20. Compilers

When a programmer writes down a solution of her problems, she writes a program on a special programming language. These programming languages are very different from the proper languages of computers, from the ***machine languages***. Therefore we have to produce the executable forms of programs created by the programmer. We need a software or hardware tool, that translates the ***source language program*** – written on a high level programming language – to the ***target language program***, a lower level programming language, mostly to a machine code program.

There are two fundamental methods to execute a program written on higher level language. The first is using an ***interpreter***. In this case, the generated machine code is not saved but executed immediately. The interpreter is considered as a special computer, whose machine code is the high level language. Essentially, when we use an interpreter, then we create a two-level machine; its lower level is the real computer, on which the higher level, the interpreter, is built. The higher level is usually realized by a computer program, but, for some programming languages, there are special hardware interpreter machines.

The second method is using a ***compiler*** program. The difference of this method from the first is that here the result of translation is not executed, but it is saved in an intermediate file called ***target program***.

The target program may be executed later, and the result of the program is received only then. In this case, in contrast with interpreters, the times of translation and execution are distinguishable.

In the respect of translation, the two translational methods are identical, since the interpreter and the compiler both generate target programs. For this reason we speak about compilers only. We will deal the these translator programs, called compilers (Figure 20.1).

source language program	$\longrightarrow$	*translator*	$\longrightarrow$	*target language program*

Figure 20.1. The translator.

Our task is to study the algorithms of compilers. This chapter will care for the translators of high level imperative programming languages; the translational methods of logical or functional languages will not be investigated.

First the structure of compilers will be given. Then we will deal with scanners, that is, lexical analysers. In the topic of parsers – syntactic analysers –, the two most successful methods will be studied: the *LL*(1) and the *LALR*(1) parsing methods. The advanced methods of semantic analysis use *O-ATG* grammars, and the task of code generation is also written by this type of grammars. In this book these topics are not considered, nor we will study such important and interesting problems as symbol table handling, error repairing or code optimising. The reader can find very new, modern and efficient methods for these methods in the bibliography.

20.1. The structure of compilers

A compiler translates the source language program (in short, source program) into a target language program (in short, target program). Moreover, it creates a list by which the programmer can check her private program. This list contains the detected errors, too.

Using the notation *program (input)(output)* the compiler can be written by

compiler (source program)(target program, list).

In the next, the structure of compilers are studied, and the tasks of program elements are described, using the previous notation.

The first program of a compiler transforms the source language program into character stream that is easy to handle. This program is the ***source handler***.

source handler (source program)(character stream).

The form of the source program depends from the operating system. The source handler reads the file of source program using a system, called operating system, and omits the characters signed the end of lines, since these characters have no importance in the next steps of compilation. This modified, "poured" character stream will be the input data of the next steps.

The list created by the compiler has to contain the original source language program written by the programmer, instead of this modified character stream. Hence we define a ***list handler*** program,

list handler (source program, errors)(list),

which creates the list according to the file form of the operating system, and puts this list on a secondary memory.

It is practical to join the source handler and the list handler programs, since they have same input files. This program is the ***source handler***.

source handler (source program, errors)(character stream, list).

The target program is created by the compiler from the generated target code. It is located on a secondary memory, too, usually in a transferable binary form. Of course this form depends on the operating system. This task is done by the ***code handler*** program.

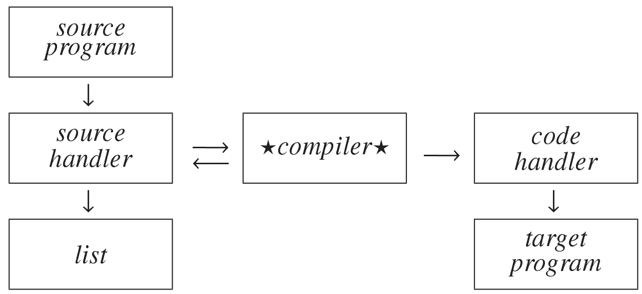

Figure 20.2. The structure of compilers.

code handler (target code)(target program).

Using the above programs, the structure of a compiler is the following (Figure 20.2):

source handler (source program, errors) (character string, list),
★*compiler*★ *(character stream)(target code, errors)*,
code handler (target code)(target program).

This decomposition is not a sequence: the three program elements are executed not sequentially. The decomposition consists of three independent working units. Their connections are indicated by their inputs and outputs.

In the next we do not deal with the handlers because of their dependentness on computers, operating system and peripherals – although the outer form, the connection with the user and the availability of the compiler are determined mainly by these programs.

The task of the program ★*compiler*★ is the translation. It consists of two main subtasks: analysing the input character stream, and to synthetizing the target code.

The first problem of the *analysis* is to determine the connected characters in the character stream. These are the symbolic items, e.g., the constants, names of variables, keywords, operators. This is done by the ***lexical analyser***, in short, ***scanner***. >From the character stream the scanner makes a ***series of symbols*** and during this task it detects ***lexical errors***.

scanner (character stream)(series of symbols, lexical errors).

This series of symbols is the input of the ***syntactic analyser***, in short, ***parser***. Its task is to check the syntactic structure of the program. This process is near to the checking the verb, the subject, predicates and attributes of a sentence by a language teacher in a language lesson. The errors detected during this analysis are the ***syntactic errors***. The result of the syntactic analysis is the syntax tree of the program, or some similar equivalent structure.

parser (series of symbols)(syntactically analysed program, syntactic errors).

The third program of the analysis is the ***semantic analyser***. Its task is to check the static semantics. For example, when the semantic analyser checks declarations and the types of variables in the expression `a + b`, it verifies whether the variables `a` and `b` are declared, do they are of the same type, do they have values? The errors detected by this program are the ***semantic errors***.

semantic analyser (syntactically analysed program)(analysed program, semantic errors).

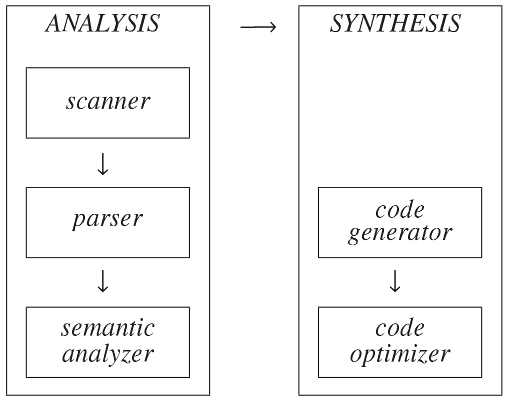

Figure 20.3. The programs of the analysis and the synthesis.

The output of the semantic analyser is the input of the programs of ***synthesis***. The first step of the synthesis is the code generation, that is made by the ***code generator***:

code generator (analysed program)(target code).

The target code usually depends on the computer and the operating system. It is usually an assembly language program or machine code. The next step of synthesis is the ***code optimisation***:

code optimiser (target code)(target code).

The code optimiser transforms the target code on such a way that the new code is better in many respect, for example running time or size.

As it follows from the considerations above, a compiler consists of the next components (the structure of the ⋆compiler⋆ program is in the Figure 20.3):

source handler (source program, errors)(character stream, list),
scanner (character stream)(series of symbols, lexical errors),
parser (series of symbols)(syntactically analysed program, syntactic errors),
semantic analyser (syntactically analysed program)(analysed program, semantic errors),
code generator (analysed program)(target code),
code optimiser (target code)(target code),
code handler(target code)(target program).

The algorithm of the part of the compiler, that performs analysis and synthesis, is the next:

⋆COMPILER⋆

determine the symbolic items in the text of source program
check the syntactic correctness of the series of symbols
check the semantic correctness of the series of symbols
generate the target code
optimise the target code

The objects written in the first two points will be analysed in the next sections.

Exercises

20.1-1 Using the above notations, give the structure of interpreters.
20.1-2 Take a programming language, and write program details in which there are lexical, syntactic and semantic errors.
20.1-3 Give respects in which the code optimiser can create better target code than the original.

20.2. Lexical analysis

The *source-handler* transforms the source program into a character stream. The main task of lexical analyser (scanner) is recognising the ***symbolic units*** in this character stream. These symbolic units are named ***symbols***.

Unfortunately, in different programming languages the same symbolic units consist of different character streams, and different symbolic units consist of the same character streams. For example, there is a programming language in which the `1.` and `.10` characters mean real numbers. If we concatenate these symbols, then the result is the `1..10` character stream. The fact, that a sign of an algebraic function is missing between the two numbers, will be detected by the next analyser, doing syntactic analysis. However, there are programming languages in which this character stream is decomposited into three components: 1 and 10 are the lower and upper limits of an interval type variable.

The lexical analyser determines not only the characters of a symbol, but the attributes derived from the surrounded text. Such attributes are, e.g., the type and value of a symbol.

The scanner assigns codes to the symbols, same codes to the same sort of symbols. For example the code of all integer numbers is the same; another unique code is assigned to variables.

The lexical analyser transforms the character stream into the series of symbol codes and the attributes of a symbols are written in this series, immediately after the code of the symbol concerned.

The output information of the lexical analyser is not „readable": it is usually a series of binary codes. We note that, in the viewpoint of the compiler, from this step of the compilation it is no matter from which characters were made the symbol, i.e. the code of the *if* symbol was made form English *if* or Hungarian *ha* or German *wenn* characters. Therefore, for a program language using English keywords, it is easy to construct another program language using keywords of another language. In the compiler of this new program language the lexical analysis would be modified only, the other parts of the compiler are unchanged.

20.2.1. The automaton of the scanner

The exact definition of symbolic units would be given by *regular grammar*, *regular expressions* or *deterministic finite automaton*. The theories of regular grammars, regular expressions and deterministic finite automata were studied in previous chapters.

Practically the lexical analyser may be a part of the syntactic analysis. The main reason to distinguish these analysers is that a lexical analyser made from regular grammar is much more simpler than a lexical analyser made from a context-free grammar. Context-free grammars are used to create syntactic analysers.

One of the most popular methods to create the lexical analyser is the following:

1. describe symbolic units in the language of regular expressions, and from this information construct the deterministic finite automaton which is equivalent to these regular expressions,
2. implement this deterministic finite automaton.

We note that, in writing of symbols regular expressions are used, because they are more comfortable and readable then regular grammars. There are standard programs as the `lex` of UNIX systems, that generate a complete syntactical analyser from regular expressions. Moreover, there are generator programs that give the automaton of scanner, too.

A very trivial implementation of the deterministic finite automaton uses multidirectional **case** instructions. The conditions of the branches are the characters of state transitions, and the instructions of a branch represent the new state the automaton reaches when it carries out the given state transition.

The main principle of the lexical analyser is building a symbol from the *longest series of symbols*. For example the string `ABC` is a three-letters symbol, rather than three one-letter symbols. This means that the alternative instructions of the **case** branch read characters as long as they are parts of a constructed symbol.

Functions can belong to the final states of the automaton. For example, the function converts constant symbols into an inner binary forms of constants, or the function writes identifiers to the symbol table.

The input stream of the lexical analyser contains tabulators and space characters, since the source-handler expunges the carriage return and line feed characters only. In most programming languages it is possible to write a lot of spaces or tabulators between symbols. In the point of view of compilers these symbols have no importance after their recognition, hence they have the name ***white spaces***.

Expunging white spaces is the task of the lexical analyser. The description of the white space is the following regular expression:

$$(space \mid tab)^* \ ,$$

where *space* and the *tab* tabulator are the characters which build the white space symbols and | is the symbol for the *or* function. No actions have to make with this white space symbols, the scanner does not pass these symbols to the syntactic analyser.

Some examples for regular expression:

Example 20.1 Introduce the following notations: Let D be an arbitrary digit, and let L be an arbitrary letter,

$$D \in \{0, 1, \ldots, 9\}, \text{ and } L \in \{a, b, \ldots, z, A, B, \ldots, Z\} \ ,$$

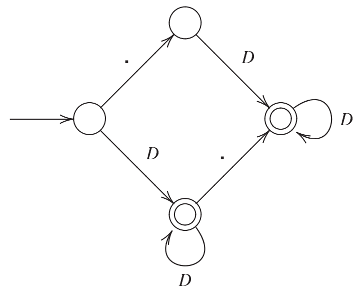

Figure 20.4. The positive integer and real number.

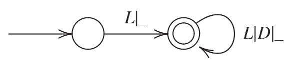

Figure 20.5. The identifier.

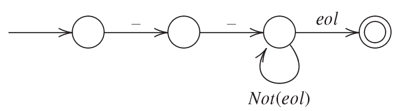

Figure 20.6. A comment.

the not-visible characters are denoted by their short names, and let ε be the name of the empty character stream. $Not(a)$ denotes a character distinct from a. The regular expressions are:

1. real number: $(+ \mid - \mid \varepsilon)D^+.D^+(\mathrm{e}(+ \mid - \mid \varepsilon)D^+ \mid \varepsilon)$,
2. positive integer and real number: $(D^+(\varepsilon \mid .)) \mid (D^*.D^+)$,
3. identifier: $(L \mid \_)(L \mid D \mid \_)^*$,
4. comment: $\text{-\,-}(Not(eol))^* eol$,
5. comment terminated by ## : $\#\#((\# \mid \varepsilon)Not(\#))^*\#\#$,
6. string of characters: $"(Not(") \mid " ")^* "$.

Deterministic finite automata constructed from regular expressions 2 and 3 are in Figures 20.4 and 20.5.

The task of lexical analyser is to determine the text of symbols, but not all the characters of a regular expression belong to the symbol. As is in the 6th example, the first and the last " characters do not belong to the symbol. To unravel this problem, a buffer is created for the scanner. After recognising of a symbol, the characters of these symbols will be in the buffer. Now the deterministic finite automaton is supplemented by a T transfer function, where $T(a)$ means that the character a is inserted into the buffer.

Example 20.2 The 4th and 6th regular expressions of the example 20.1. are supplemented by the T function, automata for these expressions are in Figures 20.6 and 20.7. The automaton of the 4th regular expression has none T function, since it recognises comments. The automaton of the 6th regular expression recognises `This is a "string"` from the character string `"This is a ""string"""`.

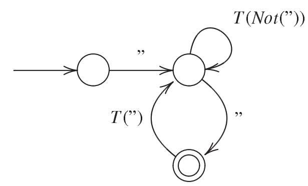

Figure 20.7. The character string.

Now we write the algorithm of the lexical analyser given by deterministic finite automaton. (The state of the set of one element will be denoted by the only element of the set).

