

Declaration on Plagiarism

Assignment Submission Form

This form must be filled in and completed by the student(s) submitting an assignment

Name(s): Ore Ibikunle
Programme: Computer Applications
Module Code: CA4003
Assignment Title: Semantic Analysis and Intermediate Representation
Submission Date: 14/12/2018
Module Coordinator: David Sinclair

I/We 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(s): _____ Ore Ibikunle _____ Date: _____ 03/11/2018 _____

Table of Contents

1. Introduction

2. Design

1. Options.....	4
2. User Code.....	4-5
3. Production Rules.....	5

3. Implementation

1. Abstract Syntax Tree.....	6-7
2. Symbol Table.....	7-8
3. Datatype Class.....	9
4. Semantic Analysis.....	9-12
5. Intermediate Representation.....	12-16

4. References

Semantic Analysis and Intermediate Representation

1. Introduction:

The aim of this assignment is to add a semantic analyser and intermediate representation generation of the lexical and syntax analyser that I implemented for the CAL language.

The purpose of Semantic Analysis in a compiler is to check the source program for semantic errors and to gather type information for the intermediate code generation phase. The semantics define the meaning of a program. It is important because not all syntactically correct programs have a valid meaning.

The goal of an Intermediate Representation (IR) is to allow multiple optimisation passes, where each pass transforms the current into an equivalent IR that runs more efficiently. There are many different types of IR, but the types of IR we will be using for this assignment are:

- **Abstract Syntax Tree (AST)**
- **3-Address Code**

In compiler construction an **Abstract Syntax Tree** is a tree representation of the abstract syntactic structure of source code written in a programming language. In the tree each node denotes a construct occurring in the source code. The tree can be described as abstract because in that sense it doesn't represent every specific detail appearing in the real syntax, but rather just the structural content-related details.

3-Address Code is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.

2. Design

As described by the notes, the **JJTree** file will consist of four sections:

1. Options
2. User Code
3. Tokens
4. Production Rules

I have explained these in the documentation for first assignment, so I won't go over them again.

The only sections I needed to add to for this assignment were the **Options**, **User Code**, and **Production Rules**.

2.1 Options:

In the option sections, will have three new additional options:

- **MULTI** – when JJTree builds the abstract syntax tree, the nodes are defined as being subclasses of a type Simple Node. By setting the value of MULTI to true it will generate different types of nodes for different types of classes.
- **VISITOR** – if this option is set to true, it will add in additional code to support the visitor pattern.
- **NODE_DEFAULT_VOID** – when this is set to true all production rules will not generate an AST node by default unless the declaration inside the production rule specifically says to do it.

```
options
{
    IGNORE_CASE = true;
    JAVA_UNICODE_ESCAPE = true;

    MULTI = true;
    VISITOR = true;
    NODE_DEFAULT_VOID = true;
}
```

2.2 User Code:

In the **User Code** section, I defined a **symbol table**, which is a class that I created that will map identifiers to their types and locations.

```
public static String scope = "global";
public static SymbolTable symTable = new SymbolTable();
```

The program will read in from a file input stream and then it instantiates a **Simple Node** that will call the **Program** production rule inside the parser. I also have two visitors in this section:

- **TypeCheckVisitor** – this will handle checks for semantic analysis.
- **ThreeAddressCodeVisitor** – this will output the code that has been parsed and represent it as intermediate representation code.

```
try
{
    parser = new Cal(new FileInputStream(args[0]));

    SimpleNode simpleNode = parser.Program();
    System.out.println("The program was parsed successfully.");
    System.out.println("--AST--");
    simpleNode.dump("");

    System.out.println("\n--Symbol Table--\n");
    symTable.output();

    System.out.println("--Semantic Analysis--");
    TypeCheckVisitor typecheckvisitor = new TypeCheckVisitor();
    simpleNode.jjtAccept(typecheckvisitor, symTable);

    System.out.println("\n--Intermediate Code Representation--\n");
    ThreeAddressCodeVisitor tac = new ThreeAddressCodeVisitor();
    simpleNode.jjtAccept(tac, null);

}
catch(FileNotFoundException e)
{
    System.out.println("File " + args[0] + " doesn't exist.");
}
catch(ParseException e)
{
    System.out.println("The program did not parse.");
    System.out.println(e.getMessage());
}
```

2.3 Production Rules:

In this section I will be using the production rules we defined from the previous assignment to create the abstract syntax tree. A parser tree structure is used to separate parsing from semantics. It is possible to write a compiler so that everything is done in by the semantic actions, but it makes it very difficult to maintain.

