two examples illustrate the case of LONGREAL:

1.0	0	1023	1.0	0 01111111111 00000000...00000000	3FF0 0000 0000 0000
0.1	0	1019	1.6	0 01111111011 10011001...10011010	3FB9 9999 9999 999A

This logarithmic form inherently excludes a value for 0. The value 0 must be treated as a special case, and it is represented by all bits being 0. With regard to numeric properties it constitutes a special case and a discontinuity. Furthermore, the IEEE standards postulate two additional special values: e = 0 (with m ≠ 0) and e = 255 (resp. e = 1023) are considered as invalid results and they are called NaN (not a number).

Normally, the programmer does not have to worry about these specifications, and the compiler designer is not affected by them. The types REAL and LONGREAL constitute abstract data types usually integrated in the hardware which features a set of instructions adapted to the floating-point representation. If this set is complete, that is, it covers all basic numeric operations, the representation may be considered as hidden, since no further, programmed operations depend on it. In many computers, instructions for floating-point operands use a special set of registers. The reason behind this is that often separate coprocessors, so-called *floating-point units* (FPUs) are used which implement all floating-point instructions and contain this set of floating-point registers.

13.2 Compatibility between numeric data types

The values of all variables with numeric data type are numbers. Therefore there is no obvious reason not to declare them all as assignment compatible. But, as already outlined, numbers of different types are differently represented in terms of bit sequences within the computer. Hence, whenever a number of type T0 is assigned to a variable of type T1, a representation conversion has to be performed which takes some finite time. The question then arises of whether this fact should remain hidden from the programmer in order to avoid distraction, or whether it should be made explicit because it affects the efficiency of the program. The latter choice is accomplished by declaring the various types as incompatible and by providing explicit, predefined conversion functions.

In any case, in order to be complete, a computer's set of instructions must also contain *conversion instructions* which convert integers into floating-point numbers and vice versa. The same holds at the level of the programming language.

In Oberon different data types may be used within an arithmetic expression, which is then said to be a *mixed expression.* In this case, the compiler must insert hidden conversion instructions yielding the representation required for the specified arithmetic operation to be applied.

The definition of Oberon provides not only two, but an entire set of numeric data types. They are ordered in the sense that the 'larger' type contains all values belonging to the 'smaller' type. This notion is called *type inclusion*:

SHORTINT ⊆ INTEGER ⊆ LONGINT ⊆ REAL ⊆ LONGREAL

The result type of an expression is defined to be equal to the 'larger' of the operands. This rule determines the conversion instructions to be inserted by the compiler.

For the assignment, certain relaxed type compatibility rules are also postulated, relaxed relative to the strict rule that the type of the destination variable must be identical to that of the source expression. Oberon postulates that the type of the variable must include the type of the expression. The following examples are therefore permitted:

```
VAR i, j: INTEGER; k: LONGINT; x, y: REAL; z: LONGREAL;

i := j;    INTEGER ⊆ INTEGER     (no conversion)
k := i;    INTEGER ⊆ LONGINT     (INTEGER to LONGINT)
z := x;    REAL ⊆ LONGREAL       (REAL to LONGREAL)
x := i;    INTEGER ⊆ REAL        (INTEGER to REAL)
```

Conversions between integer types are simple and efficient, because they consist of a sign extension only. Much more involved are conversions from integer to floating-point representation. They consist of a normalization of the mantissa such that 1 ≤ m < 2, and the packing of sign, exponent and mantissa into a single word. Typically, an appropriate instruction is provided for this task.

However, if the type of the expression is 'larger' than the type of the destination variable, the assignment is sometimes impossible, as the assigned value may be outside the range specified by the variable's type. Therefore, a run-time range check is necessary before the conversion. This fact should be made visible in the source language by an explicit conversion function. Oberon features the following functions:

```
SHORT    INTEGER to SHORTINT
         LONGINT to INTEGER
         LONGREAL to REAL
```

ENTIER REAL to LONGINT
LONGREAL to LONGINT

ENTIER(x) yields the largest integer not greater than x.

Oberon's type conversion rules are simple and easily explained. Nevertheless they harbour a pitfall. In a multiplication of two factors of type INTEGER (or REAL) in some cases it is tempting to expect a result of type LONGINT (resp. LONGREAL). However, in the assignments

```
k := i*j + k
z := x*y + z
```

the product is, according to the given rules, of type INTEGER (resp. REAL), and it is converted to its long variant for the subsequent addition. Computation of the product with higher precision is achieved by forcing the conversion to occur *before* multiplication:

```
k := LONG(i) * LONG(j) + k
z := LONG(x) * LONG(y) + z
```

The simplicity of the compatibility rules is a significant advantage for the compiler designer. The principle of expression handling is clearly prescribed. Only the compilation procedures for expressions and terms, and the one for assignment, have to be adjusted by introducing case discriminations. The larger the number of different numeric types, the more cases have to be distinguished.

13.3 The data type SET

The units of storage in computers consist of a small number of bits which are interpretable in different ways. They may represent integers with or without sign, floating-point numbers or logical data. The question about the way to introduce logical bit sequences in higher programming languages has been controversial for a long time. The proposal to introduce them as sets is due to C. A. R. Hoare (Hoare, 1972).

The proposal is attractive, because the set is a mathematically well-founded abstraction. It is appropriately represented in a computer by its *characteristic function* F. If x is a set of elements from the ordered base set M, F(x) is the sequence of truth values b_i with the meaning 'i is contained in x'. If we choose a word (consisting of N bits) to represent values of type SET, the base set consists of the integers 0, 1, ... , N–1. N is typically so small that the range of applications for the type SET is quite restricted. However, the basic set operations of intersection, union and difference are implementable extremely efficiently. Examples of sets represented by bit sequences with word length N are:

x	N–1	...	7 6 5 4 3 2 1 0
{0, 2, 4, 6, ...}	0	...	0 1 0 1 0 1 0 1
{0, 3, 6, ...}	0	...	0 1 0 0 1 0 0 1
{ }	0	...	0 0 0 0 0 0 0 0

Oberon's set operators are implemented by logical instructions available on every computer. This is possible due to the following properties of the characteristic function. Note that we use the Oberon notation for set operations, that is, x+y for x∪y and x*y for x∩y:

∀i ((i ∈ x+y): (i ∈ x) OR (i ∈ y))	Union
∀i ((i ∈ x*y): (i ∈ x) & (i ∈ y))	Intersection
∀i ((i ∈ x–y): (i ∈ x) & ~(i ∈ y))	Difference
∀i ((i ∈ x/y): (i ∈ x) ≠ (i ∈ y))	Symmetric difference

Consequently, the OR instruction can be used for set union, AND for set intersection and XOR for the symmetric difference. The result is a very efficient implementation, because the operation is executed on all elements (bits) simultaneously (in parallel). Examples with the base set {0, 1, 2, 3} are:

{0, 1} + {0, 2} = {0, 1, 2}	0011 OR 0101 = 0111
{0, 1} * {0, 2} = {0}	0011 & 0101 = 0001
{0, 1} – {0, 2} = {1}	0011 & ~0101 = 0010
{0, 1} / {0, 2} = {1, 2}	0011 XOR 0101 = 0110

We conclude by showing the code representing the set expression (a+b) * (c+d):

```
LDW   1, 0, a
LDW   2, 0, b
OR    1, 1, 2
LDW   2, 0, c
LDW   3, 0, d
OR    2, 2, 3
AND   1, 1, 2
```

The membership test i IN x is implemented by a bit test. If such an instruction is not available, the bit sequence must be shifted appropriately with subsequent sign bit test.

The type SET is particularly useful if the base set includes the ordinal numbers of a character set (CHAR). Efficiency is in this case somewhat reduced, because 256 bits (32 bytes) are typically required to represent a set value. Even in 32-bit computers 8 logical instructions are required for the execution of a set operation.

EXERCISES

13.1 Extend the language Oberon-0 and its compiler by the data type REAL (and/or LONGREAL) with its arithmetic operators +, −, * and /. The RISC architecture must be extended accordingly by a set of floating-point instructions and a set of floating-point registers. Choose one of the following alternatives:

(a) The result type of an operation is always that of the operands. The types INTEGER and REAL cannot be mixed. However, there exist the two transfer functions ENTIER(x) and REAL(i).

(b) Operands of the types INTEGER and REAL (and LONGREAL) may be mixed in expressions.

Compare the complexities of the compilers in the two cases.

13.2 Extend the language Oberon-0 and its compiler by the data type SET with its operators + (union), * (intersection) and − (difference), and with the relation IN (membership). Furthermore, set constructors are introduced by the following additional syntax. As an option, expressions in set constructors may be confined to constants.

```
factor = number | set | ...
set = "{" [element {"," element}] "}".
element = expression [".." expression].
```

13.3 Extend the language Oberon-0 and its compiler by the data type CHAR with the functions ORD(ch) (ordinal number of ch in the character set) and CHR(k) (k-th character in the character set). A variable of type CHAR occupies a single byte in store.

Chapter 14
Open Arrays, Pointers and Procedure Types

14.1 Open arrays

An *open array* is an array parameter whose length is unknown (open) at the time of compilation. Here we encounter for the first time a situation where the size of the required storage block is not given. The solution is relatively simple in the case of a reference parameter, because no storage has to be allocated anyway, and merely a reference to the actual array is passed to the called procedure.