Let $A = (Q, \Sigma, \delta, q_0, F)$ be the deterministic finite automaton, which is the scanner. We augment the alphabet Σ with a new notion: let *others* be all the characters not in Σ. Accordingly, we modify the transition function δ:

$$\delta'(q,a) = \begin{cases} \delta(q,a), & \text{if } a \neq \textit{others} \ , \\ \emptyset, & \text{otherwise} \ . \end{cases}$$

The algorithm of parsing, using the augmented automaton A', follows:

Lex-analyse($x\#, A'$)

```
 1  q ← q0, a ← first character of x
 2  s' ← analyzing
 3  while a ≠ # and s' = analyzing
 4        do if δ'(q, a) ≠ ∅
 5              then q ← δ'(q, a)
 6                   a ← next character of x
 7              else  s' ← error
 8  if s' = analyzing and q ∈ F
 9    then s' ← O.K.
10    else  s' ← ERROR
11  return s', a
```

The algorithm has two parameters: the first one is the input character string terminated by #, the second one is the automaton of the scanner. In the line 1 the state of the scanner is set to q_0, to the start state of the automaton, and the first character of the input string is determined. The variable s' indicates that the algorithm is analysing the input string, the text *analysing* is set in this variable in the line 2. In the line 5 a state-transition is executed. It can be seen that the above augmentation is needed to terminate in case of unexpected, invalid character. In line 8–10 the *O.K.* means that the analysed character string is correct, and the *ERROR* signs that a lexical error was detected. In the case of successful termination the variable a contains the # character, at erroneous termination it contains the invalid character.

We note that the algorithm Lex-Analyse recognise one symbol only, and then it is terminated. The program written in a programming language consists of a lot of symbols, hence after recognising a symbol, the algorithm have to be continued by detecting the next symbol. The work of the analyser is restarted at the state of the automaton. We propose the full algorithm of the lexical analyser as an exercise (see Exercise *20-1.*).

Example 20.3 The automaton of the identifier in the point 3 of example 20.1. is in Figure 20.5. The start state is 0, and the final state is 1. The transition function of the automaton follows:

δ	L	_	D
0	1	1	$\emptyset$
1	1	1	1

The augmented transition function of the automaton:

δ'	L	_	D	*others*
0	1	1	$\emptyset$	$\emptyset$
1	1	1	1	$\emptyset$

The algorithm Lex-Analyse gives the series of states 0111111 and sign *O.K.* to the input string `abc123#`, it gives sign *ERROR* to the input sting `9abc#`, and the series 0111 and sign *ERROR* to the input string `abc`χ`123`.

20.2.2. Special problems

In this subsection we investigate the problems emerged during running of lexical analyser, and supply solutions for these problems.

Keywords, standard words

All of programming languages allows identifiers having special names and predefined meanings. They are the ***keywords***. Keywords are used only in their original notions. However there are identifiers which also have predefined meaning but they are alterable in the programs. These words are called ***standard words***.

The number of keywords and standard words in programming languages are vary. For example, there is a program language, in which three keywords are used for the zero value: `zero`, `zeros` és `zeroes.`

Now we investigate how does the lexical analyser recognise keywords and standard words, and how does it distinguish them from identifiers created by the programmers.

The usage of a standard word distinctly from its original meaning renders extra difficulty, not only to the compilation process but also to the readability of the program, such as in the next example:

```
if if then else = then;
```

or if we declare procedures which have names `begin` and `end`:

```
begin
  begin; begin end; end; begin end;
end;
```

Recognition of keywords and standard words is a simple task if they are written using special type characters (for example bold characters), or they are between special prefix and postfix characters (for example between apostrophes).

We give two methods to analyse keywords.

1. All keywords is written as a regular expression, and the implementation of the automaton created to this expression is prepared. The disadvantage of this method is the size of the analyser program. It will be large even if the description of keywords, whose first letter are the same, are contracted.
2. Keywords are stored in a special keyword-table. The words can be determined in the character stream by a general identifier- recogniser. Then, by a simple search algorithm, we check whether this word is in the keyword- table. If this word is in the table then it is a keyword. Otherwise it is an identifier defined by the user. This method is very simple, but the efficiency of search depends on the structure of keyword-table and on the algorithm of search. A well-selected mapping function and an adequate keyword-table should be very effective.

If it is possible to write standard words in the programming language, then the lexical analyser recognises the standard words using one of the above methods. But the meaning of this standard word depends of its context. To decide, whether it has its original meaning or it was overdefined by the programmer, is the task of syntactic analyser.

Look ahead

Since the lexical analyser creates a symbol from the longest character stream, the lexical analyser has to look ahead one or more characters for the allocation of the right-end of a symbol There is a classical example for this problem, the next two FORTRAN statements:

```
DO 10 I = 1.1000
DO 10 I = 1,1000
```

In the FORTRAN programming language space-characters are not important characters, they do not play an important part, hence the character between `1` and `1000` decides that the statement is a `DO` cycle statement or it is an assignment statement for the `DO10I` identifier.

To sign the right end of the symbol, we introduce the symbol `/` into the description of regular expressions. Its name is ***lookahead operator.*** Using this symbol the description of the above `DO` keyword is the next:

$$\texttt{DO} \ / \ (letter \mid digit)^* = (letter \mid digit)^* \texttt{,}$$

This definition means that the lexical analyser says that the first two `D` and `O` letters are the `DO` keyword, if looking ahead, after the `O` letter, there are letters or digits, then there is an equal sign, and after this sign there are letters or digits again, and finally, there is a „ `,` ” character. The lookahead operator implies that the lexical analyser has to look ahead after the `DO` characters. We remark that using this lookahead method the lexical analyser recognises the `DO` keyword even if there is an error in the character stream, such as in the `DO2A=3B,` character stream, but in a correct assignment statement it does not detect the `DO` keyword.

In the next example we concern for positive integers. The definition of integer numbers is a prefix of the definition of the real numbers, and the definition of real numbers is a prefix of the definition of real numbers containing explicit power-part.

pozitív egész :	D^+
pozitív valós :	$D^+.D^+$
	és $D^+.D^+\mathtt{e}(+ \mid - \mid \varepsilon)D^+$

The automaton for all of these three expressions is the automaton of the longest character stream, the real number containing explicit power-part.

The problem of the lookahead symbols is resolved using the following algorithm. Put the character into a buffer, and put an auxiliary information aside this character. This information is „it is invalid". if the character string, using this red character, is not correct; otherwise we put the type of the symbol into here. If the automaton is in a final-state, then the automaton recognises a real number with explicit power-part. If the automaton is in an internal state, and there is no possibility to read a next character, then the longest character stream which has valid information is the recognised symbol.

Example 20.4 Consider the `12.3e+f#` character stream, where the character `#` is the endsign of the analysed text. If in this character stream there was a positive integer number in the place of character `f`, then this character stream should be a real number. The content of the puffer of lexical analyser:

`1`	*integer number*
`12`	*integer number*
`12.`	*invalid*
`12.3`	*real number*
`12.3e`	*invalid*
`12.3e+`	*invalid*
`12.3e+f`	*invalid*
`12.3e+f#`	

The recognised symbol is the `12.3` *real number*. The lexical analysing is continued at the text `e+f`.

The number of lookahead-characters may be determined from the definition of the program language. In the modern languages this number is at most two.

The symbol table

There are programming languages, for example C, in which small letters and capital letters are different. In this case the lexical analyser uses characters of all symbols without modification. Otherwise the lexical analyser converts all characters to their small letter form or all characters to capital letter form. It is proposed to execute this transformation in the source handler program.

At the case of simpler programming languages the lexical analyser writes the characters of the detected symbol into the symbol table, if this symbol is not there. After writing up, or if this symbol has been in the symbol table already, the lexical analyser returns the table address of this symbol, and writes this information into its output. These data will be important at semantic analysis and code generation.

Directives

In programming languages the ***directives*** serve to control the compiler. The lexical analyser identifies directives and recognises their operands, and usually there are further tasks with these directives.

If the directive is the `if` of the conditional compilation, then the lexical analyser has to detect all of parameters of this condition, and it has to evaluate the value of the branch. If this value is `false`, then it has to omit the next lines until the `else` or `endif` directive. It means that the lexical analyser performs syntactic and semantic checking, and creates code-style information. This task is more complicate if the programming language gives possibility to write nested conditions.

Other types of directives are the substitution of macros and including files into the source text. These tasks are far away from the original task of the lexical analyser.

The usual way to solve these problems is the following. The compiler executes a pre-processing program, and this program performs all of the tasks written by directives.

Exercises

20.2-1 Give a regular expression to the comments of a programming language. In this language the delimiters of comments are $/*$ and $*/$, and inside of a comment may occurs $/$ and $*$ characters, but $*/$ is forbidden.

20.2-2 Modify the result of the previous question if it is supposed that the programming language has possibility to write nested comments.

20.2-3 Give a regular expression for positive integer numbers, if the pre- and post-zero characters are prohibited. Give a deterministic finite automaton for this regular expression.

20.2-4 Write a program, which re-creates the original source program from the output of lexical analyser. Pay attention for nice an correct positions of the re-created character streams.

20.3. Syntactic analysis

The perfect definition of a programming language includes the definition of its *syntax* and *semantics*.

The syntax of the programming languages cannot be written by context free grammars. It is possible by using context dependent grammars, two-level grammars or attribute grammars. For these grammars there are not efficient parsing methods, hence the description of a language consists of two parts. The main part of the syntax is given using context free grammars, and for the remaining part a context dependent or an attribute grammar is applied. For example, the description of the program structure or the description of the statement structure belongs to the first part, and the type checking, the scope of variables or the correspondence of formal and actual parameters belong to the second part.

The checking of properties written by context free grammars is called *syntactic analysis* or *parsing*. Properties that cannot be written by context free grammars are called form the *static semantics*. These properties are checked by the *semantic analyser*.

The conventional semantics has the name *run-time semantics* or *dynamic semantics*. The dynamic semantics can be given by verbal methods or some interpreter methods, where the operation of the program is given by the series of state-alterations of the interpreter and its environment.

We deal with *context free grammars*, and in this section we will use *extended grammars* for the syntactic analysis. We investigate on methods of checking of properties which are written by context free grammars. First we give basic notions of the syntactic analysis, then the parsing algorithms will be studied.

Definition 20.1 *Let $G = (N, T, P, S)$ be a grammar. If $S \stackrel{*}{\Longrightarrow} \alpha$ and $\alpha \in (N \cup T)^*$ then α is a **sentential form**. If $S \stackrel{*}{\Longrightarrow} x$ and $x \in T^*$ then x is a **sentence** of the language defined by the grammar.*

The sentence has an important role in parsing. The program written by a programmer is a series of terminal symbols, and this series is a sentence if it is correct, that is, it has not syntactic errors.

Definition 20.2 *Let $G = (N, T, P, S)$ be a grammar and $\alpha = \alpha_1\beta\alpha_2$ is a sentential form ($\alpha, \alpha_1, \alpha_2, \beta \in (N \cup T)^*$). We say that β is a **phrase** of α, if there is a symbol $A \in N$, which $S \stackrel{*}{\Longrightarrow} \alpha_1 A \alpha_2$ and $A \stackrel{*}{\Longrightarrow} \beta$. We say that α is a **simple phrase** of β, if $A \to \beta \in P$.*

We note that every sentence is phrase. The leftmost simple phrase has an important role in parsing; it has its own name.

Definition 20.3 *The leftmost simple phase of a sentence is the **handle**.*

The leaves of the *syntax tree* of a sentence are terminal symbols, other points of the tree are nonterminal symbols, and the root symbol of the tree is the start symbol of the grammar.

In an ambiguous grammar there is at least one sentence, which has several syntax trees. It means that this sentence has more than one analysis, and therefore there are several target programs for this sentence. This ambiguity raises a lot of problems, therefore the compilers translate languages generated by unambiguous grammars only.

We suppose that the grammar G has properties as follows:

1. the grammar is *cycle free,* that is, it has not series of derivations rules $A \stackrel{+}{\Longrightarrow} A$ $(A \in N)$,
2. the grammar is *reduced*, that is, there are not „unused symbols" in the grammar, all of nonterminals happen in a derivation, and from all nonterminals we can derive a part of a sentence. This last property means that for all $A \in N$ it is true that $S \stackrel{*}{\Longrightarrow} \alpha A \beta \stackrel{*}{\Longrightarrow} \alpha y \beta \stackrel{*}{\Longrightarrow} xyz$, where $A \stackrel{*}{\Longrightarrow} y$ and $|y| > 0$ $(\alpha, \beta \in (N \cup T)^*, \ x, y, z \in T^*)$.

As it has shown, the lexical analyser translates the program written by a programmer into series of terminal symbols, and this series is the input of syntactic analyser. The task of syntactic analyser is to decide if this series is a sentence of the grammar or it is not. To achieve this goal, the parser creates the syntax tree of the series of symbols. From the known start symbol and the leaves of the syntax tree the parser creates all vertices and edges of the tree, that is, it creates a derivation of the program.

If this is possible, then we say that the program is an element of the language. It means that the program is syntactically correct.

Hence forward we will deal with *left to right* parsing methods. These methods read the symbols of the programs left to right. All of the real compilers use this method.

To create the inner part of the syntax tree there are several methods. One of these methods builds the syntax tree from its start symbol S. This method is called *top-down* method. If the parser goes from the leaves to the symbol S, then it uses the *bottom-up* parsing method.

We deal with top-down parsing methods in Subsection 20.3.1. We investigate bottom-up parsers in Subsection 20.3.2; now these methods are used in real compilers.

20.3.1. $LL(1)$ parser

If we analyse from top to down then we start with the start symbol. This symbol is the root of syntax tree; we attempt to construct the syntax tree. Our goal is that the leaves of tree are the terminal symbols.

First we review the notions that are necessary in the top-down parsing. Then the $LL(1)$ table methods and the recursive descent method will be analysed.

$LL(k)$ grammars

Our methods build the syntax tree top-down and read symbols of the program left to right. For this end we try to create terminals on the left side of sentential forms.

Definition 20.4 *If $A \to \alpha \in P$ then the* ***leftmost direct derivation*** *of the sentential form $xA\beta$ ($x \in T^*,\ \alpha,\beta \in (N \cup T)^*$) is $x\alpha\beta$, and*

$$xA\beta \underset{leftmost}{\Longrightarrow} x\alpha\beta \ .$$

Definition 20.5 *If all of direct derivations in $S \overset{*}{\Longrightarrow} x$ ($x \in T^*$) are leftmost, then this derivation is said to be* ***leftmost derivation****, and*

$$S \underset{leftmost}{\overset{*}{\Longrightarrow}} x \ .$$

In a leftmost derivation terminal symbols appear at the left side of the sentential forms. Therefore we use leftmost derivations in all of top-down parsing methods. Hence if we deal with top-down methods, we do not write the text "leftmost" at the arrows.

