MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY
"KHARKIV POLYTECHNIC INSTITUTE"

Svetlana Gavrylenko, Viktor Chelak

COMPILER DESIGN THEORY

The course project guide for the students of
123 – "Computer Engineering"
for full-time and part-time education

Затверджено
редакційно-видавничою радою
НТУ «ХПІ»,
протокол № 2 від 27 червня 2024р.
п. 562

Kharkiv
NTU “KhPI”
2024

Reviewer:

V.S. Breslavets, Candidate of Technical Science, Associate Professor, Professor at Information Systems Department, National Technical University “Kharkiv Polytechnic Institute”.

Svitlana Gavrylenko, Viktor Chelak.

Methodical guidelines for the implementation of the course project from the course "Compiler Design Theory" for full-time and part-time students of the specialty "123 Computer Engineering" / Gavrylenko S., Chelak V. – Kharkiv : NTU “KhPI”, 2024. – 26 p.

Department of Computer Engineering and Programming

TABLE OF CONTENTS

Introduction 4

1. Tasks for implementation of the course project 5

1.1. The purpose of the project 5

1.2. Project implementation procedure 5

1.3. Task conditions (individual area) 5

1.4. Requirements for the course project report 13

2. Construction procedure for bottom-up LR(1) parser 14

3. Example of construction of LR (1) parser 15

3.1 Construction of production rules 15

3.2. Detection of non-generating and unreachable symbols 17

3.3. Definition of grammatical occurrences 18

3.4. Finding the function F_FIRST(y) and F_FOLLOW(y) 19

3.5. Construction of go-to table 20

3.6. Construction of action table 21

3.7. The algorithm of the LR (1) parser 23

3.8. Parser simulation example 24

Bibliography 25

INTRODUCTION

Researchers studying the appearance of intelligence on our planet believe that the appearance of language played a decisive role in its development, which allowed not only to express and store knowledge, but also to exchange it.

With the creation of computers, there was a need to communicate with such devices, as it turned out to be necessary to give them orders, tasks, and a description of the work that they should perform. For this purpose, special languages began to be developed, which came to be called artificial, in contrast to the natural languages of human communication. Artificial languages should be, on the one hand, convenient and understandable for humans, and on the other hand, they should be perceived by devices. The combination of these requirements in one language turned out to be a difficult task, so there were tools for converting texts from a language understandable to a person to the language of a device. Such tools are called **translators**.

The translator can be of the **interpreting** or **compiling** type. In the first case, it is called an **interpreter of the input language**, and in the second - a **compiler**.

The interpreter sequentially reads the input language propositions, analyzes them and immediately executes them, while the compiler does not execute the language propositions, but builds a program that can later be run to obtain the result.

The input of the compiler is a text written in an input language understood by a person, and the result of the compiler's work is a text in a language understood by the device.

These methodological instructions consider the construction of a syntactic LR parser, which is one of the stages of the compiler. It is at the stage of syntactic analysis that the largest number of errors in the text of the program is revealed.

1. TASKS FOR IMPLEMENTATION OF THE COURSE PROJECT

1.1. The purpose of the project

Construction of a syntax LR parser for a given program fragment.

1.2. Project implementation procedure

1.2.1. Depending on your last name number from the group list, select the area of the individual task (paragraph 1.3). Agree with the Lecturer on a fragment of the program for analysis according to the individual task.

1.2.2. Build production rules. Check that the grammar does not contain non-generating and unreachable symbols.

1.2.3. Construct the set F_FIRST, F_FOLLOW and, if necessary, the function FOLLOW.

1.2.4. Build a Go-to table.

1.2.5. Build an Action table.

1.2.6. Check the work of the parser experimentally.

1.2.7. Write and configure the text of the parser program. The programming language and environment is chosen by the student independently.

