JenPile
Part 2 of Compiler Project
CS323 Documentation

Problem Statement:

The second assignment is to write a syntax analyzer. It can use any top-down parser such as a RDP, a predictive recursive descent parser or a table driven predictive parser. All grammar must be rewritten to remove left recursion. Arithmetic expressions should be done first, then Assignment and declarations. The Parser should print to an output file the token, lexeme, and production rules.

How to Use JenPile:

With a file input:
jenni.exe -c filename
Example: jenni.exe -c HelloWorld.jen

Without a file input, to type on the console line:
jenni.exe

Sample Test Files Included:
ShortMathTest.jen
AssignTestCode.jen
TestArithExpressions.jen

Design Of Program:

The parser is designed to be a Bottom to Top Shift Reduce Parser. It starts from the leaves of the parse tree to the root. The list from the Lexer is used as the input buffer, and a list named theStack stores and accesses the production rules. The stack was originally a stack, but traversing it to check for grammar quickly became difficult.

The variable “theStack” was originally a stack. It was also a stack of strings for the parse types, but this removed the value and token so they couldn’t be used together on part 3 if it was needed. I also briefly considered making each token also have a third type called parse to organize the parse types.

For now the choice was made to add the parse types to the TokenType list. There is an enum set up with the Parse types from that attempt, though it is not used in the submitted version. It was also debated to make a stack that consisted of a ParseType enum and a token, but that got complicated quickly.

The Parser uses the existing Lexer as input. The list or Token - Value pairs created in the Lexer is first passed into the driver for parsing. The following methods handle the parsing of the pairs.

Driver:
The driver method is used to drive the reading of the Token - Value pairs. It adds a Token at the end to designate the end of the file to be parsed. The driver skips white spaces in parsing, and calls two functions, Shift and Reduce.

Shift:
Shift handles the adding of the tokens to the stack for processing. It also writes the line.

Reduce:
Reduce calls three methods that do the reducing of expressions, assignments and statements on the stack.

CheckForExpressions:
This method checks for an identifier, operator, identifier set of tokens and reduces them to an expression. This also takes the three tokens and reduces them to one.

CheckForAssignment:
This method checks for either a keyword, identifier =, expression set of tokens, or a identifier, =, expression set of tokens. When the correct grouping is found, it reduces it to one token.

CheckForStatement:
This method checks if there is a statement, which would be an assignment or a declaration with a ; at the end. The statement function is designed to be the topmost piece that almost everything should reduce too, ending with a semicolon. This function would eventually handle if, while and do.

PrintRule:
Print Rule prints the rules when a correct rule is found.

Limitations

Rules are only printed when successfully used. This could be changed by calling the rule method when the method checking for it is called.

A limitation brought over from the Lexer and also apparent in the parser is that letters with no spaces in the middle, or no line break after them, are read together into one identifier.

In the Expression function- in arithmetic expressions, an operator is used between the two identifiers or expressions. The operator can currently include < and > , so more logic would be needed to correctly handle what operator it is. It was also considered that an expression can be a identifier, float or integer by itself, but that would not be syntactically correct to just have a word, float or integer without anything else paired with it. The choice was made it leave it as identifier operator identifier for simplicity.

Check for assignment operates on the limitation that an assignment operator will always be located at the beginning of a statement, and always has an “id = expression”, “id = float/int/id”, or “keyword id = expression/float/int or id”. It is constricted to only look at the beginning of the list when the list is long enough to have these tokens.

CheckForExpression does not change individual tokens to expression, since it is not an expression if it was by itself. It would need to be assigned to something or some operations would need to take place making it an expression.

There is no end of file token on the input string. When the input string is at the end, that is the end of the file.

In the future, the keyword token would have to be handled differently, as the parser uses certain keywords in different ways. Though, this could be handled by looking at the value after sorting out the token types.

Any shortcomings for each iterations

I did not implement printing to a file. It does implement printing to the monitor. The lexer does print to a file and could be modified to print the parsing too. My

rule outputting is different as it only prints when the rule is true. There is no error checking other than it continues along to the next token.