Nelson Yeap
CMPS 104a lab report

Lab Porject Report

Introduction

For this quarter my lab partner(Chenxing Ji) and I were tasked with building a compiler for a language that we are calling “Objective C” or OC for short. This project was broken up into a multitude of steps that were formed into individual assignments that we turned in and then put together all at the end to create OC and allow it to compile with the gcc compiler. The first part was to build the string set which involves reading in the file and then parsing it thus creating a string set. The next part was to tokenize the strings. After that we had to organize the strings in our string set and create a binary tree. Then we write a symbol table by traversing the binary tree and then we write the intermediate language to oil and compile through gcc.

String Set

This part of the project is fairly simple, We simply open up and oc file and then record all the string that are written into it into a table that we then call our string set. We can first open up the file through fopen and then get each line with the scan function while delimiting each word with spaces. Then we insert the strings into our string set data structure. With this we are done building the string set.

Scanner

For this part we had to write a file called scanner.l which uses a language called flex. Part of the project scans all the string in our string set and then tokenizes them so that we can then build an abstract tree out of the string set later. The scanner.l file is a set of flex rules in which if it sees a certain pattern of characters or a certain character it will return a token usually called TOK_INT or something similar. This is a fairly short file, but does involve the learning of flex which is not a simple task. Our scanner.l file was able to distinguish the various mathematical operators and the keywords that go with variables such as “int”, “string”, and “char”. It is even able to detect invalid strings of characters that will eventually cause the compiler to crash.

Abstract Syntax tree

For this part of the project we create a syntax tree based on the tokens that were returned to us in the scanner. To do this we use another language called bison and create a file called parser.y. The parser.y files is a bison file which is similar to the scanner file. The parser.y file is essentially another set of rules that determine how the syntx tree is built. By writing the rules we can build our sytntax tree simply by having the parser parse through our string set. For example, if a TOK_INT is followed by a TOK_IDENT we then know that the line was trying to declare a variable. TOK_IDENT will become a child of the TOK_INT in the syntax tree. The parser file is a tough one as there are many combinations of tokens to consider when building the syntax tree. Every single string in a

program is very important and will affect the syntax tree in a different way. If the syntax tree is bult incorrectly, it will become extremely difficult to read the string set and parse through the tree in the next part which is the creation of the symbol table.

Symbol Table

This part of the assignment is also not too bad, but becomes extremely difficult if any other part is done incorrectly. This part involves traversing through the syntax tree built by the parser and then assigning symbols to all the variables in the tree. So all ints, chars, and string will need to be assigned their respective symbols. If the syntax tree is built incorrect, this becomes extremely difficult as you then have to go back to the parser and fix it so that the syntax tree build correctly. We did this part through a Depth first search recursion that would scan for certain tokens such as TOK_INT and then assign the TOK_IDENT child of the TOK_INT an integer symbol and the declared variable is an integer, not the “int” keyword.

Intermediate Language

For this part of the project we would again traverse through our syntax tree, but we would then write it out to another file with a .oil extension. Again we did this through a depth first search algorithm where we would go down the tree and then output as necessary. With this we could then compile our .oil file by passing it through the gcc compiler, creating an executable and then running the executable.

Conclusion

This project was a very interesting and difficult project that took an entire quarter to complete. The project gives insight into how all programming languages are created and the difficulties some of the challenges that come with creating a language. We really gained a lot of insight into how sophisticated C and C++ really are and how much effort has gone into making these languages.