1.2.8. Create a convenient interface for the program.

1.2.9. Check the operation of the program on an example.

1.2.10. Make a course project report (content of this report are shown in paragraph 1.4).

1.3. Task conditions (Individual area)

1. Integer variables. Input operator.

```
e.g. 1.
      int i = int_constant ;
      cin >> i ;
e.g. 2.
      int i = int_constant , i = int_constant ;
      cin >> i >> i;
e.g. 3.
      int i = int_constant , i = int_constant ;
      cin >> i >> i >> i;
```

2. Character variables. Output operator.

```
e.g. 1.
      char i = char_constant ;
      cout << i ;
e.g. 2.
      char i = char_constant , i = char_constant ;
      cout << i << i;
e.g. 3.
      char i = char_constant , i = char_constant ;
      cout << i << i << i;
```

3. Integer arrays.

```
e.g. 1.
      int i[d] = {d, d, d, d, d, d, d};
e.g. 2.
      int i[] = {};
e.g. 3.
      int i[d] = {d, d, d};
```

4. Loop statement with parameter.

```
e.g. 1.
      for ( type i = expr ; expr ; expr )
          stmt ;
e.g. 2.
      for ( type i = expr ; expr ; expr ) {
          stmt ;
          stmt ;
      }
e.g. 3.
      for ( type i = expr ; expr ; expr ) {
          stmt ;
          stmt ;
          stmt ;
      }
```

5. Loop statement with precondition.

```
e.g. 1.
      while ( expr )
          stmt ;
e.g. 2.
      while ( expr ) {
          stmt ;
          stmt ;
          stmt ;
      }
e.g. 3.
      while ( expr ) {
          stmt ;
          stmt ;
          stmt ;
      }
```

6. Loop statement with postcondition.

```
e.g. 1.
      do
          stmt ;
      while ( expr ) ;
```

e.g. 2.

```
do {
    stmt ;
    stmt ;
    stmt ;
} while ( expr ) ;
```

e.g. 3.

```
do {
    stmt ;
    stmt ;
} while ( expr ) ;
```

7. If-else statement.

e.g. 1.

```
if ( expr ){
    stmt ;
}
```

e.g. 2.

```
if ( expr ){
    stmt ;
}
else {
    stmt ;
    stmt ;
    stmt ;
}
```

e.g. 3.

```
if ( expr ){
    stmt ;
    stmt ;
}
else if ( expr ){
    stmt ;
    stmt ;
}
else if ( expr ){
    stmt ;
}
else {
    stmt ;
    stmt ;
    stmt ;
}
```

8. Switch statement.

e.g. 1.

```
switch ( expr ) {
    case constant :
        stmt ;
        stmt ;
}
```

e.g. 2.

```
switch ( expr ) {
    default :
        stmt ;
}
```

e.g. 3.

```
switch ( expr ) {
    case constant :
        stmt ;
        stmt ;
        stmt ;
    case constant :
        stmt ;
        stmt ;
    default :
        stmt ;
}
```

9. Assignment operator with arithmetic expression consisting of identifiers, "+","-","*","/" operators, parentheses.

e.g. 1.

```
i = i + ( i - i ) ;
```

e.g. 2.

```
i = i * ( i + ( i / i ) - i ) ;
```

e.g. 3.

```
i = i + ( i + i ) ;
```

10. Assignment operator with arithmetic expression consisting of identifiers, "+" operator, parentheses, and mathematical functions.

e.g. 1.

```
i = i + ( i + i ) ;
```

e.g. 2.

```
i = sin ( i ) ;
```

e.g. 3.

```
i = sqrt ( sin ( i ) + cos ( i ) ) ;
```

11. Arrays of pointers.

e.g. 1.

```
type* i[d];
```

e.g. 2.

```
type*** i[d];
type** i[d];
type*** i[d];
```

e.g. 3.

```
type** i[d];
type* i[d];
type*** i[d];
type* i[d];
```

12. Real number variables.

e.g. 1.

```
float i = real_constant ;
```

e.g. 2.

```
double i = real_constant ;
```

e.g. 3.

```
float i = real_constant, i = real_constant ;
```

13. Strings. Input string operator.

e.g. 1.
```
string i, i, i, i;
getline(cin, i);
getline(cin, i);
getline(cin, i);
```
e.g. 2.
```
string i;
getline(cin, i);
```
e.g. 3.
```
string i, i;
getline(cin, i);
getline(cin, i);
```

