|  |
| --- |
|  |

Submitted: April 17, 2013

By: Devan Shah 100428864

Submitted to: Dr. Jorge Landa

|  |  |
| --- | --- |
| ENGR – 3960U | Programming Language and Compilers – Assignment 3 Report |

# Introduction

This assignment is a stepping stone in constructing a fully functional compiler of a language of your choice. For this assignment, the language that I have chosen is a subset of the Scheme language called Minifun. Minifun is a functional programming language that includes basic integer arithmetic, Booleans, relational functions, conditional statements and lists. The main goal of this phase is to implement semantic analysis, interpreter and code generation. I will be using **javacc** to scan and parse my Minifun tokens and grammar specifications. Furthermore, used java to construct the interpreter and generating MIPS assembly code; to be executed in QtSpim to see the results of the imputed program.

# Grammar Adjustment

Reverse

**Simplified Grammar**

1. <prog> ::= <s-exp>

2. | <s-exp> <prog>

3. <s-exp> ::= <def>

4. | <exp>

**5. <def> ::= (define (<var> <var>…<var>) <exp>)**

**6. | (define <var> <exp>)**

7. <exp> ::= <var>

8. | <con>

9. | (<exp\_P>)

**10. <exp> ::= <prm> <exp> ... <exp>**

**11. | <var> <exp> ... <exp>**

**12. | <cond> <cond\_P>**

**13. | (<exp><exp>) ... (<exp><exp>)**

**14. | (else <exp>)**

15. <var> ::= <a-z | A-Z | - | \_> ... <a-z | A-Z | - | \_>

16. <con> ::= <0-9> | #t | #f

17. <prm>::= + | - | \* | / | = | < | > | <= | >=

1. <S> ::= <prog> <EOF>

2. <prog> ::= <s\_exp> <prog>

3. <s\_exp> ::= <def>

4. | <exp>

5. <def> ::= (<def> <def\_P>

6. <def\_P> ::= (<var> <var> ... <var>) <exp>)

7. | <var> <exp> )

8. <exp> ::= <var>

9. | <con>

10. | (<exp\_P>)

11. <exp\_P> ::= <prm> <exp> ... <exp>

12. | <var> <exp> ... <exp>

13. | <cond> <cond\_P>

14. <cond\_P> ::= (<exp><exp>) ... (<exp><exp>)

15. | (else <exp>)

16. <var> ::= <a-z | A-Z | \_><a-z | A-Z | - | \_> ... <a-z | A-Z | - | \_>

17. <con> ::= <0-9> | #t | #f

18. <prm> ::= + | - | \* | / | = | < | > | <= | >=

Left-recursion

resolved

Left-recursion resolved

**Adjusted Grammar**

***Grammar 2***: Represents the Minifun grammar after ambiguity and left-recursion have been resolved. Once the grammar on the lefthas been resolved of all ambiguity and left-recursion, it will allow for simplicity, and an ease to conforming a successful parsing of an inputted file. This grammar after simplification ends up being an LL(2) grammar.

Ambiguity resolved

Left-recursion

resolved

***Grammar 1***: Represents the Minifun grammar that was used to construct the abstract syntax tree. Some of the grammar was un-simplified from ***Grammar 2.***

The context-free-grammar is the most important aspect of constructing a compiler for a desired programming language. The ***Grammer2*** is the grammar that was simplified from phase 1, which was used to make sure that parsing a program was successful. The factored grammar resulted in errors when I was constructing the abstract syntax tree, so I had to un-factor the grammar so that tokenizing for the AST could be done properly. The main portion of the grammar that was changed was only define and the expression grammars. Previously define was factored and broken down in to <def> and <def\_P>, but instead of doing that I put it as only <def> to allow for simplicity for the AST. Also previously expression was factored and broken down into <exp\_P> and <cond\_p>, but instead of doing that I put it as only <exp>. The altering of the grammar for the Minifun language does not affect lexical analysis or syntactical analysis. I had to un-factor the grammar due to the fact that it would make retrieving the nodes and children much easier, when creating interpret.

# Semantic Analysis

The semantic analysis is used to verify that a syntactically is correctly-formed and computes additional information about the meaning of the program. The semantic analysis also involves scope checking, which is used to determine what objects or classes are referred to by each name in the program. The construction of an AST allows to identifying how the program will be interrupted. This portion is one of the most important aspects because this step in creating a compiler helps to define how the language will evaluate the program to give an according output.

## Design

Using ***Grammar 1,*** from above I determined which nodes I will need in the interpreter and which I will not need, this way I can determine which to make void and which to give a deceleration.

**void** symbolic\_expression() *#void* :

**{}**

**{**

**LOOKAHEAD(**2**)**

define()

**|** expression()

**}**

**void** define() *#define* :

**{**

Token d;

Token e;

**int** arguments = 0;

**}**

**{**

**LOOKAHEAD(**3**)**

< LEFT\_PARENTHESIS > d = < DEFINE\_KEYWORD > < LEFT\_PARENTHESIS> e = <VARIABLE> **(**variable()**{**arguments++;**})+** **{**jjtThis.value = e.image;**}***#var(*arguments*)*< RIGHT\_PARENTHESIS > expression() **{**jjtThis.value = d.image;**}** < RIGHT\_PARENTHESIS >

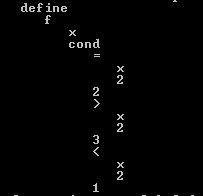
**|** < LEFT\_PARENTHESIS > d = < DEFINE\_KEYWORD > **{**jjtThis.value = d.image;**}** variable() expression() < RIGHT\_PARENTHESIS >

