Brian Alexander
Greg Miller
11/17/2019

CPSC 323 Compilers - Parser Documentation

I. Problem Statement

For assignment 2, we created a syntax analyzer using an LL(1) parser. An LL parser is a top-down table driven predictive parser and the 1 specifies that we have one character of look-ahead, namely that we consider only the next terminal when moving through the table. Our parser, given an input of terminals, is able to determine if the input is syntactically correct using our defined grammar.

The problem also required us to use our lexical analyzer developed in project one to parse a string of text into a set of tokens and lexemes which could then be further categorized into the terminal tokens used in our parser. In this way we can take a file containing string input, determine if it contains a set of valid string tokens, and then pass it to our LL parser, which will determine if the tokens are arranged in such a way that we have a valid syntax. If we experience any errors along the way a message will be indicated to the user of lexer and parser, indicating the illegal move that was attempted, the token, lexeme, and line number that were involved.

II. How To Use The Program

Make sure that all files are downloaded and are in a known location. The files that should have been downloaded are Constants.py, lexer.py, ll_parser.py, tests.py, and parser_test_string. Once these are downloaded, using the terminal, navigate to the folder that holds the above files and type python3 ll_parser.py parse_test_string and press enter. To change the input into the parser change the file parser_test_string or pass in an alternative filename. The file tests.py tests the parser without using the lexer. It allows for the direct construction of terminal strings that can be used to test functionality.

III. Program Design

The main files of my program, as mentioned above, are ll_parser.py, lexer.py, and Constants.py. Constants.py houses two important Enum objects used as constants and a string production factory that is used to create the output of rules later in the parser file. The two enums are Terminals and NonTerminals. Each houses their respective properties such as ID and NUM and STATEMENT and MORESTATEMENTS respectively. The member functions of the ProductionFactory can be chained to easily create rule strings printed later in the parser.

Also in the Constants.py file is the main component of the table driven parser is a two-level nested dictionary aptly named **TABLE**, chosen due to the sparse representation of our rules, that is used to get the productions that follow a given production (NonTerminal) and token(Terminal). The primary key is a non-terminal, which when indexed returns another dictionary is then indexed by a terminal. The result from **TABLE** when indexing a non-terminal and terminal (ex: table[NonTerminal][Terminal]) is an index which can be used to get the appropriate production rule for the given combination. Taking that key and indexing the **RULES** dictionary, one receives an array filled with the correct combination of terminals and non-terminals to represent that rule. This array will later be pushed onto the stack in reverse to keep traversing the tokens.

Some changes needed to be made to our lexer from project one for it to be compatible with the desired functionality of the parser. A simple change was the addition of keywords such as begin and end. The most major change was what was returned by the lexer. The token types output by the lexer did not have a one to one match with the parser. For example, while the lexer might output OPERATOR, it is essential for the parser to know which kind of operator it is receiving. Thus a new function was added in the parser called token_to_terminal which processed the tokens from the lexer into their proper terminals for the parser. It is at this point that we also pack the actual string representation for the Terminal and the line number where it occurs, which is essential for error handling.

Also in ll_parser.py is the function **syntactic_analysis** takes as input a list of three-element tuples that represent the token, lexeme, and terminal that are generated in lexer.py. This function uses a stack to keep track of the productions and the top of the stack will be the next production to be processed and as long as something is in the stack the while loop will keep iterating. In the while loop we will check if the top of the stack matches any of the terminal/non-terminals and if it does then we enter that statement. If we match a terminal but the terminal does not match the following production then we print out a syntax error, along with the token, lexeme, line number, and expected terminal. Although once the top of the stack matches the terminal (end of file) then we know that the input is accepted. If we match a non-terminal then we will first check our positions and if they do not match then we will print the token, lexeme from our list of rules. Otherwise, we will get the rule from the table and print the production as well as append the productions to our stack in reverse.

