Compiler Construction

Niklaus Wirth

This is a slightly revised version of the book published by Addison-Wesley in 1996

ISBN 0-201-40353-6

Zürich, May 2017

Preface

This book has emerged from my lecture notes for an introductory course in compiler design at ETH Zürich. Several times I have been asked to justify this course, since compiler design is considered a somewhat esoteric subject, practised only in a few highly specialized software houses. Because nowadays everything which does not yield immediate profits has to be justified, I shall try to explain why I consider this subject as important and relevant to computer science students in general.

It is the essence of any academic education that not only knowledge, and, in the case of an engineering education, know-how is transmitted, but also understanding and insight. In particular, knowledge about system surfaces alone is insufficient in computer science; what is needed is an understanding of contents. Every academically educated computer scientist must know how a computer functions, and must understand the ways and methods in which programs are represented and interpreted. Compilers convert program texts into internal code. Hence they constitute the bridge between software and hardware.

Now, one may interject that knowledge about the method of translation is unnecessary for an understanding of the relationship between source program and object code, and even much less relevant is knowing how to actually construct a compiler. However, from my experience as a teacher, genuine understanding of a subject is best acquired from an in-depth involvement with both concepts and details. In this case, this involvement is nothing less than the construction of an actual compiler.

Of course we must concentrate on the essentials. After all, this book is an introduction, and not a reference book for experts. Our first restriction to the essentials concerns the source language. It would be beside the point to present the design of a compiler for a large language. The language should be small, but nevertheless it must contain all the truly fundamental elements of programming languages. We have chosen a subset of the language Oberon for our purposes. The second restriction concerns the target computer. It must feature a regular structure and a simple instruction set. Most important is the practicality of the concepts taught. Oberon is a general-purpose, flexible and powerful language, and our target computer reflects the successful RISC-architecture in an ideal way. And finally, the third restriction lies in renouncing sophisticated techniques for code optimization. With these premisses, it is possible to explain a whole compiler in detail, and even to construct it within the limited time of a course.

Chapters 2 and 3 deal with the basics of language and syntax. Chapter 4 is concerned with syntax analysis, that is the method of parsing sentences and programs. We concentrate on the simple but surprisingly powerful method of recursive descent, which is used in our exemplary compiler. We consider syntax analysis as a means to an end, but not as the ultimate goal. In Chapter 5, the

transition from a parser to a compiler is prepared. The method depends on the use of attributes for syntactic constructs.

After the presentation of the language Oberon-0, Chapter 7 shows the development of its parser according to the method of recursive descent. For practical reasons, the handling of syntactically erroneous sentences is also discussed. In Chapter 8 we explain why languages which contain declarations, and which therefore introduce dependence on context, can nevertheless be treated as syntactically context free.

Up to this point no consideration of the target computer and its instruction set has been necessary. Since the subsequent chapters are devoted to the subject of code generation, the specification of a target becomes unavoidable (Chapter 9). It is a RISC architecture with a small instruction set and a set of registers. The central theme of compiler design, the generation of instruction sequences, is thereafter distributed over three chapters: code for expressions and assignments to variables (Chapter 10), for conditional and repeated statements (Chapter 11) and for procedure declarations and calls (Chapter 12). Together they cover all the constructs of Oberon-0.

The subsequent chapters are devoted to several additional, important constructs of general-purpose programming languages. Their treatment is more cursory in nature and less concerned with details, but they are referenced by several suggested exercises at the end of the respective chapters. These topics are further elementary data types (Chapter 13), and the constructs of open arrays, of dynamic data structures, and of procedure types called methods in object-oriented terminology (Chapter 14).

Chapter 15 is concerned with the module construct and the principle of information hiding. This leads to the topic of software development in teams, based on the definition of interfaces and the subsequent, independent implementation of the parts (modules). The technical basis is the separate compilation of modules with complete checks of the compatibility of the types of all interface components. This technique is of paramount importance for software engineering in general, and for modern programming languages in particular.

Finally, Chapter 16 gives a brief overview of problems of code optimization. It is necessary because of the semantic gap between source languages and computer architectures on the one hand, and our desire to use the available resources as well as possible on the other.

Acknowledgements

I express my sincere thanks to all who contributed with their suggestions and criticism to this book which matured over the many years in which I have taught the compiler design course at ETH Zürich. In particular, I am indebted to Hanspeter Mössenböck and Michael Franz who carefully read the manuscript and subjected it to their scrutiny. Furthermore, I thank Stephan Gehring, Stefan Ludwig and Josef Templ for their valuable comments and cooperation in teaching the course.