14. Strings. Built-in string functions.

e.g. 1.
```
const char * i = string_constant ;
strlen ( i ) ;
```
e.g. 2.
```
const char * i = string_constant , * i = string_constant ;
strcmp ( i , i ) ;
```
e.g. 3.
```
const char * i = string_constant , * i = string_constant ;
strlen ( i ) ;
strcmp ( i , i ) ;
strlen ( i ) ;
strlen ( i ) ;
```

15. Functions.

e.g. 1.
```
type i(type i, type i) {
   stmt ;
}
```
e.g. 2.
```
type i(type i, type i, type i) {
   stmt ;
   stmt ;
   stmt ;
   stmt ;
}
```
e.g. 3.
```
type i( ) {
   stmt ;
   stmt ;
}
```

16. Structs (records).

e.g. 1.
```
struct type {
   type i ;
} ;
```
e.g. 2.
```
struct type {
   type i ;
   type i ;
} ;
```

e.g. 3.
```
struct type {
   type i ;
   type i ;
   type i ;
} ;
```

17. Files. Input and ouput data in file.

e.g. 1.
```
ifstream in;
in.open(i);
in >> i >> i >> i;
in.close();
```
e.g. 2.
```
ofstream out;
out.open(i);
out << i << i << i;
out.close();
```
e.g. 3.
```
ifstream in;
in.open(i);
in >> i;
in.close();
```

18. Files. File functions.

e.g. 1.
```
FILE * i = fopen ( string_constant , string_constant ) ;
fclose ( i ) ;
```
e.g. 2.
```
FILE * i = fopen ( string_constant , string_constant ) ;
fputs ( i , string_constant ) ;
fgets ( i , i , i ) ;
fclose ( i ) ;
```
e.g. 3.
```
FILE * i = fopen ( string_constant , string_constant ) ;
fputs ( i , string_constant ) ;
fputs ( i , string_constant ) ;
fputs ( i , string_constant ) ;
fclose ( i ) ;
```

19. Singly linked list.

e.g. 1.
```
struct node {
   type i ;
   struct node * next ;
};
```
e.g. 2.
```
struct node {
   struct node * next ;
   type i ;
};
```
e.g. 3.
```
struct node {
   type i ;
   type i ;
   struct node * next ;
   type i ;
};
```

20. Doubly linked list.

e.g. 1.
```
struct node {
   type i ;
   struct node * prev , * next ;
};
```
e.g. 2.
```
struct node {
   struct node * prev , * next ;
   type i ;
};
```
e.g. 3.
```
struct node {
   type i ;
   type i ;
   struct node * prev , * next ;
   type i ;
};
```

21. Directives for connecting system and user modules.

e.g. 1.
```
#include<name>
#include"name.h"
#include<name>
#include<name>
```
e.g. 2.
```
#include"name.h"
#include"name.h"
#include<name>
#include"name.h"
```
e.g. 3.
```
#include<name>
```

22. Objects (classes).

e.g. 1.
```
class type {
} ;
```
e.g. 2.
```
class type {
    type i ;
    type i ;
} ;
```
e.g. 3.
```
class type {
    type i ;
    type i ;
    type i ;
    type i ;
} ;
```

23. Objects (classes). Inheritance.

e.g. 1.
```
class type {};
class type : type : type {};
```
e.g. 2.
```
class type {};
class type : type {};
class type : type {};
class type : type {};
```

e.g. 3.
```
class type {};
class type : type {};
```

24. Constants.

e.g. 1.
```
const type i = expr ;
```
e.g. 2.
```
const type i = expr , i = expr ;
```
e.g. 3.
```
const type i = expr , i = expr , i = expr ;
```

25. Objects (classes). Polymorphism.

e.g. 1.
```
class_type i = (class_type)i;
class_type i = (class_type)(class_type)i;
```
e.g. 2.
```
class_type i = (class_type)(class_type)(class_type)i;
```
e.g. 3.
```
class_type i = (class_type)i;
class_type i = (class_type)i;
class_type i = (class_type)i;
```

