Professor Anthony Le

“Rat 18” Compiler Syntax Analyzer Documentation

Members:
Michael Feldman - mikef0007@csu.fullerton.edu
Alexander Feldman - feldman0007@csu.fullerton.edu
William Clemons - wclemons@csu.fullerton.edu

First Iteration

G = (N, T, S, R)
N = {E, T, F}
T = {+, -, *, /, (,), i}
S = E
R = {

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \mathbf{i}$$

}

Original Rules	Rules After Elim. Left Recursion
E -> E + T	1) E -> TA $\boldsymbol{\delta}$ = T α = +T 2) A -> +TA 3) A -> ε
E -> E - T	1) E -> TA $\boldsymbol{\delta}$ = T α = -T 4) A -> -TA 3) A -> ε
E -> T	5) E -> T
T -> T * F	6) T -> FB $\boldsymbol{\delta}$ = F α = *F 7) B -> *FB 8) B -> ε
T -> T / F	6) T -> FB $\boldsymbol{\delta}$ = F α = /F 9) B -> /FB 8) B -> ε

T -> F	10) T -> F
F -> (E)	11) F -> (E)
F -> i	12) F -> i

Redefined grammar after left factorization and the elimination of left recursion

G = (N, T, S, R):
N = {E, A, T, B, F, X, Y}
T = {+, -, *, /, (,), i}
S = E
R = {

1) E -> TX	------>	**Condensed Rules**
2) X -> A	^	E-> TX
3) X -> ε	^	X -> A | ε
4) A -> +TA	^	A -> +TA | -TA | ε
5) A -> -TA	^	T -> FY
6) A -> ε	^	Y -> B | ε
7) T -> FY	^	B -> *FB | /FB | ε
8) Y -> B	^	F -> (E) | i
9) Y -> ε	^	
10) B -> *FB	^	
11) B -> /FB	^	
12) B -> ε	^	
13) F -> (E)	^	
14) F -> i	^	

}

First/Follow Sets

Production Rule(s)	First	Follow
E -> TX	{ (, i }	{ $,) }
X -> A | ε	{ +, - , ε }	{ $,) }
A -> +TA | -TA | ε	{ +, - , ε }	{ $,) }
T -> FY	{ (, i }	{ +, - , $,) }
Y -> B | ε	{ *, /, ε }	{ +, - , $,) }

B -> *FB | /FB | ε	{ *, /, ε }	{ +, - , $,) }
F -> (E) | i	{ (, i }	{ *, /, +, - , $,) }

Predictive Parser Table

	+	-	*	/	(	)	i	$
E					1) TX		1) TX	
X	2) A	2) A				3) ε		3) ε
A	4) +TA	5) -TE				6) ε		6) ε
T					7) FY		7) FY	
Y	9) ε	9) ε	8) B	8) B		9) ε		9) ε
B	12) ε	12) ε	10) *FB	11) /FB		12) ε		12) ε
F					13) (E)		14) i	

Predictive Parser Pseudocode

```
String InputString = file in
Int inputCursor = 0
bool doneProcessing = false
Push $
Push starting non Terminal

while(true)
if (isTerminal(stack.top()))
        if (stack.top() ==  inputString[inputCursor]) //compare terminals
                If (stack.top() == $)
                        return true // Yes! Accepted...
                inputCursor ++
                stack.pop()
        else
                Error: break (invalid grammar)
```

```
Else //Nonterminal at top of stack
        tempTop = stack.top() // save the nonterminal for expansion
        stack.pop() //now get remove so it can be replaced by expansion
        string production = Table[getIndex(tempTop)][getIndex(inputString[inputCursor])]
        If (production == "ε")
                stack.pop()
        for (int i = production.size() - 1; i >= 0; i--)  // ….in reverse order
                stack.push(production[i])
}
```

helper functions

isNonterminal(string input)
isEpsilon(string input)
isTerminal(string input)

Final Iteration

For the 2nd iteration use the production rules on page 117 of the book, problem 3.13 with the assignment operator.