N. W. December 1995

Preface to the Revised Edition of 2011

This book appeared first in 1976 in German. The source language used as a simple example was PL0, a subset of Pascal. The target computer had a stack architecture similar to the P-code interpreter used for many Pascal implementations. A strongly revised edition of the book appeared in 1995. PL0 was replaced by Oberon-0, a subset of Pascal's descendant Oberon. In the target computer a RISC architecture replaced the stack architecture. *Reduced instruction set computers* had become predominant in the early 1990s. They shared with the stack computer the underlying simplicity. The generated RISC-code was to be interpreted like the P-code by an emulator program. The target computer remained an abstract machine.

In the present new edition Oberon-0 is retained as the source language. The instruction set of the target computer is slightly extended. It is still called RISC, but the instruction set is complete like that of a conventional computer. New, however, is that this computer is available as genuine hardware, and not only as a programmed emulator. This had become possible through the use of a

field programmable gate array (FPGA). The target computer is now specified as a text in the language Verilog. From this text the circuit is automatically compiled and then loaded into the FPGA's configuration memory. The RISC thereby gains in actuality and reality. This in particular, because of the availability of a low-cost development board containing the FPGA chip. Therefore, the presented system becomes attractive for courses, in which hardware-software codesign is taught, where a complete understanding of hardware and software is the goal.

May this text be instructive not only for future compiler designers, but for all who wish to gain insight into the detailed functioning of hardware together with software.

Niklaus Wirth, Zürich, February 2014

http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf
http://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/RISC.pdf
http://www.digilentinc.com/Products/Detail.cfm?Prod=S3BOARD
http://www.xilinx.com/products/silicon-devices/fpga/spartan-3.html

Preface to the Revised Edition of 2017

In the last years, the Oberon System had been revised and implemented on an FPGA-development board featuring the RISC Computer. The Oberon-0 compiler has been adapted accordingly, as iIt does not make sense to provide an interpreter for RISC on a RISC itself. The compiler therefore now generates code in the format required by the regular Oberon loader.

The language Oberon-0, a subset of Oberon, remains unchanged with the exception of input and output statements. They now embody the successful Oberon scanner concept. Execution is triggered by the Oberon concept of commands.

Contents

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, compatibility rules among types of operators and operands define the language. 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.

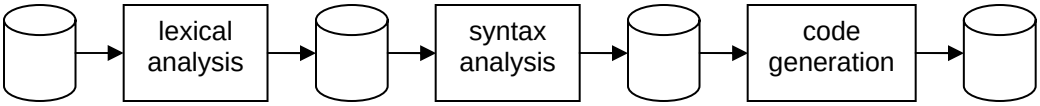


Figure 1.1. Multipass compilation.

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 it starts already 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).

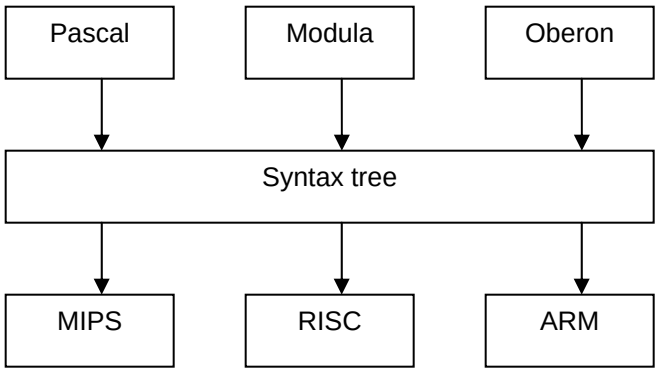


Figure 1.2. Front ends and back ends.

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.

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	Mary eats
John talks	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 ∅ stands for the empty sequence.

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

The means to do so is *recursion* which allows a substitution (here of A by "a"A) 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 in their constituents as shown in the example of Figure 2.1. The graphical representations are called *structural trees* or *syntax trees*.