However, in order to check index bounds when accessing elements of the open array parameter, the length must be known. Therefore, in addition to the array's address, its length is also passed on. In the case of a multidimensional, open array the length is also necessary to compute element addresses. Hence, the length of the array in every dimension is supplied. The unit consisting of array address and lengths is called an *array descriptor.* Consider the following example:

```
VAR a: ARRAY 10 OF ARRAY 20 OF INTEGER;

PROCEDURE P(VAR x: ARRAY OF ARRAY OF INTEGER);
BEGIN k := x[i]
END P;

P(a)
```

A descriptor with three entries is pushed onto the stack as parameter to P (see Figure 14.1) and the corresponding code is as follows:

```
ADDI   1, 0, 20      R1 := 20
PSH    1, 30, 4      push len      R30 = SP
ADDI   1, 0, 10      R1 := 10
PSH    1, 30, 4      push len
ADDI   1, 0, a       R1 := adr(a)
PSH    1, 30, 4      push adr
BSR    P             call
```

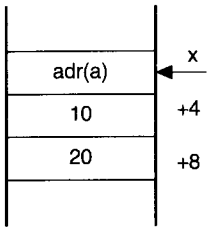


Figure 14.1 Array descriptor for open array.

If an open array parameter is passed by value, its value must be copied into its provided formal location just as in the case of a scalar value. This operation may, however, take considerable effort if the array is large. In the case of structured parameters, programmers should always use the VAR option, unless a copy is essential.

Certainly the code for the copy operation is better inserted after the prologue of the procedure rather than in the place of the call. Consequently, the code pattern for the call is the same for value and reference parameters, with the exception that for the former the copy operation is omitted from the prologue.

The formal location apparently does not hold the array, but instead the array descriptor, whose size is known. The space for the copy is allocated at the top of the stack, and the stack pointer is incremented (or decremented) by the array's size. In the case of multidimensional arrays, the size is computed (at run time) as the product of the individual lengths and the element size.

Here SP is changed at run time by amounts which are unknown at compile time. Therefore it is impossible in the general case to operate with a single address register (SP); the frame pointer FP is indeed necessary.

14.2 Dynamic data structures and pointers

The two forms of data structures provided in Oberon are the array (all elements of the same type, *homogeneous* structure) and the record (*heterogeneous* structure). More complex structures must be programmed individually, that is, they must be generated during program execution. For this reason they are said to be *dynamic* structures. Thereby the structure's components are generated one by one; storage is allocated for components individually. They do not necessarily lie in contiguous locations in store. Relationships between components are expressed explicitly by *pointers*.

For the implementation of this concept a mechanism must be available for the allocation of storage at run time. In Oberon, it is represented by the standard procedure NEW(x). This allocates storage to a dynamic variable, and assigns the address of the allocated block to the pointer variable x. From this it follows that pointers are addresses. Access to a variable referenced by a pointer is necessarily

indirect as in the case of VAR parameters. In fact, a VAR parameter represents a hidden pointer. Consider the following declarations:

```
TYPE T = POINTER TO TDesc;
  TDesc = RECORD x, y: LONGINT END;
VAR a, b: T;
```

The code for the assignment a.x := b.y with access via pointers a and b becomes

```
LDW   1, FP, b        R1 := b
LDW   2, 1, y         R2 := b.y
LDW   3, FP, a        R3 := a
STW   2, 3, x         a.x := R2
```

The step from the referencing pointer variable to the referenced record variable is called *dereferencing*. In Oberon the explicit dereferencing operator is denoted by the symbol ↑. a.x is evidently an abbreviation for the more explicit form a↑.x. The implicit dereferencing operation is recognizable when the selector symbol (dot) is preceded not by a record but by a pointer variable.

Everyone who has written programs which heavily involve pointer handling knows how easily errors can be made with catastrophic consequences. To explain why, consider the following type declarations:

```
T0 = RECORD x, y : LONGINT END;
T1 = RECORD x, y, z : LONGINT END;
```

Let a and b be pointer variables, and let a point to a record of type T0, b to a record of type T1. Then the designator a.z denotes an undefined value of a nonexistent variable, and a.z := b.x stores a value to some undefined location, perhaps corrupting another variable allocated to this location.

This dangerous situation is elegantly eliminated by *binding pointers to a data type*. This permits the validation of pointer values at the time of compilation without loss of run-time efficiency. This brilliant idea is due to C. A. R. Hoare and was implemented for the first time in Algol W (Hoare, 1972). The type to which a pointer is bound is called its *base type*.

```
P0 = POINTER TO T0;
P1 = POINTER TO T1;
```

Now the compiler can check and guarantee that only pointer values can be assigned to a pointer variable p which points to a variable of the base type of p. The value NIL, pointing to no variable at all, is considered as belonging to all pointer types. Referring to the example above, now the designator a.z is detected as incorrect, because z is not a field of the type T0 to which a is bound. If every pointer variable is initialized to NIL, it suffices to precede every access via a

pointer with a test for the pointer value NIL. In this case, the pointer points to no variable, and any designator must be erroneous.

Such a test is indeed quite simple, but because of its frequency it reduces efficiency. The need for an explicit code pattern can be circumvented by (ab)using the storage protection mechanism available on many computers. In this case, the test does not properly check whether a = NIL, but rather whether a.z is a valid, unprotected address. If as usual NIL is represented by the address 0, and if locations 0 ... N–1 are protected, mistaken references via NIL are caught only if their field offsets are less than N. Nevertheless, the method seems to be satisfactory in practice.

The introduction of pointers requires a new class of objects in the symbol table and also a new mode of items. Both are to imply indirect addressing. Because VAR parameters also require indirect addressing, a mode indicating indirection is already present, and it is only natural to use the same mode for access via pointers. However, the name Ind would now appear as more appropriate than Par.

Designator		*Mode*
x	Var	Direct addressing
x↑	Ind	Indirect addressing
x↑.y	Ind	Indirect addressing with offset

Hence, the (usually implied) dereferencing operator converts the mode of an item from Var to Ind. To summarize:

- The notion of a pointer is easily integrated into our system of type compatibility checking. Every pointer type is bound to a base type, namely the type of the referenced variable.
- x↑ denotes dereferencing, implemented by indirect addressing.
- Pointers are type safe if access is preceded by a NIL test, and if pointer variables are initialized to NIL.

Allocation of variables referenced via pointers is obtained by a call of procedure NEW(p). We postulate its existence as run-time support in operating systems. The size of the block to be allocated is given by the base type of p.

So far, we have ignored the problem of storage reclamation. It is actually irrelevant for abstract programs; for concrete ones, however, it is crucial, as stores are inherently finite. Modern operating systems offer a centralized storage management with *garbage collection.* There are various schemes for storage reclamation; but we shall not explain them here. We restrict ourselves to the only question relevant to the compiler designer: which data must be provided to the garbage collector, so that at any time all irrelevant storage blocks can safely be

identified and reclaimed? A variable is no longer relevant when there are no references to it, references emanating from declared pointer variables. In order to determine whether such references exist, the garbage collector requires the following data:

(1) the addresses of all declared pointer variables,
(2) the offsets of all pointer fields in dynamically allocated records, and
(3) the size of every dynamically allocated variable.

This information is available at compile time, and it has to be 'handed down' in such a way that it is available to the garbage collector at run time. In this sense compiler and system must be integrated. The system is here assumed to include storage management, in particular the allocator NEW and the garbage collector.

In order to make this information available at run time, procedure NEW not only allocates a block of storage, but provides it with a type description of the allocated variable. Naturally, such a descriptor must be issued only once, as it need not be duplicated for every instance (variable) of the same type. Therefore, the block is assigned merely a pointer to the type descriptor, and this pointer remains invisible to the programmer. The pointer is called a *type tag* (see Figure 14.2).

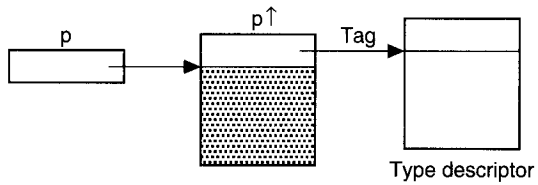


Figure 14.2 Pointer variable, referenced variable, and type descriptor.

The type descriptor specifies the size of the variable and the offset of all pointer fields (Figure 14.3).

Figure 14.3 Variable with type descriptor.

The type descriptor apparently is a reduced form of the object describing the type in the compiler's symbol table, reduced to the data relevant for storage reclamation. This concept has the following consequences:

(1) The compiler must generate a descriptor for every (record) type, and it must add it to the object file.

(2) Procedure NEW(p) obtains, in addition to the address of p, an additional, hidden parameter specifying the address of the descriptor of the base type of p.

(3) The program loader must interpret the added object file information and generate type descriptors.

This, however, is still insufficient. In order that data structures can be traversed, their roots have to be known. Therefore, the object file is also provided with a list of all declared pointer variables. This list is copied upon loading into memory. The list must also include the hidden pointers designating type descriptors. In order that descriptors do not have to be generated for all data types, Oberon restricts pointers to refer to records. This is justified when considering the role of records in dynamic data structures.

14.3 Procedure types

If in a language procedures can be passed as parameters, or if they can occur as values of variables, it becomes necessary to introduce *procedure types.* What are the characteristics of such types, that is, of the values which variables may assume?