**}**

***AST Definition 1:*** represents part of how my AST will be generated on a supplied test file, the entire jjt code can be viewed in the provided source code

## Testing

Once which methods were going to be void or not void was determine, the next test was to determine which tokens are going to show up in the representation of the AST. Once all methods have been completed according to the grammar of what the Minifun language is supposed to process, the AST needs to be tested on sample test cases. Testing this required a couple of java methods added to the previous phase to display the AST that was constructed by the program on a given test file. For example on a source files such as: **(define (f x)**

** (cond**

**((= x 2) 2)**

**((> x 2) 3)**

**((< x 2) 1)**

**)**

**)**

***Figure 1:*** this figure represents the terminal output of the source file that consisted of*the above stated source code****.*** This shows that the semantic Analysis, is constructing a correct AST, as defined by the grammar and also an interpretation of what the results is supposed to be when the code is compiled.

# Interpreter/Code Generation

The interpreter and the code generation aspect of this phase is to make sure that the code that is generated is correct according to the guidelines of Minifun that were provided. The code is generated using the interpreter as a basis of what a variable does, when and how to perform an operation or when to store values to be used later. I have constructed my generator class so that when run on a program file, the MIPS assembly code will be generated and stored in to and .asm file that can be run on QtSpim. The QtSpim is an interpreter/compiler for the MIPS assembly language, so when the code is run in QtSpim the output will be the expected out that you would expect.

## Design

The AST is used to create an interpreter that analysis the tree to determine how many children a parent node has. Using that data to construct a basis of what is the terminal node and non-terminal nodes. Below is a sample of the Generator class used to construct the .asm file and determine what that .asm file will contain.

**public** Generator(){

// Adding generic entries into the headerEntries ArrayList

headerEntries.add("# Declare main as a global function");

headerEntries.add(" .globl main");

headerEntries.add(" # All program code is placed after the");

headerEntries.add(" # .text assembler directive");

headerEntries.add(" .text");

headerEntries.add(" # The label 'main' represents the starting point");

// Adding generic entries into the dataEntries ArrayList

dataEntries.add(" .data");

dataEntries.add("newLine: .asciiz \"\\n\"");

// Adding generic entries into the mainEntries ArrayList

mainEntries.add("main:");

}

**public** **void** interpeter(SimpleNode rootNode) **throws** FileNotFoundException, ParseException {

**for** (**int** i = 0; i < rootNode.jjtGetNumChildren(); i++) {

evalMain(rootNode.jjtGetChild(i));

}

print.*printConsole*(headerEntries, mainEntries, dataEntries);

print.*printFile*(headerEntries, mainEntries, dataEntries);

}

**.**

**public** **class** mFun {

**public** **static** **void** main(String[] args){

SimpleNode root = Parser.*parser*(args[0]); //root.dump("\t");

Parser.*print\_AST*(root, " ");

Generator e = **new** Generator();

e.interpeter(root);

}

}

**public** **static** **void** printConsole(….){

**for** (**int** i = 0; i < headerEntries.size(); i++) {

System.*out*.println(headerEntries.get(i));}

**for** (**int** i = 0; i < mainEntries.size(); i++){

System.*out*.println(mainEntries.get(i));}

**for** (**int** i = 0; i < dataEntries.size(); i++){

System.*out*.println(dataEntries.get(i));}

}

**public** **static** **void** printFile(…) {

BufferedWriter out;

**try** {

out = **new** BufferedWriter(**new** FileWriter("GeneratedASMFile/Mfun.asm"));

**for** (**int** i = 0; i < headerEntries.size(); i++){

out.write(headerEntries.get(i));

out.newLine();}

.

.

.

***Code 1:*** this particular code is used to construct the .asm file, of the provided test case.

***Code 3:*** this code is from the mFun class which only contains the main method, used to parse the inputted file according the constructed AST and then running the generator class to create the .asm file an also display the MIPS code to the console.

***Code 2:*** this code is from the print class that is used to print to console and also print to .asm file to be rune in QtSpim.

# Integrated Testing

|  |  |  |
| --- | --- | --- |
| Test No. | Input | Console/QtSpim Output |
| 1 | (define y (+ 1 3))  y  (define z (+ (+ y 0) 0))  z | C:\Users\100428864\Desktop\asdfewd.pngC:\Users\100428864\Desktop\adfevgewvv.png |
| 2 | (define x 1)  x  (define y 100)  y | C:\Users\100428864\Desktop\dcdc.pngC:\Users\100428864\Desktop\hdfgfdgfhfgf.png |
| 3 | (define x 0)  (cond (x 10)  (x 11))  (define x 1)  (cond (x 10)  (x 11)) | C:\Users\100428864\Desktop\adsfcdefevee.pngC:\Users\100428864\Desktop\dsfasdjfdhskgjfdskfjdegdejfdkj.png |

# Challenges and Problems Faced

* Knowing where to start
* Learning the MIPS assembly language
* Assembling the MIPS assembly language using AST

# Discussion

This phase was a success as seen from the results that the console and the QtSpim provide, it shows that the combination of the AST and the java classes the correct output for the provided test cases is being achieved. The code generation is a key aspect for creating a compiler; this phase accomplishes this task extremely well.

# Conclusion

In Conclusion, this is the final step in constructing a compiler as the MIPS assembly code is generated with the use of the AST and java classes. The provided will generate the according MIPS code to be compiled using QtSpim to compiler the generated MIPS code.