IV. Limitations

None

V. Shortcomings

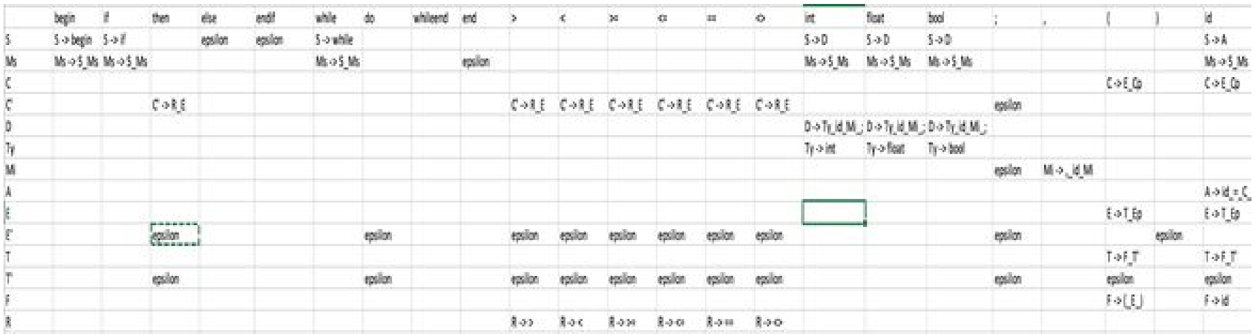
The first shortcoming is and isn’t a shortcoming of project 2. It should be noted, however, that if we are required to use the lexer from project one, it is the case that our lexer-parser system will only be able to handle tokens generated by the lexer, which does not include all of the tokens that the parser can actually handle. It is possible to manually create an input list of tuples manually that allows the parser to handle all of the listed tokens in the grammar.

During some editing of the grammar, specifically C -> ERE | E, we refactored it and ended up in a state where we can no longer have conditional statements wrapped in parenthesis. The resulting factorization was C -> EC’ and C’ -> RE | epsilon. This only occurred to us at our final testing phase and thus would have been a very large rework to implement. A way to solve this problem and expand the capabilities of the language would have been to adjust the grammar as follows:

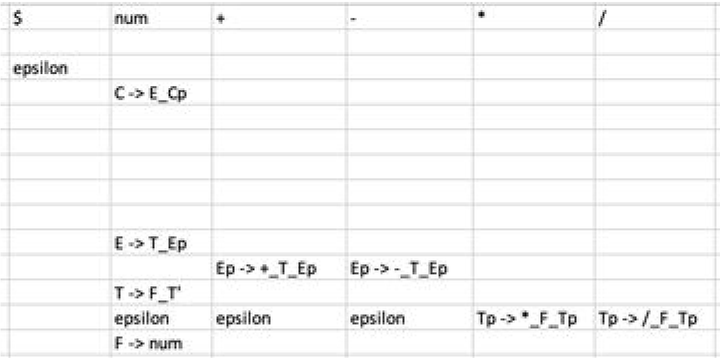
Q -> EQ’	E-> TE’
Q’ -> >Z | <Z | ==EQ’ | <>EQ’ | epsilon	E-> +TE’ | -TE’ | epsilon
Z -> ED’ | =ED’	T-> FT’
F-> (D) | num | id	T’ -> *FT’ | /FT’ | epsilon

This above grammar adjustment allows for conditional statements to be included in the general concept of an expression. This would allow for (5 > 2) + 6, which would result in 7 when (5 > 2) resolves to a positive 1 (boolean true value). This would therefore allow for complex nested boolean evaluations with complex expressions.