26. Graphics. The use of graphics functions.

e.g. 1.
```
LineTo ( i , i , i );
```
e.g. 2.
```
Rectangle ( i , i , i , i , i );
Ellipse ( i , i , i , i , i );
```
e.g. 3.
```
LineTo ( i , i , i );
LineTo ( i , i , i );
Ellipse ( i , i , i , i , i );
Rectangle ( i , i , i , i , i );
Ellipse ( i , i , i , i , i );
Rectangle ( i , i , i , i , i );
LineTo ( i , i , i );
```

27. Unary and binary operations.

e.g. 1.
```
i + i + i + i ^ i + ~i;
```
e.g. 2.
```
-i + -i - i / ~i;
```
e.g. 3.
```
~i | i | i & i + -i * i / i;
```

28. Math functions.

e.g. 1.
```
sin ( i ) ;
```
e.g. 2.
```
cos ( i ) ;
sin ( i ) ;
cos ( i ) ;
```

```
e.g. 3.
      sin ( i ) ;
      cos ( i ) ;
      atan2 ( i , i );
      sqrt ( i ) ;
      pow ( i , i );
```

29. Arrays of symbols.

```
e.g. 1.
      char i[d] = "sssssss";
e.g. 2.
      char i[d] = {'s', 's', 's', 's', '\0'};
e.g. 3.
      char i[d] = {'s', '\0'};
      char i[d] = "ss";
      char i[d] = {'\0'};
      char i[d] = "sss";
```

30. Output formatting.

```
e.g. 1.
      cout << fixed << i ;
e.g. 2.
      cout << fixed << i << i ;
e.g. 3.
      cout << scientific << i << fixed << i ;
```

1.4. Requirements for the course project report

Course project report should have the followed chapters:

- Title page (1 page);

- Table of Contest (1 page);

- Introduction (1 page);

- LR parser design (5-10 pages, tables, productions, description all what you did to complete 1.2.1-1.2.6)

- Development of the LR parser simulator (5 pages. Describe the key points when creating a software application. Specify the main problems that arose. List what additional data structures you used, your decisions in terms of input format, how you mark them, etc.)

- Source code section (unlimited);

- Conclusions (1 page);

2. CONSTRUCTION PROCEDURE FOR BOTTOM-UP LR(1) PARSER

1. Build production rules for a given program fragment.

2. Determine whether the grammar contains non-generating and unreachable symbols. If such symbols are present, they must be removed from the grammar rules.

3. Change the rules using grammatical occurrences.

4. Find for the resulting grammar the functions F_FIRST, F_FOLLOW and FOLLOW (if there are empty production rules).

5. Construct a Go-to table that has one column for each grammatical symbol and one row for each grammatical occurrence and bottom marker h0. If the table constructed in step 5 is non-deterministic, then it must be converted into a deterministic table. The states obtained in this table should be used as store symbols (if available). Empty states are treated as forbidden.

6. Construct an Action table. Empty table states are considered forbidden. If as a result of the execution it is not possible to build the Action table, then the given grammar is not an LR(0) or LR(1) grammar and it is impossible to build a parser for it. In this case, it is necessary to build other rules of grammar or construct another, more complex parser.

7. Check the parser operation using the given program fragment.

8. Write the source code that simulates the parser prosses.

9. Create a convenient program interface.

3. EXAMPLE OF CONSTRUCTION OF LR(1) PARSER

Task: Build an LR(1) parser to describe an arithmetic expression that contains: identifier i, =, +, (,), ;. Brackets can be nested.

3.1 Construction of production rules

The basis for creating production rules is the method of extracting the structure of a given set of strings. This method involves dismembering the strings included in each set into their parts in such a way as to reveal the repeated parts of the strings and the parts included in all the strings in an unchanged form. This dismemberment into parts is the discovery of the strings structure of a given set.

For each detected element of the structure, we will enter a notation. A set of such notations forms the basis of a dictionary of non-terminal symbols of a certain grammar. The next step of the construction is to identify the sequences in which the elements of the structure can be included in the given chains. Such sequences are the basis for building grammar rules. To show how the structure of chains is reflected in the rules of grammar, consider examples.