3. Implementation:

In this section I will discuss how I built my Abstract Syntax Tree, Symbol Table, Datatype class, and visitors.

3.1 Abstract Syntax Tree:

An abstract syntax tree is a clean interface between parsing and semantic analysis. The abstract syntax tree captures the phrase structure of the input.

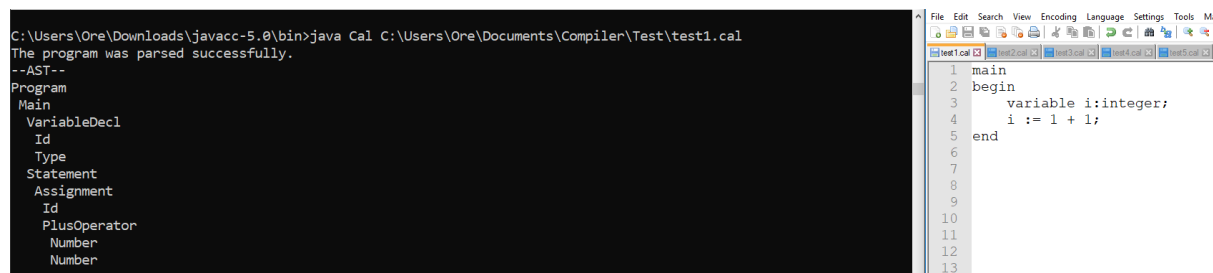
To build my abstract syntax tree some of my production rules will have its name and the number of children as a decoration as shown below in my **Function** production rule. I didn't make all production rules in my abstract syntax tree a node because I wanted to remove unnecessary information from my abstract syntax tree.

```
void Function() #FuncBody: { String type; String id; }
{
    type = Type() id = Id() { symTable.push(scope, "function", id, type); scope = id; } < LP > Parameter_list() < RP > < IS >
    Decl_list()
    < BEGIN >
    Statement_block()
    < RETURN > < LP > (Expression() | {}) < RP > < SEMIC > #FunctionReturn
    < END >
}
```

For arithmetic operators, logical operators, and comparison operators, I pushed two child nodes onto the tree as demonstrated in my **Comp_Op** production rule shown below.

```
void Comp_op() : { Token t; }
{
    Expression()
    (
        t = < EQUAL_TO > Expression() { jjtThis.value = t.image; } #EqualTo(2)
        | t = < NEQUAL_TO > Expression() { jjtThis.value = t.image; } #NequalTo(2)
        | t = < LESS_THAN > Expression() { jjtThis.value = t.image; } #LessThan(2)
        | t = < LEQUAL_TO > Expression() { jjtThis.value = t.image; } #LequalTo(2)
        | t = < GREATER_THAN > Expression() { jjtThis.value = t.image; } #GreaterThan(2)
        | t = < GEQUAL_TO > Expression() { jjtThis.value = t.image; } #GequalTo(2)
    )
}
```

This is so when I implemented my semantic analyser, I could check to see that the two child nodes are of the same type. For example, If I used the plus operator, two child nodes will be pushed onto the tree with the plus operator as their parent node.



