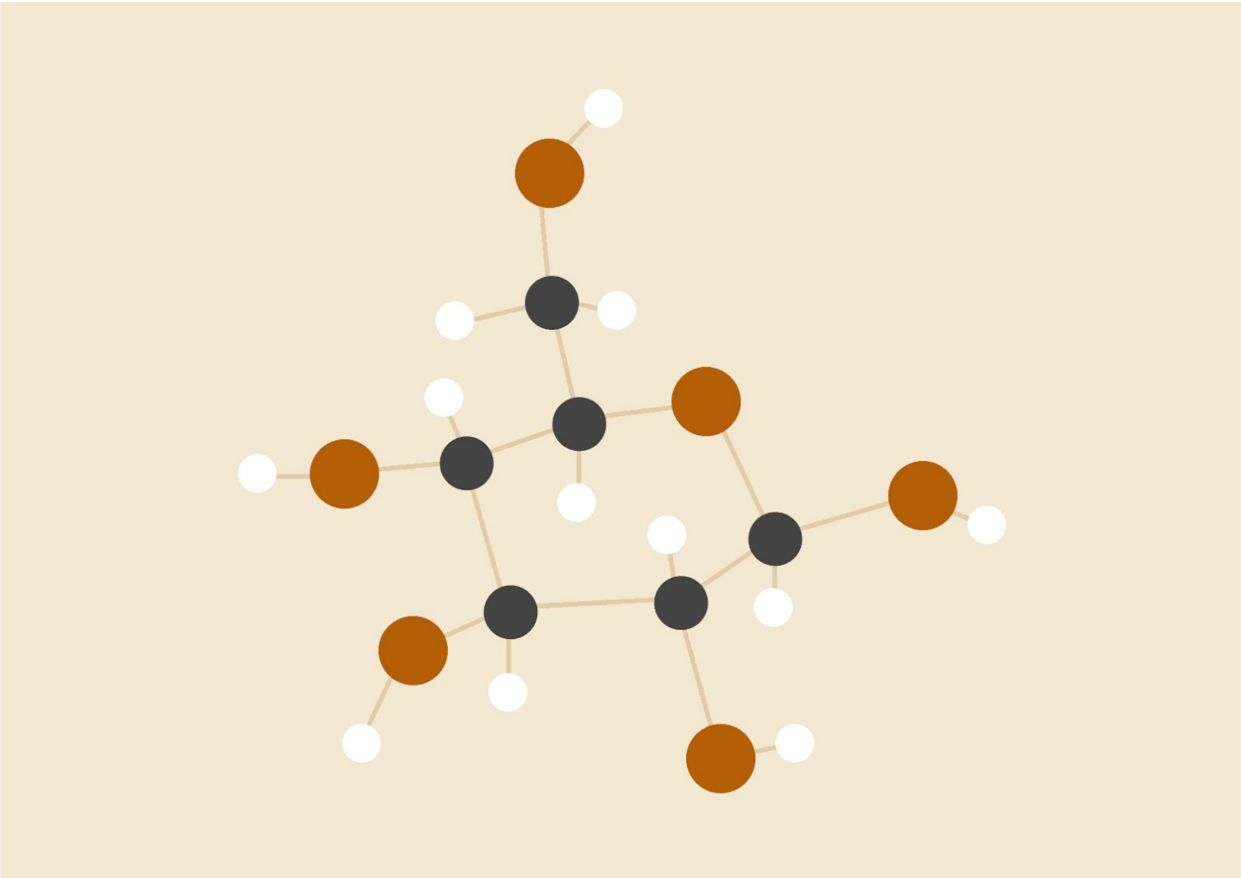
Semantic Analysis and Intermediate Representation

CA4003 Assignment Two



Niall Lyons - 13493628

17/12/18

CA4003 Compiler Construction

Dr. David Sinclair

Table Of Contents

DECLARATION

I declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study. I/We have read and understood the referencing guidelines found at http://www.dcu.ie/info/regulations/plagiarism.shtml , https://www4.dcu.ie/students/az/plagiarism and/or recommended in the assignment guidelines.

