David Nguyen

Professor Le

CPSC 323-04

16 December 2019

Project 3: Assembly and Symbol Table

1. Problem Statement

For this project, the programmer must build a symbol table handling and generate assembly code from the given grammar. There are many approaches such as using the current implementation from assignment 2 or implement a Bottom-Up approach for the Syntax Analyzer or add more implementation to the Top-Down Syntax Analyzer. The current grammar must be the same as the previous assignment and that the program has no function definitions. Also, there should be no “real” type. There are some semantics that needs to be followed: No arithmetic operations are allowed for Booleans, “true” has an integer value of 1 and “false” has an integer value of 0, and the types must match for arithmetic operations (no conversions). For the symbol table handling, every identifier declared should be in the symbol table and accessed by symbol table functions. Each entry in the symbol table should hold the lexeme, corresponding memory address, and data type. Also, the programmer must create a function that checks if the identifier is declared before being used for assignments and operations. Also, the programmer must print out each entry of the symbol table. There should be error-handling in case of declaring a variable more than once, using an undeclared variable, and mismatch type handling. For generating the assembly code, the programmer must modify the parser and add code to the parser that will create assembly code instructions. The instruction will be kept in a data structure so that the I/O functions will output the assembly code.

2. How to use your program

After obtaining the ZIP file and extracting on to the user's computer, open the CPSC 323 folder. The source code for this program is located at …\CPSC323. The executable and test files are located at …\CPSC323\cmake-build-debug. There are two methods to run my program: the client and the developer side. For ease of use, click on the executable file and the program will parse the input files and output the assembly code and symbol table onto the output file. The default files that I have selected for this program are "syntax_input.txt" and "syntax_output.txt". For the developers, if Clion is being used, just open my file and run my project (shift + f10). If using other IDE just import the files into your environment and run the project.

3. Design of your program

There are three ways to approach this project. I chose to remodify modify my production so that it can parse the assignment, declarative, and conditional statements. I change my parser from LL to LR so that it can be easier to implement the 3AC and the symbol table. Why is it so easy to implement the 3AC and the symbol table because there is no need to worry about left recursion. Now each production has its own final state and it is easier to know when to create the corresponding assembly code. However, creating the state table is very tedious, each production has its own state and each state has its own items. Since, the state table must include every possibility that the marker (input position) must account, it has more items which means there are more states than the LR state table. After, creating the state table by hand, I must implement the state table onto the header file so that the driver function can look up values. The driver function has many ways of handling the incoming input. First, the initial state ($) and then the '0' state must be pushed onto the stack. Then, the top of the stack and the incoming token will be used to index the state table (*state_table[TOS,i]*). If indexing the state table returns a shift, then the incoming input token and the state of the shift will be shifted/pushed onto the stack. If indexing the state table returns a reduction, then depending on how many characters are on the RHS times two will be popped from the stack. After popping the stack, save the top of the stack (state) that will hold the next lookup/peek, and then pushed the LHS of the reduction's production. Then, index the state table with the saved state and the LHS to get the go-to part (state). If indexing the state table returns "accepted", then the current line of token input is accepted. If indexing the state table returns -1, then the input is incorrect will throw a run time error. After implementing the LR parser, the symbol table can be built. Building a symbol table requires a tuple to hold all of the attributes of a symbol (Identifier, Memory Location, Type) and a vector that will hold every symbol. Next, a symbol table handling procedure must input a declaration onto the symbol table with its attributes. After finishing the symbol table, I built the assembly code generator. The assembly code generator requires a tuple that will also be inputted into a vector. The assembly code generator and the parser will work together to parse and obtain assembly code.

4. Any Limitations

Due to finals, it was pretty hard to implement more error handling functions to improve the program. After the semester end, I will add more error handling functions.

5. Any Shortcomings

None.