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 I double	13	Block → { Statements } I S ;
5	FuncName → id	14	Condition → Keyword CompareOP Keyword
6	Paras → ε I Para	15	Keyword → id I num
7	Para → DataType id	16	CompareOP → > I >= I < I <= I !=
8	Statements → S ; Statements I 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								
Non-terminal	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 I double	DataType → int I double						
FuncName				FuncName → id					
Paras		Paras → ε l Para	Paras → ε l Para						
Para		Para → DataType id	Para → DataType id						
Statements		Statements \rightarrow S ; Statements I ϵ	Statements \rightarrow S ; Statements I ϵ	Statements \rightarrow S ; Statements I ϵ					
PickValue				PickValue → id = num ;					
Condition				Condition → Keyword CompareOP Keyword	Condition → Keyword CompareOP Keyword				
Block		Block → { Statements } IS;	Block → { Statements } IS;	Block → { Statements } IS;					
s		S → DataType PickValue	S → DataType PickValue	S → PickValue					
Keyword				Keyword → id I num	Keyword → id I num				
CompareOP									
ReturnClause									

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

Non towning!	Input Symbol							
Non-terminal	>	>=	<	<=	==	!=		
Program								
FuncBlock								
DataType								
FuncName								
Paras								
Para								
Statements								
PickValue								
Condition								
Block								
s								
Keyword								
CompareOP	>= < <= ==	CompareOP → > >= < <= == !=	CompareOP → > >= < <= == !=	CompareOP → > >= < <= == !=	CompareOP → > >= < <= == !=	CompareOP → > >= < <= == !=		
ReturnClause								

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

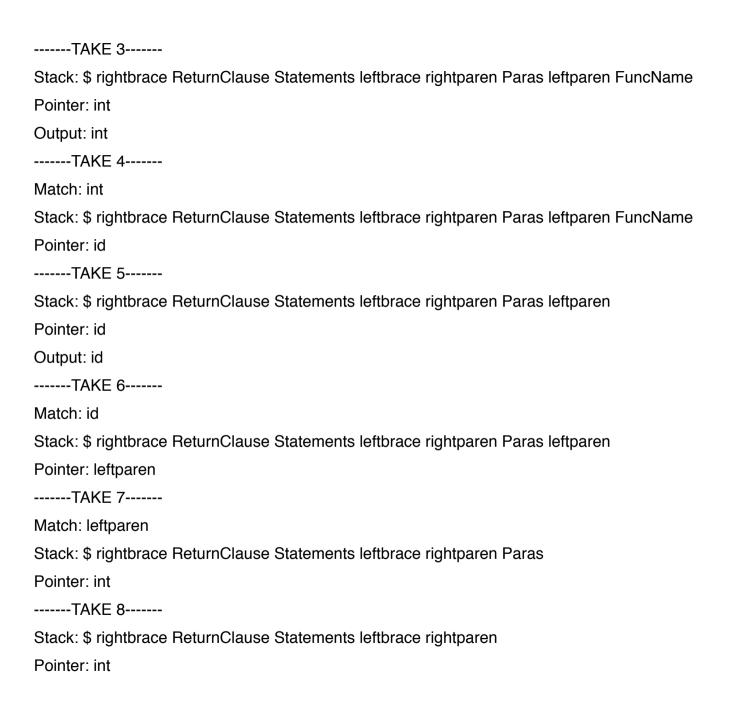
4. Algorithm(Defined in Figure 4.20. The Dragon Book, pp 227)

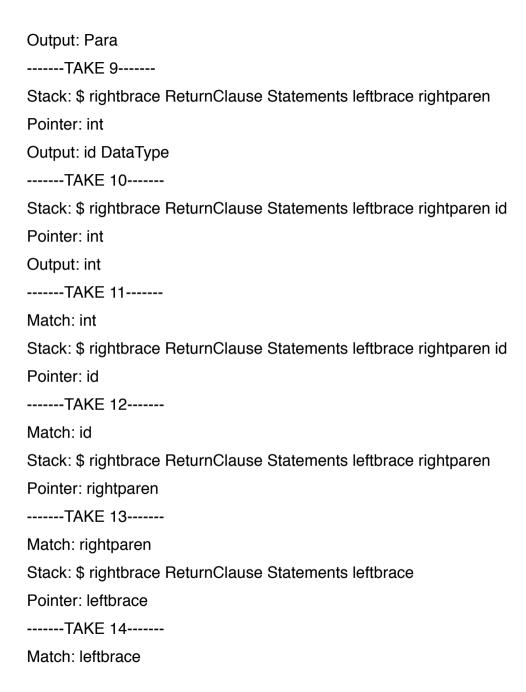
H. Use Cases on Running

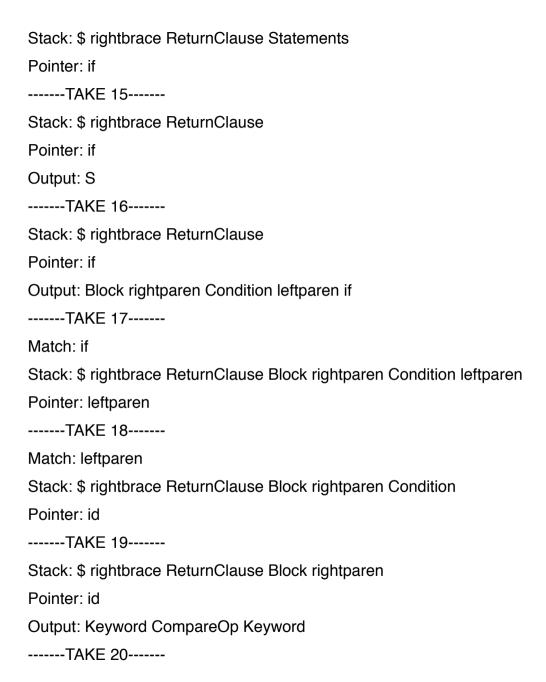
I used the program.txt to check the correctness.

 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







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.