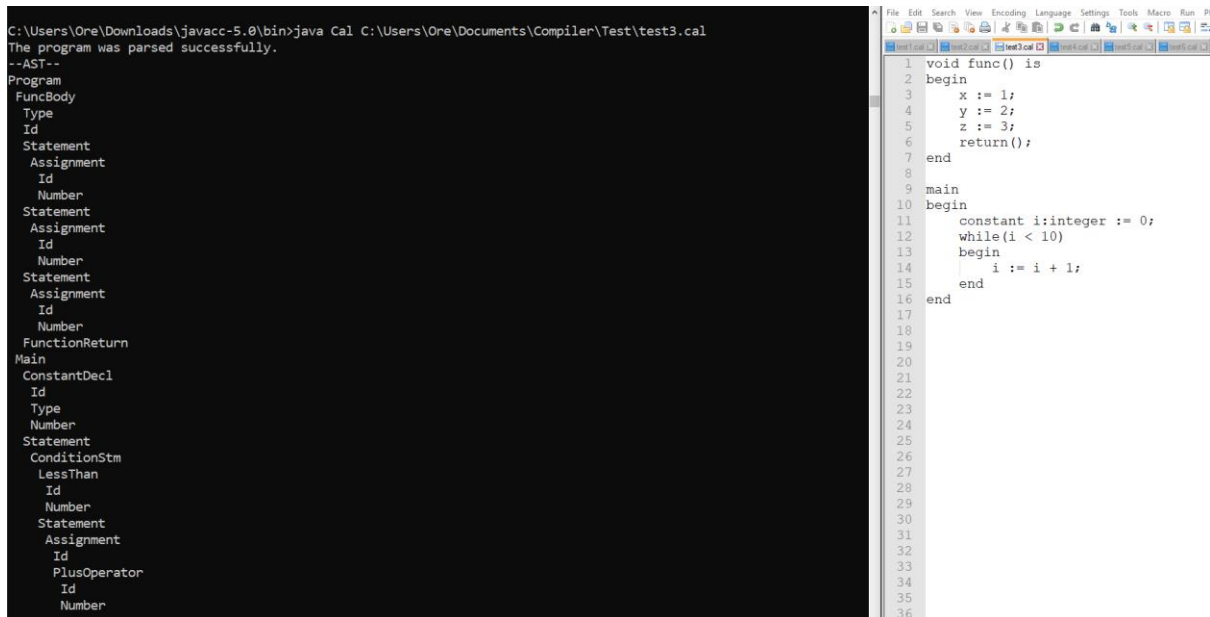
```
C:\Users\Ore\Downloads\javacc-5.0\bin>java Cal C:\Users\Ore\Documents\Compiler\Test\test1.cal
The program was parsed successfully.
--AST--
Program
Main
VariableDecl
Id
Type
Statement
Assignment
Id
PlusOperator
Number
Number
```

```
1 main
2 begin
3     variable i:integer;
4     i := 1 + 1;
5 end
6
7
8
9
10
11
12
13
```

As shown in my **Comp_Op** production rule I also kept the types of each operator so that it can be used later when I build my semantic analyser. I also used this method for arithmetic and logical operators.

The main idea I had when building my abstract syntax tree was to remove unnecessary nodes from the tree, but also to make sure that it was possible to visit every node in the tree so that implementing my visitors would be more straightforward.

Here is an example of my AST for a program I wrote below. Logically the program does nothing, this is simply just a demonstration.



The image shows a terminal window on the left and a code editor on the right. The terminal window displays the output of a Java compiler (javac) and the resulting Abstract Syntax Tree (AST) for a program. The code editor shows the source code of the program, which is a simple demonstration program.

```
C:\Users\Ore\Downloads\javacc-5.0\bin>java Cal C:\Users\Ore\Documents\Compiler\Test\test3.cal
The program was parsed successfully.
--AST--
Program
  FuncBody
    Type
      Id
    Statement
      Assignment
        Id
        Number
    Statement
      Assignment
        Id
        Number
    Statement
      Assignment
        Id
        Number
    FunctionReturn
  Main
    ConstantDecl
      Id
      Type
      Number
    Statement
      ConditionStm
        LessThan
          Id
          Number
        Statement
          Assignment
            Id
            PlusOperator
            Id
            Number
```

```
1 void func() is
2 begin
3   x := 1;
4   y := 2;
5   z := 3;
6   return();
7 end
8
9 main
10 begin
11   constant i:integer := 0;
12   while(i < 10)
13   begin
14     i := i + 1;
15   end
16 end
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

3.2 Symbol Table:

A symbol table is an important data structure created and maintained by compilers to store information about the occurrence of various entities such as variable names and function names. A symbol table can be implemented using a Hash Table. My symbol table will handle three types of scope:

1. global
2. main
3. function

To create my symbol table, I created a symbol table class that has three hash tables that map:

- the scope of all variables defined to a linked list
- the same key to a keyword, e.g. variable or constant

- identifiers and their scope to their type, e.g. as shown below, the variable *i* is in the scope “**main**”, this will be its key and it will be mapped to type integer.

