Syntax Parser Report

121250156

Wang Tianyu

2014.11.19

A. Aiming

In the last lab, we finished programming a lexical parser. And now we have got these tokens and want to analyse the grammar rules in the context. So we build this syntax parser to find out this problem.

B. Content Description

Build a syntax parser to check the context grammar.

C. Ideas

We use LL(1) to create the predictive parsing table and use stacks to simulate the process.

D. Assumptions

We assume that this program can recognise language defined by last lab description. And in order to avoid ambiguous grammar, omit 'else' clause.

E. Related FA Descriptions

I defined the following key words as tokens:

void, int, while, double, if, else(omit), return, +(omit), -(omit), *(omit), /(omit), >, <, =, >=, <=, !=, ;, ,, ., (,), [(omit),](omit), {, }.

F. Description of Important Data Structure

Use stack to simulate this process.

Use PPTElement and PPT to construct the PPT defined below.

G. Description of Core Algorithms

1. Define productions:

No.	Production	No.	Production
1	Program → FuncBlock	10	S → PickValue
2	FuncBlock → void FuncName (Paras) { Statements }	11	PickValue → id = num ;
3	FuncBlock → DataType FuncName (Paras) { Statements ReturnClause }	12	S → if (Condition) Block
4	DataType → int | double	13	Block → { Statements } | S ;
5	FuncName → id	14	Condition → Keyword CompareOP Keyword
6	Paras → ε | Para	15	Keyword → id | num
7	Para → DataType id	16	CompareOP → > | >= | < | <= | == | !=
8	Statements → S ; Statements | ε	17	S → while (Condition) Block
9	S → DataType PickValue	18	ReturnClause → return Keyword ;

2. Calculate First(E) & Follow(E)

E	First(E)	Follow(E)
Program	{void, int, double}	{$\$_R$}
FuncBlock	{void, int, double}	{$\$_R$}
DataType	{int, double}	{id}
FuncName	{id}	{(}
Paras	{int, double, ε}	{)}
Para	{int, double}	{,,)}
Statements	{int, double, id, if, while, ε}	{}}
PickValue	{id}	{int, double, id, if, while}
Condition	{id, num}	{)}
Block	{{, int, double, id, if, while}	{int, double, id, if, while}
S	{int, double, id, if, while}	{int, double, id, if, while, ;}
Keyword	{id, num}	{>, >=, <, <=, ==, !=,)}
CompareOP	{>, >=, <, <=, ==, !=}	{id, num}
ReturnClause	{return}	{}}

3. Construct predictive parsing table

Non-terminal	Input Symbol				
	void	int	double	id	num
Program	Program → FuncBlock	Program → FuncBlock	Program → FuncBlock		
FuncBlock	FuncBlock → void FuncName (Paras) { Statements }	FuncBlock → DataType FuncName (Paras) { Statements ReturnClause }	FuncBlock → DataType FuncName (Paras) { Statements ReturnClause }		
DataType		DataType → int l double	DataType → int l double		
FuncName				FuncName → id	
Paras		Paras → ε l Para	Paras → ε l Para		
Para		Para → DataType id	Para → DataType id		
Statements		Statements → S ; Statements l ε	Statements → S ; Statements l ε	Statements → S ; Statements l ε	
PickValue				PickValue → id = num ;	
Condition				Condition → Keyword CompareOP Keyword	Condition → Keyword CompareOP Keyword
Block		Block → { Statements } l S ;	Block → { Statements } l S ;	Block → { Statements } l S ;	
S		S → DataType PickValue	S → DataType PickValue	S → PickValue	
Keyword				Keyword → id l num	Keyword → id l num
CompareOP					
ReturnClause					

Non-terminal	Input Symbol				
	if	while	return	,	;
Program					
FuncBlock					
DataType					
FuncName					
Paras					
Para					
Statements	Statements → S ; Statements I ε	Statements → S ; Statements I ε			
PickValue					
Condition					
Block	Block → { Statements } I S ;	Block → { Statements } I S ;			
S	S → if (Condition) Block	S → while (Condition) Block			
Keyword					
CompareOP					
ReturnClause			ReturnClause → return Keyword ;		