Procedure types have been in use since the advent of Algol 60. There, they occurred implicitly only. A parameter in Algol 60 can be a procedure (formal procedure). Its type, however, is not specified; it is merely known that the parameter denotes some procedure or function. The type specification is incomplete or missing, and this constitutes an unfortunate loophole in Algol's type system. In Pascal, it was retained as a concession to Algol compatibility. Modula-2, however, requires a complete, type-safe specification, and besides parameters, variables with procedures as their values are also allowed. Thereby procedure types achieve the same standing as other data types. In this respect, Oberon has adopted the same concept as Modula-2 (Wirth, 1982).

What does this type-safe specification, called the procedure's *signature,* consist of? It contains all specifications necessary to validate the compatibility between actual and formal parameters, namely their number, the type of each parameter, its kind (value or reference) and the type of the result in the case of function procedures. The following example illustrates the case:


```
PROCEDURE F(x, y : REAL): REAL;
BEGIN
...
END F

PROCEDURE H(f: PROCEDURE (u, v : REAL): REAL);
  VAR a, b: REAL;
BEGIN a : = f(a + b, a – b)
END H
```

Upon compilation of the declaration of H the type compatibility between a + b and u, respectively that between a – b and v, is checked, as well as whether the result type of f is assignable to a. In the call H(F) the compatibility between the parameters, and that of the result type of the actual F and the formal f is verified, that is, between x and u and between y and v. Note that the identifiers u and v do not occur in the program, except as the names of the formal parameters of the formal procedure f. Hence, they are actually superfluous, but they may be useful as comments to the reader if meaningful names are chosen.

Pascal, Modula and Oberon assume *name compatibility* as the basis for establishing type consistency. In the case of procedure parameters, an exception was made; *structural compatibility* suffices. If name compatibility were required, the type (signature) of every procedure used as an actual parameter would have to be given an explicit name. This was considered as too cumbersome when the language was designed. However, structural compatibility requires that a compiler be capable of comparing two parameter lists for type correspondence.

A procedure may thus be assigned to a variable under the condition that the two parameter lists correspond. The assigned procedure is activated by referring to the procedure variable. The call is indirect. This is actually the basis of object-oriented programming, where procedures are bound to fields of record variables called *objects.* These bound procedures are called *methods.* In contrast to Oberon, methods, once declared and bound, cannot be altered. All instances of a class refer to the same methods.

The implementation of procedure types and methods turns out to be surprisingly simple, if the problem of type compatibility checking is ignored. The value of a variable or record field with procedure type is simply the entry address of the assigned procedure. This holds only if we require that only *global* procedures, that is, procedures which are not embedded in some context, can be assigned. This readily acceptable restriction is explained with the aid of the following example which breaches this restriction. Upon execution of Q alias v the context containing variables a and b is missing.

```
TYPE T = PROCEDURE (u: INTEGER);
VAR v: T;  r: INTEGER;

PROCEDURE P;
  VAR a, b: INTEGER;
```

```
  PROCEDURE Q(VAR x: INTEGER);
  BEGIN x := a+b END Q;

BEGIN v := Q
END P;

... P; v(r) ...
```

EXERCISES

14.1 Extend the language Oberon-0 and its compiler with open arrays:
- (a) for one-dimensional VAR parameters,
- (b) for multidimensional VAR parameters,
- (c) for value parameters.

14.2 Extend the language Oberon-0 and its compiler with function procedures:
- (a) for scalar result types (INTEGER, REAL, SET),
- (b) for any type.

14.3 A certain module M manages a data structure whose details are to be kept hidden. In spite of this hiding it must be possible to apply any given operation P on all elements of the structure. For this purpose, a procedure Enumerate is exported from M, which allows P to be specified as parameter. As a simple example, we choose for P the counting of the elements currently in the data structure and display the desired solution:

```
PROCEDURE Enumerate(P: PROCEDURE (e: Element));
PROCEDURE CountElements*;
  VAR n: INTEGER;
  PROCEDURE Cnt(e: Element); BEGIN n := n+1 END Cnt;
BEGIN n := 0; M.Enumerate(Cnt); Texts.WriteInt(W, n, 6)
END CountElements;
```

Unfortunately, this solution violates a restriction postulated for the language Oberon. The restriction specifies that procedures used as parameters must be declared globally. This forces us to declare Cnt outside of CountElements and thereby also the counter variable n, although both definitely have no global function.

Implement procedure types in such a way that the mentioned restriction can be lifted, and that the proposed solution is admissible. What is the price?

Chapter 15
Modules and Separate Compilation

15.1 The principle of information hiding

Algol 60 introduced the principles of textual locality of identifiers and that of limited lifetime of the identified objects during execution. The range of visibility of an identifier in the text is called *scope* (Naur, 1960), and it extends over the block in which the identifier is declared. According to the syntax, blocks may be nested, with the consequence that the rules about visibility and scopes must be refined. Algol 60 postulates that identifiers declared in a block B are visible within B and within all blocks contained in B. But they are invisible in the environment of B.

From this rule, the implementer concludes that storage must be allocated to a variable x local to B as soon as control enters B, and that storage may be released as soon as control leaves B. Not only is x invisible outside B, but x ceases to exist when control is outside B. This implies the significant advantage that storage need not remain allocated to all variables of a program.

In some cases, however, the continued existence of a variable during a period of invisibility is highly desirable. Variable x then seems to reappear with its previous value as soon as control re-enters block B. This special case was covered in Algol 60 by the feature of 'own' variables. But this solution was soon discovered to be quite unsatisfactory, in particular in connection with recursive procedures.

An elegant and highly useful solution to the own-problem was discovered around 1972 with the structure of the module. It was adopted in the languages *Modula* (Wirth, 1977) and *Mesa* (Mitchell, Maybury and Sweet, 1978), and later under the name package in *Ada*. Syntactically, a module resembles a procedure and consists of local declarations followed by statements. In contrast to a procedure, however, a module is not called, but its statements are executed once only, namely when the module is loaded. The locally declared objects are static and remain in existence as long as the module remains loaded. The statements in the module body merely serve to initialize the module's variables. These variables are invisible outside the module; effectively they are hidden. D. L. Parnas has coined the term *information hiding*, and it has become an important notion in software construction. Oberon features the possibility of specifying selected

identifiers declared in modules as visible in the module's environment. These identifiers are then said to be *exported.*

The own variable x declared within the Algol procedure P now will be declared, like P itself, local to a module M. P is exported, but not x. In the environment of M the details of the implementation of P as well as the variable x are hidden, but x retains its existence and its value between calls of P.

The desire to hide certain objects and details is particularly justified if a system consist of various parts whose tasks are relatively well separated, and if the parts themselves are of a certain complexity. This is typically the case in an organizational unit which 'manages' a data structure. Then the data structure is hidden within a module, and it is accessible only via exported procedures. The programmer of this module postulates certain *invariants,* such as consistency conditions, which govern the data structure. These invariants can be *guaranteed* to hold, because they cannot be violated by parts of the system outside the module. As a consequence, the programmer's responsibility is effectively limited to the procedures within the module. This *encapsulation* of details solely responsible for the specified invariants is the true purpose of information hiding and of the module concept.

Typical examples of modules and information hiding are the file system hiding the structure of files and their dictionary, the scanner of a compiler hiding the source text and its lexicographic structure, or the code generator of a compiler hiding the generated code and the structure of the target architecture.

15.2 Separate compilation

It is tempting to postulate that modules be nestable like procedures. This facility is offered for example by the language Modula-2. In practice, however, this flexibility has hardly been fruitful. A flat module structure usually suffices. Hence, we consider all modules as being global, and their environment as the universe.

Much more relevant than their nestability is the possibility of developing and compiling modules separately. The latter is clearly feasible only if the modules are global, that is, not nested. The reason for this demand is simply the fact that software is never planned, implemented and tested in straight sequence, but that it is developed in steps, each step incorporating some additions or adaptations. Software is not 'written', but grows. The module concept is of fundamental importance in this connection, because it allows development of individual modules separately under the assumption of constant interfaces of their imports. The set of exported objects effectively constitutes a module's *interface* with its partners. If an interface remains unchanged, a module's implementation can be improved (and corrected) without needing to adapt and recompile the module's clients. This is the real justification for separate compilation.

The advantage of this concept becomes particularly relevant if software is developed by teams. Once agreement is reached about the partitioning of a system into modules and about their interfaces, the team members can proceed independently in implementing the module assigned to them. Even if in practice it

turns out that later changes in the specification of interfaces are avoidable only rarely, the simplification of teamwork through the concept of separate compilation of modules can hardly be overestimated. The successful development of complex systems crucially depends on the concept of modules and their separate compilation.

At this point, the reader may think that all this is not new, that the independent programming of modules and their binding by the program loader, as symbolized in Figure 15.1, has been in common use since the era of assemblers and the first Fortran compilers.

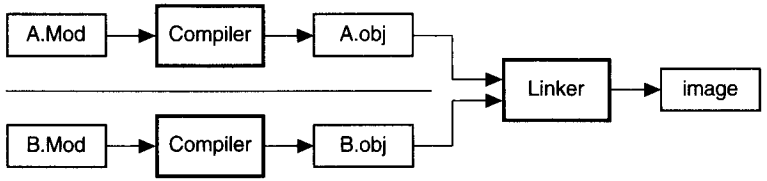


Figure 15.1 Independent compilation of modules A and B.