One might as well say that we create all possible syntax trees. Reading leaves from left to right, we take sentences of the language. Then we compare these sentences with the parseable text and if a sentence is same as the parseable text, then we can read the steps of parsing from the syntax tree which is belongs to this sentence. But this method is not practical; generally it is even impossible to apply.

A good idea is the following. We start at the start symbol of the grammar, and using leftmost derivations we try to create the text of the program. If we use a not suitable derivation at one of steps of parsing, then we find that, at the next step, we can not apply a proper derivation. At this case such terminal symbols are at the left side of the sentential form, that are not same as in our parseable text.

For the leftmost terminal symbols we state the theorem as follows.

Theorem 20.6 *If $S \overset{*}{\Longrightarrow} x\alpha \overset{*}{\Longrightarrow} yz$ ($\alpha \in (N \cup T)^*,\ x,y,z \in T^*$) és $|x| = |y|$, then $x = y$.*

The proof of this theorem is trivial. It is not possible to change the leftmost terminal symbols x of sentential forms using derivation rules of a context free grammar.

This theorem is used during the building of syntax tree, to check that the leftmost terminals of the tree are same as the leftmost symbols of the parseable text. If they are different then we created wrong directions with this syntax tree. At this case we have to make a backtrack, and we have to apply an other derivation rule. If it is impossible (since for example there are no more derivation rules) then we have to apply a backtrack once again.

General top-down methods are realized by using backtrack algorithms, but these backtrack steps make the parser very slow. Therefore we will deal only with grammars such that

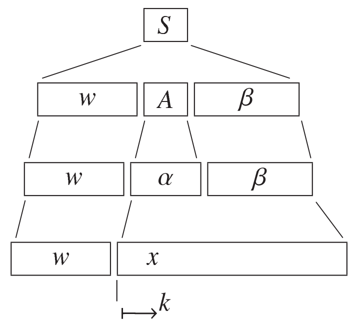

Figure 20.8. $LL(k)$ grammar.

have parsing methods without backtracks.

The main properties of $LL(k)$ grammars are the following. If, by creating the leftmost derivation $S \overset{*}{\Longrightarrow} wx$ $(w, x \in T^*)$, we obtain the sentential form $S \overset{*}{\Longrightarrow} wA\beta$ $(A \in N,\ \beta \in (N \cup T)^*)$ at some step of this derivation, and our goal is to achieve $A\beta \overset{*}{\Longrightarrow} x$, then the next step of the derivation for nonterminal A is determinable unambiguously from the first k symbols of x.

To look ahead k symbols we define the function $First_k$.

Definition 20.7 *Let* $\boldsymbol{First_k(\alpha)}$ $(k \ge 0,\ \alpha \in (N \cup T)^*)$ *be the set as follows.*

$$First_k(\alpha) = \{x \mid \alpha \overset{*}{\Longrightarrow} x\beta \text{ and } |x| = k\} \cup \{x \mid \alpha \overset{*}{\Longrightarrow} x \text{ and } |x| < k\} \ \ (x \in T^*,\ \beta \in (N \cup T)^*)\ .$$

The set $First_k(x)$ consists of the first k symbols of x; for $|x| < k$, it consists the full x. If $\alpha \overset{*}{\Longrightarrow} \varepsilon$, then $\varepsilon \in First_k(\alpha)$.

Definition 20.8 *The grammar G is a* ***LL(k) grammar*** $(k \ge 0)$*, if for derivations*

$$S \overset{*}{\Longrightarrow} wA\beta \Longrightarrow w\alpha_1\beta \overset{*}{\Longrightarrow} wx\ ,$$
$$S \overset{*}{\Longrightarrow} wA\beta \Longrightarrow w\alpha_2\beta \overset{*}{\Longrightarrow} wy$$

$(A \in N,\ x, y, w \in T^*,\ \alpha_1, \alpha_2, \beta \in (N \cup T)^*)$ *the equality*

$$First_k(x) = First_k(y)$$

implies

$$\alpha_1 = \alpha_2\ .$$

Using this definition, if a grammar is a $LL(k)$ grammar then the k symbol after the parsed x determine the next derivation rule unambiguously (Figure 20.8).

One can see from this definition that if a grammar is an $LL(k_0)$ grammar then for all $k > k_0$ it is also an $LL(k)$ grammar. If we speak about $LL(k)$ grammar then we also mean that k is the least number such that the properties of the definition are true.

Example 20.5 The next grammar is a $LL(1)$ grammar. Let $G = (\{A, S\}, \{a, b\}, P, S)$ be a grammar whose derivation rules are:

$S \to AS \mid \varepsilon$
$A \to aA \mid b$

We have to use the derivation $S \to AS$ for the start symbol S if the next symbol of the parseable text is a or b. We use the derivation $S \to \varepsilon$ if the next symbol is the mark #.

Example 20.6 The next grammar is a $LL(2)$ grammar. Let $G = (\{A, S\}, \{a, b\}, P, S)$ be a grammar whose the derivation rules are:

$S \to abA \mid \varepsilon$
$A \to Saa \mid b$

One can see that at the last step of derivations

$$S \Longrightarrow abA \Longrightarrow abSaa \overset{S \to abA}{\Longrightarrow} ababAaa$$

and

$$S \Longrightarrow abA \Longrightarrow abSaa \overset{S \to \varepsilon}{\Longrightarrow} abaa$$

if we look ahead one symbol, then in both derivations we obtain the symbol a. The proper rule for symbol S is determined to look ahead two symbols (ab or aa).

There are context free grammars such that are not $LL(k)$ grammars. For example the next grammar is not $LL(k)$ grammar for any k.

Example 20.7 Let $G = (\{A, B, S\}, \{a, b, c\}, P, S)$ be a grammar whose the derivation rules are:

$S \to A \mid B$
$A \to aAb \mid ab$
$B \to aBc \mid ac$

$L(G)$ consists of sentences $a^i b^i$ és $a^i c^i$ ($i \geq 1$). If we analyse the sentence $a^{k+1}b^{k+1}$, then at the first step we can not decide by looking ahead k symbols whether we have to use the derivation $S \to A$ or $S \to B$, since for all k $First_k(a^k b^k) = First_k(a^k c^k) = a^k$.

By the definition of the $LL(k)$ grammar, if we get the sentential form $wA\beta$ using leftmost derivations, then the next k symbol determines the next rule for symbol A. This is stated in the next theorem.

Theorem 20.9 *Grammar G is a $LL(k)$ grammar iff*

$$S \overset{*}{\Longrightarrow} wA\beta, \text{ és } A \to \gamma \mid \delta \;\; (\gamma \neq \delta,\; w \in T^*,\; A \in N,\; \beta, \gamma, \delta \in (N \cup T)^*)$$

implies

$$First_k(\gamma\beta) \cap First_k(\delta\beta) = \emptyset \;.$$

If there is a $A \to \varepsilon$ rule in the grammar, then the set $First_k$ consists the k length prefixes of terminal series generated from β. It implies that, for deciding the property $LL(k)$, we have to check not only the derivation rules, but also the infinite derivations.

We can give good methods, that are used in the practice, for $LL(1)$ grammars only. We define the follower-series, which follow a symbol or series of symbols.

Definition 20.10 ***$Follow_k(\beta)$*** $= \{x \mid S \overset{*}{\Longrightarrow} \alpha\beta\gamma$ *and* $x \in First_k(\gamma)\}$, *and if* $\varepsilon \in Follow_k(\beta)$, *then* $Follow_k(\beta) = Follow_k(\beta) \setminus \{\varepsilon\} \cup \{\#\} \;\; (\alpha, \beta, \gamma \in (N \cup T)^*,\; x \in T^*)$.

The second part of the definition is necessary because if there are no symbols after the β in the derivation $\alpha\beta\gamma$, that is $\gamma = \varepsilon$, then the next symbol after β is the mark # only.

$Follow_1(A)$ $(A \in N)$ consists of terminal symbols that can be immediately after the symbol A in the derivation

$$S \stackrel{*}{\Longrightarrow} \alpha A\gamma \stackrel{*}{\Longrightarrow} \alpha A w \ (\alpha, \gamma \in (N \cup T)^*,\ w \in T^*).$$

Theorem 20.11 *The grammar G is a $LL(1)$ grammar iff, for all nonterminal A and for all derivation rules $A \to \gamma \mid \delta$,*

$$First_1(\gamma Follow_1(A)) \cap First_1(\delta Follow_1(A)) = \emptyset\ .$$

In this theorem the expression $First_1(\gamma Follow_1(A))$ means that we have to concatenate to γ the elements of set $Follow_1(A)$ separately, and for all elements of this new set we have to apply the function $First_1$.

It is evident that theorem 20.11 is suitable to decide whether a grammar is $LL(1)$ or it is not.

Hence forward we deal with $LL(1)$ languages determined by $LL(1)$ grammars, and we investigate the parsing methods of $LL(1)$ languages. For the sake of simplicity, we omit indexes from the names of functions $First_1$ és $Follow_1$.

The elements of the set $First(\alpha)$ are determined using the next algorithm.

First(α)

```
if α = ε
  then F ← {ε}
if α = a, where a ∈ T
  then F ← {a}
if α = A, where A ∈ N
  then if A → ε ∈ P
         then F ← {ε}
         else F ← ∅
       for all A → Y1Y2...Ym ∈ P (m ≥ 1)
          do F ← F ∪ (First(Y1) \ {ε})
             for k ← 1 to m − 1
                do if Y1Y2...Yk =*=> ε
                      then F ← F ∪ (First(Yk+1) \ {ε})
             if Y1Y2...Ym =*=> ε
               then F ← F ∪ {ε}
if α = Y1Y2...Ym (m ≥ 2)
  then F ← (First(Y1) \ {ε})
       for k ← 1 to m − 1
          do if Y1Y2...Yk =*=> ε
               then F ← F ∪ (First(Yk+1) \ {ε})
       if Y1Y2...Ym =*=> ε
         then F ← F ∪ {ε}
return F
```

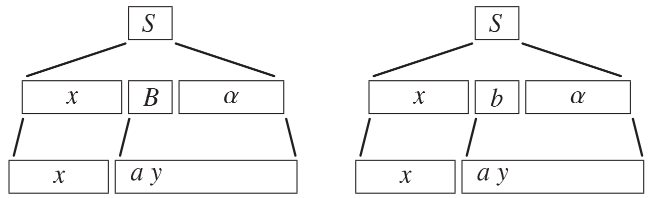

Figure 20.9. The sentential form and the analysed text.

In lines 1–4 the set is given for ε and a terminal symbol a. In lines 5–15 we construct the elements of this set for a nonterminal A. If ε is derivated from A then we put symbol ε into the set in lines 6–7 and 14–15. If the argument is a symbol stream then the elements of the set are constructed in lines 16–22. We notice that we can terminate the **for** cycle in lines 11 and 18 if $Y_k \in T$, since in this case it is not possible to derive symbol ε from $Y_1 Y_2 \ldots Y_k$.

In theorem 20.11 and hereafter, it is necessary to know the elements of the set *Follow*(A). The next algorithm constructs this set.

Follow(A)

1 **if** $A = S$
2   **then** $F \leftarrow \{\#\}$
3   **else** $F \leftarrow \emptyset$
4 **for** all rules $B \rightarrow \alpha A \beta \in P$
5   **do if** $|\beta| > 0$
6   **then** $F \leftarrow F \cup (\text{First}(\beta) \setminus \{\varepsilon\})$
7   **if** $\beta \stackrel{*}{\Longrightarrow} \varepsilon$
8   **then** $F \leftarrow F \cup \text{Follow}(B)$
9   **else** $F \leftarrow F \cup \text{Follow}(B)$
10 **return** F

The elements of the *Follow*(A) set get into the set F. In lines 4–9 we check that, if the argumentum is at the right side of a derivation rule, what symbols may stand immediately after him. It is obvious that no ε is in this set, and the symbol # is in the set only if the argumentum is the rightmost symbol of a sentential form.

Parsing with table

Suppose that we analyse a series of terminal symbols xay and the part x has already been analysed without errors. We analyse the text with a top-down method, so we use leftmost derivations. Suppose that our sentential form is $xY\alpha$, that is, it has form $xB\alpha$ or $xb\alpha$ ($Y \in (N \cup T)$, $B \in N$, $a, b \in T$, $x, y \in T^*$, $\alpha \in (N \cup T)^*$) (Figure 20.9).

In the first case the next step is the substitution of symbol B. We know the next element of the input series, this is the terminal a, therefore we can determine the correct substitution of symbol B. This substitution is the rule $B \rightarrow \beta$ for which $a \in First(\beta Follow(B))$. If there is such a rule then, according to the definition of $LL(1)$ grammar, there is exactly one. If

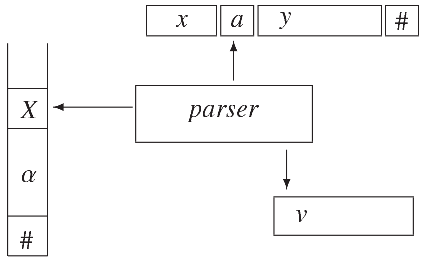

Figure 20.10. The structure of the $LL(1)$ parser.

there is not such a rule, then a *syntactic error* was found.

In the second case the next symbol of the sentential form is the terminal symbol b, thus we look out for the symbol b as the next symbol of the analysed text. If this comes true, that is, $a = b$, then the symbol a is a correct symbol and we can go further. We put the symbol a into the already analysed text. If $a \neq b$, then here is a *syntactic error*. We can see that the position of the error is known, and the erroneous symbol is the terminal symbol a.

The action of the parser is the following. Let # be the sign of the right end of the analysed text, that is, the mark # is the last symbol of the text. We use a stack through the analysing, the bottom of the stack is signed by mark #, too. We give serial numbers to derivation rules and through the analysing we write the number of the applied rule into a list. At the end of parsing we can write the syntax tree from this list (Figure 20.10).

We sign the *state of the parser* using triples $(ay\#, X\alpha\#, v)$. The symbol $ay\#$ is the text not analysed yet. $X\alpha\#$ is the part of the sentential form corresponding to the not analysed text; this information is in the stack, the symbol X is at the top of the stack. v is the list of the serial numbers of production rules.

If we analyse the text then we observe the symbol X at the top of the stack, and the symbol a that is the first symbol of the not analysed text. The name of the symbol a is *actual symbol*. There are pointers to the top of the stack and to the actual symbol.