<table>
<tr><th rowspan="2">Non-terminal</th><th colspan="6">Input Symbol</th></tr>
<tr><th>></th><th>>=</th><th><</th><th><=</th><th>==</th><th>!=</th></tr>
<tr><td>Program</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>FuncBlock</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>DataType</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>FuncName</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Paras</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Para</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Statements</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>PickValue</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Condition</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Block</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>S</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Keyword</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>CompareOP</td><td>CompareOP → > | >= | < | <= | == | !=</td><td>CompareOP → > | >= | < | <= | == | !=</td><td>CompareOP → > | >= | < | <= | == | !=</td><td>CompareOP → > | >= | < | <= | == | !=</td><td>CompareOP → > | >= | < | <= | == | !=</td><td>CompareOP → > | >= | < | <= | == | !=</td></tr>
<tr><td>ReturnClause</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Non-terminal	Input Symbol				
	(	)	{	}	$_R$
Program					Program → ε
FuncBlock					FuncBlock → ε
DataType					
FuncName					
Paras				Paras → ε	
Para					
Statements				Statements → ε	
PickValue					
Condition					
Block			Block → { Statements } l S ;		
S					
Keyword					
CompareOP					
ReturnClause					

4. Algorithm(Defined in Figure 4.20. The Dragon Book, pp 227)

H. Use Cases on Running

I used the program.txt to check the correctness.

- TestCase:

```
int main(int x) {
    if (x != 4) {
        while (3 >= 2){
            y = 2.5;
        }
    }
    return x;
}
```

- Process:

-------TAKE 0-------

Stack: $ Program

Pointer: int

-------TAKE 1-------

Stack: $

Pointer: int

Output: FuncBlock

-------TAKE 2-------

Stack: $

Pointer: int

Output: rightbrace ReturnClause Statements leftbrace rightparen Paras leftparen FuncName DataType