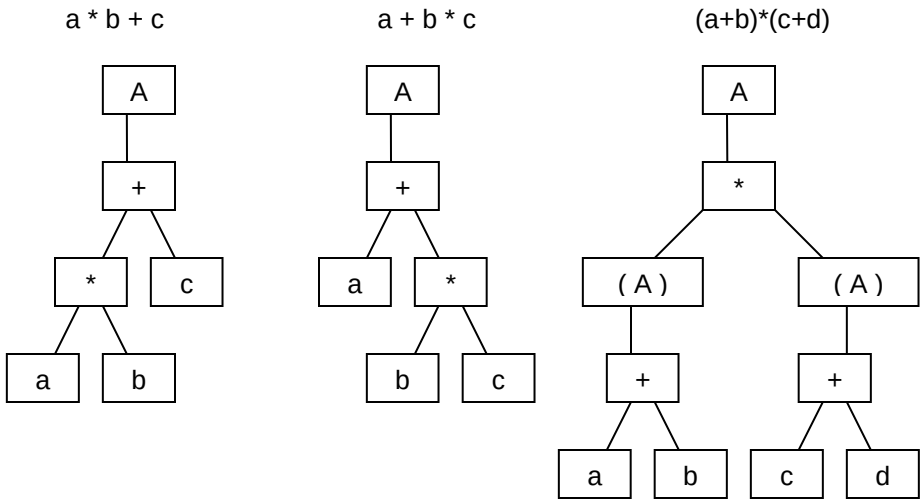


Figure 2.1. Structure of expressions

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 | ∅.
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

$$A = AB \mid \varnothing.$$

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 ∅ 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.

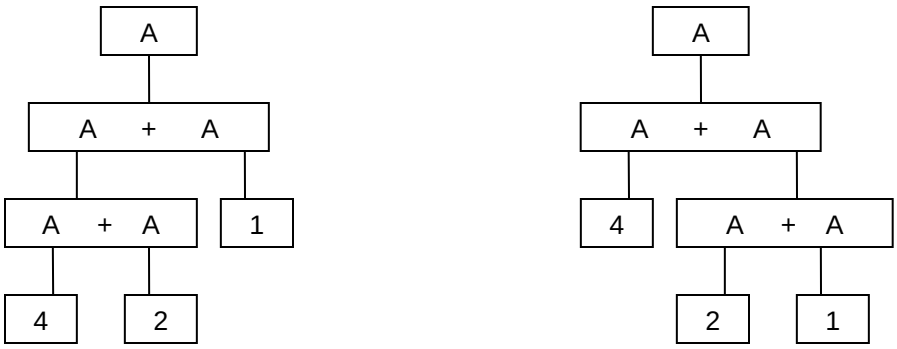


Figure 2.2. Differing structural trees for the same expression.

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).

2.1. 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:

a + b = c + d
a * -b
a < b & c < d

Evaluate the following expressions:

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

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 indispensible. 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. Therein 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
fac0 fac1 ... facn	Pr(fac0); Pr(fac1); ... Pr(facn

```
term0 | term1 | ... | termn   CASE sym OF
                                  first(term0): Pr(term0)
                              |   first(term1): Pr(term1)
                              ...
                              |   first(termn): Pr(termn)
                              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 translations 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)**
term0 | term1	The terms must not have any common start symbols.
fac0 fac1	If fac0 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 sym 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;
```

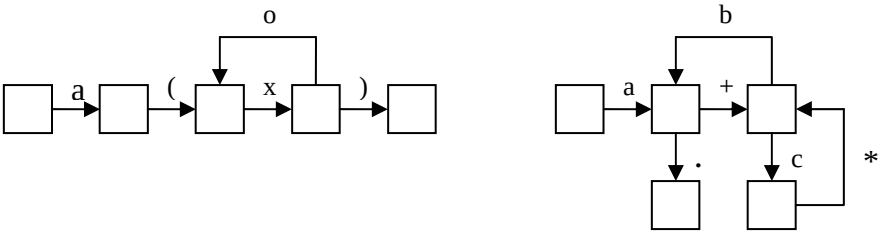
Note that the abstract reading operation is now represented by the concrete call *Texts.Read(R, ch). R* is a globally declared *Reader* specifying the source text. Also note that variable *ch* must be global, because at the end of *GetSym* it may contain the first character belonging to the next symbol.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

3.1. Exercise

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.