```

C:\Users\Ore\Downloads\javacc-5.0\bin>java Cal C:\Users\Ore\Documents\Compiler\Test\test1.cal
The program was parsed successfully.
--AST--
Program
Main
  VariableDecl
    Id
    Type
  Statement
  Assignment
    Id
    PlusOperator
    Number
    Number
--Symbol Table--
The scope is: main
(variable, integer, i)
The scope is: global

```

```

1 main
2 begin
3   variable i:integer;
4   i := 1 + 1;
5 end
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

```

```

public class SymbolTable extends Object
{
    public Hashtable<String, LinkedList<String>> symTable;
    public Hashtable<String, String> dataTypes;
    public Hashtable<String, String> keywords;

    public SymbolTable()
    {
        this.symTable = new Hashtable<>();
        this.dataTypes = new Hashtable<>();
        this.keywords = new Hashtable<>();

        symTable.put("global", new LinkedList<>());
    }
}

```

In my symbol table class, I created a **push** method to add values to my symbol table and an **output** method to show the values added to my symbol table. I also have two helper methods here called **getDataType** and **getKeyword**. I'll explain why these methods are useful later. Along with these two helper methods, I have two other methods:

- detectScope** – this method checks if the identifier you supplied as a parameter is defined in the scope that you supplied as a parameter.
- dedups** – short for detect duplicates, this method is used to check for duplicates and will return an output error if there are duplicate variables defined in the same scope.

Whenever a program is parsed, it will create the abstract syntax tree and symbol table at the same time, this occurs anywhere an **identifier** is used as demonstrated below by my function production rule:

```

void Function() #FuncBody: { String type; String id; }
{
    type = Type() id = Id() { symTable.push(scope, "function", id, type); scope = id; } < LP > Parameter_list() < RP > < IS >
    Decl_list()
    < BEGIN >
    Statement_block()
    < RETURN > < LP > (Expression() | {}) < RP > < SEMIC > #FunctionReturn
    < END >
}

```


3.3 Datatype Class:

To carry out semantic analysis I will need to have a file that defines the datatypes for the Cal language. In my datatype file I declared an enumeration of datatypes as shown below:

```
public enum DataType
{
    SimpleNode,
    Program,
    VariableDecl,
    ConstantDecl,
    FuncBody,
    FunctionReturn,
    Assignment,
    FunctionAssignment,
    ParameterList,
    Main,
    Statement,
    ConditionStm,
    Number,
    BooleanOperator,
    ArgList,
    TypeUnknown
}
```

3.4 Semantic Analysis:

To carry out Semantic Analysis I created a **TypeCheckVisitor** class that visits each node that was created in my abstract syntax tree.

- Is every identifier declared within scope before it is used?

To check if every identifier is declared before it is used, I must check if its parent node is not a constant or variable declaration. If it is not a declaration, then it has no type yet, I check the scope and type of that identifier using my `getDataType` method I defined in the symbol table and check for its appropriate type. If it isn't in that scope, then you check if the identifier is defined in the global scope.

```
C:\Users\Ore\Downloads\javacc-5.0\bin>java Cal C:\Users\Ore\Documents\Compiler\Test\test1.cal
The program was parsed successfully.
--AST--
Program
Main
  VariableDecl
    Id
    Type
  Statement
  Assignment
    Id
    PlusOperator
    Number
    Number
--Symbol Table--
The scope is: main
(variable, integer, i)
The scope is: global
```

```
1 main
2 begin
3   variable i:integer;
4   i := 1 + 1;
5 end
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```

public Object visit(ASTId node, Object data)
{
    SimpleNode parentNode = (SimpleNode)node.jjtGetParent();
    String dataType = parentNode.toString();

    if(dataType != "ConstantDecl" && dataType != "VariableDecl" && dataType != "FuncBody")
    {
        String value = (String) node.jjtGetValue();
        boolean check = symTable.detectScope(scope,value);
        if (check)
        {
            String type = symTable.getDataType(scope,value);
            if(type.equals("boolean")) return DataType.BooleanOperator;
            if(type.equals("integer")) return DataType.Number;
            else
            {
                return DataType.TypeUnknown;
            }
        }
        else
        {
            boolean global = symTable.detectScope("global",value);
            if (global) {
                String type = symTable.getDataType("global", value);
                if(type.equals("boolean")) return DataType.BooleanOperator;
                if(type.equals("integer")) return DataType.Number;
                else
                {
                    return DataType.TypeUnknown;
                }
            }
        }
        declared = false;
    }
    return DataType.TypeUnknown;
}

```

If it isn't defined, an appropriate error message is shown:

```

--Semantic Analysis--
Problem: i was assigned value of wrong type.
Expected TypeUnknown but found Number.

You are using a variable, constant, or function that has not been declared yet.

All binary operations for plus and minus are legal.
All comparisons are legal.
All logical comparisons for AND and OR are legal.
There are no duplicate variables.

