CA4003 Compiler Construction

Assignment #2 Semantic Analysis and Intermediate Representation

A report submitted to Dublin City University, School of Computing for module CA4009: Search Technologies, 2018/2019. I understand that the University regards breaches of academic integrity and plagiarism as grave and serious. I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the sources cited are identified in the assignment references. I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work. By signing this form or by submitting this material online I confirm that this assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy (available at: http://www.dcu.ie/registry/examinations/index.shtml)

Name: Olan Buckeridge (15461022)
Date: 15/12/2018
Programme: CASE4

0. Overview

The aim of this assignment is to add semantic analysis checks and intermediate representation generation to the lexical and syntax analyser you have implement in Assignment 1. The generated intermediate code should be a 3-address code and stored in a file with the ".ir" extension.

You will need to extend your submission for Assignment 1 to:

- Generate an Abstract Syntax Tree.
- Add a Symbol Table **that can handle scope**.
- Perform a set of semantic checks. This following is a list of typical semantic checks:
    - Is every identifier declared within scope before its is used?
    - Is no identifier declared more than once in the same scope?
    - Is the left-hand side of an assignment a variable of the correct type?
    - Are the arguments of an arithmetic operator the integer variables or integer constants?
    - Are the arguments of a boolean operator boolean variables or boolean constants?
    - Is there a function for every invoked identifier?
    - Does every function call have the correct number of arguments?
    - Is every variable both written to and read from?
    - Is every function called?
- Generate an Intermediate Representation using 3-address code.

1. Introduction

For this assignment I applied the knowledge learned about Abstract Syntax Trees, Symbol Tables and Intermediate Code Representations from lecture notes. By using examples in the notes I was able to structure and grasp the assignment better.

While going back to my CALParser code from Assignment #1, I found that it was quite messy and not structured properly. I added some formatting to make the code more legible and easier to understand.

I personally found this assignment particularly challenging and time consuming.

2. Abstract Syntax Tree

In order to implement the Abstract Syntax Tree, I took a systematic approach. I decided to choose an example program to walkthrough and see what nodes it would create. To create the Abstract Syntax Tree each production rule and a number of child nodes is decorated.

For instance, the rule function_body always will have 3 child notes, decl_list, statement_list and a return_statement. Therefore we use #FunctionBody(3).

```
void function_body ( ) #FunctionBody(3): {}
{
    <IS>
    decl_list()
    <BEGIN>
       statement_list()
       return_statement()
    <END>
}
```

I've refactored rules that call another rule multiple times. For instance, (function( ))* is put inside its own rule named function_list with the decoration #FunctionList.

```
void function_list ( ) #FunctionList: {}
{
    ( function() )*
}
```

Another example is binary_arith_op() which is capable of performing two different actions.

```
void binary_arith_op ( ) : {}
{
    ( <PLUS> expression() #Add(2) |
      <MINUS> expression() #Subtract(2) |
      {} )
}
```

If an assignment statement of a negative identifier, there should only a single child node	If it's a calculation, there should be two child nodes
..... | Assignment | | ID | | Negative | | | ID	..... | Assignment | | ID | | Subtract | | | ID | | | ID

In order to fix this issue, we put <MINUS> fragment( ) into fragment( ) and decorate that node as #Negative, then give the #Subtract decoration to the binary_arith_op( ) condition. The same is also repeated for <PLUS>.

You can find each decoration in the CALOlanBuckeridge.jjt file. The Abstract Syntax Tree is generated in the CALOlanBuckeridge.jjt files user code by calling root.dump(“”); on the parser.

Example output below;

```
File name: ../CAL/test2.cal
**** Abstract Syntax Tree ****
Program
 DeclList
 FunctionList
  FunctionDecl
   Type
   ID
   ParameterList
   FunctionBody
    DeclList
    StatementList
    Return
 Main
  DeclList
  StatementList
   FunctionCall
    ID
*****************************
```

3. Symbol Table

SemanticCheckVisitor is where the Symbol Table is created. All the symbols in the table are saved as an STC object. The following components below are found in each object;

- Token name
- Token type
- DataType datatype
- String scope;
- LinkedHashMap<String, Object> values
- int numArgs = -1;
- boolean isRead
- boolean isCalled

In order to fill out the Symbol Table, some production rules are refactored again to return Tokens. For instance, we create a rule called identifier() which will return that token for every call of identifier( ).

```
void identifier ( ) #ID: { Token t; }
{
    t = <IDENTIFIER> { jjtThis.value = t; }
}
```

In the SemanticCheckVisitor we create a data structure HashMap of HashMaps for the Symbol Table. After some research I found that using Hash Tables would be the most efficient route to take. An example of the symbol table can be seen below.

```
{ "Program":
        { "x": STC[x, integer, Variable, "Program" {x=2}, -1, true, false],
          "y": STC[y, boolean, Constant, "Program" {y=true}, -1, false, false],
           ... },
        ...
}
```

The Symbol Table is displayed in the SemanticCheckVisitor.java file after visiting every node in the Abstract Syntax Tree.

Sample output is as follows;

```
**** Symbol Table ****
Scope: test_fn
x
 DataType: ParamVariable
 Type: integer
 Initial Value: {}
 Is written to?: false
 Is read from?: true
i
 DataType: Variable
 Type: integer
 Initial Value: {i=2}
 Is written to?: true
 Is read from?: false

Scope: Program
test_fn
 DataType: Function
 Type: integer
 Parameters: {x=integer}
 Is called?: true
i
 DataType: Variable
 Type: integer
 Initial Value: {}
 Is written to?: false
 Is read from?: false

Scope: Main
i
 DataType: Variable
 Type: integer
 Initial Value: {i=1}
 Is written to?: true
 Is read from?: true

**********************
```

4. Semantic Analysis

In the SemanticCheckVisitor, the following check are completed.

- Is every identifier declared within scope before it is used?
- Is no identifier declared more than once in the same scope?
- Is the left-hand side of an assignment a variable of the correct type?
- Are the arguments of an arithmetic operator the integer variables or integer constants?
- Are the arguments of a boolean operator boolean variables or boolean constants?
- Is there a function for every invoked identifier?
- Does every function call have the correct number of arguments?
- Is every variable both written to and read from?
- Is every function called?

There are some extra checks implemented along with these checks.

★ Function is already declared with the same number of parameters
★ Function attempting to be declared with one or more parameters with the same id
★ Argument in a function call is of incorrect type
★ Variable is not declared when trying to assign it a value
★ Variable has no value when attempting to do an operation

All of errors are displayed and directing the line and column where the error is found. The number of errors are also output. Certain errors might only happen when the previous errors are addressed. For instance,

```
integer add (x:integer, y:integer) {
var z:integer;
z := x + y;
return x;
}
```

The x and y variables are saying that they have not to been read to, but this is due to this case as they do not have any value yet. The visit methods in the SemanticCheckVisitor.java file describes how many, and also what type of child nodes the current node will have.
The three address code ir will be generated if there are no errors.

5. Three Address Code

AddressCode is an object that I created to simply hold 4 string values, for the IR.
A visit method is used in the ThreeAddressVisitor.java for each node and shows the CAL code in three address code. Sample output below;

```
**** IR using 3-address code ****
L1
 return
L2
 funcCall func
 goto L1
********************************
```

6. How to Run

You can run with ./build.sh using the script included in the folder.

```
#!/bin/bash

jjtree CALOlanBuckeridge.jjt
echo ""
javacc CALOlanBuckeridge.jj
echo ""
javac *.java
echo "Built"
```