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

$term_0 \mid term_1$

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 record(id: Identifier; class: INTEGER);
BEGIN (*enter id in appropriate list of identifiers*)
END record;

PROCEDURE expression;

   PROCEDURE term;

      PROCEDURE factor;
      BEGIN
         IF sym = ident THEN record(id, 1); GetSym
         ELSIF sym = literal THEN record(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*) record(id, 2); 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 ")" .
```

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;
```

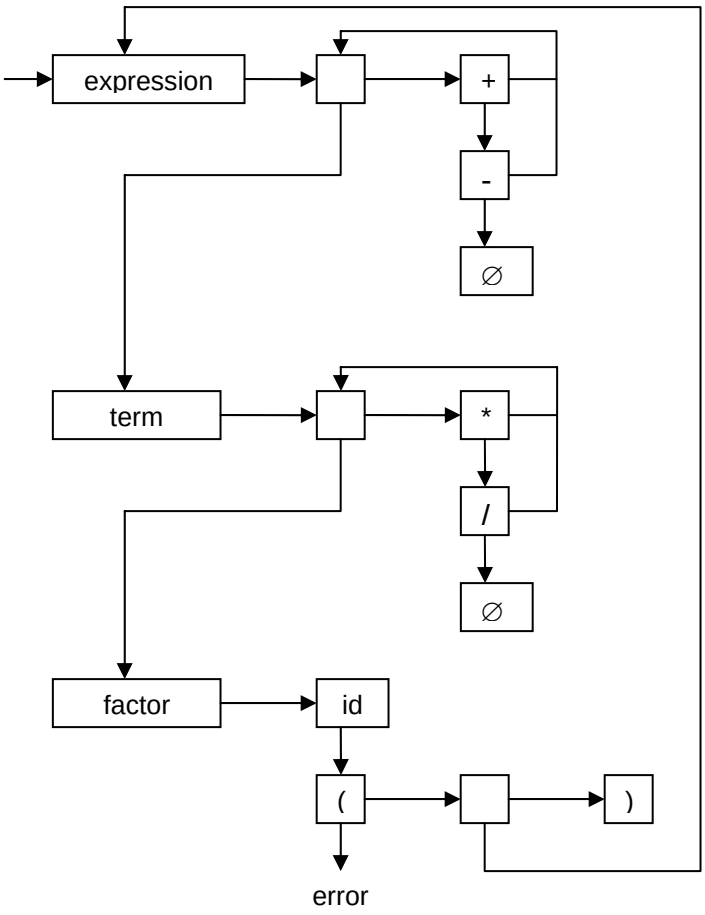


Figure 4.1. Syntax as data structure

The following remarks must be kept in mind:

1. We tacitly assume that terms always are of the form

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

   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 is it 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 read subsequence corresponds to some sentential construct, that is, the right part of a production. If this is the case, the read 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 further detail.

4. 4. 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 "]" .
```

Which 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 read syntax.

5. Attributed Grammars and Semantics

In attributed grammars certain attributes are associated with individual constructs, that is, with nonterminal symbols. The symbols are parameterized and represent whole classes of variants. This serves to simplify the syntax, but is, in practice, indispensible 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(T0) = term(T1)  |                   T0 := T1
          exp(T1) "+" term(T2)  |       T0 := T1         T1 = T2
          exp(T1) "-" term(T2).         T0 := T1         T1 = T2
```

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

T0 := if (T1 = INTEGER) & (T2 = INTEGER) then INTEGER else REAL,

T1 = INTEGER or T1 = REAL
T2 = INTEGER or T2 = 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(T0) = term(T1) {"+" term(T2)}.        exp(T0) = ["-"] term(T1).
```

are split into pairs of terms, namely

```
exp(T0) = term(T1)  |                     exp(T0) =  term(T1)  |
          exp(T1) "+" term(T2).                      "-" term(T1).
```

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(v0)	=	term(v1) |	v0 := v1
		exp(v1) "+" term(v2) |	v0 := v1 + v2
		exp(v1) "-" term(v2).	v0 := v1 - v2
term(v0)	=	factor(v1) |	v0 := v1
		term(v1) "*" factor(v2) |	v0 := v1 * v2
		term(v1) "/" factor(v2).	v0 := v1 / v2
factor(v0)	=	number(v1) |	v0 := v1
		"(" exp(v1) ")".	v0 := v1

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 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 utopia, but the idea goes back to the 1960s. It is represented schematically by Figure 5.1.

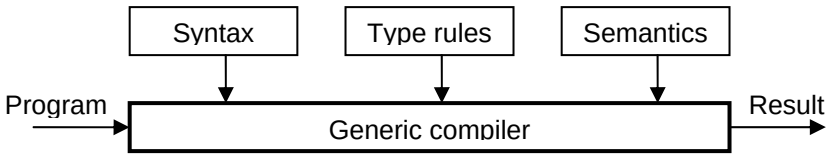


Figure 5.1. Schema of a general, parametrized compiler.

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 from the syntax automatically 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.

5.4. 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).

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 we 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-. while-, and repeat 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 parameterizing 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 is primarily 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".
RepeatStatement  =  “REPEAT” Statement Sequence “UNTIL” expression.
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. It also contains calls to specific, predefined procedures *OpenInput, ReadInt, WriteInt, WriteLn*, and *eot()* whose purpose is evident. Note that every command which asks for input, must start with a call to *OpenInput.*