However, this ignores the fact that higher programming languages offer significantly increased protection against mistakes and inconsistencies through their static type concept. This inestimable – but all too often underestimated – gain is swept aside if type consistency checks are guaranteed only within modules, but not across module boundaries. This implies that type information about all imported objects must be available whenever a module is compiled. In contrast to *independent compilation* (Figure 15.1), where this information is not available, compilation as shown in Figure 15.2 with type consistency checks across module boundaries is called *separate compilation.*

Information about the imported objects is essentially an excerpt of the symbol table as presented in Chapter 8. This excerpt of the symbol table, transformed into a sequential form, is called a *symbol file.* Compilation of a module A which imports (objects from) modules B_1 ... B_n now requires, in addition to the source text of A, the symbol files of B_1 ... B_n. And in addition to the object code (A.obj) it also generates a symbol file (A.sym).

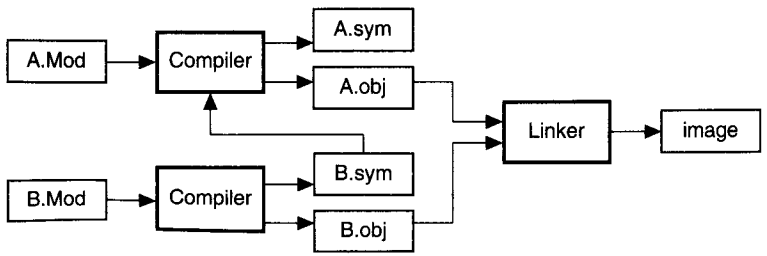


Figure 15.2 Separate compilation of modules A and B.

15.3 Implementation of symbol files

From the foregoing considerations we may first conclude that compilation of a module's import list causes a symbol file to be read for each module identifier in the list. The symbol table of the compiled module is initialized by the imported symbol files. Second, it follows that at the end of compilation the new symbol table is traversed, and a symbol file is output with an entry corresponding to every symbol table element marked for export. Figure 15.3 shows as an example the relevant excerpt of the symbol table during compilation of a module A importing B. Within B, T and f are marked with an asterisk for export.

```
MODULE A;
  IMPORT B;
  VAR x: B.T;
BEGIN x.f := 1; ...
END A.

MODULE B;
  TYPE T* = RECORD f*: INTEGER; ... END;
BEGIN ...
END B.
```

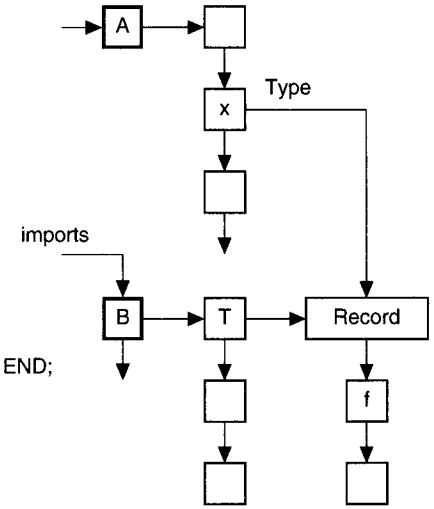


Figure 15.3 Symbol table of A with imports from B.

Let us first consider the generation of the symbol file M.sym of a module M. At first sight, the task merely consists of traversing the table and emitting an entry corresponding to every marked element in an appropriately sequentialized form. The symbol table is essentially a list of objects with pointers to type structures which are trees. In this case the sequentialization of structures using a characteristic prefix for every element is perhaps the most appropriate technique. It is illustrated by an example in Figure 15.4.

A problem arises because every object contains at least a pointer referring to its type. Writing pointer values into a file is problematic, to say the least. Our solution consists in writing the type description into the file the first time it is encountered when scanning the symbol table. Thereby the type entry is marked and obtains a unique *reference number*. The number is stored in an additional record field of the type ObjectDesc. If the type is referenced again later, the reference number is output instead of the structure.


```
VAR x: ARRAY 10 OF INTEGER;
  b: ARRAY 8 OF ARRAY 20 OF REAL
```

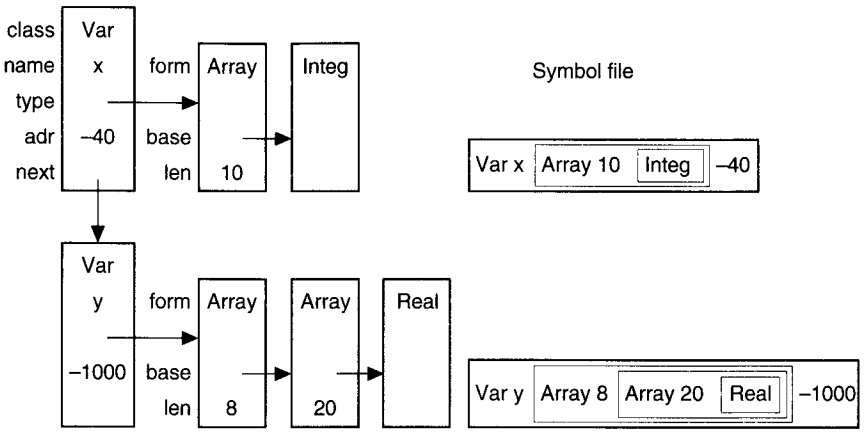


Figure 15.4 Sequentialized form of a symbol table with two arrays.

This technique not only avoids the repeated writing of the same type descriptions, but also solves the problem of recursive references, as shown in Figure 15.5.

```
TYPE P = POINTER TO R;
  R = RECORD x, y: INTEGER; next: P END
```

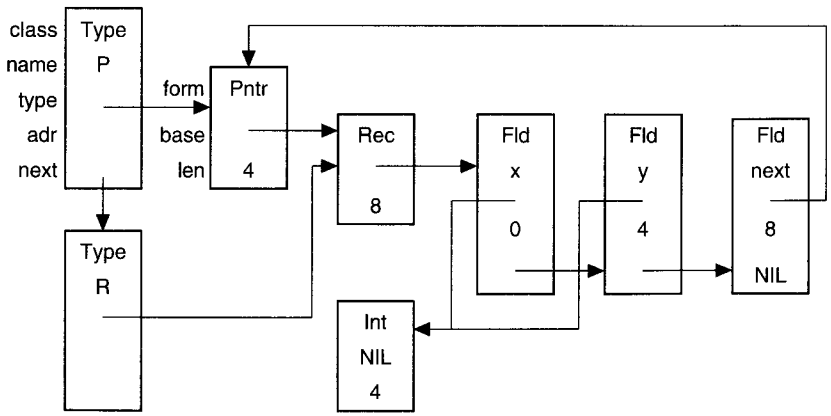


Figure 15.5 Cyclic reference of type node.

Positive values are used for reference numbers. As an indication that the reference number is used for the first time, and that it is therefore immediately

followed by the type description, the number is given a negative sign. While reading a symbol file, a type table T is constructed with references to the respective type structures. If a positive reference number r is read, T[r] is the needed pointer; if r is negative, the subsequent type data is read, and the pointer referring to the newly constructed descriptor is assigned to T[–r].

Type information can, in contrast to data about other objects, be imported and at the same time be re-exported. Therefore it is necessary to specify the module from which the exported type stems. In order to make this possible, we use a so-called *module anchor*. In the heading of every symbol file there is a list of anchor objects, one for each imported module which is re-exported, that is, which contains a type that is referenced by an exported object. Figure 15.6 illustrates such a situation; module C imports modules A and B, whereby a variable x is imported from B whose type stems from A. The type compatibility check for an assignment like y := x rests on the assumption that the type pointers of x and y both refer to the same type descriptor. If they do not, an error is indicated.

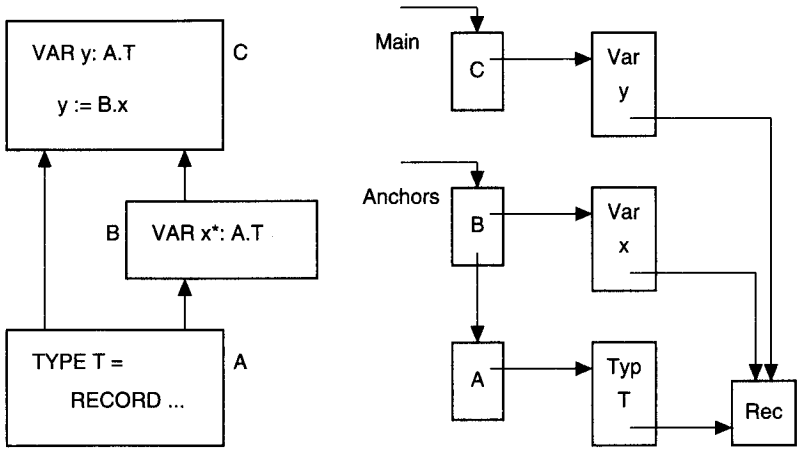


Figure 15.6 Re-export of type A.T from module B.