- A string consisting of given characters *abc* corresponds to the rule: $I \rightarrow abc$.
- A string starting with a given symbol *a* corresponds to the rule: $I \rightarrow a$A.
- The string ending with the given symbol *a* corresponds to the rule: $I \rightarrow Aa$.
- The rule $I \rightarrow aAb$ corresponds to the string starting and ending with given symbols *a* and *b*.
- A string containing the symbol *a* inside is matched by a rule: $I \rightarrow AaB$.
- A string of a given length $l = 2$ corresponds to the following rules: A$\rightarrow aB$ and $B \rightarrow a$.
- A string consisting of repeating symbols *a* is matched by the rules $A \rightarrow aA$ and $A \rightarrow a$.
- A string consisting of alternating symbols *a* and *b* corresponds to the following rules: $A \rightarrow aB$ and $B \rightarrow bA$.

Consider the construction of grammars for sequences of symbols with separators for lists.

Let us denote the sequence element *a*. The simplest sequence can consist of one element a. All other sequences can be obtained by assigning one more element to the already constructed sequence. If we denote the constructed part of the sequence by the non-terminal

symbol R, and the sequence by the symbol *L*, then we obtain the grammar rules in the following form:

$$L \rightarrow aR \ (1)$$
$$R \rightarrow aR \ (2)$$
$$R \rightarrow \$ \quad (3)$$

In the previous problem, it was assumed that the list L must contain at least one element. If we assume that the set of chains generated by grammar rules can include an empty symbol, then one more rule L→$ must be added to the constructed rules. In this case, the set of rules looks like this:

$$L \rightarrow aR \ (1)$$
$$R \rightarrow aR \ (2)$$
$$R \rightarrow \$ \ (3)$$
$$L \rightarrow \$ \ (4)$$

Let us consider the construction of a list, the elements of which should have separators between them. Let us choose a comma as the separator. The simplest list, as in the previous case, consists of one element, and the construction of a list of several elements can be performed by assigning to the already constructed part of the list a separator with a list element. The rules corresponding to this construction are as follows:

$$L \rightarrow aR \ (1)$$
$$R \rightarrow ,aR \ (2)$$
$$R \rightarrow \$ \ (3)$$

If the delimited list can be empty, then the above set of rules must be supplemented with another rule with an empty right part. As a result, we get:

$$L \rightarrow aR \ (1)$$
$$R \rightarrow ,aR \ (2)$$
$$\text{R} \rightarrow \$. \ (3)$$
$$L \rightarrow \$ \ (4)$$

In the general case, if a set of strings representing a certain language is described, and it is necessary to construct a grammar that generates this set of strings, then it should be done as follows:

- ✓ write several examples from a given set of chains;
- ✓ analyze the structure of chains, highlighting the beginning, the end, repeated or symbols from a group of symbols;

- ✓ introduce notation for complex structures consisting of groups of symbols; such notations are non-terminal symbols of the grammar;
- ✓ build rules for each of the selected structures, using recursive rules for the task of repeated structures;
- ✓ combine all rules;
- ✓ check with the help of deductions the possibility of obtaining chains with different structures.

For our task, the grammar looks like this:

G: V_T={i,+,=,(,),;}, V_a={I,A,C}
1. $I \rightarrow i=A;$
2. $A \rightarrow iC$
3. $A \rightarrow (A)C$
4. $C \rightarrow +A$
5. $C \rightarrow \$$

3.2. Detection of non-generating and unreachable symbols

A symbol $x \in V_a$ is called non-generating if no finite terminal sequence can be derived from it.

Considering the rules of grammar, we can conclude that when all the symbols on the right side are productive, then the symbol standing on the left side is also productive. The last statement allows you to organize the procedure for detecting non-generating symbols in the following form:

- Make a list of non-terminal symbols for which there is at least one rule, the right part of which contains terminal symbols or empty ($).
- If such a rule is found and all non-terminal symbols that are on its right side are already listed, then it should be added to the list the non-terminal symbol standing on its left side.
- If in step 2 the list is no longer replenished, then we have obtained a list of all productive non-terminal symbols of the grammar, and all non-terminal symbols that do not fall into it are non-productive.