```
MODULE Samples;

  PROCEDURE Multiply*;
      VAR x, y, z: INTEGER;
  BEGIN OpenInput; ReadInt(x); ReadInt(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 ;
      WriteInt(x, 4); WriteInt(y, 4); WriteInt(z, 6); WriteLn
  END Multiply;

  PROCEDURE Divide*;
      VAR x, y, r, q, w: INTEGER;
  BEGIN OpenInput; ReadInt(x); ReadInt(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 ;
      WriteInt(x.4); WriteInt(y, 4); WriteInt(q, 4); WriteInt(r, 4); WriteLn
  END Divide;

  PROCEDURE Sum*;
      VAR n, s: INTEGER;
  BEGIN OpenInput; s:= 0;
      WHILE ~eot() DO ReadInt(n); WriteInt(n, 4); s := s + n END ;
      WriteInt(s, 6); WriteLn
  END Sum;

END Samples.
```

Corresponding commands are:

```
Samples.Multiply 7 9
Samples.Divide 65 7
Samples.Sum 1 2 3 4 5~
```

6.1. Exercise

6.1. Determine the code for the computer defined in Chapter 9, generated from the program listed at the end of this 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 UNTIL ( [ ~ := ;
END ELSE ELSIF IF WHILE REPEAT
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 identifiers 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; leq = 12; gtr = 13; geq = 14;
      period = 18; int = 21; false = 23; true = 24;
      not = 27; lparen = 28; lbrak = 29;
      ident = 31; if = 32; while = 34;
      repeat = 35;
      comma = 40; colon = 41; becomes = 42; rparen = 44;
      rbrak = 45; then = 47; of = 48; do = 49;
      semicolon = 52; end = 53;
      else = 55; elsif = 56; until = 57;
      array = 60; record = 61; const = 63; type = 64;
      var = 65; procedure = 66; begin = 67;  module = 69;
      eof = 70;

   TYPE Ident = ARRAY IdLen OF CHAR;

   VAR val: INTEGER;
      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:

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 occur in a well-formed program text.

7.2. The parser

The construction of the parser strictly follows 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)	
selector	. [	*
factor	(~ integer ident	
term	(~ integer ident	
SimpleExpression	+ - (~ integer ident	
expression	+ - (~ integer ident	
assignment	ident	
ProcedureCall	ident	
statement	ident IF WHILE REPEAT	*
StatementSequence	ident IF WHILE REPEAT	*
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 & + - OR = # < <= > >= ,)] := OF THEN DO ; END ELSE ELSIF UNTIL
factor	* DIV MOD & + - OR = # < <= > >= ,)] OF THEN DO ; END ELSE ELSIF UNTIL
term	+ - OR = # < <= > >= ,)] OF THEN DO ; END ELSE ELSIF UNTIL
SimpleExpression	= # < <= > >= ,)] OF THEN DO ; END ELSE ELSIF UNTIL
expression	,)] OF THEN DO ; END ELSE ELSIF UNTIL
assignment	; END ELSE ELSIF UNTIL
ProcedureCall	; END ELSE ELSIF UNTIL
statement	; END ELSE ELSIF UNTIL
StatementSequence	END ELSE ELSIF UNTIL
FieldList	; END
type	) ;
FPSection	) ;
FormalParameters	;
ProcedureHeading	;

```
ProcedureBody          ;
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.