VI. Image Reference

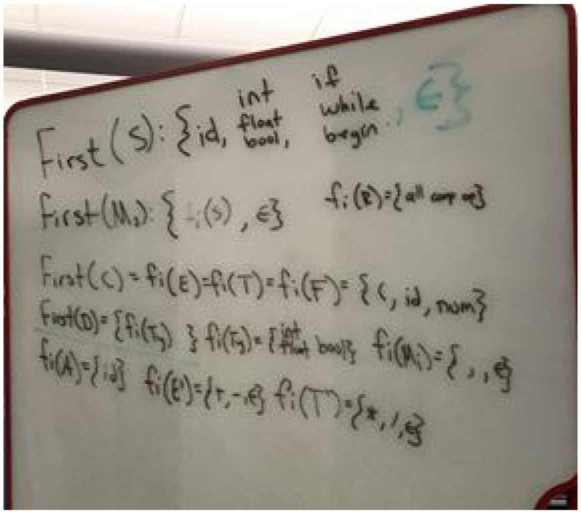


	begin	if	then	else	endif	while	do	whileend	end	>	<	>=	<=	==	<>	int	float	bool	;	,	(	)	id
S	S -> begin	S -> if		epsilon	epsilon	S -> while										S -> D	S -> D	S -> D					S -> A
Ms	Ms -> S_Ms	Ms -> S_Ms				Ms -> S_Ms			epsilon							Ms -> S_Ms	Ms -> S_Ms	Ms -> S_Ms					Ms -> S_Ms
C																					C -> E_Cp		C -> E_Cp
C'			C' -> R_E							C' -> R_E	C' -> R_E	C' -> R_E	C' -> R_E	C' -> R_E	C' -> R_E				epsilon				
D																D -> Ty_id_Mi_;	D -> Ty_id_Mi_;	D -> Ty_id_Mi_;					
Ty																Ty -> int	Ty -> float	Ty -> bool					
Mi																			epsilon	Mi -> ,_id_Mi			
A																							A -> id_=_C_
E																					E -> T_Ep		E -> T_Ep
E'			epsilon				epsilon			epsilon	epsilon	epsilon	epsilon	epsilon	epsilon				epsilon			epsilon	
T																					T -> F_T'		T -> F_T'
T'			epsilon				epsilon			epsilon	epsilon	epsilon	epsilon	epsilon	epsilon				epsilon		epsilon		epsilon
F																					F -> (_E_)		F -> id
R										R -> >	R -> <	R -> >=	R -> <=	R -> ==	R -> <>								



$	num	+	-	*	/
epsilon					
	C -> E_Cp				
	E -> T_Ep				
		Ep -> +_T_Ep	Ep -> -_T_Ep		
	T -> F_T'				
	epsilon	epsilon	epsilon	Tp -> *_F_Tp	Tp -> /_F_Tp
	F -> num				

Brian Alexander
Greg Miller
11/17/2019



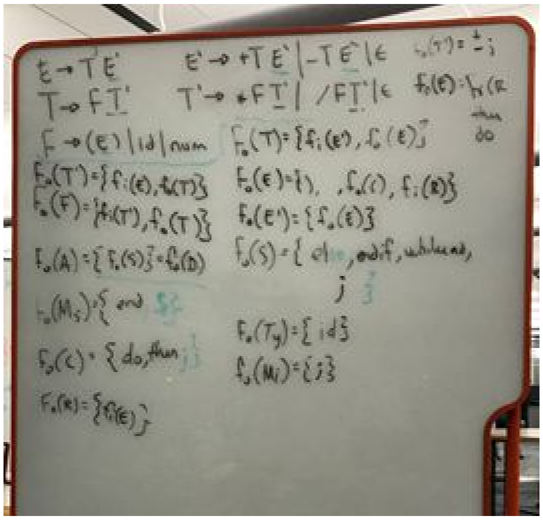
First(S): {id, int float bool, if while begin, ε}

First(M_s): {fi(S), ε}   fi(R)={all comp op}

First(C) = fi(E)=fi(T)=fi(F)= {(, id, num}

First(D)= {fi(T_y) }   fi(T_y)= {int float bool}   fi(M_i)= {, , ε}

fi(A)={id}   fi(E')={+, -, ε}   fi(T')={*, /, ε}



E → TE'   E' → +TE' | -TE' | ε f_o(T')= ± ;

T → FT'   T' → *FT' | /FT' | ε f_o(E): fi(R then do

F → (E) | id | num f_o(T)={fi(E'), f_o(E)}

f_o(T')={fi(E), f_o(T)} f_o(E)={), , f_o(C), fi(R)}

f_o(F)={fi(T'), f_o(T')} f_o(E')={f_o(E)}

f_o(A)={f_o(S)}=f_o(D) f_o(S)={ else, endif, whileend, ; }

f_o(M_s)={end $}

f_o(C)= {do, then ;} f_o(T_y)={id}

f_o(R)={fi(E)} f_o(M_i)={;}