Hence we conclude that upon compilation of a module M, not only the symbol tables of the explicitly imported modules must be present, but also those of modules from which types are referenced either directly or indirectly. This is a cause for concern, because the compilation of any module might necessitate the reading of symbol files of entire module hierarchies. It might even reach down to the deepest level of an operating environment, from where neither variables nor procedures are imported, but perhaps only a single type. The result would not only be the superfluous loading of large amounts of data, but also a waste of much memory space. It turns out that, although our concern is justified, the consequences are much less dramatic than might be expected (Franz, 1993). The reason is that most symbol tables requested are present already for other reasons.

As a consequence, the additional effort remains small. Nevertheless it is worth pondering over the possibility of avoiding the extra effort. Indeed, the first compilers for Modula and Oberon have adopted the following technique.

Let a module M import types from modules M0, M1, and so on, either directly or indirectly. The solution consists of including in the symbol file of M complete descriptions of the imported types, thereby avoiding references to the secondary modules M0, M1, and so on. However, this fairly obvious solution causes complication. In the example illustrated by Figure 15.6, the symbol file of B evidently contains a complete description of type T. The consistency check for the assignment y := B.x, in order to be highly efficient, merely compares two type pointers. The configuration shown on the right of Figure 15.6 must therefore be present after loading. This implies that in symbol files re-exported types not only specify their home module, but that when loading a symbol file a test must verify whether or not the read type is already present. This may be the case because the symbol file of the module defining the type has already been loaded, or because the type has already been read when loading other symbol files.

At this point we also mention another, small complication in connection with types that arises because types may appear under different names (aliases). Although use of aliases is rare, the language definition (unfortunately) allows it. They are moderately meaningful only if the synonyms stem from different modules, as shown in Figure 15.7.

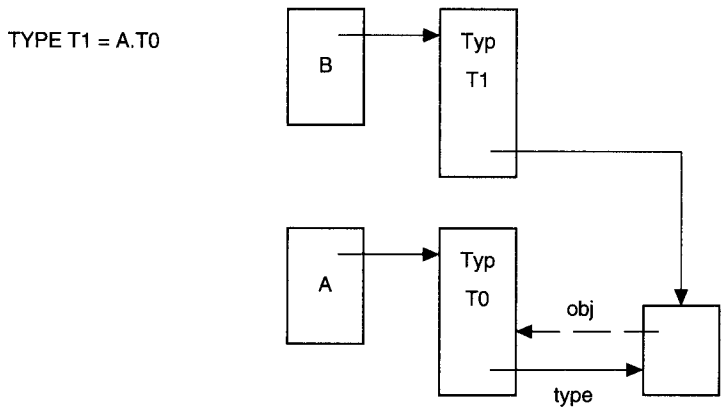


Figure 15.7 Type with aliases.

When loading the symbol file of B it is recognized that B.T1 and A.T0, both pointing to a type object, must actually point to the same object descriptor. In order to determine which of the two descriptors should be discarded and which one retained, type nodes (type Structure) are supplied with a back-pointer to the original type object (type Object), here to T0.

15.4 Addressing external objects

The principal advantage of separate compilation is that changes in a module M do not invalidate clients of M, if the interface of M remains unaffected. We recall that the interface consists of the entire set of exported declarations. Changes which do not affect the interface may occur, so to say, under cover, and without client programmers being aware of them. Such changes must not even require recompilation of the clients using new symbol files. For the sake of honesty, we hasten to add that exported procedures must not have altered in their semantics, because compilers could not detect such changes reliably. Hence, if we say that an interface remains unchanged, we explicitly refer to the declarations of types and variables, and to the signatures of procedures, and only implicitly to their semantics.

If in a certain module non-exported procedures and variables are changed, added or deleted, their addresses necessarily also change, and as a consequence so do those of other, possibly exported variables and procedures. This leads to a change of the symbol table, and thereby also to an invalidation of client modules. But this obviously contradicts the requirements postulated for separate compilation.

The solution to this dilemma lies in avoiding the inclusion of addresses in a symbol file. This has the consequence that addresses must be computed when loading and binding a module. Hence, in addition to its address (for module-internal use), an exported object is given a unique number. This number assumes the place of the address in the symbol file. Typically, these numbers are allocated strictly sequentially.

As a consequence, when compiling a client, only module specific numbers are available, but no addresses. These numbers must, as mentioned before, be converted into absolute addresses upon loading. For this task, knowledge about the positions of such incomplete address fields must be available. Instead of supplying the object file with a list of all locations of such addresses, the elements of this fixup list are embedded in the instructions at the very places of the yet unknown addresses. This mirrors the technique used for the completion of addresses of forward jumps (see Chapter 11). If all such addresses to be completed are collected in a single fixup list, then this corresponds to Figure 15.8 (a). Every element must be identified with a pair consisting of a module number (mno) and an entry number (eno). It is simpler to provide a separate list for every module. In the object file, not just a single fixup root, but one for each list is required. This corresponds to Figure 15.8 (b). Part (c) shows the extreme solution where a separate fixup list is specified for every imported object. Which of the three presented solutions is adopted, depends on how much information can be put into the place of an absolute address, by which it is ultimately replaced.

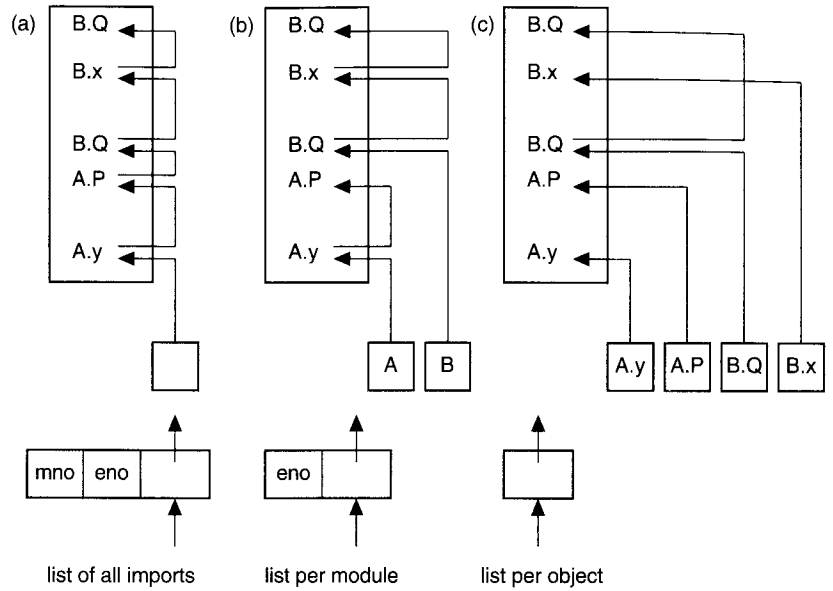


Figure 15.8 Three forms of fixup lists in object files.

15.5 Checking configuration consistency

It may seem belated if we now pose the question: Why are symbol files introduced at all? Let us assume that a module M is to be compiled which imports M0 and M1. A rather straightforward solution would be to recompile M0 and M1 immediately preceding the compilation of M, and to unite the three symbol tables obtained. The compilations of M0 and M1 might easily be triggered by the compilation of M reading the import list.

Although the repeated compilation of the same source text is a waste of time, this technique is used by various commercial compilers for (extended) Pascal and Modula. The serious shortcoming inherent in this method, however, is not so much the additional effort needed, but the lack of a guarantee for the consistency of the modules being bound. Let us assume that M is to be recompiled after some changes have been made in the source text. Then it is quite likely that after the original formulation of M and after its compilation, changes have also been made to M0 and M1. These changes may invalidate M. Perhaps even the source versions of M0 and M1 currently available to the programmer of client M no longer comply with the actual object files of M0 and M1. This fact, however, cannot be determined by a compilation of M, but it almost certainly leads to disaster when the inconsistent parts are bound and executed.

Symbol files, however, do not permit changes like source files; they are encoded and not visible through a text editor. They can only be replaced as a whole. In order to establish consistency, every symbol file is provided with a unique *key*. Symbol files thus make it possible to make modules available without giving away the source text. A client may rely on the specified interface definition and, thanks to the key, the consistency of the definition with the present implementations is also guaranteed. Unless this guarantee is provided, the entire notion of modules and separate compilation is perhaps enticing, but hardly a useful tool.

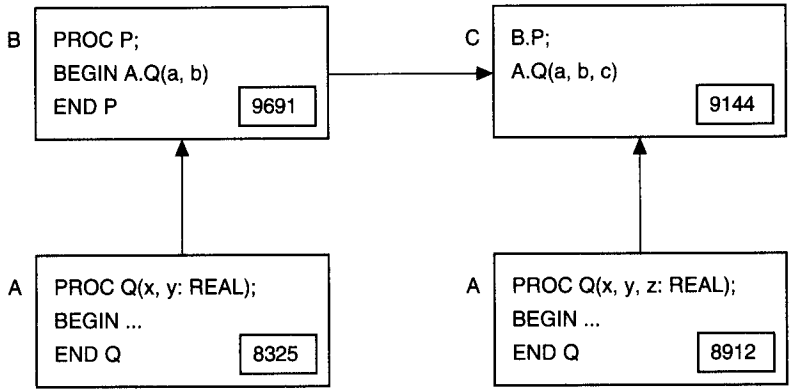


Figure 15.9 Inconsistency of module versions.