--Intermediate Code Representation--
main:
    mt1 = 1 + 1
    i = mt1
C:\Users\Ore\Downloads\javacc-5.0\bin>

```

In my *ASTId* checking for appropriate types also allows me to check the child nodes of our operators are of the right type, e.g. the following would produce an error message:

```

--Semantic Analysis--
Problem: incorrect type used for: PlusOperator.
Expected two integers but got: Number and BooleanOperator.

Problem: c was assigned value of wrong type.
Expected Number but found TypeUnknown.

Problem: incorrect type used for: PlusOperator.
Expected two integers but got: Number and BooleanOperator.

All comparisons are legal.
All logical comparisons for AND and OR are legal.
There are no duplicate variables.
All variables, constants, and functions are declared before being used.

--Intermediate Code Representation--
main:
    a = 1
    b = true
    c = 1
    mt1 = a + b
    c = mt1
C:\Users\Ore\Downloads\javacc-5.0\bin>

```

- Is no identifier declared more than once in the same scope?

For this I used my *dedups* method that I created in my symbol table in my *ASTProgram* node. If a variable is declared more than once in the same scope, the appropriate message is outputted.

```
--Semantic Analysis--
Problem: i has already been defined in the main.

All binary operations for plus and minus are legal.
All comparisons are legal.
The left hand side of assignments are valid.
All logical comparisons for AND and OR are legal.
All variables, constants, and functions are declared before being used.

--Intermediate Code Representation--
main:
    mt1 = 1 + 1
    i = mt1
```

```
1 main
2 begin
3     variable i:integer;
4     variable i:integer;
5     i := 1 + 1;
6 end
7
8
9
10
11
12
13
14
```

- Is the left-hand side of an assignment a variable of the correct type?

The assignment operator has two children, the first child is on the left-hand side and the second child is on the right-hand side. We check if the two children are of the same type, if they aren't an appropriate error message is show.

```
public Object visit(ASTAssignment node, Object data) {
    SimpleNode node1 = (SimpleNode) node.jjtGetChild(0);
    SimpleNode node2 = (SimpleNode) node.jjtGetChild(1);
    DataType t1 = (DataType) node1.jjtAccept(this, data);
    DataType t2 = (DataType) node2.jjtAccept(this, data);

    if(t1 != t2)
    {
        assign = false;
        System.out.printf("\nProblem: %s was assigned value of wrong type.\nExpected %s but found %s.\n", node1.value, t1, t2);
    }

    node.childrenAccept(this, data);
    return DataType.Assignment;
}
```

```
--Semantic Analysis--
Problem: i was assigned value of wrong type.
Expected Number but found BooleanOperator.

All binary operations for plus and minus are legal.
All comparisons are legal.
All logical comparisons for AND and OR are legal.
There are no duplicate variables.
All variables, constants, and functions are declared before being used.

--Intermediate Code Representation--
main:
    i = true

C:\Users\Ore\Downloads\javacc-5.0\bin>
```

```
1 main
2 begin
3     variable i:integer;
4     i := true;
5 end
6
7
8
9
10
11
12
13
14
15
16
```

- Are the arguments of an arithmetic operator the integer variables or integer constants?

Both the plus and minus operator have two child nodes, I simply just check to see if both are both a Number. If either of them is not a number, then the appropriate message is shown.

- Are the arguments of a boolean operator boolean variables or boolean constants?
- Boolean operators will have two child nodes, if both child nodes are a boolean operator, then you return a boolean operator datatype, otherwise you output the appropriate error message.

```

public Object visit(ASTAndCond node, Object data) {
    SimpleNode node1 = (SimpleNode)node.jjtGetChild(0);
    SimpleNode node2 = (SimpleNode)node.jjtGetChild(1);
    DataType t1 = (DataType)node1.jjtAccept(this, data);
    DataType t2 = (DataType)node2.jjtAccept(this, data);

    if((t1 == DataType.BooleanOperator) && (t2 == DataType.BooleanOperator))
    {
        return DataType.BooleanOperator;
    }

    logical = false;
    System.out.printf("\nProblem: incorrect types used for: %s.\nExpected two boolean operators but got: %s and %s.\n", node,t1,t2);
    return DataType.TypeUnknown;
}

```

For comparison operators, I check if both child nodes are of type number, if they are then you return a boolean operator. If they aren't then you output an error message.

