CA4003 Compiler Construction

Part 2 Semantic Analysis and Intermediate Representation for the basicL Language

Eoin Murphy 11487358

I declare that this material, which I now submit for assessment, is entirely my own work and has notbeen 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. I understand that plagiarism,collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, 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.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

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. I have read and understood the referencing guidelines found at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.

I understand that I may be required to discuss with the module lecturer/s the contents of this submission.

Abstract Syntax Tree

In generating my Syntax tree I had to make a couple of modifications to my original grammar. The first change I had to make was to separate Addition and multiplication Expressions so I could create a node for each type. After this I found a problem with my grammer where any if statement followed by a function would result in a Mult node in the tree rather then a Identifier node This was due to how I originally wrote the condition rule to exploit the fact if an expression can match a function. After the changes I made to the expression rule the function rule was found in fragment and as such when a fragment was matched it resulted in a mult node.

To fix this I created a Function call production rule and added it into the main expression rule as a or. Unfortunately this caused a choice conflict with the Identifier in fragment so I had to used a lookahead of 2 for expression as I could not find another solution.

```
//fixed expression to allow for the function call in condition to work properly!
void Expression() : {}
{
  LOOKAHEAD(2)
  FunctionCall()
  | AddExpression()
}

void AddExpression() : {}
{
  (
    MultExpression() ( AddOp() MultExpression() )*
  ) #Add(>1)
}

void MultExpression() : {}
{
  (
    Fragment() ( MultOp() Fragment() )*
  ) #Mult(>1)
}

void AddOp() #AddOp : {}
{
    <PLUS_SIGN> { jjtThis.value = token;}
  | <MINUS_SIGN> { jjtThis.value = token;}
}

void MultOp() #MultOp : {}
{
    <MULT_SIGN> { jjtThis.value = token;}
  | <DIV_SIGN> { jjtThis.value = token;}
}

void Fragment() : {}
{
    Indentifier() //[ Args_list() ] -- removed as reults in mult node for function calls
  | Bool()
  | Number()
  | Real()
  | <LEFT_BRACKET> Expression() <RIGHT_BRACKET>
```

I also created additional production rules for the tokens that I would need to return their value to put into the symbol tree. I also split the function declerations and the functions body into two separate rules so I could make nodes for each. This was to make it easier to handle a functions scope in the symbol tree and also to help in checking function calls had to correct parameters.

```
void Function_Decl() #Function_Decl : {}
{
  ( Type() Indentifier() <LEFT_BRACKET> Param_List() <RIGHT_BRACKET> )
  ( Function() )
}

void Function() #Function_body   : {}
{
  <BEGIN>
  ( Decl() )*
  ( Statment() <SEMICOLON> ) *
  (<RETURN> ( Expression() | {} ) <SEMICOLON>)
  <END>
}
```

Example of AST for sum_primes.bl:

C:\Windows\system32\cmd.exe

```
C:\Users\Eoin\Documents\GitHub\Complier_Construction_CA4003\Assignment_two>java
BasicL sum_primes.bl
Abstract Syntax Tree:
 Program
  Decl
   Ident_list
    Id
   Type
  ConstDecl
   Id
   Type
   Num
  Function_Decl
   Type
   Id
   Params
    Id
    Type
   Function_body
    Decl
     Ident_list
      Id
     Type
     Ident_list
      Id
     Type
    Condition
     Id
     BoolOpp
     Num
     BoolOpp
     Id
     BoolOpp
     Num
    Assign
     Id
     Bool
    Assign
     Id
     Num
    Assign
     Id
     Bool
    Condition
     Id
     BoolOpp
     Id
     BoolOpp
     Id
     BoolOpp
     Bool
    Condition
     Add
      Id
      AddOp
      Mult
       Mult
        Id
        MultOp
        Id
       MultOp
       Id
     BoolOpp
     Num
    Assign
     Id
     Bool
    Assign
     Id
     Mult
      Id
      MultOp
      Num
    Assign
     Id
     Add
      Id
      AddOp
      Num
    Id
  Main
   Decl
    Ident_list
```

Symbol Table & Semantic Analysis

To generate the symbol table I used a visitor which also performs the Semantic checks rather then doing it directly in the production rules. The reason for this was the fact I could make us of the SimpleNode class to help in navigation of the tree and give me more control over how the table was generated.

The data structure I used for the symbol table that allowed for scope was:

HashMap<String,HashMap<String,STC>>

Where the first key is the scope of the symbol and the key of the inner HashMap is the symbols identifier which uses my STC as its value for that identifier.

The first step of the visitor is when it visits the program node it creates a new symbol table and then visits all the child nodes for that node.

```
public Object visit(ASTProgram node, Object data)
{
    //Add a new Node
    ST.put(scope,new HashMap<String,STC>());

    //vistit all nodes in the program
    node.childrenAccept(this,data);
```

When the visitor visits any decleration nodes (Delcl,ConstDecl) it creates the STC for each identifier and then adds them to the table for the current scope. Before it adds them to the table it checks to see if they are already found in the table for the current scope if so it returns an error to the user saying it is already declared:

```
------Symantic Analysis------

Identifier: result
        Already declared in scope: Program
Error Line: 7 Column: 5
```

```
6    var result:int;
7    var result:int;
8    const N:int = 100;
9
```

In my visitor the scope is changed when it visits a child node of program that is not a deceleration, the new scopes name is the image of the child node identifier. And once it has finished visiting the node the scope is changed back to previous one.