We use a top down parser, therefore the initial content of the stack is $S\#$. If the initial analysed text is xay, then the initial state of the parsing process is the triple $(xay\#, S\#, \varepsilon)$, where ε is the sign of the empty list.

We analyse the text, the series of symbols using a *parsing table* The rows of this table sign the symbols at the top of the stack, the columns of the table sign the next input symbols, and we write mark # to the last row and the last column of the table. Hence the number of rows of the table is greater by one than the number of symbols of the grammar, and the number of columns is greater by one than the number of terminal symbols of the grammar.

The element $T[X, a]$ of the table is as follows.

$$T[X, a] = \begin{cases} (\beta, i), & \text{ha } X \to \beta \text{ az } i\text{-th derivation rule} , \\ & a \in \mathit{First}(\beta) \text{ or} \\ & (\varepsilon \in \mathit{First}(\beta) \text{ and } a \in \mathit{Follow}(X)) , \\ \mathit{pop}, & \text{if } X = a , \\ \mathit{accept}, & \text{if } X = \# \text{ and } a = \# , \\ \mathit{error} & \text{otherwise} . \end{cases}$$

We fill in the parsing table using the following algorithm.

LL(1)-TABLE-FILL-IN(G)

1 **for** all $A \in N$
2 **do if** $A \to \alpha \in P$ the i-th rule
3 **then for** all $a \in$ FIRST(α)- ra
4 **do** $T[A, a] \leftarrow (\alpha, i)$
5 **if** $\varepsilon \in$ FIRST(α)
6 **then for** all $a \in$ FOLLOW(A)
7 **do** $T[A, a] \leftarrow (\alpha, i)$
8 **for** all $a \in T$
9 **do** $T[a, a] \leftarrow pop$
10 $T[\#, \#] \leftarrow accept$
11 **for** all $X \in (N \cup T \cup \{\#\})$ and all $a \in (T \cup \{\#\})$
12 **do if** $T[X, a] =$ „empty”
13 **then** $T[X, a] \leftarrow error$
14 **return** T

At the line 10 we write the text *accept* into the right lower corner of the table. At the lines 8–9 we write the text *pop* into the main diagonal of the square labelled by terminal symbols. The program in lines 1–7 writes a tuple in which the first element is the right part of a derivation rule and the second element is the serial number of this rule. In lines 12–13 we write *error* texts into the empty positions.

The actions of the parser are written by state-transitions. The initial state is $(x\#, S\#, \varepsilon)$, where the initial text is x, and the parsing process will be finished if the parser goes into the state $(\#, \#, w)$, this state is the *final state* If the text is $ay\#$ in an intermediate step, and the symbol X is at the top of stack, then the possible state-transitions are as follows.

$$(ay\#, X\alpha\#, v) \rightarrow \begin{cases} (ay\#, \beta\alpha\#, vi), & \text{ha } T[X, a] = (\beta, i)\ , \\ (y\#, \alpha\#, v), & \text{ha } T[X, a] = pop\ , \\ O.K., & \text{ha } T[X, a] = accept\ , \\ ERROR, & \text{ha } T[X, a] = error\ . \end{cases}$$

The letters *O.K.* mean that the analysed text is syntactically correct; the text *ERROR* means that a syntactic error is detected.

The actions of this parser are written by the next algorithm.

LL(1)-PARSER$(xay\#, T)$

1 $s \leftarrow (xay\#, S\#, \varepsilon)$, $s' \leftarrow$ *analyze*
2 **repeat**
3 **if** $s = (ay\#, A\alpha\#, v)$ és $T[A, a] = (\beta, i)$
4 **then** $s \leftarrow (ay\#, \beta\alpha\#, vi)$
5 **else if** $s = (ay\#, a\alpha\#, v)$
6 **then** $s \leftarrow (y\#, \alpha\#, v)$ ▷ Then $T[a, a] = pop$.
7 **else** **if** $s = (\#, \#, v)$
8 **then** $s' \leftarrow O.K.$ ▷ Then $T[\#, \#] = accept$.
9 **else** $s' \leftarrow ERROR$ ▷ Then $T[A, a] = error$.
10 **until** $s' = O.K.$ **or** $s' = ERROR$
11 **return** s', s

The input parameters of this algorithm are the text xay and the parsing table T. The variable s' describes the state of the parser: its value is *analyse,* during the analysis, and it is either $O.K.$ or $ERROR.$ at the end. The parser determines his action by the actual symbol a and by the symbol at the top of the stack, using the parsing table T. In the line 3–4 the parser builds the syntax tree using the derivation rule $A \rightarrow \beta$. In lines 5–6 the parser executes a shift action, since there is a symbol a at the top of the stack. At lines 8–9 the algorithm finishes his work if the stack is empty and it is at the end of the text, otherwise a syntactic error was detected. At the end of this work the result is $O.K.$ or $ERROR$ in the variable s', and, as a result, there is the triple s at the output of this algorithm. If the text was correct, then we can create the syntax tree of the analysed text from the third element of the triple. If there was an error, then the first element of the triple points to the position of the erroneous symbol.

Example 20.8 Let G be a grammar $G = (\{E, E', T, T', F\}, \{+, *, (,), i\}, P, E)$, where the set P of derivation rules:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid i$

>From these rules we can determine the $Follow(A)$ sets. To fill in the parsing table, the following sets are required:

$First(TE') = \{(, i\}$,
$First(+TE') = \{+\}$,
$First(FT') = \{(, i\}$,
$First(*FT') = \{*\}$,
$First((E)) = \{(\}$,
$First(i) = \{i\}$,
$Follow(E') = \{), \#\}$,
$Follow(T') = \{+,), \#\}$.

The parsing table is as follows. The empty positions in the table mean *errors*

	$+$	$*$	(	)	i	$\#$
E			$(TE',1)$		$(TE',1)$	
E'	$(+TE',2)$			$(\varepsilon,3)$		$(\varepsilon,3)$
T			$(FT',4)$		$(FT',4)$	
T'	$(\varepsilon,6)$	$(*FT',5)$		$(\varepsilon,6)$		$(\varepsilon,6)$
F			$((E),7)$		$(i,8)$	
$+$	*pop*					
$*$		*pop*				
(			*pop*			
)				*pop*		
i					*pop*	
$\#$						*accept*

Example 20.9 Using the parsing table of the previous example, analyse the text $i + i * i$.

$$
\begin{array}{lllrrll}
(i+i*i\#, S\#, \varepsilon) & \xrightarrow{(TE',1)} & (& i+i*i\#, & TE'\#, & 1 &) \\
 & \xrightarrow{(FT',4)} & (& i+i*i\#, & FT'E'\#, & 14 &) \\
 & \xrightarrow{(i,8)} & (& i+i*i\#, & iT'E'\#, & 148 &) \\
 & \xrightarrow{pop} & (& +i*i\#, & T'E'\#, & 148 &) \\
 & \xrightarrow{(\varepsilon,6)} & (& +i*i\#, & E'\#, & 1486 &) \\
 & \xrightarrow{(+TE',2)} & (& +i*i\#, & +TE'\#, & 14862 &) \\
 & \xrightarrow{pop} & (& i*i\#, & TE'\#, & 14862 &) \\
 & \xrightarrow{(FT',4)} & (& i*i\#, & FT'E'\#, & 148624 &) \\
 & \xrightarrow{(i,8)} & (& i*i\#, & iT'E'\#, & 1486248 &) \\
 & \xrightarrow{pop} & (& *i\#, & T'E'\#, & 1486248 &) \\
 & \xrightarrow{(*FT',5)} & (& *i\#, & *FT'E'\#, & 14862485 &) \\
 & \xrightarrow{pop} & (& i\#, & FT'E'\#, & 14862485 &) \\
 & \xrightarrow{(i,8)} & (& i\#, & iT'E'\#, & 148624858 &) \\
 & \xrightarrow{pop} & (& \#, & T'E'\#, & 148624858 &) \\
 & \xrightarrow{(\varepsilon,6)} & (& \#, & E'\#, & 1486248586 &) \\
 & \xrightarrow{(\varepsilon,3)} & (& \#, & \#, & 14862485863 &) \\
 & \xrightarrow{accept} & & O.K. & & &
\end{array}
$$

The syntax tree of the analysed text is the Figure 20.11.

Recursive-descent parsing method

There is another frequently used method for the backtrackless top-down parsing. Its essence is that we write a real program for the applied grammar. We create procedures to the symbols of grammar, and using these procedures the recursive procedure calls realize the stack of the parser and the stack management. This is a top-down parsing method, and the procedures call each other recursively; it is the origin of the name of this method, that is, ***recursive-descent method***

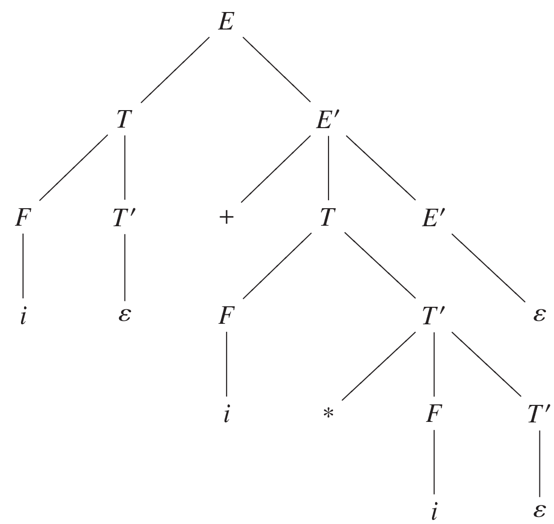

Figure 20.11. The syntax tree of the sentence $i + i * i$.

To check the terminal symbols we create the procedure *Check*. Let the parameter of this procedure be the „expected symbol", that is the leftmost unchecked terminal symbol of the sentential form, and let the *actual symbol* be the symbol which is analysed in that moment.

```
procedure Check(a);
begin
  if actual_symbol = a
      then Next_symbol
      else Error_report
end;
```

The procedure *Next_symbol* reads the next symbol, it is a call for the lexical analyser. This procedure determines the next symbol and put this symbol into the *actual_symbol* variable. The procedure *Error_report* creates an error report and then finishes the parsing.

We create procedures to symbols of the grammar as follows. The procedure of the nonterminal symbol A is the next.

```
procedure  A;
begin
  T(A)
end;
```

where `T(A)` is determined by symbols of the right part of derivation rule having symbol A in its left part.

The grammars which are used for syntactic analysis are *reduced grammars*. It means that no unnecessary symbols in the grammar, and all of symbols occur at the left side at least one reduction rule. Therefore, if we consider the symbol A, there is at least one $A \to \alpha$

production rule.

1. If there is only one production rule for the symbol A,

   (a) let the program of the rule $A \to a$ is as follows: `Check(a)`,

   (b) for the rule $A \to B$ we give the procedure call `B` ,

   (c) for the rule $A \to X_1 X_2 \ldots X_n$ $(n \geq 2)$ we give the next block:

```
begin
  T(X_1);
  T(X_2);
  ...
  T(X_n)
end;
```

2. If there are more rules for the symbol A:

   (a) If the rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ are ε-free, that is from α_i $(1 \leq i \leq n)$ it is not possible to deduce ε, then `T(A)`

```
case actual_symbol of
  First(alpha_1) : T(alpha_1);
  First(alpha_2) : T(alpha_2);
  ...
  First(alpha_n) : T(alpha_n)
end;
```

   where `First(alpha_i)` is the sign of the set $First(\alpha_i)$.

   We note that this is the first point of the method of recursive-descent parser where we use the fact that the grammar is an $LL(1)$ grammar.

   (b) We use the $LL(1)$ grammar to write a programming language, therefore it is not comfortable to require that the grammar is a ε-free grammar. For the rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_{n-1} \mid \varepsilon$ we create the next `T(A)` program:

```
case actual_symbol of
  First(alpha_1)     : T(alpha_1);
  First(alpha_2)     : T(alpha_2);
  ...
  First(alpha_(n-1)) : T(alpha_(n-1));
  Follow(A)          : skip
end;
```

   where `Follow(A)` is the set $Follow(A)$.

   In particular, if the rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ for some i $(1 \leq i \leq n)$ $\alpha_i \stackrel{*}{\Longrightarrow} \varepsilon$, that is $\varepsilon \in First(\alpha_i)$, then the i-th row of the `case` statement is

```
Follow(A) : skip
```

In the program `T(A)`, if it is possible, we use `if-then-else` or `while` statement instead of the statement `case`.

The start procedure, that is the main program of this parsing program, is the procedure which is created for the start symbol of the grammar.

We can create the recursive-descent parsing program with the next algorithm. The input of this algorithm is the grammar G, and the result is parsing program P. In this algorithm we use a Write-program procedure, which concatenates the new program lines to the program P. We will not go into the details of this algorithm.

Create-rec-desc(G)

```
1  P ← ∅
2  Write-program(
3      procedure Check(a);
4      begin
5      if actual_symbol = a
6         then Next_symbol
7         else Error_report
8      end;
9      )
10 for all symbol A ∈ N of the grammar G
11     do if A = S
12          then Write-program(
13                    program S;
14                    begin
15                      Rec-desc-stat(S, P)
16                    end.
17                    )
18          else Write-program(
19                    procedure A;
20                    begin
21                      Rec-desc-stat(A, P)
22                    end;
23                    )
24 return P
```

The algorithm creates the *Check* procedure in lines 2–9/ Then, for all nonterminals of grammar G, it determines their procedures using the algorithm Rec-desc-stat. In the lines 11–17, we can see that for the start symbol S we create the main program. The output of the algorithm is the parsing program.

Rec-desc-stat(A, P)

```
if there is only one rule A → α
  then Rec-desc-stat1(α, P)                          ▷ A → α.
  else Rec-desc-stat2(A, (α_1, ..., α_n), P)         ▷ A → α_1 | ··· | α_n.
return P
```

The form of the statements of the parsing program depends on the derivation rules of the symbol A. Therefore the algorithm Rec- desc-stat divides the next tasks into two parts. The algorithm Rec-desc-stat1 deals with the case when there is only one derivation rule, and the algorithm Rec-desc-stat2 creates the program for the alternatives.