```

public Object visit(ASTGreaterThan node, Object data) {
    SimpleNode node1 = (SimpleNode)node.jjtGetChild(0);
    SimpleNode node2 = (SimpleNode)node.jjtGetChild(1);
    DataType t1 = (DataType)node1.jjtAccept(this, data);
    DataType t2 = (DataType)node2.jjtAccept(this, data);

    if((t1 == DataType.Number) && (t2 == DataType.Number))
    {
        return DataType.BooleanOperator;
    }

    compop = false;
    System.out.printf("\nProblem: incorrect types used for: %s.\nExpected two boolean operators but got: %s and %s.\n", node,t1, t2);
    return DataType.TypeUnknown;
}

```

3.5 Intermediate Representation:

Three-address code is a type of intermediate code which is easy to generate. It can be easily converted to machine code. In three-address code there is at most one operator on the right-hand side of an instruction.

To implement IR, I created a ThreeAddressCodeVisitor class that visits every node in my abstract syntax tree. It is like what I did in my TypeCheckVisitor class.

Three-Address Code is built on the concepts of addresses and instructions. An address can be a *name* or a *constant*. The names can be a temporary variable created by the compiler, which will be removed in subsequent optimization process

- Arithmetic Instructions

$x = y \text{ op } z$ – where op is a binary arithmetic (e.g. =, -, *, /) or binary logical (e.g. &&, ||).

To implement arithmetic operations, I created a temporary variable known as *tmp*, which I set to 1. Any time the visitor visits a node that has an operator, *tmp* is incremented:

```

public class ThreeAddressCodeVisitor implements CalVisitor{

    public static boolean activeLabel = false;
    public static int labelCounter = 1;
    public static int tmp = 1;
}

```

```

public Object visit(ASTPlusOperator node, Object data) {
    String node1 = (String)node.jjtGetChild(0).jjtAccept(this, data);
    String node2 = (String)node.jjtGetChild(1).jjtAccept(this, data);

    String mt = "mt" + tmp++;
    System.out.println("\t" + mt + " = " + node1 + " " + node.jjtGetValue() + " " + node2);
    return mt;
}

```

For *ASTAnd* and *ASTOr*, and *ASTEqualTo* I replaced *node.jjtGetValue()* with “&&”, “||”, and “==” respectively.

Here is some example code to demonstrate:

```

--Intermediate Code Representation--
main:
  a = 1
  b = 1
  c = 0
  mt1 = a + b
  c = mt1
C:\Users\One\Downloads\javacc-5.0\bin>

1 main
2 begin
3   constant a:integer := 1;
4   constant b:integer := 1;
5   constant c:integer := 0;
6   c := a + b;
7 end
8
9

```

- Functions:

The arguments to procedure and function calls are defined by the *param* instructions. Parameters are placed on the stack in reverse order.

call p, n – procedure *p* takes *n* arguments.

In three-address code a function return is either:

- *return* if you return no value
- *return x* if you decide to return a value

Here is an example of a simple function to demonstrate:

```

--Intermediate Code Representation--
add:
  x = 2
  y = 2
  mt1 = x + y
  z = mt1
  return z
main:
  param y
  param x
  call add, 2
C:\Users\One\Downloads\javacc-5.0\bin>

1 integer add(x:integer,y:integer) is
2 constant x:integer := 2;
3 constant y:integer := 2;
4 variable z:integer;
5 begin
6   z := x + y;
7   return(z);
8 end
9
10 main
11 begin
12   add(x,y);
13 end

```

As you can see, when there is a call to a procedure, you first place the parameters on the stack in reverse order.

Also, as stated by the *CAL* documentation, an invoked procedure can also be assigned to an identifier, meaning that it can also be an expression. This can be represented in IR as:

y = call p, n – an invocation of function *p* that takes *n* arguments. The result of the call to *p* is returned and stored in *y*. This is implemented in my *ASTAssignment* visitor method.

Below is a simple example to demonstrate:

```

--Intermediate Code Representation--
add:
  x = 2
  y = 2
  mt1 = x + y
  z = mt1
  return z
main:
  param y
  param x
  result = call add, 2
C:\Users\One\Downloads\javacc-5.0\bin>

1 integer add(x:integer,y:integer) is
2 constant x:integer := 2;
3 constant y:integer := 2;
4 variable z:integer;
5 begin
6   z := x + y;
7   return(z);
8 end
9
10 main
11 begin
12   variable result:integer;
13   result := add(x,y);
14 end
15

```

Below is how I implemented this in my *ThreeAddressCodeVisitor*.