Name: Niall Lyons

Date: 17/12/18

INTRODUCTION

" 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. "

In this report I will discuss and outline how I implemented the following sections:

- Abstract Syntax Tree
- Symbol Table
- Semantic Analysis
- 3 Address Code

ABSTRACT SYNTAX TREE

An abstract syntax tree represents all of the syntactical elements of a programming language, similar to syntax trees that linguists use for human languages. The tree focuses on the rules rather than elements like braces or semicolons that terminate statements in some languages. The tree is hierarchical, with the elements of programming statements broken down into their parts. For example, a tree for a conditional statement has the rules for variables hanging down from the required operator. [1]

In order to see exactly where we are in the programme and also the tokens that are being consumed, I added a name to each of the production rules in the JJT file. The parser will then go down through the programme, starting at the parent node and then to each child node, this will construct the AST. When we reach the bottom of the programme the the tree is then fully developed and I was then able to evaluate in order to start looking at the symbol tree.

This was implemented in the JJT file, and as can be seen in it, the AST is generated when *root.dump("")* is called on the parser. An example can be seen below, of test4.cal and the corresponding AST.

```
variable i : integer ;
integer test_fn (x : integer) is
variable i:integer;
begin
    i := 2;
    return (x);
end
main
begin
    variable i:integer;
    i := 1;

end
```

```
***** ABSTRACT SYNTAX TREE *****
Programme
 DeclarationList
  VarDeclaration
   ID
   TypeValue
  DeclarationList
 FunctionList
  Function
   TypeValue
   ID
   ParamList
    Params
     ID
     TypeValue
   FunctionBody
    DeclarationList
     VarDeclaration
      ID
      TypeValue
     DeclarationList
    StatementBlock
     Assignment
      ID
      Digit
     StatementBlock
    ReturnStatement
     FunctionCall
      ID
  FunctionList
 Main
  DeclarationList
   VarDeclaration
    ID
    TypeValue
   DeclarationList
  StatementBlock
   Assignment
    ID
    Digit
   StatementBlock
***** END SYNTAX TREE *****
```

SYMBOL TABLE

A symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. [2]

After I created the AST tree I was now able to develop a Symbol table, this was developed in the Visitor.java file.

I created a Symbols.java file which holds the attributes of each symbol in the symbol table, in this way we are able to track each one. Examples of these are :

- Token name
- Token type
- DataType symbolType
- String scope
- LinkedHashMap<String, LinkedList<VarTypes>> values
- int numberOfArgs = -1
- boolean isRead = false
- boolean isCalled = false

Above we can see symbolType, this allowed me to track the different types of symbols in the symbol table, variable, constant or function. While developing the symbol table I had to add to my production rules, here I added a rule identifier() and digit() that will return the given tokens when called.

To then track assignments that are made to each symbol I created a map called values as seen above. I also created a string called scope to track if it was in function or the main.

The two booleans represent if a symbol was read or called when the visitor is running. Also, numberOfArgs represents in a function the number of arguments that a symbol has.

As an example I will use the same test4.cal file as above and this is the resulting symbol table.

```
***** START SYMBOL TABLE ***
***** START test_fn SCOPE *****
Name: x
         The SymbolType: PARAM
         The Values: No assignments made
         It is written to: false
         It is read from: false

Name: i
         The SymbolType: VAR
         The Values: { 2: integer, }
         It is written to: true
         It is read from: false

***** END test_fn SCOPE *****
***** START Main SCOPE *****
Name: i
         The SymbolType: VAR
         The Values: { 1: integer, }
         It is written to: true
         It is read from: false

***** END Main SCOPE *****
***** START Programme SCOPE *****
Name: test_fn
         The SymbolType: FUNC
         The Parameters: { x: integer, }
         Is called?: false

Name: i
         The SymbolType: VAR
         The Values: No assignments made
         It is written to: false
         It is read from: false

***** END Programme SCOPE *****
***** END SYMBOL TABLE *****
```

SEMANTIC ANALYSIS

I made the following 12 semantic analysis checks which were made by Visitor.java using the above symbol table:

- 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?
- Constants are not reassigned?
- Variables are assigned a value before used?
- Does the function already exist with the same name?

At the beginning, I took the wrong approach to this part of the assignment as I was letting the parser go through the entire completed symbol table and then trying to check them. This wasn't working for me so I then took the approach of calling back on prior symbols that were parsed or the current node. Although some of my checks required the entire tree to be checked I was able to implement this approach in a much easier way.

