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.

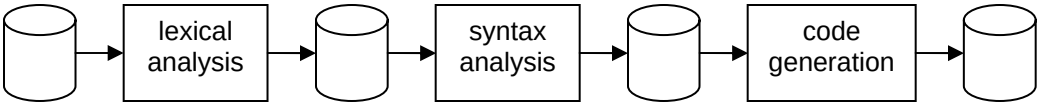


Figure 1.1. Multipass compilation.

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).

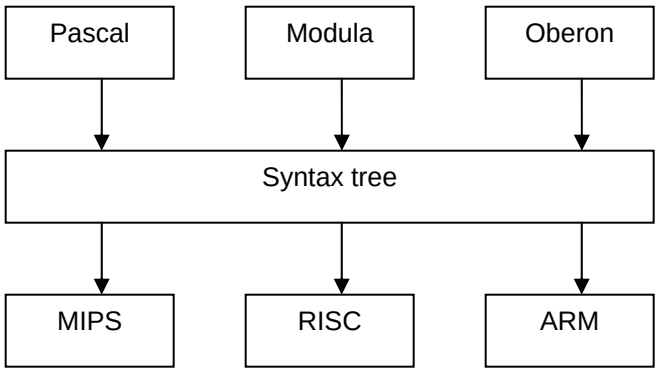


Figure 1.2. Front ends and back ends.

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*.

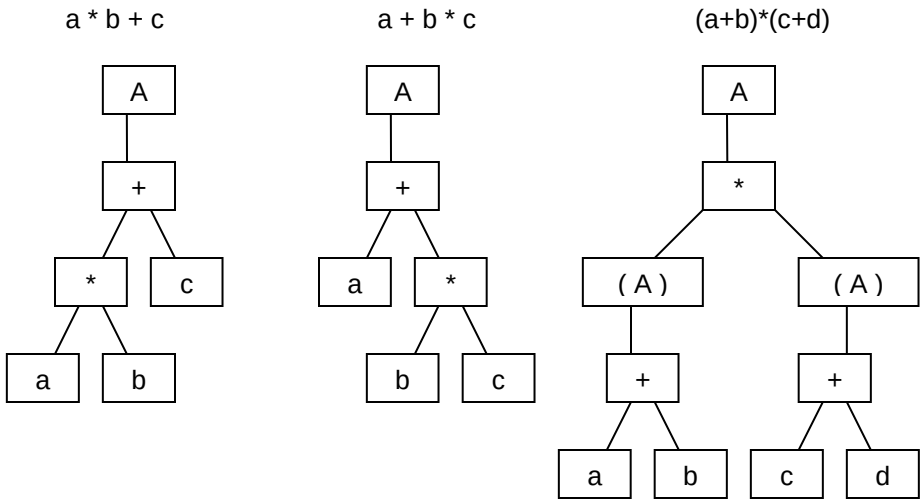


Figure 2.1. Structure of expressions

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.

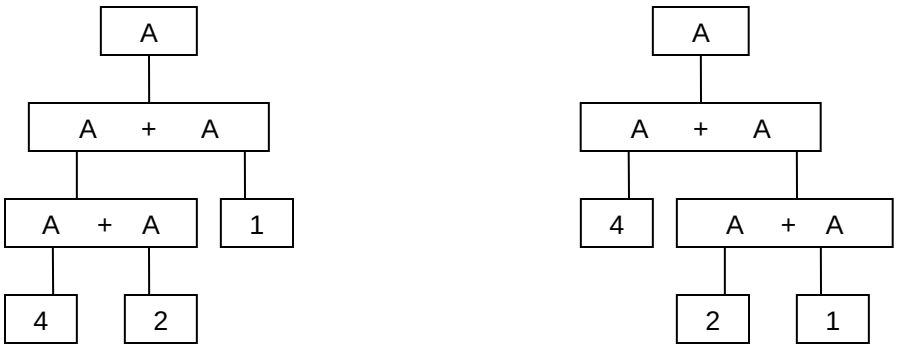


Figure 2.2. Differing structural trees for the same expression.

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;
```

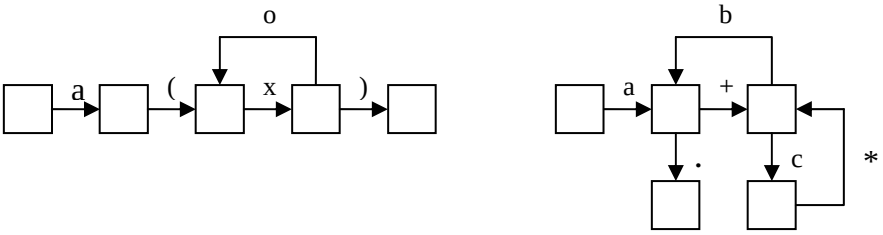
Note that the abstract reading operation is now represented by the concrete call *Texts.Read(R, ch). R* is a globally declared *Reader* specifying the source text. Also note that variable *ch* must be global, because at the end of *GetSym* it may contain the first character belonging to the next symbol.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

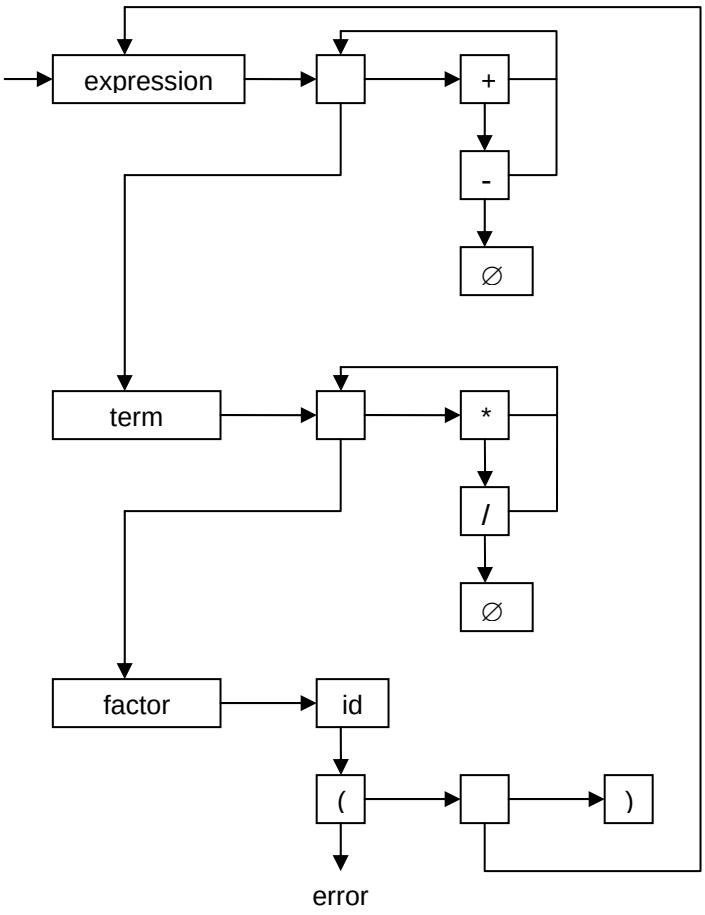


Figure 4.1. Syntax as data structure

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.