-------TAKE 3-------

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen Paras leftparen FuncName

Pointer: int

Output: int

-------TAKE 4-------

Match: int

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen Paras leftparen FuncName

Pointer: id

-------TAKE 5-------

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen Paras leftparen

Pointer: id

Output: id

-------TAKE 6-------

Match: id

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen Paras leftparen

Pointer: leftparen

-------TAKE 7-------

Match: leftparen

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen Paras

Pointer: int

-------TAKE 8-------

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen

Pointer: int

Output: Para

-------TAKE 9-------

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen

Pointer: int

Output: id DataType

-------TAKE 10-------

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen id

Pointer: int

Output: int

-------TAKE 11-------

Match: int

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen id

Pointer: id

-------TAKE 12-------

Match: id

Stack: $ rightbrace ReturnClause Statements leftbrace rightparen

Pointer: rightparen

-------TAKE 13-------

Match: rightparen

Stack: $ rightbrace ReturnClause Statements leftbrace

Pointer: leftbrace

-------TAKE 14-------

Match: leftbrace

Stack: $ rightbrace ReturnClause Statements

Pointer: if

-------TAKE 15-------

Stack: $ rightbrace ReturnClause

Pointer: if

Output: S

-------TAKE 16-------

Stack: $ rightbrace ReturnClause

Pointer: if

Output: Block rightparen Condition leftparen if

-------TAKE 17-------

Match: if

Stack: $ rightbrace ReturnClause Block rightparen Condition leftparen

Pointer: leftparen

-------TAKE 18-------

Match: leftparen

Stack: $ rightbrace ReturnClause Block rightparen Condition

Pointer: id

-------TAKE 19-------

Stack: $ rightbrace ReturnClause Block rightparen

Pointer: id

Output: Keyword CompareOp Keyword

-------TAKE 20-------

Stack: $ rightbrace ReturnClause Block rightparen Keyword CompareOp

Pointer: id

Output: id

-------TAKE 21-------

Match: id

Stack: $ rightbrace ReturnClause Block rightparen Keyword CompareOp

Pointer: notequal

-------TAKE 22-------

Stack: $ rightbrace ReturnClause Block rightparen Keyword

Pointer: notequal

Output: notequal

-------TAKE 23-------

Match: notequal

Stack: $ rightbrace ReturnClause Block rightparen Keyword

Pointer: num

-------TAKE 24-------

Stack: $ rightbrace ReturnClause Block rightparen

Pointer: num

Output: num

-------TAKE 25-------

Match: num

Stack: $ rightbrace ReturnClause Block rightparen

Pointer: rightparen

-------TAKE 26-------

Match: rightparen

Stack: $ rightbrace ReturnClause Block

Pointer: leftbrace

-------TAKE 27-------

Stack: $ rightbrace ReturnClause

Pointer: leftbrace

Output: rightbrace Statements leftbrace

-------TAKE 28-------

Match: leftbrace

Stack: $ rightbrace ReturnClause rightbrace Statements

Pointer: while

-------TAKE 29-------

Stack: $ rightbrace ReturnClause rightbrace

Pointer: while

Output: S

-------TAKE 30-------

Stack: $ rightbrace ReturnClause rightbrace

Pointer: while

Output: Block rightparen Condition leftparen while

-------TAKE 31-------

Match: while

Stack: $ rightbrace ReturnClause rightbrace Block rightparen Condition leftparen

Pointer: leftparen

-------TAKE 32-------

Match: leftparen

Stack: $ rightbrace ReturnClause rightbrace Block rightparen Condition

Pointer: num

-------TAKE 33-------

Stack: $ rightbrace ReturnClause rightbrace Block rightparen

Pointer: num

Output: Keyword CompareOp Keyword

-------TAKE 34-------

Stack: $ rightbrace ReturnClause rightbrace Block rightparen Keyword CompareOp

Pointer: num

Output: num

-------TAKE 35-------

Match: num

Stack: $ rightbrace ReturnClause rightbrace Block rightparen Keyword CompareOp

Pointer: greaterofequal

-------TAKE 36-------

Stack: $ rightbrace ReturnClause rightbrace Block rightparen Keyword

Pointer: greaterofequal

Output: greaterofequal

-------TAKE 37-------

Match: greaterofequal

Stack: $ rightbrace ReturnClause rightbrace Block rightparen Keyword

Pointer: num

-------TAKE 38-------

Stack: $ rightbrace ReturnClause rightbrace Block rightparen

Pointer: num

Output: num

-------TAKE 39-------

Match: num

Stack: $ rightbrace ReturnClause rightbrace Block rightparen

Pointer: rightparen

-------TAKE 40-------

Match: rightparen

Stack: $ rightbrace ReturnClause rightbrace Block

Pointer: leftbrace

-------TAKE 41-------

Stack: $ rightbrace ReturnClause rightbrace

Pointer: leftbrace

Output: rightbrace Statements leftbrace

-------TAKE 42-------

Match: leftbrace

Stack: $ rightbrace ReturnClause rightbrace rightbrace Statements

Pointer: id

-------TAKE 43-------

Stack: $ rightbrace ReturnClause rightbrace rightbrace

Pointer: id

Output: S

-------TAKE 44-------

Stack: $ rightbrace ReturnClause rightbrace rightbrace

Pointer: id

Output: PickValue

-------TAKE 45-------

Stack: $ rightbrace ReturnClause rightbrace rightbrace

Pointer: id

Output: semicolon num assignop id

-------TAKE 46-------

Match: id

Stack: $ rightbrace ReturnClause rightbrace rightbrace semicolon num assignop

Pointer: assignop

-------TAKE 47-------

Match: assignop

Stack: $ rightbrace ReturnClause rightbrace rightbrace semicolon num

Pointer: num

-------TAKE 48-------

Match: num

Stack: $ rightbrace ReturnClause rightbrace rightbrace semicolon

Pointer: semicolon

-------TAKE 49-------

Match: semicolon

Stack: $ rightbrace ReturnClause rightbrace rightbrace

Pointer: rightbrace

-------TAKE 50-------

Match: rightbrace

Stack: $ rightbrace ReturnClause rightbrace

Pointer: rightbrace

-------TAKE 51-------

Match: rightbrace

Stack: $ rightbrace ReturnClause

Pointer: return

-------TAKE 52-------

Stack: $ rightbrace

Pointer: return

Output: semicolon Keyword return

-------TAKE 53-------

Match: return

Stack: $ rightbrace semicolon Keyword

Pointer: id

-------TAKE 54-------

Stack: $ rightbrace semicolon

Pointer: id

Output: id

-------TAKE 55-------

Match: id

Stack: $ rightbrace semicolon

Pointer: semicolon

-------TAKE 56-------

Match: semicolon

Stack: $ rightbrace

Pointer: rightbrace

-------TAKE 57-------

Match: rightbrace

Stack: $

Pointer: $

Complete!

I. Problems Occurred and Related Solution

When I finished this ppt, I found that the 'statements' was an ambiguous grammar. So when I wrote this program, I dismissed this rule. So the main part can only contains one statement. (Sorry for this)

J. Feelings and Comments

Again, this lab is also meaningful to me. It's quite difficult but fun.