Using the same test4.cal file as above the following semantic analysis results are given,

```
***** HERE ARE THE SEMANTIC CHECK RESULTS! ******
1 functions are declared but have not been used.
4 variables have not been accessed:
x,i,i,i
1 variables have not been initialised:
i
NO ERRORS HERE!
```

From the above symbol tree that was produced for this test file, we can see from the semantic analysis results that there is one function declared but not used and one variable has not been initialised, while four variables have not been accessed.

We can also see that there are no errors during the parse, these are tracked by the visitor also, and if an error occurs it is put into an array to print at the end. If this array length gets higher than zero, we stop. As if we have an error, the three address representation will not be made.

3 ADDRESS CODE

In order to implement three address code into this compiler I made a AddressCode object that will hold four string values for the Intermediate Representation. In order to parse I created the ThreeAddressCoder.java file. Here, while going through each node, all representations, labels for functions and conditions etc are held in a table.

As in the assignment description , the 3 address code is printed and store in a ".ir" file. I named my file "TAC.ir". To do this I used, PrintStream, FileOutputStream and also just incase, a FileNotFoundException.

The following three address code is generated in the TAC.ir file for the same test4.cal file that I have been using:

```
****** THREE-ADDRESS CODE REPRESENTATION *****
L1
 = i 2
 return
L2
 = i 1
 END
***** END THREE-ADDRESS CODE REPRESENTATION *****
```

Here we can see that L1 is the beginning of the programme where i is equal to 2 and moving to L2 where i is equal to 1.

However as there is no loops etc, let's look at test6.cal and its corresponding 3 address code:

```
integer multiply (x : integer , y : integer ) is
    variable result : integer;
    variable minus_sign : boolean;
begin
// f i g u r e out si g n o f r e s u l t and c o n v e r t a r g s to a b s ol u t e v al ue s
    if ( x < 0 & y >= 0)
    begin
        minus_sign := true;
        x := -x;
    end
    else
    begin
        if y < 0 & x >= 0
        begin
            minus_sign := true;
            y := -y;
        end
        else
        begin
            if (x < 0) & y < 0
            begin
                minus_sign := false;
                x := -x;
                y := -y;
            end
            else
            begin
                minus_sign := false;
            end
        end
    end
    result := 0;
    while ( y > 0)
    begin
        result := result + x;
        y := y - 1;
    end
    if minus_sign = true
    begin
        result := -result;
    end
    else
    begin
        skip;
    end
        return (result);
end
main
begin
variable arg_1 : integer;
variable arg_2 : integer;
variable result : integer ;
constant five : integer := 5;
arg_1 := -6;
arg_2 := five;
result := multiply (arg_1 , arg_2);
end
```

```
****** THREE-ADDRESS CODE REPRESENTATION *****
L1
 if
 goto L4
 goto L19
 L4
 = result 0
 functionCall y
 goto null
 L21
 while
 goto L24
 if
 functionCall minus_sign
 goto null
 goto L29
 goto L31
 L30
 return
L2
 = minus_sign true
 goto L4
L3
 = x x
 goto L4
L5
 if
 goto L9
 goto L18
 L9
L7
 = minus_sign true
 goto L9
L8
 = y y
 goto L9
L10
 if
 goto L15
 goto L17
 L14
L12
 = minus_sign false
 goto L14
L13
 = x x
 goto L14
L14
 = y y
L16
 = minus_sign false
L22
 functionCall result
 goto null
 functionCall x
 goto null
 = result
L23
 functionCall y
 goto null
 = y
L24
 goto L21
L28
 = result result
 goto L30
L29
```

Here we are able to see “goto” used for loops and if else statements.

HOW TO RUN

To run this file I created a bash script, which consisted of :

```
#!/usr/bin/env bash
jjtree assignment2.jjt;
javacc -debug_parser assignment2.jj;
javac *.java
```

Then to test the parser on a file I used:

- Java assignment2 inputfile

REFERENCES

[1] Techopedia, Abstract Syntax Tree (AST), *https://www.techopedia.com/definition/22431/abstract-syntax-tree-ast* *, 2017*

[2] TutorialsPoint, Compiler Design, *https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.html* *, 2016*

Stanford.edu, Three Address Code Examples, *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/240%20TAC%20Examples.pdf* *, 2012*

University Of Copenhagen, Basics of Compiler Design, *http://hjemmesider.diku.dk/~torbenm/Basics/basics_lulu2.pdf* *, 2011*