CS323 Documentation

Luc Dang

ID: 888858644

1. **Problem Statement**

   The assignment is to build a syntax analyzer using a top down parser of our choice.

2. **How to use my program**

   From the terminal of a Unix operating system (preferably Ubuntu), run pre-compiled executable file "parser" with following command:

   ./parser.

   If there is any problem with "*parser*" file, compile "*syntax_analyzer_FINAL.cpp*" with following command, then execute again.

   clang++ -std=c++17 syntax_analyzer_FINAL.cpp -o parser

   The program will prompt for input file name for syntax analysis. You must include the file extension in the name in order for the file to be read because the program will not assume .txt for reading. The program will then prompt for the desired name of your output file which the analysis will be printed to. The program will then print a list of tokens and lexemes, as well as the grammar rules for each token, to both the output file and the terminal window.

3. **Design of my program**

   For my program I chose to implement a table driven predictive parser. I enumerated a set of Symbols for each terminal and non-terminal, and used a stack of symbols to parse. Using my table (which is shown below) I matched the symbol at the top of the stack with the current token to find the correct rule to use. (My set of production rules are shown on the next page.) I then pushed the rules backwards onto the stack, and popped it if the token and the symbol at the top of the stack matched.

		0 id	1 =	2 +	3 -	4 *	5 /	6 (	7)	8 ;	9 ,	10 int	11 float	12 bool	13 $	14 if	15 then	16 else	17 endif	18 do	19 while	20 whileend	21 begin	22 end	23 <	24 >
0	**S**	1										2	2	2		21					22		23				
1	**A**	3																									
2	**E**	4						4																			
3	**E'**			5	6				ε	ε					ε		ε			ε					ε	ε
4	**T**	8						8																			
5	**T'**			ε	ε	9	10		ε	ε					ε		ε			ε					ε	ε
6	**F**	12						13																			
7	**D**											14	14	14	ε												
8	**Ty**											16	17	18													
9	**M**									ε	19				ε												
10	**C**	26/27						26/27																			
11	**Ms**									24														ε			
12	**R**																								28	29	

G = (N, T, S, R)		
N = (S, A, E, E', T, T', F, D, Ty, M, C, Ms, R)		
T = (id, =, +, -, *, /, (,), ';' , ',' , int, float, bool, if, then, else, endif, do, while, whileend, begin, end, < , >)		
S = S		
R = {		
(1) S -> A | **(2)** D	First(S) = { id, int, float, bool, if, while, begin }	Follow(S) = { $, else, endif, whileend, ; , end }
(3) A -> id=E;	First(A) = { id }	Follow(A) = { $, else, endif }
(4) E -> TE'	First(E) = { id, (}	Follow(E) = {), ; , $, do, <, >, then }
(5) E' -> +TE' | **(6)** -TE' | **(7)** ε	**First(E') = { +, -, ε }**	**Follow(E') = {), ; , $, do, <, >, then }**
(8) T -> FT'	First(T) = { id, (}	Follow(T) = { +, -,), ; , $, do, <, >, then }
(9) T' -> *FT' | **(10)** /FT' | **(11)** ε	**First(T') = { *, /, ε }**	**Follow(T') = { +, -,), ; , $, do, <, >, then }**
(12) F -> id | **(13)** (E)	First(F) = { id, (}	Follow(F) = { *, /, +, -,), ; , $, do, <, > }
(14) D -> T^{y}idM; | **(15)** ε	**First(D) = { int, float, bool, ε }**	**Follow(D) = { $ }**
(16) T^{y} -> int | **(17)** float | **(18)** bool	First(Ty) = { int, float, bool }	Follow(Ty) = { id }
(19) M -> ,idM | **(20)** ε	**First(M) = { ',' , ε }**	**Follow(M) = { ; , $ }**
(21) S -> if C then S else S endif	First(C) = { id, (}	Follow(C) = { then }
(22) S-> while E do S whileend	**First(Ms) = { ; , ε}**	**Follow(Ms) = { end }**
(23) S -> begin S Ms end	First(R) = { < , > }	Follow(R) = { id, (}
(24) Ms -> ; S Ms | **(25)** ε		
(26) C -> E R E | **(27)** E		
(28) R -> < | **(29)** >		

4. Any Limitations

My program does not optimize memory very well because in order to integrate my syntax analyzer well I had to process all the tokens first, then analyze the syntax. My program is broken down into 2 stages instead of a integrated lexer and syntax parser.

I also chose not to push the End of Stack symbol ($), since it seemed redundant to check both the end of stack and end of string with a "$" when there are functions in C++ that do it anyway.

5. Any Shortcomings

- Program cannot parser if-while grammar very well
- Since lexer did not identify compound operators (>=), the syntax analyzer will treat them as separate tokens and will sometimes throw an error because of it.
- If file is not ended with a separator or operator and reached the end of file, lexer will fail and therefore the parser will fail.