Brian Alexander
Greg Miller
12/16/2019

CPSC 323 Compilers - Project 3
Symbol Table and Assembly Code Generation Documentation

I. Problem Statement

For assignment 3, we created a top-down predictive recursive decent parser. The parser will generate a symbol table that will print the identifier, memory location, and the identifiers type and the instruction table that will print the address, assembly operation, and the operand. The lexer and parser will determine if the input given was accepted and print out the symbol and instruction table for that input.

The assignment also required us to use our lexical analyzer developed in assignment 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. If we experience an error along the way the parser will throw a syntax error.

II. How To Use The Program

Make sure that all files are downloaded and are in a known location. The required files are: Constands.py, lexer.py, utilities.py, test_file.txt, and semantic_analyzer.py. Once these are downloaded, using the terminal, navigate to the folder that holds the above files and type "python3 semantic_analyzer.py test_file.txt" and press enter. To change the input into the parser, change the file test_file.txt or pass in an alternative filename.

III. Program Design

The recursive descent parser created for this assignment relies on several functions. To do the syntax parsing portion, we rely on the following functions to represent the different rules in our grammar: **Type**, **Declarative**, **Assignment**, **Statement**, **Expression**, **Expression_Prime**, **Conditional**, **Conditional_Prime**, **Relop**, **Term**, **Term_Prime**, and **Factor**. These functions call each other and create a recursive stack used for parsing so it is not necessary to create a stack object in this implementation. They also rely on two helper functions, expect and accept. The accept function checks the current terminal to see if it is a valid option and if it is, increments the pointer to the next terminal. It is used in decision structures to determine where to go next in the syntax analyzer. The expect function also calls the accept function to see if the next terminal is valid, but will error and end parsing if it is not the symbol we were expecting.

Using the recursive descent parser implementation, we can easily integrate intermediate code generation at the appropriate locations. The symbol table is defined as a dictionary **symbol_table** with 3 keys, "identifier", "memory_location", and "type". Using the dictionary with a function **add_symbol_to_table** and a memory address counter

next_available_memory_location it was trivial to check if a symbol was already defined, and if not, add it to the table at a specific memory address and increment the index.

To generate the code listing, we used an array named **code_listing** and had an index counter, **code_listing_index**, initialized to one to keep track of which index we were currently on. To get the ordering of the instructions correct we were required to have an additional stack, **op_stack**. Any operations such as **ADD**, **SUB**, **MUL**, or **DIV** had to be pushed onto the stack one at a time and then popped off once the semi-color was reached, so that they appeared in a verse order from which they were defined in our test text file. **PUSHI, PUSHM,** and **POPM** could be pushed to the instruction table directly.

Error handling is handled by the function **error** that takes one argument, the error message, which defaults to an empty string. It is called at several different locations throughout the program offering information about where the error occurred, including the lexeme and line number. After an error is encountered, parsing and code generation stops as the program exits.

Besides the things mentioned above, the lexer.py file was unchanged from the previous assignment as the changes that we made to it during assignment 2 were sufficient for this assignment. The utilities.py file contains the **token_to_terminal** function from assignment two. which processes the tokens from the lexer into their proper terminals for the parser. It also contains two functions for displaying the results in the terminal. First is **print_instruction_table** for printing the instruction table and **print_symbol_table** for printing the symbol table.

IV. Limitations

None

V. Shortcomings

Although this functionality was not necessary for completion of the project, we will list it here as shortcomings. Our current implementation is only capable of variable declaration (int a; int a, b, c;), assignment (a = 10; b = c;), and mathematical operations, such as addition, subtraction, multiplication, and division (sum = a+b; value = a+b*c-d/f;), but does not consider the order of operations. It does not support decision structures or loop structures at this time.