```

public Object visit(ASTArglist node, Object data) {
    for(int i = node.jjtGetNumChildren()-1; i >= 0; i--)
    {
        SimpleNode identifier = (SimpleNode)node.jjtGetChild(i);
        String iden = (String)identifier.jjtAccept(this, data).toString();
        printer("param " + iden);
    }
    return null;
}

public Object visit(ASTFunctionAssignment node, Object data) {
    String identifier = (String)node.jjtGetChild(0).jjtAccept(this, data);
    Node arglist = node.jjtGetChild(1);
    int argsCounter = 0;
    for(int i = 0; i < arglist.jjtGetNumChildren(); i++)
    {
        argsCounter++;
    }
    node.childrenAccept(this, data);
    System.out.println("\tcall " + identifier + ", " + argsCounter);
    return null;
}

//return value is always the second last child node in my Abstract Syntax Tree
//in three address code a function return is either
//'return' if you return no value
//or 'return x' if you decide to return a value
//the least number of child nodes can only be 3, because FuncBody -> Type(1) -> Id(2) -> FunctionReturn(3)
public Object visit(ASTFunctionReturn node, Object data) {
    SimpleNode parent = (SimpleNode)node.jjtGetParent();
    int numOfChildren = parent.jjtGetNumChildren();
    if(parent.jjtGetNumChildren() == 3)
    {
        System.out.println("\treturn");
    }
    else if(parent.jjtGetNumChildren() > 3)
    {
        SimpleNode position = (SimpleNode)parent.jjtGetChild(numOfChildren-2);
        String value = "";
        if(position.toString().equals("Id"))
        {
            value = "\treturn " + position.value;
        }
        else
        {
            value = "\treturn";
        }
        System.out.println(value);
    }
    return null;
}

```

- Branches:

name – defines a *name* as a label. A label must be defined on a line. Every *TACi* program must have a *main*: label.

goto L – Unconditional jump to label *L*.

if x relop y goto L – conditional jump where control is passed to label *L* if *x relop y* is **true** and *relop* is a binary relational operator (e.g. *>*, *>=*, *==*, *!=*, *<=*, etc.).

Below is an example of a simple while loop to demonstrate.

```
--Intermediate Code Representation--
main:
  sum = 0
  j = 0
L2:
  if j < 10 goto L3
  goto L4
L3:
  mt1 = sum + 1
  sum = mt1
  mt2 = j + 1
  j = mt2
  goto L2
L4:
  sum = 2

C:\Users\Ore\Downloads\javacc-5.0\bin>

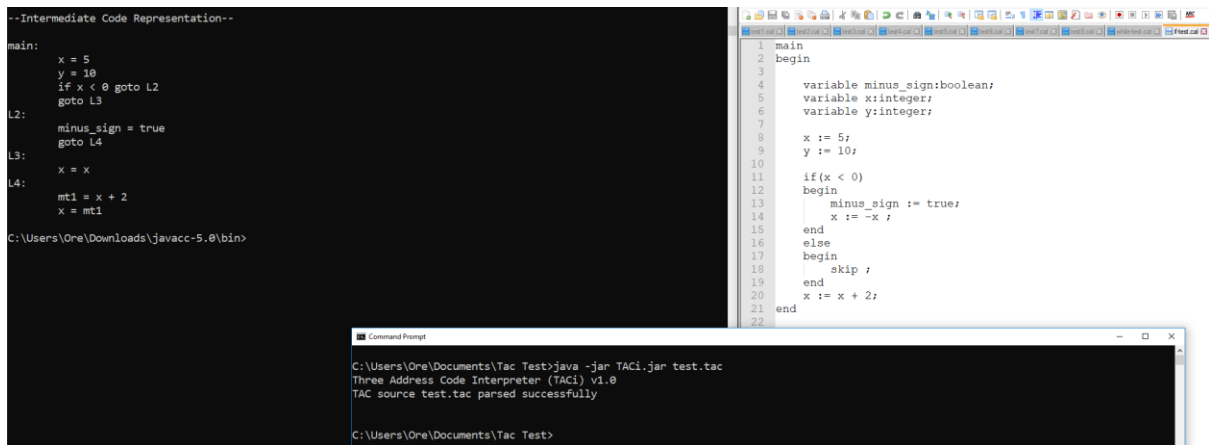
C:\Users\Ore\Documents\TAC Test> java -jar TACi.jar test.tac
Three Address Code Interpreter (TACi) v1.0
TAC source test.tac parsed successfully
C:\Users\Ore\Documents\TAC Test>
```

Here is how I implemented it in my *ThreeAddressCodeVisitor* class:

