Madison Jordan

CS323 Documentation – Project 2

Problem Statement

Parse code to check whether it is syntactically correct. The parser should take an input file and output tokens, lexemes, and the production rule used. If the input string is not valid, an error should be generated.

Compilation / Execution Instructions

Step 1) go to command line
Step 2) type "`.proj2.exe`"
Step 3) Hit the CR
Step 4) type the `<input filename>`
Step 5) Hit the CR
Step 6) See output at "`out_<input filename>`"

Note: Any lines that don't have a semicolon will error out and that line won't be parsed.

Program Design

references:

- *Course Textbook (pseudocode)*
- *https://andrewbegel.com/cs164/ll1.html*
- *https://slaystudy.com/ll1-parsing-table-program-in-c/*
- *http://www.dailyfreecode.com/code/ll1-parsing-1665.aspx*

Grammar Productions:

use left-recursion elimination to get rid of non-terminals on the left side that would cause an infinite loop. (Given)

Production Rules:

Arithmetic

- Addition
  - E -> E+E
- Multiplication
  - E -> E * E
- Parentheses (balanced)
  - E -> (E)
  - E -> Epsilon
    - empty – no parentheses is also balanced
- Variable names
  - E -> id

First():

- For the current non-terminal, get the first character in each of the columns. Unless it's an error value.

Follow():

- For the "current" non-terminal, find this non-terminal in the production table. Then find the terminals after it.
- If there is always the same non-terminal after it, pass that "next" non terminal to the First() function then pass it ("next") to the follow function. The terminals will be the same for this "current".
- If you get an epsilon, find which First(non-terminal) it came from and use that non-terminal in the Follow() function to get the terminal values to replace the epsilon.

Production Table / Generation:

- From the sets from the First() and Follow() functions above, fill in values for the columns where the terminals exist. Otherwise, the value in the column (terminal) /row (non-terminal) is an error.
- The values used should be taken from the grammar productions (left-recursion eliminated) based on the operator used in each of the options/branches.
    - Eg.) T' -> *F T' has the * and goes in the * col.

TABLE

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

	id	**=**	**+**	**-**	*****	**/**	**(**	**)**	**$**
E	**TQ**	*Error*	*Error*	*Error*	*Error*	*Error*	**TQ**	*Error*	*Error*
Q	*Error*	**=TQ**	**+TQ**	**-TQ**	*Error*	*Error*	*Error*	**ε**	**ε**
T	**FR**	*Error*	*Error*	*Error*	*Error*	*Error*	**FR**	*Error*	*Error*
R	*Error*	**ε**	**ε**	**ε**	***FR**	**/FR**	*Error*	**ε**	**ε**
F	**id**	*Error*	*Error*	*Error*	*Error*	*Error*	**(E)**	*Error*	*Error*

Parser:

- Create a stack for the characters to process in this program.
- Get line / code and append “$” to the string
- Push the characters from the string including the “$” onto the stack.

Error Conditions:

String is not in the language

- stack is empty but input stream is not at “$”

Limitations

- none

Shortcomings

- Errors are not especially detailed.
    - Doesn’t print out line numbers, etc.
- Doesn’t parse comments
    - Only the terminals in the production rules