As an example, Figure 15.9 shows on its left side the situation upon compilation of module B, and on the right side that upon compilation of module C. Between the two compilations, A was changed and recompiled. The symbol files of B and C therefore contain module anchors of A with *differing* keys, namely 8325 in B and 8912 in C. The compiler checks the keys, notices the difference, and issues an error message. If, however, module A is changed *after* the recompilation of C (with changed interface), then the inconsistency can and must be detected upon loading and binding the module configuration. For this purpose, the same keys are also included in the respective object files. Therefore it is possible to detect the inconsistency of the import of A in B and C *before* execution is attempted. This is absolutely essential.

The key and the name are taken as the characteristic pair of every module, and this pair is contained in the heading of every symbol and object file. As already mentioned, the names of modules in the import list are also supplemented by their key. These considerations lead to the structure of symbol files as specified in Appendix A.

Unique module keys can be generated by various algorithms. The simplest is perhaps the use of current time and date which, suitably encoded, yield the desired key. A drawback is that this method is not entirely reliable. Even if the

resolution of the clock is one second, simultaneous compilations on different computers may generate the same key. Somewhat more significant is the argument that two compilations of the same source text should always generate the same key; but they do not. Hence, if a change is made in a module which is later detected to be in error, recompilation of the original version nevertheless results in a new key which lets old clients appear as invalidated.

A better method to generate a key is to use the symbol file itself as argument, like in the computation of a checksum. But this method is also not entirely safe, because different symbol files may result in the same key. But it features the advantage that every recompilation of the same text generates the same key. Keys computed in this way are called *fingerprints*.

EXERCISES

15.1 Incorporate separate compilation into your Oberon-0 compiler. The langauge is extended to include an import list (see Appendix A) and a marker in the exported identifier's declaration. Use the technique of symbol files and introduce the rule that exported variables may not be assigned values from outside, that is, in importing modules that are considered to be read-only variables.

15.2 Implement a fingerprint facility for generating module keys.

Chapter 16
Optimizations and the Frontend/Backend Structure

16.1 General considerations

If we analyse the code generated by the compiler developed in the preceding chapters, we can easily see that it is correct and fairly straightforward, but in many instances also improvable. The reason primarily lies in the directness of the chosen algorithm which translates language constructs independently of their context into fixed patterns of instruction sequences. It hardly perceives special cases and does not take advantage of them. The directness of this scheme leads to results that are only partially satisfactory as far as economy of storage and execution speed are concerned. This is not surprising, as source and target languages do not correspond in simple ways. In this connection we can observe the semantic gap between programming language on the one hand and instruction set and machine architecture on the other.

In order to generate code which utilizes the available instructions and machine resources more effectively, more sophisticated translation schemes must be employed. They are called *optimizations*, and compilers using them are said to be optimizing compilers. It must be pointed out that this term, although in widespread use, basically is a euphemism. Nobody would be willing to claim that the code generated by them could be optimal in all cases, that is, in no way improvable. The so-called optimizations are nothing more than improvements. However, we shall comply with the common vocabulary and will also use the term optimization.

It is fairly evident that the more sophisticated the algorithm, the better the code obtained. In general it can be claimed that the better the generated code and the faster its execution, the more complex, larger and slower will be the compiler. In some cases, compilers have been built which allow a choice of an *optimization level*: while a program is under development, a low, and after its completion a high, degree of optimization is selected for compilation. As an aside, note that optimization may be selected with different goals, such as towards faster execution or towards denser code. The two criteria usually require different code generation algorithms and are often contradictory, a clear indication that there is no such thing as a well-defined optimum.

It is hardly surprising that certain measures for code improvement may yield considerable gains with modest effort, whereas others may require large

increases in compiler complexity and size while yielding only moderate code improvements, simply because they apply in rare cases only. Indeed, there are tremendous differences in the ratio of effort to gain. Before the compiler designer decides to incorporate sophisticated optimization facilities, or before deciding to purchase a highly optimizing, slow and expensive compiler, it is worth while clarifying this ratio, and whether the promised improvements are truly needed.

Furthermore, we must distinguish between optimizations whose effects could also be obtained by a more appropriate formulation of the source program, and those where this is impossible. The first kind of optimization mainly serves the untalented or sloppy programmer, but merely burdens all the other users through the increased size and decreased speed of the compiler. As an extreme example, consider the case of a compiler which eliminates a multiplication if one factor has the value 1. The situation is completely different for the computation of the address of an array element, where the index must be multiplied by the size of the elements. Here, the case of a size equal to 1 is frequent, and the multiplication cannot be eliminated by a clever trick in the source program.

A further criterion in the classification of optimization facilities is whether or not they depend on a given target architecture. There are measures which can be explained solely in terms of the source language, independent of any target. Examples of target-independent optimizations are suggested by the following well-known identities:

```
x + 0 = x
x * 2 = x + x

b & TRUE = b
b & ~b = FALSE

IF TRUE THEN A ELSE B END  = A
IF FALSE THEN A ELSE B END = B
```

On the other hand, there are optimizations that are justified only through the properties of a given architecture. For example, computers exist which combine a multiplication and an addition, or an addition, a comparison and a conditional branch in a single instruction. A compiler must then recognize the code pattern which allows the use of such a special instruction.

Lastly, we must also point out that the more optimizations with sizeable effects that can be incorporated in a compiler, the poorer its original version must have been. In this connection, the cumbersome structures of many commercial compilers, whose origin is difficult to fathom, lead to surprisingly poor initial performance, which makes optimizing features seem absolutely indispensable.

16.2 Simple optimizations

First, let us consider optimizations that are implementable with modest effort, and which therefore are practically mandatory. This category includes the cases which

can be recognized by inspection of the immediate context. A prime example is the evaluation of expressions with constants. This is called *constant folding* and is already contained in the compiler presented.

Another example is multiplication by a power of 2, which can be replaced by a simple, efficient shift instruction. Also, this case can be recognized without considering any context:

```
IF (y.mode = Const) & (y.a # 0) THEN
  n := y.a; k := 0;
  WHILE ~ODD(n) DO n := n DIV 2; k := k+1 END;
  IF n = 1 THEN PutShift(x, k) ELSE PutOp(MUL, x, y) END
ELSE ...
END
```

Division (of integers) is treated in the same way. If the divisor is 2^k for some integer k, the dividend is merely shifted k bits to the right. For the modulo operator, the least significant k bits are simply masked out.

16.3 Avoiding repeated evaluation

Perhaps the best known case among the target-independent optimizations is the *elimination of common subexpressions.* At first sight, this case may be classified among the elective optimizations, because the re-evaluation of the same subexpression can be achieved by a simple change of the source program. For example, the assignments

```
x := (a + b) / c;  y := (a + b) / d
```

can easily be replaced by three simpler assignments when using an auxiliary variable u:

```
u := a + b;  x := u / c;  b := u / d
```

Certainly, this is an optimization with respect to the number of arithmetic operations, but not with respect to the number of assignments or the clarity of the source text. Therefore the question remains open as to whether this change constitutes an improvement at all.

More critical is the case where the improvement is impossible to achieve by a change of the source text, as is shown in the following example:

```
a[i, j] := a[i, j] + b[i, j]
```

Here, the same address computation is performed three times, and each time it involves at least one multiplication and one addition. The common subexpressions

are implicit and not directly visible in the source. An optimization can be performed only by the compiler.

Elimination of common expressions is only worth while if they are evaluated repeatedly. This may even be the case if the expression occurs only once in the source:

```
WHILE i > 0 DO z := x+y; i := i–1 END
```

Since x and y remain unchanged during the repetition, the sum need be computed only once. The compiler must pull the assignment to z out of the loop. The technical term for this feat is *loop invariant code motion.*

In all the latter cases code can only be improved by selective analysis of context. But this is precisely what increases the effort during compilation very significantly. The compiler presented for Oberon-0 does not constitute a suitable basis for this kind of optimization.

Related to the pulling out of constant expressions from loops is the technique of simplifying expressions by using the values computed in the previous repetition, that is, by considering recurrence relations. If, for example, the address of an array element is given by adr(a[i]) = k*i + a0, then adr(a[i+1]) = adr(a[i]) + k. This case is particularly frequent and therefore relevant. For instance, the addresses of the indexed variables in the statement

```
FOR i := 0 TO N–1 DO a[i] := b[i] * c[i] END
```

can be computed by a simple addition of a constant to their previous values. This optimization leads to significant reductions in computation time. A test with the following example of a matrix multiplication showed surprising results:

```
FOR i := 0 TO 99 DO
  FOR j := 0 TO 99 DO
    FOR k := 0 TO 99 DO a[i, j] := a[i, j] + b[i, k] * c[k, j] END
  END
END
```

The use of registers instead of memory locations to hold index values and sums, and the elimination of index bound tests resulted in a speed increased by a factor of 1.5. The replacement of indexed addressing by linear progression of addresses as described above yielded a factor of 2.75. And the additional use of a combined multiplication and addition instruction to compute the scalar products increased the factor to 3.90.

Unfortunately, not even consideration of simple context information suffices in this case. A sophisticated control and data flow analysis is required, as well as detection of the fact that in each repetition an index is incremented monotonically by 1.