REC-DESC-STAT1(α, P)

if $\alpha = a$
  then WRITE-PROGRAM(
      `Check(a)`
)
if $\alpha = B$
  then WRITE-PROGRAM(
      `B`
)
if $\alpha = X_1 X_2 \ldots X_n$ $(n \geq 2)$
  then WRITE-PROGRAM(
      `begin`
       REC-DESC-STAT1(X_1, P) ;
       REC-DESC-STAT1(X_2, P) ;
       ...
       REC-DESC-STAT1(X_n, P)
      `end;`
return P

REC-DESC-STAT2($A, (\alpha_1, \ldots, \alpha_n), P$)

if the rules $\alpha_1, \ldots, \alpha_n$ are ε- free
  then WRITE-PROGRAM(
      `case actual_symbol of`
       `First(alpha_1) :` REC-DESC-STAT1 (α_1, P) ;
       `...`
       `First(alpha_n) :` REC-DESC-STAT1 (α_n, P)
      `end;`
)
if there is a ε-rule, $\alpha_i = \varepsilon$ $(1 \leq i \leq n)$
  then WRITE-PROGRAM(
      `case actual_symbol of`
       `First(alpha_1)      :` REC-DESC-STAT1 (α_1, P) ;
       `...`
       `First(alpha_(i-1)) :` REC-DESC-STAT1 (α_{i-1}, P) ;
       `Follow(A)          : skip;`
       `First(alpha_(i+1)) :` REC-DESC-STAT1 (α_{i+1}, P) ;
       `...`
       `First(alpha_n)      :` REC-DESC-STAT1 (α_1, P)
      `end;`
)
return P

These two algorithms create the program described above.

Checking the end of the parsed text is achieved by the recursive- descent parsing method with the next modification. We generate a new derivation rule for the end mark #. If the start symbol of the grammar is S, then we create the new rule $S' \to S\#$, where the new symbol S' is the start symbol of our new grammar. The mark # is considered as terminal symbol. Then we generate the parsing program for this new grammar.

Example 20.10 We augment the grammar of the Example 20.8. in the above manner. The production rules are as follows.

$S' \to E\#$
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid i$

In the example 20.8. we give the necessary *First* and *Follow* sets. We use the next sets:

$First(+TE') = \{+\}$,
$First(*FT') = \{*\}$,
$First((E)) = \{(\}$,
$First(i) = \{i\}$,
$Follow(E') = \{), \#\}$,
$Follow(T') = \{+,), \#\}$.

In the comments of the program lines we give the using of these sets. The first characters of the comment are the character pair `--`.

The program of the recursive-descent parser is the following.

```
program S';
begin
  E;
  Check(#)
end.
procedure E;
begin
  T;
  E'
end;
procedure E';
begin
  case actual_symbol of
  +     : begin                -- First(+TE')
              Check(+);
              T;
              E'
          end;
  ),#   : skip                 -- Follow(E')
  end
end;
procedure T;
begin
  F;
  T'
end;
procedure T';
```

```
begin
  case actual_symbol of
  *     : begin               -- First(*FT')
              Check(*);
              F;
              T'
          end;
  +,),# : skip                -- Follow(T')
  end
end;
procedure F;
begin
  case actual_symbol of
  (     : begin               -- First((E))
              Check(();
              E;
              Check())
          end;
  i     : Check(i)           -- First(i)
  end
end;
```

We can see that the main program of this parser belongs to the symbol S'.

20.3.2. $LR(1)$ parsing

If we analyse from bottom to up, then we start with the program text. We search the handle of the sentential form, and we substitute the nonterminal symbol that belongs to the handle, for this handle. After this first step, we repeat this procedure several times. Our goal is to achieve the start symbol of the grammar. This symbol will be the root of the syntax tree, and by this time the terminal symbols of the program text are the leaves of the tree.

First we review the notions which are necessary in the parsing.

To analyse bottom-up, we have to determine the handle of the sentential form. The problem is to create a good method which finds the handle, and to find the best substitution if there are more than one possibilities.

Definition 20.12 *If $A \to \alpha \in P$, then the **rightmost substitution** of the sentential form $\beta A x$ ($x \in T^*$, $\alpha, \beta \in (N \cup T)^*$) is $\beta\alpha x$, that is*

$$\beta A x \underset{rightmost}{\Longrightarrow} \beta\alpha x \ .$$

Definition 20.13 *If the derivation $S \overset{*}{\Longrightarrow} x$ ($x \in T^*$) all of the substitutions were rightmost substitution, then this derivation is a **rightmost derivation**,*

$$S \overset{*}{\underset{rightmost}{\Longrightarrow}} x \ .$$

In a rightmost derivation, terminal symbols are at the right side of the sentential form.

By the connection of the notion of the handle and the rightmost derivation, if we apply the steps of a rightmost derivation backwards, then we obtain the steps of a bottom-up parsing. Hence the bottom-up parsing is equivalent with the „inverse" of a rightmost derivation. Therefore, if we deal with bottom-up methods, we will not write the text "rightmost" at the arrows.

General bottom-up parsing methods are realized by using backtrack algorithms. They are similar to the top-down parsing methods. But the backtrack steps make the parser very slow. Therefore we only deal with grammars such that have parsing methods without backtracks.

Hence forward we produce a very efficient algorithm for a large class of context-free grammars. This class contains the grammars for the programming languages.

The parsing is called $LR(k)$ parsing; the grammar is called $LR(k)$ grammar. *LR* means the "Left to Right" method, and k means that if we look ahead k symbols then we can determine the handles of the sentential forms. The $LR(k)$ parsing method is a shift-reduce method.

We deal with $LR(1)$ parsing only, since for all $LR(k)$ ($k > 1$) grammar there is an equivalent $LR(1)$ grammar. This fact is very important for us since, using this type of grammars, it is enough to look ahead one symbol in all cases.

Creating $LR(k)$ parsers is not an easy task. However, there are such standard programs (for example the `yacc` in UNIX systems), that create the complete parsing program from the derivation rules of a grammar. Using these programs the task of writing parsers is not too hard.

After studying the $LR(k)$ grammars we will deal with the $LALR(1)$ parsing method. This method is used in the compilers of modern programming languages.

$LR(k)$ grammars

As we did previously, we write a mark # to the right end of the text to be analysed. We introduce a new nonterminal symbol S' and a new rule $S' \to S$ into the grammar.

Definition 20.14 *Let G' be the* ***augmented grammar*** *belongs to grammar $G = (N, T, P, S)$, where G'* ***augmented grammar***

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S') \ .$$

Assign serial numbers to the derivation rules of grammar, and let $S' \to S$ be the 0th rule. Using this numbering, if we apply the 0th rule, it means that the parsing process is concluded and the text is correct.

We notice that if the original start symbol S does not happen on the right side of any rules, then there is no need for this augmentation. However, for the sake of generality, we deal with augmented grammars only.

Definition 20.15 *The augmented grammar G' is an* ***$LR(k)$ grammar*** ($k \geq 0$)*, if for derivations*

$$S' \stackrel{*}{\Longrightarrow} \alpha A w \Longrightarrow \alpha\beta w \ ,$$
$$S' \stackrel{*}{\Longrightarrow} \gamma B x \Longrightarrow \gamma\delta x = \alpha\beta y$$

($A, B \in N,\ x, y, w \in T^*,\ \alpha, \beta, \gamma, \delta \in (N \cup T)^*$) the equality

$$First_k(w) = First_k(y)$$

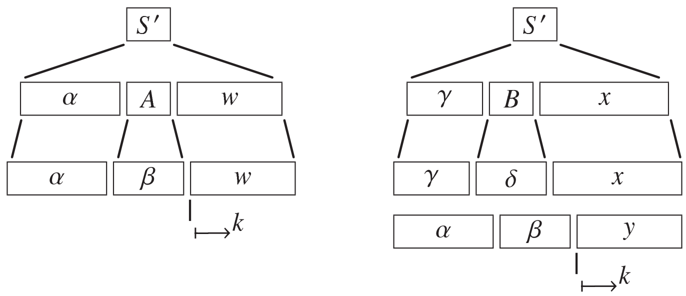

Figure 20.12. The $LR(k)$ grammar.

implies

$$\alpha = \gamma, A = B \text{ és } x = y \ .$$

The feature of $LR(k)$ grammars is that, in the sentential form $\alpha\beta w$, looking ahead k symbol from w unambiguously decides if β is or is not the handle. If the handle is *beta*, then we have to reduce the form using the rule $A \to \beta$, that results the new sentential form is $\alpha A w$. Its reason is the following: suppose that, for sentential forms $\alpha\beta w$ and $\alpha\beta y$, (their prefixes $\alpha\beta$ are same), $First_k(w) = First_k(y)$, and we can reduce $\alpha\beta w$ to $\alpha A w$ and $\alpha\beta y$ to $\gamma B x$. In this case, since the grammar is a $LR(k)$ grammar, $\alpha = \gamma$ and $A = B$ hold. Therefore in this case either the handle is β or β never is the handle.

Example 20.11 Let $G' = (\{S', S\}, \{a\}, P', S')$ be a grammar and let the derivation rules be as follows.
$S' \to S$
$S \to Sa \mid a$
This grammar is not an $LR(0)$ grammar, since using notations of the definition, in the derivations

$$S' \stackrel{*}{\Longrightarrow} \underset{\alpha}{\varepsilon}\ \underset{A}{S'}\ \underset{w}{\varepsilon} \Longrightarrow \underset{\alpha}{\varepsilon}\ \underset{\beta}{S}\ \underset{w}{\varepsilon},$$

$$S' \stackrel{*}{\Longrightarrow} \underset{\gamma}{\varepsilon}\ \underset{B}{S'}\ \underset{x}{\varepsilon} \Longrightarrow \underset{\gamma}{\varepsilon}\ \underset{\delta}{Sa}\ \underset{x}{\varepsilon} = \underset{\alpha}{\varepsilon}\ \underset{\beta}{S}\ \underset{y}{a},$$

it holds that $First_0(\varepsilon) = First_0(a) = \varepsilon$, and $\gamma B x \neq \alpha A y$.

Example 20.12
The next grammar is a $LR(1)$ grammar. $G = (\{S', S\}, \{a, b\}, P', S')$, the derivation rules are:
$S' \to S$
$S \to SaSb \mid \varepsilon$

In the next example we show that there is a context-free grammar, such that is not $LR(k)$ grammar for any k. ($k \geq 0$).

Example 20.13 Let $G' = (\{S', S\}, \{a\}, P', S')$ be a grammar and let the derivation rules be
$S' \to S$

$S \to aSa \mid a$

Now for all k ($k \geq 0$)

$$S' \overset{*}{\Longrightarrow} a^k S a^k \Longrightarrow a^k a a^k = a^{2k+1} ,$$
$$S' \overset{*}{\Longrightarrow} a^{k+1} S a^{k+1} \Longrightarrow a^{k+1} a a^{k+1} = a^{2k+3} ,$$

and

$$First_k(a^k) = First_k(aa^{k+1}) = a^k ,$$

but

$$a^{k+1} S a^{k+1} \neq a^k S a^{k+2} .$$

It is not sure that, for a $LL(k)$ ($k > 1$) grammar, we can find an equivalent $LL(1)$ grammar. However, $LR(k)$ grammars have this nice property.

Theorem 20.16 *For all $LR(k)$ ($k > 1$) grammar there is an equivalent LR(1) grammar.*

The great significance of this theorem is that it makes sufficient to study the $LR(1)$ grammars instead of $LR(k)$ ($k > 1$) grammars.

LR(1) canonical sets

Now we define a very important notion of the LR parsings.

Definition 20.17 *If β is the handle of the $\alpha\beta x$ ($\alpha, \beta \in (N \cup T)^*, x \in T^*$) sentential form, then the prefixes of $\alpha\beta$ are the **viable prefixes** of $\alpha\beta x$.*

Example 20.14 Let $G' = (\{E, T, S'\}, \{i, +, (,)\}, P', S')$ be a grammar and the derivation rule as follows.

(0) $S' \to E$
(1) $E \to T$
(2) $E \to E + T$
(3) $T \to i$
(4) $T \to (E)$

$E + (i + i)$ is a sentential form, and the first i is the handle. The viable prefixes of this sentential form are E, $E+$, $E + ($, $E + (i$.

By the above definition, symbols after the handle are not parts of any viable prefix. Hence the task of finding the handle is the task of finding the longest viable prefix.

For a given grammar, the set of viable prefixes is determined, but it is obvious that the size of this set is not always finite.

The significance of viable prefixes are the following. We can assign states of a deterministic finite automaton to viable prefixes, and we can assign state transitions to the symbols of the grammar. From the initial state we go to a state along the symbols of a viable prefix. Using this property, we will give a method to create an automaton that executes the task of parsing.

Definition 20.18 *If $A \to \alpha\beta$ is a rule of a G' grammar, then let*

$$[A \to \alpha.\beta, a] , \quad (a \in T \cup \{\#\}) ,$$

*be a **LR(1)-item**, where $A \to \alpha.\beta$ is the **core** of the $LR(1)$-item, and a is the **lookahead symbol** of the $LR(1)$-item.*

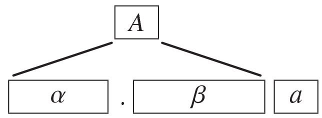

Figure 20.13. The $[A \to \alpha.\beta, a]$ $LR(1)$ -item.

The lookahead symbol is instrumental in reduction, i.e. it has form $[A \to \alpha., a]$. It means that we can execute reduction only if the symbol a follows the handle *alpha*.

Definition 20.19 *The $LR(1)$-item $[A \to \alpha.\beta, a]$ is **valid** for the viable prefix $\gamma\alpha$ if*

$$S' \overset{*}{\Longrightarrow} \gamma A x \Longrightarrow \gamma\alpha\beta x \; (\gamma \in (N \cup T)^*, \; x \in T^*) \; ,$$

and a is the first symbol of x or if $x = \varepsilon$ then $a = \#$.

Example 20.15 Let $G' = (\{S', S, A\}, \{a, b\}, P', S')$ a grammar and the derivation rules as follows.
(0) $S' \to S$
(1) $S \to AA$
(2) $A \to aA$
(3) $A \to b$
Using these rules, we can derive $S' \overset{*}{\Longrightarrow} aaAab \Longrightarrow aaaAab$. Here aaa is a viable prefix, and $[A \to a.A, a]$ is valid for this viable prefix. Similarly, $S' \overset{*}{\Longrightarrow} AaA \Longrightarrow AaaA$, and $LR(1)$-item $[A \to a.A, \#]$ is valid for viable prefix Aaa.

Creating a $LR(1)$ parser, we construct the canonical sets of $LR(1)$-items. To achieve this we have to define the *closure* and *read* functions.