```
public Object visit(ASTConditionStm node, Object data) {
    String value = (String)node.jjtGetValue();
    String label;
    String ifOrWhile;
    String gotoLabel;
    String exitLabel;

    if(value.equals("while"))
    {
        label = "L" + LabelCounter;
        isLabelActive(label + ":");
        gotoLabel = "L" + LabelCounter;
        LabelCounter++;
        ifOrWhile = (String)node.jjtGetChild(0).jjtAccept(this, data);
        printer("if " + ifOrWhile + " goto " + gotoLabel);
        exitLabel = "L" + LabelCounter;
        LabelCounter++;
        printer("goto " + exitLabel);
        isLabelActive(gotoLabel + ":");
        for(int i = 1; i < node.jjtGetNumChildren(); i++)
        {
            node.jjtGetChild(i).jjtAccept(this, data);
        }
        printer("goto " + label);
        isLabelActive(exitLabel + ":");
        return null;
    }
}
```

Below is an example of a simple program with a simple if statement:



The screenshot displays three windows illustrating the compilation and execution of a simple program with an if statement.

Intermediate Code Representation:

```
--Intermediate Code Representation--
main:
  x = 5
  y = 10
  if x < 0 goto L2
  goto L3
L2:
  minus_sign = true
  goto L4
L3:
  x = x
L4:
  mt1 = x + 2
  x = mt1
C:\Users\One\Downloads\javacc-5.0\bin>
```

Test Code:

```
1 main
2 begin
3
4   variable minus_sign:boolean;
5   variable x:integer;
6   variable y:integer;
7
8   x := 5;
9   y := 10;
10
11  if(x < 0)
12  begin
13    minus_sign := true;
14    x := -x ;
15  end
16  else
17  begin
18    skip ;
19  end
20  x := x + 2;
21
22  end
```

Command Prompt:

```
C:\Users\One\Documents\Tac Test>java -jar TACi.jar test.tac
Three Address Code Interpreter (TACi) v1.0
TAC source test.tac parsed successfully
C:\Users\One\Documents\Tac Test>
```

Here is how I implemented it in my ThreeAddressCodeVisitor:

```
else if(value.equals("if"))
{
    if(node.jjtGetNumChildren() > 2)
    {
        label = "L" + LabelCounter;
        LabelCounter++;
        gotoLabel = "L" + LabelCounter;
        LabelCounter++;
        exitLabel = "L" + LabelCounter;
        LabelCounter++;
        ifOrWhile = (String)node.jjtGetChild(0).jjtAccept(this, data);
        printer("if " + ifOrWhile + " goto " + label);
        printer("goto " + gotoLabel);
        isLabelActive(label + ":");
        node.jjtGetChild(1).jjtAccept(this,data);
        printer("goto " + exitLabel);

        isLabelActive(gotoLabel + ":");
        node.jjtGetChild(2).jjtAccept(this,data);
        isLabelActive(exitLabel + ":");
    }
    else
    {
        label = "L" + LabelCounter;
        LabelCounter++;
        gotoLabel = "L" + LabelCounter;
        LabelCounter++;
        ifOrWhile = (String)node.jjtGetChild(0).jjtAccept(this, data);
        printer("if " + ifOrWhile + " goto " + label);
        printer("goto " + gotoLabel);
        isLabelActive(label + ":");
        node.jjtGetChild(1).jjtAccept(this,data);

        isLabelActive(gotoLabel + ":");
    }
    return null;
}
```


References for assignment 2:

https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm

<https://www.geeksforgeeks.org/three-address-code-compiler/>

<https://www.computing.dcu.ie/~davids/courses/CA4003/taci.pdf>

https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_Semantic_Analysis_2p.pdf

https://www.computing.dcu.ie/~davids/courses/CA341/CA341_Introduction_2p.pdf

<https://www.geeksforgeeks.org/hashtable-in-java/>

http://cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:compi:comp_sesion21_aux2.pdf

https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_JJTreeVid/index.html

References for assignment 1:

https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_JavaCC_2p.pdf

https://www.computing.dcu.ie/~davids/courses/CA4003/CA4003_JavaCC_2p.pdf

<https://www.computing.dcu.ie/~hamilton/teaching/CA423/notes/javacc4.pdf>

<https://www.computing.dcu.ie/~hamilton/teaching/CA448/notes/JavaCClex4.pdf>

<https://stackoverflow.com/questions/47122927/how-to-eliminate-this-left-recursion>