16.4 Register allocation

The dominant theme in the subject of optimization is the use and allocation of processor registers. In the Oberon-0 compiler presented registers are used exclusively to hold anonymous intermediate results during the evaluation of expressions. For this purpose, usually a few registers suffice. Modern processors, however, feature a significant number of registers with access times considerably shorter than that of main memory. Using them for intermediate results only would imply a poor utilization of the most valuable resources. A primary goal of good code optimization is the most effective use of registers in order to reduce the number of accesses to the relatively slow main memory. A good strategy of register usage yields more advantages than any other branch of optimization.

A widespread technique is register allocation using graph colouring. For every value occurring in a computation, that is, for every expression, the point of its generation and the point of its last use are determined. They delimit its *range of relevance*. Obviously, values of different expressions can be stored in the same register, if and only if their ranges do not overlap. The ranges are represented by the nodes of a graph, in which an edge between two nodes signifies that the two ranges overlap. The allocation of N available registers to the occurring values may then be understood as the colouring of the graph with N colours in such a way that neighbouring nodes always have different colours. This implies that values with overlapping ranges are always allocated to different registers.

Furthermore, selected, scalar, local variables are no longer allocated in memory at all, but rather in dedicated registers. In order to approach an optimal register utilization, sophisticated algorithms are employed to determine which variables are accessed most frequently. Evidently, the necessary bookkeeping about variable accesses grows, and thereby compilation speed suffers. Also, care has to be taken that register values are saved in memory before procedure calls and are restored after the procedure return. The lurking danger is that the effort necessary for this task surpasses the savings obtained. In many compilers, local variables are allocated to registers only in procedures which do not contain any calls themselves (leaf procedures), and which therefore are also called most frequently, as they constitute the leaves in the tree representing the procedure call hierarchy.

A detailed treatment of all these optimization problems is beyond the scope of an introductory text about compiler construction. The above outline shall therefore suffice. In any case such techniques make it clear that for a nearly optimal code generation significantly more information about context must be considered than is the case in our relatively simple Oberon-0 compiler. Its structure is not well suited to achieving a high degree of optimization. But it serves excellently as a fast compiler producing quite acceptable, although not optimal code, as is appropriate in the development phase of a system, particularly for educational purposes. Section 16.5 indicates another, somewhat more complex compiler structure which is better suited for the incorporation of optimization algorithms.

16.5 The frontend/backend compiler structure

The most significant characteristic of the compiler developed in Chapters 7–12 is that the source text is read exactly once. Code is thereby generated on the fly. At each point, information about the operands is restricted to the items denoting the operand and to the symbol table representing declarations. The so-called frontend/backend compiler structure, which was briefly mentioned in Chapter 1, deviates decisively in this respect. The *frontend* part also reads the source text once only, but instead of generating code it builds a data structure representing the program in a form suitably organized for further processing. All information contained in statements is mapped into this data structure. It is called a *syntax tree*, because it also mirrors the syntactic structure of the text. Somewhat oversimplifying the situation, we may say that the front end compiles declarations into the symbol table and statements into the syntax tree. These two data structures constitute the interface to the *backend* part whose task is code generation. The syntax tree allows fast access to practically all parts of a program, and it represents the program in a preprocessed form. The resulting compilation process is shown in Figure 16.1.

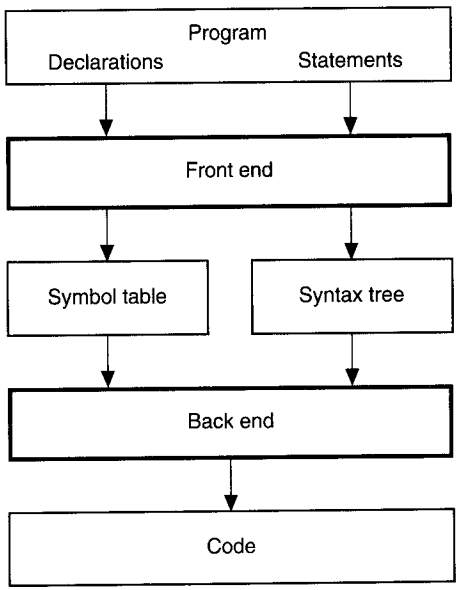


Figure 16.1 Compiler consisting of front end and back end.

We pointed out one significant advantage of this structure in Chapter 1: the partitioning of a compiler into a target-independent front end and a target-dependent back end. In the following, we focus on the interface between the two parts, namely the structure of the syntax tree. Furthermore, we show how the tree is generated.

Exactly as in a source program where statements refer to declarations, so does the syntax tree refer to entries in the symbol table. This gives rise to the understandable desire to declare the elements of the symbol table (objects) in such a fashion that it is possible to refer to them from the symbol table itself as well as from the syntax tree. As our basic type we introduce the type Object which may assume different forms as appropriate to represent constants, variables, types, and procedures. Only the attribute type is common to all. Here and subsequently we make use of Oberon's feature called *type extension* (Reiser and Wirth, 1992).

```
Object = POINTER TO ObjDesc;
ObjDesc = RECORD type: Type END;
ConstDesc = RECORD (ObjDesc) value: LONGINT END;
VarDesc = RECORD (ObjDesc) adr, level: LONGINT END;
```

The symbol table consists of lists of elements, one for each scope (see Section 8.2). The elements consist of the name (identifier) and a reference to the identified object.

```
Ident = POINTER TO IdentDesc;
IdentDesc = RECORD
  name: ARRAY 32 OF CHAR;
  obj: Object;
  next: Ident
END;
Scope = POINTER TO ScopeDesc;
ScopeDesc = RECORD first: Ident; dsc: Scope END;
```

The syntax tree is best conceived as a binary tree. We call its elements Nodes. If a syntactic construct has the form of a list, it is represented as a degenerate tree in which the last element has an empty branch.

```
Node = POINTER TO NodeDesc;
NodeDesc = RECORD (Object)
  op: INTEGER;
  left, right: Object
END
```

Let us consider the following brief excerpt of a program text as an example:

```
VAR x, y, z: INTEGER;
BEGIN z := x + y – 5; ...
```

The front end parses the source text and builds the symbol table and the syntax tree as shown in Figure 16.2. Representations of data types are omitted.

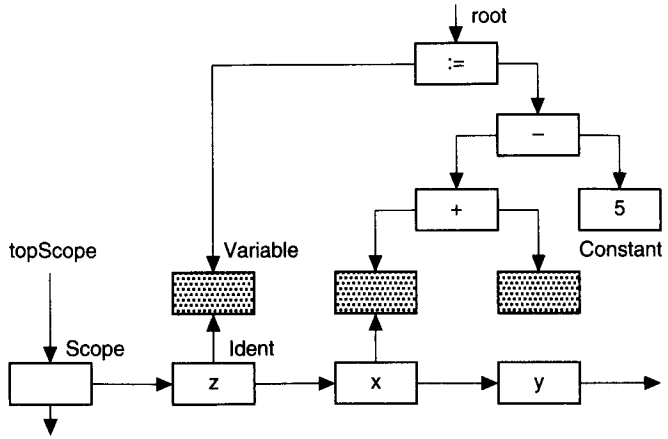


Figure 16.2 Symbol table (below) and syntax tree (above).

Representations of procedure calls, the IF and WHILE statements and the statement sequence are shown in Figures 16.3–16.5.

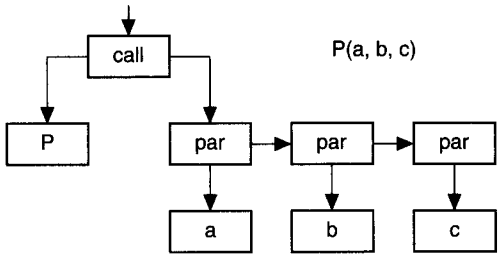


Figure 16.3 Procedure call.

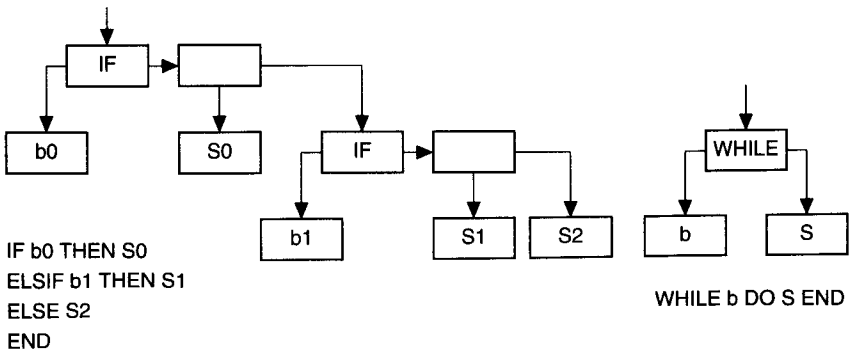


Figure 16.4 IF and WHILE statements.

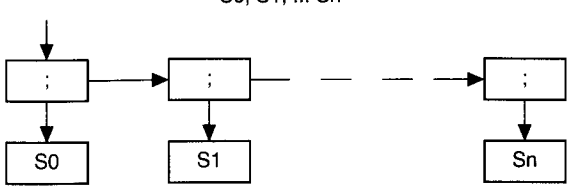


Figure 16.5 Statement sequence.