Let us define non-generating symbols for grammar. In the first step, we enter the symbol C in the list. Then, according to the second rule, we enter the symbol A in the list. In the third step, the symbols I are entered in the list. All non-terminal symbols are included in the list, therefore, the grammar does not contain non-generating symbols.

Let us determine whether the grammar G contains unreachable symbols.

A symbol $x \in V_t \cup V_a$ is called unreachable in the Context-free grammar G if x does not appear in any derived sequence.

Looking at the rules of the grammar, you can see that if the non-terminal symbol on the left side of the rule is reachable, then all the symbols on the right side are reachable. This property of the rules is the basis of the unreachable character detection procedure, which can be described as follows:

- Create a one-element list consisting of the initial symbol of grammar I.
- If a rule is found, the left part of which is already in the list, then include in the list all symbols that are contained in its right part.
- If in step 2 no more new non-terminal symbols are added to the list, then a list of all reachable non-terminal symbols is obtained, and the rest of the symbols not in the list are unreachable.

Let us define unattainable symbols for grammar G.

In the first step, we add the symbol I to the list. In the second step, we add the symbol A. In the third step, we supplement the list with the symbol C. All non-terminal symbols are included in the list, therefore, the grammar does not contain unreachable symbols.

3.3. Definition of grammatical occurrences

Since each grammatical symbol can be included in a grammar rule one or more times, and can also be included in different rules, depending on the location of grammatical symbols in the rules, different actions can be performed: symbol transfer or convolution, therefore the concept of grammatical occurrence is introduced.

The grammatical occurrence of a grammar symbol is given by the number of the rule and the number of the position of the symbol in this rule. The leftmost symbol is the first. If a character is included in a rule once, it is indicated by the number of the rule. If a symbol

occurs in the grammar only once, then it may not be numbered. The initial symbol of the grammar is denoted by I_0 and is called the initial value.

Let's rewrite the rules of grammar *G*, using grammatical occurrences.

$I \rightarrow i_1 = A_1;$ (1)
$A \rightarrow i_2\ C_2$ (2)
$C \rightarrow + A_3$ (3)
$A \rightarrow (A_4)\ C_4$ (4)
$C \rightarrow \$$ (5)

Note: if a grammar contains empty production rules, there is only LR(1) parser that can work with such grammar.

3.4. Finding the function F_FIRST(Y) and F_FOLLOW(Y)

The functions F_FIRST(*Y*) and F_FOLLOW(*Y*) are used when constructing the ascending recognizer.

The function F_FIRST(*Y*) defines the set of symbols that can be first in the strings derived from *Y*. It includes *Y* itself and all symbols derived from *Y* without empty production rules. The function F_FIRST(*Y*) is also defined for the initial symbol of the grammar *I*.

For example, the function F_FIRST(*Y*) for the first grammar rule would look like

F_FIRST (i_1) ={i_1}

F_FIRST (=) ={=}

F_FIRST (A_1) = {A_1, i_2,(}

F_FIRST (;) = {;}

The F_FOLLOW(Y) function determines the set of characters that can occur immediately after Y in the strings derived from the grammar's initial character.

If the symbol *Z* appears after *Y*, then $\alpha \rightarrow \varphi YZ$, and F_FOLLOW (*Y*)= F_FIRST (*Z*).

In addition, the function F_FOLLOW(h_0) is defined here, which is equal to F_FIRST(I_0), i.e. F_FOLLOW (h_0)= F_FIRST (I_0).

For the given grammar, we construct the F_FOLLOW (*Y*) function.