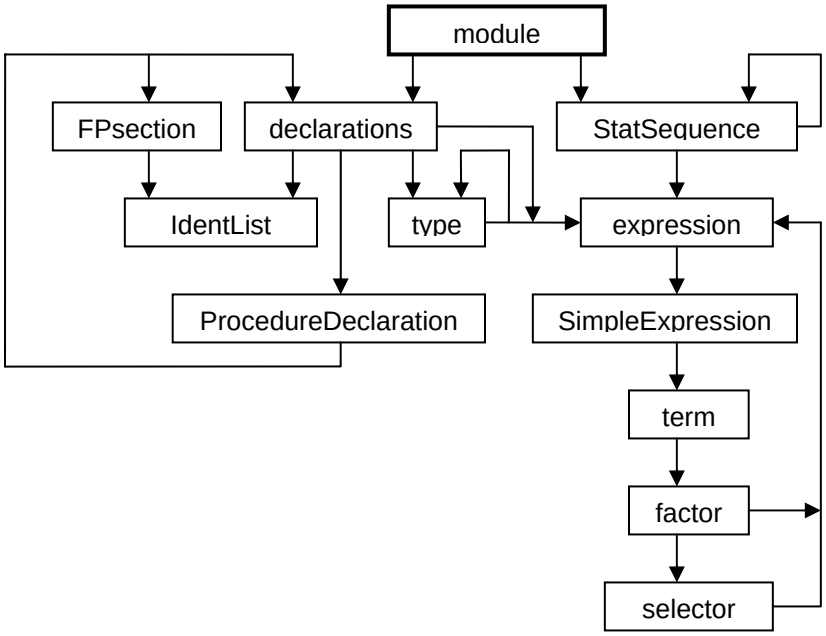


Figure 7.1. Dependence diagram of parsing procedures

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 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 inacceptable 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 must be detected in a single scan through the text.
2. As few additional assumptions as possible about the language are to 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 are being skipped. The process is resumed when either a correct start symbol or a strong symbol is read.

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

PROCEDURE StatSequence;
BEGIN (*sync*)
  IF ~((sym = OSS.ident) OR (sym >= OSS.if) & (sym <= OSS.repeat)
```

```
        OR (sym >= OSS.semicolon)) THEN Mark("Statement?");
      REPEAT Get(sym) UNTIL (sym = ident) OR (sym >= if)
   END ;
   ...
END StatSequence;

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

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

Evidently, a certain ordering among symbols is assumed at this point. This ordering had 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 read symbol 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

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

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 (here, 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 being replaced by

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

```
REPEAT
   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") END
UNTIL (sym # const) & (sym # type) & (sym # var)
```

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.

It should now have become clear that there is no such thing as a perfect strategy of error handling which would 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, albeit with possibilities for improvement. 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 language.

7.4. Exercises

7.1. The scanner 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.

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 times of assemblers, when identifiers were called symbols. Also, 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
```

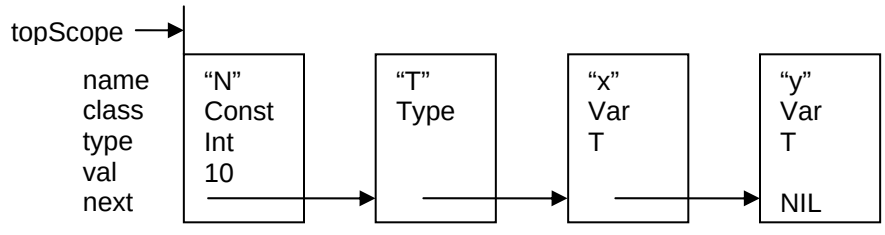


Figure 8.1. Symbol table representing objects with names and attributes.

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.

```
PROCEDURE NewObj(VAR obj: Object; class: INTEGER);
   VAR new, x: Object;
BEGIN x := topScope;
   WHILE (x.next # NIL) & (x.next.name # id) DO x := x.next END ;
   IF x.next = NIL THEN
      NEW(new); new.name := id; new.class := class; new.next := NIL;
      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.

A procedure *find* serves to access the object with name *id*. It represents a simple linear search, proceeding through the list of scopes, and in each scope through the list of objects.

```
PROCEDURE find(VAR obj: OSG.Object);
   VAR s, x: Object;
BEGIN s := topScope;
   REPEAT x := s.next;
      WHILE (x # NIL) & (x.name # id) DO x := x.next END ;
      s := s.dsc
   UNTIL (x # NIL) OR (s = NIL);
   IF x = NIL THEN x := dummy; OSS.Mark("undef") END ;
   obj := x
END find;
```