Definition 20.20 *Let the set $\mathcal{H}$ be a set of $LR(1)$-items for a given grammar. The set **closure($\mathcal{H}$)** consists of the next $LR(1)$-items:*

1. *every element of the set $\mathcal{H}$ is an element of the set closure($\mathcal{H}$),*
2. *if $[A \to \alpha.B\beta, a] \in closure(\mathcal{H})$, and $B \to \gamma$ is a derivation rule of the grammar, then $[B \to .\gamma, b] \in closure(\mathcal{H})$ for all $b \in First(\beta a)$,*
3. *the set closure($\mathcal{H}$) is needed to expand using the step 2 until no more items can be added to it.*

By definitions, if the $LR(1)$-item $[A \to \alpha.B\beta, a]$ is valid for the viable prefix $\delta\alpha$, then the $LR(1)$-item $[B \to .\gamma, b]$ is valid for the same viable prefix in the case of $b \in First(\beta a)$. (Figure 20.14). It is obvious that the function *closure* creates all of $LR(1)$-items which are valid for viable prefix $\delta\alpha$.

We can define the function $closure(\mathcal{H})$, i.e. the closure of set $\mathcal{H}$ by the following algorithm. The result of this algorithm is the set $\mathcal{K}$.

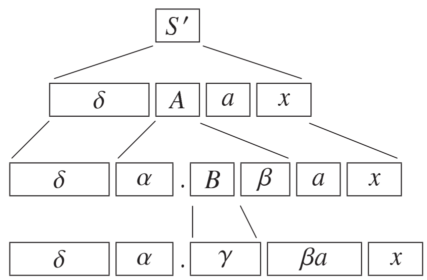

Figure 20.14. The function $closure([A \to \alpha.B\beta, a])$.

```
Closure-set-of-items(𝓗)
𝒦 ← ∅
for all E ∈ 𝓗 LR(1)-item
    do 𝒦 ← 𝒦 ∪ Closure-item(E)
return 𝒦
```

```
Closure-item(E)
 1  𝒦_E ← {E}
 2  if the LR(1)-item E has form [A → α.Bβ, a]
 3     then I ← ∅
 4          J ← 𝒦_E
 5          repeat
 6                for for all LR(1)-items ∈ J which have form [C → γ.Dδ, b]
 7                    do for for all rules D → η ∈ P
 8                           do for for all symbols c ∈ First(δb)
 9                                  do I ← I ∪ [D → .η, c]
10                J ← I
11                if I ≠ ∅
12                   then 𝒦_E ← 𝒦_E ∪ I
13                        I ← ∅
14          until J ≠ ∅
15  return 𝒦_E
```

The algorithm Closure-item creates $\mathcal{K}_E$, the closure of item E. If, in the argument E, the "point" is followed by a terminal symbol, then the result is this item only (line 1). If in E the "point" is followed by a nonterminal symbol B, then we can create new items from every rule having the symbol B at their left side (line 9). We have to check this condition for all new items, too, the **repeat** cycle is in line 5–14. These steps are executed until no more items can be added (line 14). The set J contains the items to be checked, the set I contains the new items. We can find the operation $J \leftarrow I$ in line 10.

Definition 20.21 *Let $\mathcal{H}$ be a set of LR(1)-items for the grammar G. Then the set*

***read*($\mathcal{H}, X$)** *($X \in (N \cup T)$) consists of the following $LR(1)$-items.*

1. *if $[A \to \alpha.X\beta, a] \in \mathcal{H}$, then all items of the set closure($[A \to \alpha X.\beta, a]$) are in read($\mathcal{H}, X$),*
2. *the set read($\mathcal{H}, X$) is extended using step 1 until no more items can be added to it.*

The function *read*($\mathcal{H}, X$) "reads symbol X" in items of $\mathcal{H}$, and after this operation the sign "point" in the items gets to the right side of X. If the set $\mathcal{H}$ contains the valid $LR(1)$-items for the viable prefix γ then the set *read*($\mathcal{H}, X$) contains the valid $LR(1)$-items for the viable prefix γX.

The algorithm Read-set-of-items executes the function *read*. The result is the set $\mathcal{K}$.

Read-set-of-items($\mathcal{H}, Y$)

```
K ← ∅
for all E ∈ H
    do K ← K ∪ Read-item(E, Y)
return K
```

Read-item(E, Y)

```
if E = [A → α.Xβ, a] and X = Y
  then K_{E,Y} ← Closure-item([A → αX.β, a])
  else K_{E,Y} ← ∅
return K_{E,Y}
```

Using these algorithms we can create all of items which writes the state after reading of symbol Y.

Now we introduce the following notation for $LR(1)$-items, to give shorter descriptions. Let

$$[A \to \alpha.X\beta, a/b]$$

be a notation for items

$$[A \to \alpha.X\beta, a] \text{ and } [A \to \alpha.X\beta, b] .$$

Example 20.16 The $LR(1)$-item $[S' \to .S, \#]$ is an item of the grammar in the example 20.15.. For this item

$$closure([S' \to .S, \#]) = \{[S' \to .S, \#], [S \to .AA, \#], [A \to .aA, a/b], [A \to .b, a/b]\} .$$

We can create the *canonical sets of $LR(1)$-items* or shortly the *$LR(1)$-canonical sets* with the following method.

Definition 20.22 *Canonical sets of $LR(1)$-items $\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_m$ are the following.*

- $\mathcal{H}_0 = closure([S' \to .S, \#])$,

- *Create the set read*($\mathcal{H}_0, X$) *for a symbol X. If this set is not empty and it is not equal to canonical set* $\mathcal{H}_0$ *then it is the next canonical set* $\mathcal{H}_1$.

  Repeat this operation for all possible terminal and nonterminal symbol X. If we get a nonempty set which is not equal to any of previous sets then this set is a new canonical set, and its index is greater by one as the maximal index of previously generated canonical sets.

- *repeat the above operation for all previously generated canonical sets and for all symbols of the grammar until no more items can be added to it.*

  The sets

$$\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_m$$

are the canonical sets of $LR(1)$*-items of the grammar G.*

The number of elements of $LR(1)$-items for a grammar is finite, hence the above method is terminated in finite time.

The next algorithm creates canonical sets of the grammar G.

Create-canonical-sets(G)

```
 1  i ← 0
 2  H_i ← Closure-item([S′ → .S, #])
 3  I ← {H_i}, K ← {H_i}
 4  repeat
 5        L ← K
 6        for all M ∈ I-re
 7            do I ← I \ M
 8               for all X ∈ T ∪ N-re
 9                   do J ← Closure-set-of-items(Read-set-of-items(M, X))
10                      if J ≠ ∅ and J ∉ K
11                        then i ← i + 1
12                             H_i ← J
13                             K ← K ∪ {H_i}
14                             I ← I ∪ {H_i}
15  until K = L
16  return K
```

The result of the algorithm is K. The first canonical set is the set $\mathcal{H}_0$ in the line 2. Further canonical sets are created by functions Closure-set-of-items(Read-set-of-items) in the line 9. The program in the line 10 checks that the new set differs from previous sets, and if the answer is true then this set will be a new set in lines 11–12. The **for** cycle in lines 6–14 guarantees that these operations are executed for all sets previously generated. In lines 3–14 the **repeat** cycle generate new canonical sets as long as it is possible.

Example 20.17 The canonical sets of $LR(1)$-items for the example 20.15. are as follows.

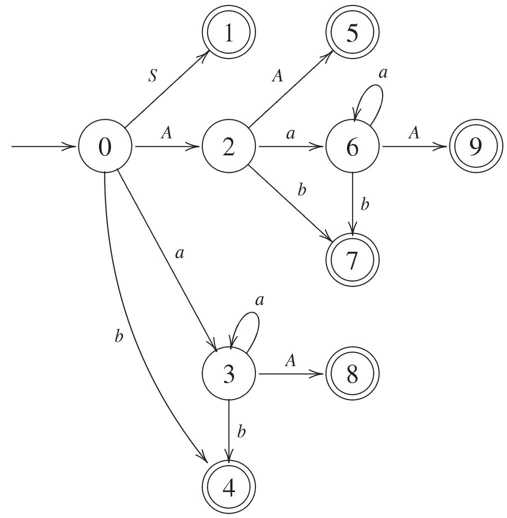

Figure 20.15. The automaton of the example 20.15..

$$
\begin{array}{lllll}
\mathcal{H}_0 & & = closure([S' \to .S]) & = & \{[S' \to .S, \#], [S \to .AA, \#], \\
 & & & & [A \to .aA, a/b], [A \to .b, a/b]\} \\
\mathcal{H}_1 & = read(\mathcal{H}_0, S) & = closure([S' \to S., \#]) & = & \{[S' \to S., \#]\} \\
\mathcal{H}_2 & = read(\mathcal{H}_0, A) & = closure([S' \to A.A, \#]) & = & \{[S \to A.A, \#], [A \to .aA, \#], \\
 & & & & [A \to .b, \#]\} \\
\mathcal{H}_3 & = read(\mathcal{H}_0, a) & = closure([A \to a.A, a/b]) & = & \{[A \to a.A, a/b], [A \to .aA, a/b], \\
 & & & & [A \to .b, a/b]\} \\
\mathcal{H}_4 & = read(\mathcal{H}_0, b) & = closure([A \to b., a/b]) & = & \{[A \to b., a/b]\} \\
\mathcal{H}_5 & = read(\mathcal{H}_2, A) & = closure([S \to AA., \#]) & = & \{[S \to AA., \#]\} \\
\mathcal{H}_6 & = read(\mathcal{H}_2, a) & = closure([A \to a.A, \#]) & = & \{[A \to a.A, \#], [A \to .aA, \#], \\
 & & & & [A \to .b, \#]\} \\
\mathcal{H}_7 & = read(\mathcal{H}_2, b) & = closure([A \to b., \#]) & = & \{[A \to b., \#]\} \\
\mathcal{H}_8 & = read(\mathcal{H}_3, A) & = closure([A \to aA., a/b]) & = & \{[A \to aA., a/b]\} \\
 & \ read(\mathcal{H}_3, a) & = \mathcal{H}_3 & & \\
 & \ read(\mathcal{H}_3, b) & = \mathcal{H}_4 & & \\
\mathcal{H}_9 & = read(\mathcal{H}_6, A) & = closure([A \to aA., \#]) & = & \{[A \to aA., \#]\} \\
 & \ read(\mathcal{H}_6, a) & = \mathcal{H}_6 & & \\
 & \ read(\mathcal{H}_6, b) & = \mathcal{H}_7 & &
\end{array}
$$

The automaton of the parser is in Figure 20.15.

LR(1) parser

If the canonical sets of $LR(1)$-items

$$\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_m$$

were created, then assign the state k of an automaton to the set $\mathcal{H}_k$. Relation between the states of the automaton and the canonical sets of $LR(1)$-items is stated by the next theorem. This theorem is the ***"great" theorem of the LR(1)-parsing***.

Theorem 20.23 *The set of the $LR(1)$-items being valid for a viable prefix γ can be assigned to the automaton-state k such that there is path from the initial state to state k labeled by gamma.*

This theorem states that we can create the automaton of the parser using canonical sets. Now we give a method to create this $LR(1)$ parser from canonical sets of $LR(1)$-items.

The deterministic finite automaton can be described with a table, that is called $LR(1)$ *parsing table*. The rows of the table are assigned to the states of the automaton.

The parsing table has two parts. The first is the *action* table. Since the operations of parser are determined by the symbols of analysed text, the *action* table is divided into columns labeled by the terminal symbols. The *action* table contains information about the action performing at the given state and at the given symbol. These actions can be shifts or reductions. The sign of a shift operation is sj, where j is the next state. The sign of the reduction is ri, where i is the serial number of the applied rule. The reduction by the rule having the serial number zero means the termination of the parsing and that the parsed text is syntactically correct; for this reason we call this operation *accept*.

The second part of the parsing table is the *goto* table. In this table are informations about shifts caused by nonterminals. (Shifts belong to terminals are in the action table.)

Let $\{0, 1, \ldots, m\}$ be the set of states of the automata. The i-th row of the table is filled in from the $LR(1)$-items of canonical set $\mathcal{H}_i$.

The i-th row of the *action* table:

- if $[A \to \alpha.a\beta, b] \in \mathcal{H}_i$ and $read(\mathcal{H}_i, a) = \mathcal{H}_j$ then $action[i, a] = sj$,

- if $[A \to \alpha., a] \in \mathcal{H}_i$ and $A \neq S'$, then $action[i, a] = rl$, where $A \to \alpha$ is the l-th rule of the grammar,

- if $[S' \to S., \#] \in \mathcal{H}_i$, then $action[i, \#] = accept$.

The method of filling in the *goto* table:

- if $read(\mathcal{H}_i, A) = \mathcal{H}_j$, then $goto[i, A] = j$.

- In both table we have to write the text *error* into the empty positions.

These *action* and *goto* tables are called *canonical parsing tables*.

Theorem 20.24 *The augmented grammar G' is $LR(1)$ grammar iff we can fill in the parsing tables created for this grammar without conflicts.*

We can fill in the parsing tables with the next algorithm.

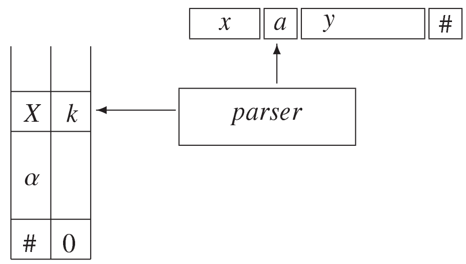

Figure 20.16. The structure of the *LR*(1) parser.

FILL-IN-LR(1)-TABLE(G)

1 **for** all LR(1) canonical sets $\mathcal{H}_i$
2 **do for** all LR(1)-items
3 **if** $[A \to \alpha.a\beta, b] \in \mathcal{H}_i$ and $read(\mathcal{H}_i, a) = \mathcal{H}_j$
4 **then** $action[i, a] = sj$
5 **if** $[A \to \alpha., a] \in \mathcal{H}_i$ and $A \neq S'$ and $A \to \alpha$ the l-th rule
6 **then** $action[i, a] = rl$
7 **if** $[S' \to S., \#] \in \mathcal{H}_i$
8 **then** $action[i, \#] = accept$
9 **if** $read(\mathcal{H}_i, A) = \mathcal{H}_j$
10 **then** $goto[i, A] = j$
11 **for** all $a \in (T \cup \{\#\})$
12 **do if** $action[i, a] =$ „empty”
13 **then** $action[i, a] \leftarrow error$
14 **for** all $X \in N$
15 **do if** $goto[i, X] =$ „empty”
16 **then** $goto[i, X] \leftarrow error$
17 **return** *action*, *goto*

We fill in the tables its line-by-line. In lines 2–6 of the algorithm we fill in the *action* table, in lines 9–10 we fill in the *goto* table. In lines 11–13 we write the *error* into the positions which remained empty.

Now we deal with the steps of the *LR*(1) parsing. (Figure 20.16).