G = { N, T, S, R }
N = {S, E, Q, T, F, R }
T = { i, =, +, -, *, /, (,), ;}
S = { S }
R = {

$$
\begin{aligned}
S &\rightarrow \mathbf{i} = E \\
E &\rightarrow TQ \\
Q &\rightarrow +TQ \mid -TQ \mid \epsilon \\
T &\rightarrow FR \\
R &\rightarrow *FR \mid /FR \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{i}
\end{aligned}
$$

}

Rules written out for reference in table

1) S -> i = E;
2) E -> TQ
3) Q -> +TQ
4) Q -> -TQ
5) Q -> ε
6) T -> FR
7) R -> *FR
8) R -> /FR
9) R -> ε
10) F -> (E)
11) F -> i

First / Follow Sets

Production Rule(s)	First	Follow
S -> i = E;	{ i }	{ $ }
E -> TQ	{ (, i }	{ ;,) }
Q -> +TQ | -TQ | ε	{+, -, ε }	{ ;,) }
T -> FR	{ (, i }	{ +, -, ;,) }
R -> *FR | /FR | ε	{ *, /, ε }	{ +, -, ;,) }
F -> (E) | i	{ (, i }	{ *, /, +, -, ;,) }

Predictive Parser Table

	i	=	+	-	*	/	(	)	;	$
S	1) i = E;									
E	2) TQ						2) TQ			
Q			3)+TQ	4)-TQ				5) ε	5) ε	
T	6) FR						6) FR			

R			9) ε	9) ε	7)*FR	8) /FR		9) ε	9) ε	
F	11) i						10) (E)			

1. Problem Statement

We were tasked to create a Parser from following the production rules mentioned above, with assignment operator rules being added into our final iteration. The program should implement the 5 RDP functions with the assignment operator, and print out tokens, lexemes, and production rules used. The parser should also generate meaningful error messages when needed.

2. How to Use Your Program

To execute the program, the steps will vary depending on the environment you're running. For simplicity, directions will assume you are able to access an IDE able to compile up to the latest version of C++.

- Download the zip file from Titanium.

The environment in which the code was successfully compiled and tested on was in Windows 10, running Visual Studio 2017. To compile this project from start to finish, download the files and ensure that they are all in the same project folder. To ensure that the lexical analyzers works properly, you should copy and paste your Rat-18 source code into the file "test2.txt". If you wish to use your own text file, then you must ensure that the text file is in the same project folder as the compiler. Also, you must change the name of the file to be opened in compiler.cpp on line 5. To test the syntax analyzer, change the input string passed in line 19 of compiler.cpp to be the line of source code you want to parse. Keep in mind that our syntax analyzer can only process assignment expressions fully at the moment. Once these things are done, simply compile the code, and the program should successfully run and display a table containing the lexemes and token types from the specified input file; furthermore, the syntax analyzer will also display the step by step procedure taken to accept or reject the input string you have passed. Note that when executing the program, the lexical analyzer will run first, and then the program will pause before the syntax analyzer runs. Simply hit enter when prompted to resume the program. By default, the output of the program will print to the terminal for both compiler components. If you want the program to write output to a file, do both of the following. For the lexer, call writeTable() in compiler.cpp instead of printTable(). The lexer's output will be written to "output.txt" in the project directory you are using. For the syntax analyzer, open ParserV3.cpp and comment out each occurrence of the printParserStatus function call and uncomment each occurrence of the writeParserStatus function call. The parser's output file will be written to

“parserOutput.txt” in the project directory you are using. To view the output from the program, go to the directory and double click either file to open with a text editor, or drag the files into your project so you can view them in Visual Studios.

3. Design

We implemented our syntax analyzer using a predictive parse table. The table is a matrix with the columns as terminal values and rows as non-terminal values, both corresponding to a grammar defined by the Rat-18 language. We ensured that left recursion was eliminated from the set of production rules of this grammar; furthermore, we also left factorized rules to reduce ambiguity between these rules. After generating the first and follow sets from our production rules, we populated our predictive table accordingly. We implemented our predictive parse table in conjunction with a stack; moreover, we expanded production rules into the stack by selecting rules from our predictive table. We cross referenced the contents of the stack with our input string to determine if that input was accepted by the Rat-18 grammar. Accepted strings should be fully processed and result in a stack with only a ‘$’ symbol, and these accepted strings are to be considered as syntactically correct.

4. Any Limitations

At the moment, the syntax analyzer does not have mechanisms to analyze all aspects of the Rat-18 language. For example, we do not have a system in place to determine valid function headers/prototypes, if/else blocks, and looping constructs. Furthermore, assembly code cannot be constructed using this compiler.

5. Any shortcomings

There aren’t any shortcomings in the final iteration. In previous iterations, we had not linked the lexer and the parser, but we were able to do so after hammering out the core functionality of both components of the compiler.