To conclude, the following examples demonstrate how the described data structures are generated. The reader should compare these compiler excerpts with the corresponding procedures of the Oberon-0 compiler listed in Appendix C. All subsequent algorithms make use of the auxiliary procedure New, which generates a new node.

```
PROCEDURE New(op: INTEGER; x, y: Object): Item;
  VAR z: Item;
BEGIN New(z); z.op := op; z.left := x; z.right := y; RETURN z
END New;

PROCEDURE factor(): Object;
  VAR x: Object; c: Constant;
BEGIN
  IF sym = ident THEN x := This(name); Get(sym); x := selector(x)
  ELSIF sym = number THEN NEW(c); c.value := number; Get(sym); x := c
  ELSIF sym = lparen THEN Get(sym); x := expression();
    IF sym = rparen THEN Get(sym) ELSE Mark(22) END
  ELSIF sym = not THEN Get(sym); x := New(not, NIL, factor())
  ELSE ...
  END;
  RETURN x
END factor;

PROCEDURE term(): Object;
  VAR op: INTEGER; x: Object;
BEGIN x := factor();
  WHILE (sym >= times) & (sym <= and) DO
    op := sym; Get(sym); x := New(op, x, factor())
  END;
  RETURN x
END term;

PROCEDURE statement(): Object;
  VAR x: Object;
```



```
BEGIN
  IF sym = ident THEN
    x := This(name); Get(sym); x := selector(x);
    IF sym = becomes THEN
      Get(sym); x := New(becomes, x, expression())
    ELSIF ...
    END
  ELSIF sym = while THEN
    Get(sym); x := expression();
    IF sym = do THEN Get(sym) ELSE Mark(25) END;
    x := New(while, x, statseq());
    IF sym = end THEN Get(sym) ELSE Mark(20) END
  ELSIF ...
  END;
  RETURN x
END statement;
```

These excerpts clearly show that the structure of the front end is predetermined by the parser. The program has even become slightly simpler. But it must be kept in mind that type checking has been omitted in the above procedures for the sake of brevity. However, as a target-independent task, type checking clearly belongs to the front end.

EXERCISES

16.1 Improve code generation of the Oberon-0 compiler such that values and addresses, once loaded into a register, may possibly be reused without reloading. For the example

```
z := (x – y) * (x + y); y := x
```

the compiler presented generates the instruction sequence

```
LDW  1, 0, x
LDW  2, 0, y
SUB  1, 1, 2
LDW  2, 0, x
LDW  3, 0, y
ADD  2, 2, 3
MUL  1, 1, 2
STW  1, 0, z
LDW  1, 0, x
STW  1, 0, y
```

The improved version is to generate

```
LDW  1, 0, x
LDW  2, 0, y
SUB  3, 1, 2
ADD  4, 1, 2
MUL  5, 3, 4
STW  5, 0, z
STW  1, 0, y
```

Measure the performance improvement achieved by a reasonably large number of test cases.

16.2 Which additional instructions of the RISC architecture of Chapter 9 would be desirable to facilitate the implementations of the preceding exercises, and to generate shorter and more efficient code?

16.3 Optimize the Oberon-0 compiler in such a way that scalar variables are allocated in registers instead of memory if possible. Measure the performance improvement achieved and compare it with the one obtained in Exercise 16.1. How are variables treated as VAR parameters?

16.4 Construct a module OSGx which replaces OSG (see listing in Appendix C) and generates code for a CISC architecture x. The given interface of OSG should be retained as far as possible in order that modules OSS and OSP remain unchanged.

Appendix A
Syntax

A.1 Oberon-0

```
ident = letter {letter | digit}.
integer = digit {digit}.

selector = {"." ident | "[" expression "]"}.
factor = ident selector | integer | "(" expression ")" | "~" factor.
term = factor {("*" | "DIV" | "MOD" | "&") factor}.
SimpleExpression = ["+"|"–"] term {("+"|"–" | "OR") term}.
expression = SimpleExpression
  [("=" | "#" | "<" | "<=" | ">" | ">=") SimpleExpression].
assignment = ident selector ":=" expression.
ActualParameters = "(" [expression {"," expression}] ")".
ProcedureCall = ident [ActualParameters].
IfStatement = "IF" expression "THEN" StatementSequence
  {"ELSIF" expression "THEN" StatementSequence}
  ["ELSE" StatementSequence] "END".
WhileStatement = "WHILE" expression "DO" StatementSequence "END".
statement = [assignment | ProcedureCall | IfStatement | WhileStatement].
StatementSequence = statement {";" statement}.

IdentList = ident {"," ident}.
ArrayType = "ARRAY" expression "OF" type.
FieldList = [IdentList ":" type].
RecordType = "RECORD" FieldList {";" FieldList} "END".
type = ident | ArrayType | RecordType.
FPSection = ["VAR"] IdentList ":" type.
FormalParameters = "(" [FPSection {";" FPSection}] ")".
ProcedureHeading = "PROCEDURE" ident [FormalParameters].
ProcedureBody = declarations ["BEGIN" StatementSequence] "END".
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
declarations = ["CONST" {ident "=" expression ";"}]
  ["TYPE" {ident "=" type ";"}]
  ["VAR" {IdentList ":" type ";"}]
```

```
  {ProcedureDeclaration ";"}.
module = "MODULE" ident ";" declarations
  ["BEGIN" StatementSequence] "END" ident ".".
```

A.2 Oberon

```
ident = letter {letter | digit}.
number = integer | real.
integer = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = ("E" | "D") ["+" | "–"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant = "'" character "'" | digit {hexDigit} "X".
string = '"' {character} '"'.

identdef = ident ["*"].
qualident = [ident "."] ident.
ConstantDeclaration = identdef "=" ConstExpression.
ConstExpression = expression.
TypeDeclaration = identdef "=" type.
type = qualident | ArrayType | RecordType | PointerType | ProcedureType.
ArrayType = ARRAY length {"," length} OF type.
length = ConstExpression.
RecordType = RECORD ["(" BaseType ")"] FieldListSequence END.
BaseType = qualident.
FieldListSequence = FieldList {";" FieldList}.
FieldList = [IdentList ":" type].
IdentList = identdef {"," identdef}.
PointerType = POINTER TO type.
ProcedureType = PROCEDURE [FormalParameters].
VariableDeclaration = IdentList ":" type.

designator = qualident {"." ident | "[" ExpList "]" | "(" qualident ")" | "↑" }.
ExpList = expression {"," expression}.
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression = ["+"|"–"] term {AddOperator term}.
AddOperator = "+" | "–" | OR.
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | DIV | MOD | "&".
factor = number | CharConstant | string | NIL | set |
  designator [ActualParameters] | "(" expression ")" | "~" factor.
set = "{" [element {"," element}] "}".
```



```
element = expression [".." expression].
ActualParameters = "(" [ExpList] ")".
statement = [assignment | ProcedureCall |
  IfStatement | CaseStatement | WhileStatement | RepeatStatement |
  LoopStatement | ForStatement | WithStatement | EXIT |
  RETURN [expression] ].
assignment = designator ":=" expression.
ProcedureCall = designator [ActualParameters].
StatementSequence = statement {";" statement}.
IfStatement = IF expression THEN StatementSequence
  {ELSIF expression THEN StatementSequence}
  [ELSE StatementSequence] END.
CaseStatement = CASE expression OF case {"|" case}
  [ELSE StatementSequence] END.
case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
WhileStatement = WHILE expression DO StatementSequence END.
RepeatStatement = REPEAT StatementSequence UNTIL expression.
LoopStatement = LOOP StatementSequence END.
ForStatement = FOR ident ":=" expression TO expression
  [BY ConstExpression] DO StatementSequence END.
WithStatement = WITH qualident ":" qualident DO
  StatementSequence END.

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading = PROCEDURE ["*"] identdef [FormalParameters].
ProcedureBody = DeclarationSequence
  [BEGIN StatementSequence] END.
ForwardDeclaration = PROCEDURE "↑" ident ["*"] [FormalParameters].
DeclarationSequence = {CONST {ConstantDeclaration ";"} |
  TYPE {TypeDeclaration ";"} | VAR {VariableDeclaration ";"}}
  {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection = [VAR] ident {"," ident} ":" FormalType.
FormalType = {ARRAY OF} (qualident | ProcedureType).
ImportList = IMPORT import {"," import} ";".
import = ident [":=" ident].
module = MODULE ident ";" [ImportList] DeclarationSequence
  [BEGIN StatementSequence] END ident ".".
```

A.3 Symbol files

```
SymFile = BEGIN key {name key}                    imported modules
  [ CONST {type name value} ]                     constants
  [ VAR {type name} ]                             variables
  [ PROC {type name                               procedures
    {[VAR] type name} END} ]                      and parameters
  [ ALIAS {type name} ] [NEWTYP {type} ]          renamed types
  END.
type = basicType | [Module] OldType | NewType.
basicType = BOOL | CHAR | INTEGER | REAL | ...
NewType = ARRAY type name intval
  | DYNARRAY type name
  | POINTER type name
  | RECORD type name                              record types
    {type name} END                               and fields
  | PROCTYP type name                             procedure types
    {[VAR] type name} END.                        and parameters
```

Words consisting of upper-case letters denote terminal symbols. In the symbol file, they are encoded as integers. OldType and Module denote type and module numbers, that is, they are references to previously defined objects.

Appendix B
The ASCII Character Set

	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
A	lf	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[	k	{
C	ff	fs	,	<	L	\	l	|
D	cr	gs	–	=	M	]	m	}
E	so	rs	.	>	N	↑	n	~
F	si	us	/	?	O	_	o	del