Compiling Principles
Lab2 Syntax Analyzer
Report

141250017

陈自强

2016/11/12

Table of Contents

1. Motivation/Aim

In chapter4 Syntax Analysis, we learned about how to analyze expressions by using Top-Down Parsing method or Bottom-Up Parsing method. If use Top-Down Parsing method, we can construct a LL(1) Parser which produces a sequence of derivations. We can also use Bottom-Up method, by constructing SLR(1), LR(1) or LALR(1), we can produces a sequence of reductions.

In this lab, we try to imitate the process of constructing a LR(1) and analyze the simple context free grammar.

2. Content description

The Syntax Analyzer is expected to perform the following functions:

1)Constructing the Parsing Table based on the input file

2)Accept a sequence of characters from the users

3)Parse the sequence based on the Parsing Table

4)Output the sequence of derivations

3. Ideas/Methods

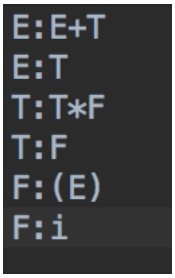
Generally, there are two ways available while syntax analysis, LL or LR.

In the lab, we use LR since it is adaptable to a wilder field.

The process of LR is fairly straight. First, we assume a certain equations as fundamental grammar which is unambiguous and context free. Then we construct the parse table manually. Finally, we program to handle the user input based on the table.

4. Assumptions

- The CFG must be unambiguous and be input like the following pictures.



- The Parse Table must be correspondent to the CFG and the format should like the following picture.

```
0 i,S5 (,S4 E,1 T,2 F,3
1 +,S6 $,r0
2 +,r2 *,S7 ),r2 $,r2
3 +,r4 *,r4 ),r4 $,r4
4 i,S5 (,S4 E,8 T,2 F,3
5 +,r6 *,r6 ),r6 $,r6
6 i,S5 (,S4 T,9 F,3
7 i,S5 (,S4 F,10
8 +,S6 ),S11
9 +,r1 *,S7 ),r1 $,r1
10 +,r3 *,r3 (,r3 $,r3
11 +,r5 *,r5 (,r5 $,r5
```

- User input should be specified in the "input.txt" and must be ended with '$'. A sample input is like the following.

```
i+i*(i+i)+i*i*i$
```

- All the symbols are single characters

5. Description of Important Data Structures

Totally, there are five classes in this Java program.

Class Analyzer is the core of the program.

There are several important data structures in this class.

- `private Map<Pair, Integer> actionTable;`

A simple map storing the function between the Pair(Pair is an inner class represents the symbol and the state) the next action. When the value is 0, it denotes r0, or acceptable. When the value > 0, it refers to shift to the next state n where n is the value. When the value < 0, it refers that it is reducible and the reduction number is the absolute value of the integer.

- `private Map<Pair, Integer> gotoTable;`

Identical with the above map, stores the function between Pair and the next GOTO state.

- `private List<Reduction> reductions;`

Stores the reductions of the CFG.

- `Stack<Character> symbolStack = new Stack<>();`
- `Stack<Integer> stateStack = new Stack<>();`

The symbol stack stores the symbols and the states.

6. Description of core Algorithms

In this experiment, to fully test the LR(1) Syntax Analyzer, I construct three CFGs and their Predictive Parsing Tables:

Context Free Grammar1:

S:E
E:E+T
E:T
T:T*F
T:F
F:(E)
F:i

Corresponding Predictive Parsing Table:

state	ACTION						GOTO		
	i	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				accept			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Context Free Grammar2:

```
S:A
A:CC
C:cC
C:d
```

Corresponding Predictive Parsing Table:

State	Action			GOTO	
	c	d	$	A	C
0	S3	S4		1	2
1			r0		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Context Free Grammar 3: (Ambiguous Grammar)

Since it is an ambiguous grammar, we stipulate that:

1. To solve the precedence problem, we assume that: Priority: if > ';' > else
2. We assume that ';' is left associative
3. To solve the "Dangling Else", we assume that else is corresponded the most adjacent 'if'

```
E:S
S:iSeS
S:iS
S:S;S
S:a
```

Predictive Parsing Table:

State	Action					GOTO
	if	else	;	a	$	S
0	S2			S3		1
1			S4		r0	
2	S6			S7		5
3			r4		r4	
4	S2			S3		8
5		S9	r2		r2	
6	S6			S7		11
7		r4	r4		r4	
8			r3		r3	
9	S2			S3		12
10	S6			S7		13
11		S14	r2		r2	
12			S4		r1	
13		r3	r3		r3	
14	S6			S7		15
15		r1	S10		r1	

Since much preparation work has been done before the algorithm, the algorithm itself is fairly simple.

```
// Stack initialization
symbolStack.push('$');
stateStack.push(0);


int state;
Pair tempPair;
Reduction reduction;
char currentChar = userInput[0];
for (int i = 0;;) {


    state = stateStack.peek();
    if (symbolStack.size() > stateStack.size()) {
        tempPair = new Pair(state, symbolStack.peek());
    } else {
        tempPair = new Pair(state, currentChar);
    }


    if (actionTable.containsKey(tempPair)) {

        int next = actionTable.get(tempPair);
        if (next == 0) {              // r0 acceptable
            break;
        } else if (next > 0) {        // shift
            symbolStack.push(currentChar);
            stateStack.push(next);
            currentChar = userInput[++i];
        } else {                      // reducible

            reduction = reductions.get( -next - 1 );
            reductionsSeq.add(reduction);

            int redLen = reduction.getLength();
            for (int j = 0; j < redLen; j++) {
                symbolStack.pop();
                stateStack.pop();


            }
            symbolStack.push(reduction.nonTerminal);



        }


    } else if (gotoTable.containsKey(tempPair)) {
        stateStack.push(gotoTable.get(tempPair));


    }
}
```

Just based on the two stacks and the current char of the user input to judge what to do by searching the parse table.

7. Use cases on running

Test case1:

CFG&PPT are based on the first one mentioned in part6:

Input:

```
i+i*(i+i)+i*i*i$
```

Output:

```
========== Test Case 1 ==========
F -> i
T -> F
E -> T
F -> i
T -> F
F -> i
T -> F
E -> T
F -> i
T -> F
E -> E+T
F -> (E)
T -> T*F
E -> E+T
F -> i
T -> F
F -> i
T -> T*F
F -> i
T -> T*F
E -> E+T
S -> E
```

Test case2:

CFG&PPT are based on the second one mentioned in part6:

Input:

```
cccdccccccccd$
```

Output:

```
========== Test Case 2 ==========
C -> d
C -> cC
C -> cC
C -> cC
C -> d
C -> cC
C -> cC
C -> cC
C -> cC
C -> cC
C -> cC
C -> cC
A -> CC
S -> A
```

Test case3:

CFG&PPT are based on the third one mentioned in part6:

Input:

Output:

```
========== Test Case 3 ==========
S -> a
S -> a
S -> a
S -> a
S -> iS
S -> S;S
S -> a
S -> a
S -> iS
S -> iSeS
S -> iS
S -> S;S
S -> iSeS
S -> S;S
E -> S
```

8. Problems occurred and related solutions

Since the lab is rather easy relatively, I did not meet any arduous problems during the process. Just a bit error-prone during constructing the parse table manually and transform it into the .txt file.

9. Your feelings and comments

I like coding rather than doing the paper exercise! The feeling of constructing a Syntax Analyzer is gorgeous even it is basic and incomplete.