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;
```

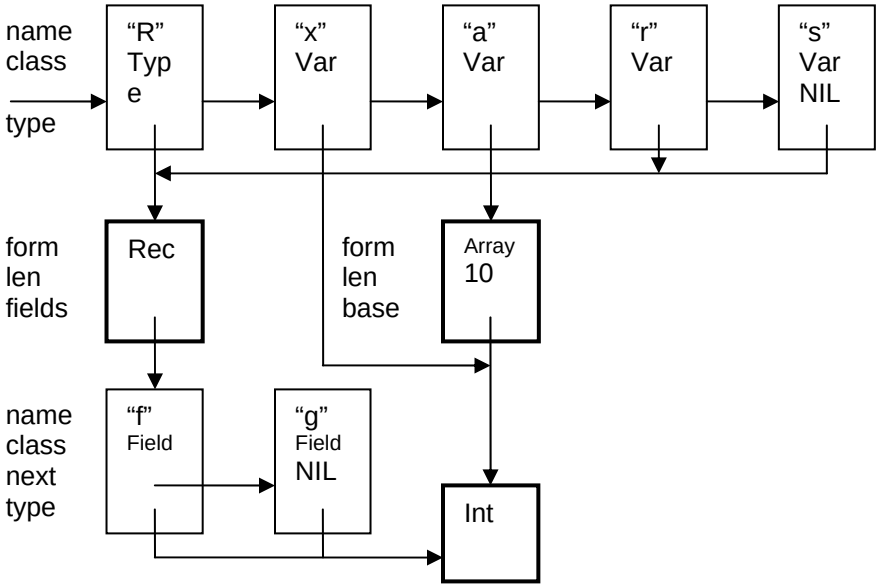


Figure 8.2. Symbol table representing declared objects.

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). In the present language Oberon-0, there exist only the two basic, scalar data types: INTEGER and BOOLEAN. In the

computer used here, the former occupies 4 bytes, the latter a single byte. However, in general every type has a *size*, and 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 T0
VAR a: T

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

size(T) = n * size(T0)
adr(a[x]) = adr(a) + x * size(T0)

For multi-dimensional arrays, the corresponding formulas (see Figure 8.3) are:

TYPE T = ARRAY n_{k-1}, ... , n_1, n_0 OF T0

size(T) = n_{k-1} * ... * n_1 * n_0 * size(T0)

adr(a[x_{k-1}, ... , x_1, x_0]) = adr(a)
+ x_{k-1} * n_{k-2} * ... * n_0 * size(T0) + ...
+ x_2 * n_1 * n_0 * size(T0) + x_1 * n_0 * size(T0) + x_0 * size(T0)
= adr(a) + (((... x_{k-1} * n_{k-2} + ... + x_2) * n_1 + x_1) * n_0 + x_0) * size(T0) (Horner schema)

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.

a: ARRAY 2 OF ARRAY 2 OF INTEGER

a[0]	0
a[0, 0]	0
a[0, 1]	4
a[1]	8
a[1, 0]	8
a[1, 1]	12

Figure 8.3. Representation of a matrix.

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 f0: T0; f1: T1; ... ; fk-1: Tk-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) + ... + size(T_{k-1})
adr(r.fi) = adr(r) + offset(f_i)
offset(fi) = size(T_0) + ... + 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), assuming a size of 2 for integers, 4 for real numbers. 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.

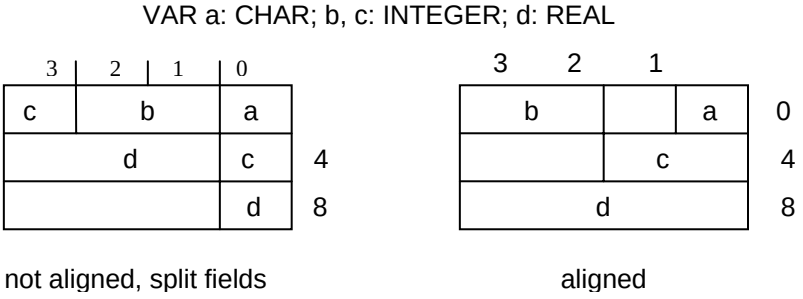


Figure 8.4. Alignment in address computation.

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 # NIL 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("no :") 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;
      REPEAT
        IF sym = ident THEN
          IdentList(Fld, first); Type1(tp); obj := first;
          WHILE obj # NIL 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("no ;")
        END
      UNTIL sym # ident;
      type.fields := topScope.next; CloseScope;
      IF sym = end THEN Get(sym) ELSE Mark("END?") END
    ELSE Mark("ident ?")
    END
  END Type1;
```

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*.

8.4. 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?