F_FOLLOW (i_1) = {=};
F_FOLLOW (=) = {A_1, i_2, (};
F_FOLLOW (A_1) = {;};
F_FOLLOW (A_3) = {$};
F_FOLLOW (() = {A_4, i_2, (};
F_FOLLOW (A_4) = {)};

F_FOLLOW (;) = {$}; F_FOLLOW ()) ={$C_4$, +};
F_FOLLOW (i_2) = {C_2, +}; F_FOLLOW (C_4) = {$};
F_FOLLOW (C_2) = {$}; F_FOLLOW ($h_0$) = {$I_0$, i_1};
F_FOLLOW (+) = {A_3, i_2, (}; F_FOLLOW (I_0) = {$};

Let's build the FOLLOW function for the rightmost symbols in all rules:

FOLLOW (A) = {), ;}; FOLLOW (C) = {), ;}; FOLLOW (I) = {$}

3.5. Construction of Go-to table

Using the F_FOLLOW(Y) function, a transition table is built that determines the change in states of the automaton. A Go-to table is used to define the grammatical occurrences that are written to the store. The table is built as follows:

1) each grammatical occurrence corresponds to a row of the table, each grammatical symbol corresponds to a column.

2) the cells of the table are filled with the elements of the F_FOLLOW (Y) function;

3) the element x_k, which belongs to the set of the function F_FOLLOW (Y_j), is entered in the cell located at the intersection of row Y_j and column x;

For example, for a line marked by grammatical occurrence i_1, the function F_FOLLOW(i_1)={=}. So, at the intersection of row i_1 and column = we put an element from the set F_FOLLOW(i_1), i.e. =. We perform similar actions with all lines of the table of transitions (Table 3.1).

Table 3.1 – **Go-to table**

Grammatical Occurrences	Grammatical symbols								
	I	*i*	=	*A*	*C*	+	(	)	;
i_1			=						
=		i_2		A_1			(		
A_1									;
;									
i_2					C_2	+			
C_2									
+		i_2		A_3			(		
A_3									
(		i_2		A_4			(		
A_4								)	
)					C_4	+			
C_4									
h_0	I_0	i_1							
I_0									

It should be noted that when constructing a Go-to table, there may be several grammatical occurrences of the corresponding symbols in the cells. Such a table is non-deterministic and must be made deterministic using the means used to transform tables of finite state machines. As a result, we will get a table in which the rows are marked with sets of grammatical occurrences.

3.6. Construction of Action table

To describe the order of actions of the parser, an Action table is built, in which:

- the letter *S* denotes the Shift operation;
- the letter *R(k)* denotes the Reduce operation, where *k* is the number of the rule;
- the letter *A* denotes the Allow/Accept operation, i.e. the entire input is parsed;
- the letter *E* (or an empty cell) indicates the Error operation, which means that is an error has occurred and further parsing process is impossible.

The table contains rows, which are grammatical occurrences, and columns, which are terminal symbols (characters of the input string). The table also has a ⊥ or $ column that marks the end of the input line.

The table is filled in as follows:

- At the intersection of the line marked with the symbol I_0 and the column marked with the marker ⊥, we mark as operation A.
- If the line is marked by a grammatical occurrence R_{ij}, which is not the rightmost occurrence of any rule, and if the Go-to table element at the intersection of the row R_{ij} and column F is not empty, then the shift operation (S) is fill in the action table at the intersection of the row R_{ij} and column F.
- If the row is marked by the rightmost grammatical occurrence p_{ij}, and there is a production rule A→ α p_{ij} with the number k, then for each input symbol x belonging to the set FOLLOW(A), the Reduce operation ($R(k)$) is entered in the action table (at the intersection of row p_{ij} and column x).
- If the grammar contains an empty production rule A→$, with the number k, then it is necessary to mark the lines for which the F_FOLLOW function contains A_{ij}. At the intersection of the given rows and columns corresponding to the function FOLLOW(A), we apply the Reduce operation $R(k)$.

The Action table constructed for a given grammar is shown in Table 3.2.