The *state of the parsing* is written by configurations. A configuration of the *LR*(1) parser consists of two parts, the first is the stack and the second is the unexpended input text.

The stack of the parsing is a double stack, we write or read two data with the operations *push* or *pop*. The stack consists of pairs of symbols, the first element of pairs there is a terminal or nonterminal symbol, and the second element is the serial number of the state of automaton. The content of the start state is #0.

The *start configuration* is $(\#0, z\#)$, where z means the unexpected text.

The parsing is successful if the parser moves to *final state*. In the final state the content of the stack is #0, and the parser is at the end of the text.

Suppose that the parser is in the configuration $(\#0\dots Y_k i_k, ay\#)$. The next move of the parser is determined by $action[i_k, a]$.

State transitions are the following.

- If $action[i_k, a] = sl$, i.e. the parser executes a shift, then the actual symbol a and the new state l are written into the stack. That is, the new configuration is

$$(\#0\dots Y_k i_k, ay\#) \to (\#0\dots Y_k i_k a i_l, y\#) \ .$$

- If $action[i_k, a] = rl$, then we execute a reduction by the i-th rule $A \to \alpha$. In this step we delete $|\alpha|$ rows, i.e. we delete $2|\alpha|$ elements from the stack, and then we determine the new state using the *goto* table. If after the deletion there is the state i_{k-r} at the top of the stack, then the new state is $goto[i_{k-r}, A] = i_l$.

$$(\#0\dots Y_{k-r} i_{k-r} Y_{k-r+1} i_{k-r+1}\dots Y_k i_k, y\#) \to (\#0\dots Y_{k-r} i_{k-r} A i_l, y\#) \ ,$$

  where $|\alpha| = r$.

- If $action[i_k, a] = accept$, then the parsing is completed, and the analysed text was correct.
- If $action[i_k, a] = error$, then the parsing terminates, and a *syntactic error* was discovered at the symbol a.

The $LR(1)$ parser is often named *canonical* $LR(1)$ *parser*.

Denote the *action* and *goto* tables together by T. We can give the following algorithm for the steps of parser.

LR(1)-PARSER$(xay\#, T)$

1 $s \leftarrow (\#0, xay\#)$, $s' \leftarrow parsing$
2 **repeat**
3 $\quad s = (\#0\dots Y_{k-r} i_{k-r} Y_{k-r+1} i_{k-r+1}\dots Y_k i_k, ay\#)$
4 $\quad$ **if** $action[i_k, a] = sl$
5 $\quad\quad$ **then** $s \leftarrow (\#0\dots Y_k i_k a i_l, y\#)$
6 $\quad\quad$ **else if** $action[i_k, a] = rl$ and $A \to \alpha$ is the l-th rule and
7 $\quad\quad\quad$ $|\alpha| = r$ and $goto[i_{k-r}, A] = i_l$
8 $\quad\quad\quad$ **then** $s \leftarrow (\#0\dots Y_{k-r} i_{k-r} A i_l, ay\#)$
9 $\quad\quad\quad$ **else if** $action[i_k, a] = accept$
10 $\quad\quad\quad\quad$ **then** $s' \leftarrow O.K.$
11 $\quad\quad\quad\quad$ **else** $s' \leftarrow ERROR$
12 **until** $s' = O.K.$ **or** $s' = ERROR$
13 **return** s', s

The input parameters of the algorithm are the text xay and table T. The variable s' indicates the action of the parser. It has value *parsing* in the intermediate states, and its value is *O.K.* or *ERROR* at the final states. In line 3 we detail the configuration of the parser, that is necessary at lines 6–8. Using the *action* table, the parser determines its move from the symbol x_k at the top of the stack and from the actual symbol a. In lines 4–5 we execute a shift step, in lines 6–8 a reduction. The algorithm is completed in lines 9–11. At this moment, if the parser is at the end of text and the state 0 is at the top of stack, then the text is correct, otherwise a syntax error was detected. According to this, the output of the algorithm is *O.K.* or *ERROR*, and the final configuration is at the output, too. In the case of error, the first symbol of the second element of the configuration is the erroneous symbol.

Example 20.18 The *action* and *goto* tables of the $LR(1)$ parser for the grammar of example 20.15. are as follows. The empty positions denote *errors*.

state	*action*			*goto*	
	a	b	#	S	A
0	$s3$	$s4$		1	2
1			*accept*		
2	$s6$	$s7$			5
3	$s3$	$s4$			8
4	$r3$	$r3$			
5			$r1$		
6	$s6$	$s7$			9
7			$r3$		
8	$r2$	$r2$			
9			$r2$		

Example 20.19 Using the tables of the previous example, analyse the text $abb\#$.

$$
\begin{array}{llll|l}
 & & & & rule \\
(\#0, & aab\#) & \xrightarrow{s3} & (\#0a3, \quad bb\#) & \\
 & & \xrightarrow{s4} & (\#0a3b4, \quad b\#) & \\
 & & \xrightarrow{r3} & (\#0a3A8, \quad b\#) & A \rightarrow b \\
 & & \xrightarrow{r2} & (\#0A2, \quad b\#) & A \rightarrow aA \\
 & & \xrightarrow{s7} & (\#0A2b7, \quad \#) & \\
 & & \xrightarrow{r3} & (\#0A2A5, \quad \#) & A \rightarrow b \\
 & & \xrightarrow{r1} & (\#0S1, \quad \#) & S \rightarrow AA \\
 & & \xrightarrow{elfogad} & O.K. &
\end{array}
$$

The syntax tree of the sentence is in Figure 20.17.

LALR(1) parser

Our goal is to decrease the number of states of the parser, since not only the size but the speed of the compiler is dependent on the number of states. At the same time, we wish not to cut radically the set of $LR(1)$ grammars and languages, by using our new method.

There are a lot of $LR(1)$-items in the canonical sets, such that are very similar: their core are the same, only their lookahead symbols are different. If there are two or more canonical sets in which there are similar items only, then we merge these sets.

If the canonical sets $\mathcal{H}_i$ és a $\mathcal{H}_j$ are mergeable, then let $\mathcal{K}_{[i,j]} = \mathcal{H}_i \cup \mathcal{H}_j$.

Execute all of possible merging of $LR(1)$ canonical sets. After renumbering the indexes we obtain sets $\mathcal{K}_0, \mathcal{K}_1, \ldots, \mathcal{K}_n$; these are the *merged* $LR(1)$ *canonical sets* or $LALR(1)$

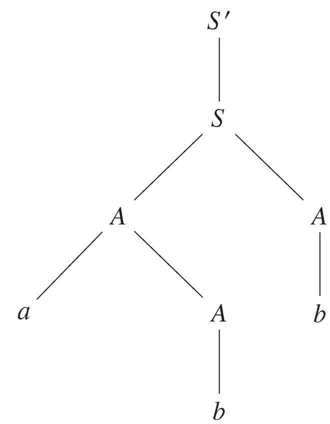

Figure 20.17. The syntax tree of the sentence *aab*.

canonical sets.

We create the $LALR(1)$ parser from these united canonical sets.

Example 20.20 Using the $LR(1)$ canonical sets of the example 20.17., we can merge the next canonical sets:
$\mathcal{H}_3$ and $\mathcal{H}_6$,
$\mathcal{H}_4$ and $\mathcal{H}_7$,
$\mathcal{H}_8$ and $\mathcal{H}_9$.
In the Figure 20.15 it can be seen that mergeable sets are in equivalent or similar positions in the automaton.

There is no difficulty with the function *read* if we use merged canonical sets. If

$$\mathcal{K} = \mathcal{H}_1 \cup \mathcal{H}_2 \cup \ldots \cup \mathcal{H}_k \ ,$$

$$read(\mathcal{H}_1, X) = \mathcal{H}_1^{'}, read(\mathcal{H}_2, X) = \mathcal{H}_2^{'}, \ldots, read(\mathcal{H}_k, X) = \mathcal{H}_k^{'} \ ,$$

and

$$\mathcal{K}' = \mathcal{H}_1^{'} \cup \mathcal{H}_2^{'} \cup \ldots \cup \mathcal{H}_k^{'} \ ,$$

then

$$read(\mathcal{K}, X) = \mathcal{K}' \ .$$

We can prove this on the following way. By the definition of function *read*, the set $read(\mathcal{H}, X)$ depends on the core of $LR(1)$-items in $\mathcal{H}$ only, and it is independent of the lookahead symbols. Since the cores of $LR(1)$-items in the sets $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_k$ are the same, the cores of $LR(1)$-items of

$$read(\mathcal{H}_1, X), read(\mathcal{H}_2, X), \ldots, read(\mathcal{H}_k, X)$$

are also the same. It follows that these sets are mergeable into a set $\mathcal{K}'$, thus $read(\mathcal{K}, X) = \mathcal{K}'$.

However, after merging canonical sets of $LR(1)$-items, elements of this set can raise difficulties. Suppose that

$\mathcal{K}_{[i,j]} = \mathcal{H}_i \cup \mathcal{H}_j$.

- After merging there are not *shift-shift conflicts*. If
$$[A \to \alpha.a\beta, b] \in \mathcal{H}_i$$
and
$$[B \to \gamma.a\delta, c] \in \mathcal{H}_j$$
then there is a shift for the symbol a and we saw that the function *read* does not cause problem, i.e. the set $read(\mathcal{K}_{[i,j]}, a)$ is equal to the set $read(\mathcal{H}_i, a) \cup read(\mathcal{H}_j, a)$.
- If there is an item
$$[A \to \alpha.a\beta, b]$$
in the canonical set $\mathcal{H}_i$ and there is an item
$$[B \to \gamma., a]$$
in the set a $\mathcal{H}_j$, then the merged set is an inadequate set with the symbol a, i.e. there is a *shift-reduce conflict* in the merged set.

  But this case never happens. Both items are elements of the set $\mathcal{H}_i$ and of the set $\mathcal{H}_j$. These sets are mergeable sets, thus they are different in lookahead symbols only. It follows that there is an item $[A \to \alpha.a\beta, c]$ in the set $\mathcal{H}_j$. Using the theorem 20.24 we get that the grammar is not a $LR(1)$ grammar; we get shift-reduce conflict from the set $\mathcal{H}_j$ for the $LR(1)$ parser, too.
- However, after merging *reduce-reduce conflict* may arise. The properties of $LR(1)$ grammar do not exclude this case. In the next example we show such a case.

Example 20.21 Let $G' = (\{S', S, A, B\}, \{a, b, c, d, e\}, P', S')$ be a grammar, and the derivation rules are as follows.

$S' \to S$
$S \to aAd \mid bBd \mid aBe \mid bAe$
$A \to c$
$B \to c$

This grammar is a $LR(1)$ grammar. For the viable prefix ac the $LR(1)$-items

$$\{[A \to c., d], [B \to c., e]\} ,$$

for the viable prefix bc the $LR(1)$-items

$$\{[A \to c., e], [B \to c., d]\}$$

create two canonical sets.

After merging these two sets we get a reduce-reduce conflict. If the input symbol is d or e then the handle is c, but we cannot decide that if we have to use the rule $A \to c$ or the rule $B \to c$ for reducing.

Now we give the method for creating a $LALR(1)$ parsing table. First we give the canonical sets of $LR(1)$-items

$$\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_m$$

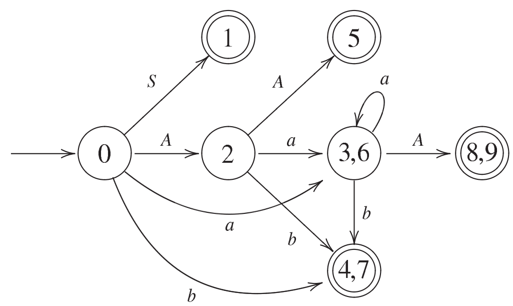

Figure 20.18. The automaton of the example 20.22..

, then we merge canonical sets in which the sets constructed from the core of the items are identical ones. Let

$$\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_n \quad (n \leq m)$$

be the $LALR(1)$ canonical sets.

For the calculation of the size of the *action* and *goto* tables and for filling in these tables we use the sets $\mathcal{K}_i \quad (1 \leq i \leq n)$. The method is the same as it was in the $LR(1)$ parsers. The constructed tables are named by $LALR(1)$ *parsing tables*.

Definition 20.25 *If the filling in the* $LALR(1)$ *parsing tables do not produce conflicts then the grammar is said to be an* $\mathbf{LALR(1)}$ *grammar.*

The run of $LALR(1)$ parser is the same as it was in $LR(1)$ parser.

Example 20.22 Denote the result of merging canonical sets $\mathcal{H}_i$ and $\mathcal{H}_j$ by $\mathcal{K}_{[i,j]}$. Let $[i, j]$ be the state which belonging to this set.

The $LR(1)$ canonical sets of the grammar of example 20.15. were given in the example 20.17. and the mergeable sets were seen in the example 20.20.. For this grammar we can create the next $LALR(1)$ parsing tables.

állapot	*action*			*goto*	
	a	b	#	S	A
0	$s\,[3,6]$	$s\,[4,7]$		1	2
1			*accept*		
2	$s\,[3,6]$	$s\,[4,7]$			5
$[3,6]$	$s\,[3,6]$	$s\,[4,7]$			$[8,9]$
$[4,7]$	$r3$	$r3$	$r3$		
5			$r1$		
$[8,9]$	$r2$	$r2$	$r2$		

The filling in the $LALR(1)$ tables are conflict free, therefore the grammar is an $LALR(1)$ grammar. The automaton of this parser is in Figure 20.18.

Example 20.23 Analyse the text *abb#* using the parsing table of the previous example.

$$
\begin{array}{llll|l}
 & & & & \textit{rule} \\
(\#0, \quad aab\#) & \xrightarrow{s[3,6]} & (\#0a\,[3,6]\,, & bb\#) & \\
 & \xrightarrow{s[4,7]} & (\#0a\,[3,6]\,b\,[4,7]\,, & b\#) & \\
 & \xrightarrow{r3} & (\#0a\,[3,6]\,A[8,9], & b\#) & A \to b \\
 & \xrightarrow{r2} & (\#0A2, & b\#) & A \to aA \\
 & \xrightarrow{s[4,7]} & (\#0A2b\,[4,7]\,, & \#) & \\
 & \xrightarrow{r3} & (\#0A2A5, & \#) & A \to b \\
 & \xrightarrow{r1} & (\#0S\,1, & \#) & S \to AA \\
 & \xrightarrow{elfogad} & O.K. & &
\end{array}
$$

The syntax tree of the parsed text is in the Figure 20.17.

As it can be seen from the previous example, the *LALR*(1) grammars are *LR*(1) grammars. The converse assertion is not true. In the example 20.21. there is a grammar which is *LR*(1), but it is not *LALR*(1) grammar.

Programming languages can be written by *LALR*(1) grammars. The most frequently used methods in compilers of programming languages is the *LALR*(1) method. The advantage of the *LALR*(1) parser is that the sizes of parsing tables are smaller than the size of *LR*(1) parsing tables.

For example, the *LALR*(1) parsing tables for the Pascal language have a few hundreds of lines, whilst the *LR*(1) parsers for this language have a few thousands of lines.

Exercises

20.3-1 Find the *LL*(1) grammars among the following grammars (we give their derivation rules only).

1. $\begin{array}{lll} S & \to & ABc \\ A & \to & a \mid \varepsilon \\ B & \to & b \mid \varepsilon \end{array}$

2. $\begin{array}{lll} S & \to & Ab \\ A & \to & a \mid B \mid \varepsilon \\ B & \to & b \mid \varepsilon \end{array}$

3. $\begin{array}{lll} S & \to & ABBA \\ A & \to & a \mid \varepsilon \\ B & \to & b \mid \varepsilon \end{array}$

4. $\begin{array}{lll} S & \to & aSe \mid A \\ A & \to & bAe \mid B \\ B & \to & cBe \mid d \end{array}$

20.3-2 Prove that the next grammars are *LL*(1) grammars (we give their derivation rules only).

1. $\begin{array}{lll} S & \to & Bb \mid Cd \\ B & \to & aB \mid \varepsilon \\ C & \to & cC \mid \varepsilon \end{array}$

2. $S \rightarrow aSA \mid \varepsilon$
   $A \rightarrow c \mid bS$

3. $S \rightarrow AB$
   $A \rightarrow a \mid \varepsilon$
   $B \rightarrow b \mid \varepsilon$

20.3-3 Prove that the next grammars are not $LL(1)$ grammars (we give their derivation rules only).

1. $S \rightarrow aAa \mid Cd$
   $A \rightarrow abS \mid c$

2. $S \rightarrow aAaa \mid bAba$
   $A \rightarrow b \mid \varepsilon$

3. $S \rightarrow abA \mid \varepsilon$
   $A \rightarrow Saa \mid b$

20.3-4 Show that a $LL(0)$ language has only one sentence.

20.3-5 Prove that the next grammars are $LR(0)$ grammars (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow aSa \mid aSb \mid c$

2. $S' \rightarrow S$
   $S \rightarrow aAc$
   $A \rightarrow Abb \mid b$

20.3-6 Prove that the next grammars are $LR(1)$ grammars. (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow aSS \mid b$

2. $S' \rightarrow S$
   $S \rightarrow SSa \mid b$

20.3-7 Prove that the next grammars are not $LR(k)$ grammars for any k (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow aSa \mid bSb \mid a \mid b$

2. $S' \rightarrow S$
   $S \rightarrow aSa \mid bSa \mid ab \mid ba$

20.3-8 Prove that the next grammars are $LR(1)$ but are not $LALR(1)$ grammars (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow Aa \mid bAc \mid Bc \mid bBa$
   $A \rightarrow d$
   $B \rightarrow d$

2.
$$
\begin{array}{lcl}
S' & \to & S \\
S & \to & aAcA \mid A \mid B \\
A & \to & b \mid Ce \\
B & \to & dD \\
C & \to & b \\
D & \to & CcS \mid CcD
\end{array}
$$

20.3-9 Create parsing table for the above $LL(1)$ grammars.
20.3-10 Using the recursive descent method, write the parsing program for the above $LL(1)$ grammars.
20.3-11 Create canonical sets and the parsing tables for the above $LR(1)$ grammars.
20.3-12 Create merged canonical sets and the parsing tables for the above $LALR(1)$ grammars.

Problems

20-1. Lexical analysis of a program text
The algorithm LEX-ANALYSE in the section 20.2 gives a scanner for the text that is described by *only one* regular expression or deterministic finite automaton, i.e. this scanner is able to analyse only one symbol. Create an automaton which executes total lexical analysis of a program language, and give the algorithm LEX-ANALYSE-LANGUAGE for this automaton. Let the input of the algorithm be the text of a program, and the output be the series of symbols. It is obvious that if the automaton goes into a finite state then its new work begins at the initial state, for analysing the next symbol. The algorithm finishes his work if it is at the end of the text or a lexical error is detected.

20-2. Series of symbols augmented with data of symbols
Modify the algorithm of the previous task on such way that the output is the series of symbols augmented with the appropriate attributes. For example, the attribute of a variable is the character string of its name, or the attribute of a number is its value and type. It is practical to write pointers to the symbols in places of data.

20-3. LALR(1) parser from LR (0) canonical sets
If we omit lookahead symbols from the $LR(1)$-items then we get ***LR(0)-items***. We can define functions *closure* and *read* for $LR(0)$-items too, doing not care for lookahead symbols. Using a method similar to the method of $LR(1)$, we can construct ***LR(0) canonical sets***

$$\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_n .$$

One can observe that the number of merged canonical sets is equal to the number of $LR(0)$ canonical sets, since the cores of $LR(1)$-items of the merged canonical sets are the same as the items of the $LR(0)$ canonical sets. Therefore the number of states of $LALR(1)$ parser is equal to the number of states of its $LR(0)$ parser.

Using this property, we can construct $LALR(1)$ canonical sets from $LR(0)$ canonical sets, by completing the items of the $LR(0)$ canonical sets with lookahead symbols. The result of this procedure is the set of $LALR(1)$ canonical sets.

It is obvious that the right part of an $LR(1)$-item begins with symbol point only if this item was constructed by the function closure. (We notice that there is one exception, the $[S' \to .S]$ item of the canonical set $\mathcal{H}_0$.) Therefore it is no need for all items of $LR(1)$ canonical sets. Let the ***kernel*** of the canonical set $\mathcal{H}_0$ be the $LR(1)$-item $[S' \to .S, \#]$, and let the kernel of any other canonical set be the set of the $LR(1)$-items such that there is no point at the first position on the right side of the item. We give an $LR(1)$ canonical set by its kernel, since all of items can be construct from the kernel using the function *closure*.

If we complete the items of the kernel of $LR(0)$ canonical sets then we get the kernel of the merged $LR(1)$ canonical sets. That is, if the kernel of an $LR(0)$ canonical set is $\mathcal{I}_j$, then from it with completions we get the kernel of the $LR(1)$ canonical set, $\mathcal{K}_j$.

If we know $\mathcal{I}_j$ then we can construct $read(\mathcal{I}_j, X)$ easily. If $[B \to \gamma.C\delta] \in \mathcal{I}_j$, $C \xrightarrow{*} A\eta$ and $A \to X\alpha$, then $[A \to X.\alpha] \in read(\mathcal{I}_j, X)$. For $LR(1)$-items, if $[B \to \gamma.C\delta, b] \in \mathcal{K}_j$, $C \xrightarrow{*} A\eta$ and $A \to X\alpha$ then we have to determine also the lookahead symbols, i.e. the symbols a such that $[A \to X.\alpha, a] \in read(\mathcal{K}_j, X)$.

If $\eta\delta \neq \varepsilon$ and $a \in First(\eta\delta b)$ then it is sure that $[A \to X.\alpha, a] \in read(\mathcal{K}_j, X)$. In this case, we say that the lookahead symbol was ***spontaneously generated*** for this item of canonical set $read(\mathcal{K}_j, X)$. The symbol b do not play important role in the construction of the lookahead symbol.

If $\eta\delta = \varepsilon$ then $[A \to X.\alpha, b]$ is an element of the set $read(\mathcal{K}_j, X)$, and the lookahead symbol is b. In this case we say that the lookahead symbol is ***propagated*** from $\mathcal{K}_j$ into the item of the set $read(\mathcal{K}_j, X)$.

If the kernel $\mathcal{I}_j$ of an $LR(0)$ canonical set is given then we construct the propagated and spontaneously generated lookahead symbols for items of $read(\mathcal{K}_j, X)$ by the following algorithm.

For all items $[B \to \gamma.\delta] \in \mathcal{I}_j$ we construct the set $\mathcal{K}_j = closure([B \to \gamma.\delta, @])$, where @ is a dummy symbol,

- if $[A \to \alpha.X\beta, a] \in \mathcal{K}_j$ and $a \neq @$ then $[A \to \alpha X.\beta, a] \in read(\mathcal{K}_j, X)$ and the symbol a is spontaneously generated into the item of the set $read(\mathcal{K}_j, X)$,
- if $[A \to \alpha.X\beta, @] \in \mathcal{K}_j$ then $[A \to \alpha X.\beta, @] \in read(\mathcal{K}_j, X)$, and the symbol @ is propagated from $\mathcal{K}_j$ into the item of the set $read(\mathcal{K}_j, X)$.

The kernel of the canonical set $\mathcal{K}_0$ has only one element. The core of this element is $[S' \to .S]$. For this item we can give the lookahead symbol # directly. Since the core of the kernel of all $\mathcal{K}_j$ canonical sets are given, using the above method we can calculate all of propagated and spontaneously generated symbols.

Give the algorithm which constructs $LALR(1)$ canonical sets from $LR(0)$ canonical sets using the methods of propagation and spontaneously generation.

Chapter notes

The theory and practice of compilers, computers and program languages are of the same age. The construction of first compilers date back to the 1950's. The task of writing compilers was a very hard task at that time, the first Fortran compiler took 18 man-years to implement [3]. From that time more and more precise definitions and solutions have been given to the problems of compilation, and better and better methods and utilities have been used in the construction of translators.

The development of formal languages and automata was a great leap forward, and we can say that this development was urged by the demand of writing of compilers. In our days this task is a simple routine project. New results, new discoveries are expected in the field of code optimisation only.

One of the earliest nondeterministic and backtrack algorithms appeared in the 1960's. The first two dynamic programming algorithms were the CYK (Cocke–Younger–Kasami) algorithm from 1965–67 and the Earley-algorithm from 1965. The idea of precedence parsers is from the end of 1970's and from the beginning of 1980's. The *LR*(k) grammars was defined by Knuth in 1965; the definition of *LL*(k) grammars is dated from the beginning of 1970's. *LALR*(1) grammars were studied by De Remer in 1971, the elaborating of *LALR*(1) parsing methods were finished in the beginning of 1980's [1, 2, 3].

To the middle of 1980's it became obvious that the *LR* parsing methods are the real efficient methods and since than the *LALR*(1) methods are used in compilers [1].

A lot of very excellent books deal with the theory and practice of compiles. Perhaps the most successful of them was the book of Gries [10]; in this book there are interesting results for precedence grammars. The first successful book which wrote about the new *LR* algorithms was of Aho and Ullman [2], we can find here also the CYK and the Early algorithms. It was followed by the "dragon book" of Aho and Ullman[3]; the extended and corrected issue of it was published in 1986 by authors Aho, Ullman and Sethi [1].

Without completeness we notice the books of Fischer and LeBlanc [8], Tremblay and Sorenson [18], Waite and Goos [20], Hunter[12], Pittman [16] and Mak [14]. Advanced achievements are in recently published books, among others in the book of Muchnick [15], Grune, Bal, Jacobs and Langendoen [11], in the book of Cooper and Torczon [6] and in a chapter of the book by Louden [13].

There are many books on topics of the theory of formal languages and automata in Hungarian. For example the book of Judit Bánkfalvi, Zsolt Bánkfalvi and Gábor Bognár published in 1978 [5]. There are books in which a chapter deals with compilers, for example in the book of Iván Bach [4], Zoltán Fülöp [9], György Révész [17] and László Varga [19], as well as in the electronic course notes of Pál Dömösi, Attila Fazekas, Géza Horváth and Zoltán Mecsei [7]. In [4, 7, 17] the CYK and the Earley algorithms are investigated. About precedence parsers we can read the books [4] and [19]. [4, 5, 9] treat also *LR* parsing methods.

Bibliography

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986. 998

[2] A. V. Aho, J. D. Ullman. *The Theory of Parsing, Translation and Compiling Vol. I.* Prentice-Hall, 1972. 998

[3] A. V. Aho, J. D. Ullman. *The Theory of Parsing, Translation and Compiling Vol. II.* Prentice-Hall, 1973. 997, 998

[4] I. Bach. *Formális nyelvek (Formal Languages)*. Typotex, 2001. 998

[5] J. Bánkfalvi, Zs. Bánkfalvi, G. Bognár. *A formális nyelvek szintaktikus elemzése*. Közgazdasági és Jogi Kiadó, 1978. 998

[6] K. D. Cooper, L. Torczon. *Engineering a Compiler*. Morgan Kaufman Publisher, 2004. 998

[7] P. Dömösi, A. Fazekas, G. Horváth, Z. Mecsei. Formális nyelvek és automaták (Formal Languages and Automaton). Digitális kézirat, 2004. 998

[8] C. N. Fischer, R. LeBlanc (szerkesztők). *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, 1988. 998

[9] Z. Fülöp. *Formális nyelvek és szintaktikus elemzésük (Formal Languages and their Syntactical Analysis)*. Polygon, 2004 (second edition). 998

[10] D. Gries. *Compiler Construction for Digital Computers*. John Wiley & Sons, 1971. 998

[11] D. Grune, H. Bal, C. J. H. Jacobs, K. Langendoen. *Modern Compiler Design*. John Wiley & Sons, 2000. 998

[12] R. W. Hunter. *Compilers, Their Design and Construction using Pascal*. John Wiley & Sons, 1985. 998

[13] K.. Louden. Compilers and interpreters. In A. B. Tucker (szerkesztő), *Handbook of Computer Science*, 99/1–99/30. o. Chapman & Hall/CRC, 2004. 998

[14] R. Mak. *Writing Compilers and Interpreters*. Addison-Wesley, 1991. 998

[15] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman Publisher, 1997. 998

[16] T. Pittman. *The Art of Compiler Design, Theory and Practice*. Prentice Hall, 1992. 998

[17] Gy. Révész. *Bevezetés a formális nyelvek elméletébe (Introduction to the Theory of Formal Languages)*. Akadémiai Kiadó, 1979. 998

[18] J-P. Tremblay, P. G. Sorenson. *Compiler Writing*. McGraw-Hill Book Co., 1985. 998

[19] L. Varga. *Rendszerprogramok elmélete és gyakorlata (Theory and Practice of System Programs)*. Akadémiai Kiadó, 1980. 998

[20] W. Waite, G. Goos. *Compiler Construction*. Springer-Verlag, 1984. 998

Subject Index

Name index

Contents