Table 3.2 – Action table

Grammatical Occurrences	Terminal symbols						
	i	=	+	(	)	;	⊥
i_1		S					
=	S			S			
A_1						S	
;							R (1)
i_2			S		R (5)	R (5)	
C_2					R (2)	R (2)	
+	S			S			
A_3					R (3)	R (3)	
(	S			S			
A_4					S		
)			S		R (5)	R (5)	
C_4					R (4)	R (4)	
h_0	S						
I_0							A

3.7. The algorithm of the LR (1) parser

The algorithm uses a go-to table and an action table and works as follows:

1. Read the current symbol of the input string *x*.
2. Read the symbol on the top of the stack Y_{K1}.
3. Read the cases of the action table located in row Y_{K1} and column *x*.
4. If the action is Error Action (*Ej*) or Allow action (*A*), then the process should be finished, because the result is obtained.
5. If the action is Shift Action (*S*), then read the Grammatical occurrence Z_{ij} in the Goto table, located in row Y_{K1} and column *x*. Write the Grammatical occurrence Z_{ij} to the stack.
6. If the action is Reduce Action (*R(n)*) in the non-terminal symbol *Z* (for example, *n.* $Z \rightarrow dA_2$), then read the Grammatical occurrence Z_{ij} in the Goto table, located in the column *Z* and the row corresponding to the top symbol of the stack Y_{K1} that does not participate in the

Reduce Action. Accordantly production *n.* $Z \rightarrow dA_2$ we should drop from the stack the right part *(dA2)* of the production rule *n.* $Z \rightarrow dA_2$ and write to the stack symbol Z_{ij}, located in row Y_{K1} and column *Z*.

3.8. Parser simulation example

Let's check the work of the recognizer on the example of recognizing a input string: i=((i+i+i));. An example of work is given in table 3.3.

Table 3.3 – Parser simulation example

Stack	Input String	Action
h_0	$i=((i+i+i));\ \perp$	*S*
$h_0 i_1$	$=((i+i+i));\ \perp$	*S*
$h_0 i_1 =$	$((i+i+i));\ \perp$	*S*
$h_0 i_1 = ($	$(i+i+i));\ \perp$	*S*
$h_0 i_1 = (($	$i+i+i));\ \perp$	*S*
$h_0 i_1 = ((i_2$	$+i+i));\ \perp$	*S*
$h_0 i_1 = ((i_2+$	$i+i));\ \perp$	*S*
$h_0 i_1 = ((i_2+i_2$	$+i));\ \perp$	*S*
$h_0 i_1=((i_2+i_2+$	$i));\ \perp$	*S*
$h_0 i_1=((i_2+i_2+i_2$	$));\ \perp$	*R(5)*
$h_0 i_1=((i_2+i_2+i_2C_2$	$));\ \perp$	*R(2)*
$h_0 i_1=((i_2+i_2+A_3$	$));\ \perp$	*R(3)*
$h_0 i_1=((i_2+i_2C_2$	$));\ \perp$	*R(2)*
$h_0 i_1=((i_2+A_3$	$));\ \perp$	*R(3)*
$h_0 i_1=((i_2C_2$	$));\ \perp$	*R(2)*
$h_0 i_1=((A_4$	$));\ \perp$	*S*
$h_0 i_1=((A_4)$	$);\ \perp$	*R(5)*
$h_0 i_1=((A_4)C_4$	$);\ \perp$	*R(4)*
$h_0 i_1=(A_4$	$);\ \perp$	*S*
$h_0 i_1=(A_4)$	$;\ \perp$	*R(5)*
$h_0 i_1=(A_4)C_4$	$;\ \perp$	*R(4)*
$h_0 i_1=A_1$	$;\ \perp$	*S*
$h_0 i_1=A_1;$	$\perp$	*R(1)*
$h_0 I_0$	$\perp$	*A*

BIBLIOGRAPHY

1. Chapman, Nigel P., LR Parsing: Theory and Practice, Cambridge University Press, 1987. ISBN 0-521-30413-X

2. Pager, D., A Practical General Method for Constructing LR(k) Parsers. Acta Informatica 7, 249 - 268 (1977)

3. "Compiler Construction: Principles and Practice" by Kenneth C. Louden. ISBN 0-534-939724

4. Practical Translators for LR(k) Languages, by Frank DeRemer, MIT PhD dissertation 1969.

The educational electronic publication

Svitlana GAVRYLENKO
Viktor CHELAK

Methodical guidelines

for the implementation of the course project from the course "Compiler Design Theory" for full-time and part-time students of the specialty "123 Computer Engineering"

The work was recommended for publication by N.I. Zapolovskiy

In the author's edition