Example Symbol table output for sum_prime.bl:

```
--------Symbol Table---------

Scope: Program
        ID: result| DataType: Var| Type: int| Values: {}
        ID: N| DataType: Const| Type: int| Values: {fragment=100}
        ID: is_prime| DataType: Function| Type: bool| Values: {}
Scope: Main
        ID: i| DataType: Var| Type: int| Values: {}
        ID: sum| DataType: Var| Type: int| Values: {}
Scope: is_prime
        ID: res| DataType: Var| Type: bool| Values: {}
        ID: x| DataType: ParamVar| Type: int| Values: {}
        ID: i| DataType: Var| Type: int| Values: {}
```

Symantec Checks

The STVisitor performs the following symantec checks:

- Is every identifier declared within scope before its is used?
- Is every identifier declared within scope before its is used?
- is the left-hand side of an assignment a variable of the correct type?
- Is the program trying to overwrite a constant?
- are condition parameters of boolean variables?
- Is there a function for every invoked identifier?

To show the Symantec check results I created a version of the sum_primes.bl test file called sum_primes_broken.bl that contains one of each of these errors to highlight the type of symantic checks I performed.

Symantic Analysis of sum_primes_broken.bl:

```
------Symantic Analysis------

Identifier: result
        Already declared in scope: Program
Error Line: 7 Column: 5

Invalid type assign:
        res:bool <- 1:int
Error Line: 26 Column: 20

Cannot Asign to constant: N
Error Line: 8 Column: 7

Invalid type assign:
        i:int <- false:bool
Error Line: 72 Column: 8

Identifier: is_primeinvalid
        Not declared in scope: Main Or Program
Error Line: 73 Column: 8

Unreachable Condition: is_prime_int
         Not a Boolean Identifier
Error Line: 76 Column: 8

Identifier: invalidsum
        Not declared in scope: Main Or Program
Error Line: 78 Column: 14

--------Symbol Table---------

Scope: is_prime_int
        ID: res| DataType: Var| Type: bool| Values: {}
        ID: x| DataType: ParamVar| Type: int| Values: {}
        ID: i| DataType: Var| Type: int| Values: {}
Scope: Program
        ID: result| DataType: Var| Type: int| Values: {}
        ID: is_prime_int| DataType: Function| Type: int| Values: {}
        ID: N| DataType: Const| Type: int| Values: {fragment=100}
        ID: is_prime| DataType: Function| Type: bool| Values: {}
Scope: Main
        ID: i| DataType: Var| Type: int| Values: {}
        ID: sum| DataType: Var| Type: int| Values: {}
Scope: is_prime
        ID: res| DataType: Var| Type: bool| Values: {}
        ID: x| DataType: ParamVar| Type: int| Values: {}
        ID: i| DataType: Var| Type: int| Values: {}

Errors: 7

C:\Users\Eoin\Documents\GitHub\Complier_Construction_CA4003\Assignment_two>
```

```
var result:int;
var result:int; --ERROR 1
const N:int = 100;

    res:= 1; --ERROR 2
    i := i * 1;

    var i:int, sum:int;
    N := 101; --ERROR 3

sum := 0
i := false; --ERROR 4
i := is_primeinvalid (i); --ERROR 5

begin
  if is_prime_int (i) --ERROR 6
  then
    sum := invalidsum +1;  --ERROR 7
  else
```

Intermediate Representation

For generating the 3 address code I first created a class Quadruple to hold the data for each instuction. The 3 address code I generate uses the same instruction sets as those in the notes.

The 3-address code generation is done through a visitor that is called from STVisitor only when no errors are found after the symantic checks, it passes the symbol table in as its data in case we need to work with it:

```
if(numErrors > 0)
{
  System.out.println("\nErrors: "+numErrors);
}
else
{
  ThreeAddressVisitor tv = new ThreeAddressVisitor();
  node.jjtAccept(tv,ST);
}
```

I again used a HashMap as my data structure to hold the address code in the form of:

HashMap<String,Vector<Quadruple>>

Where the key is the block label and the value is a vector of Quadruples.

The visitor starts at program and visits each child node and where there are any assignments or expressions it generates the instruction sets for them and creates temporary variables where needs be.

```
public Object visit(ASTAssign node, Object data)
{
  Vector<Quadruple> temp = addrCode.get(label);
  if(temp == null)
    temp = new Vector<Quadruple>();
  //assign is just a copy instuction generate any nessacry temp vars mult and add
  Quadruple decl = new Quadruple(
      "=",
      (String)node.jjtGetChild(0).jjtAccept(this,null),
      (String)node.jjtGetChild(1).jjtAccept(this,null)
    );
  temp.add(decl);
  addrCode.put(label,temp);
  return null;

}

public Object visit(ASTAdd node, Object data)
{
  Vector<Quadruple> temp = addrCode.get(label);
  if(temp == null)
    temp = new Vector<Quadruple>();

  curTempCount++;
  String tempName = "t"+curTempCount;
  Quadruple add = new Quadruple(
      (String)node.jjtGetChild(1).jjtAccept(this,null),
      tempName,
      (String)node.jjtGetChild(0).jjtAccept(this,null),
      (String)node.jjtGetChild(2).jjtAccept(this,null)
    );
  temp.add(add);
  addrCode.put(label,temp);
  return tempName;
}

public Object visit(ASTMult node, Object data)
{
  Vector<Quadruple> temp = addrCode.get(label);
  if(temp == null)
    temp = new Vector<Quadruple>();

  curTempCount++;
  String tempName = "t"+curTempCount;
  Quadruple mult = new Quadruple(
      (String)node.jjtGetChild(1).jjtAccept(this,null),
      tempName,
      (String)node.jjtGetChild(0).jjtAccept(this,null),
      (String)node.jjtGetChild(2).jjtAccept(this,null)
    );
  temp.add(mult);
  addrCode.put(label,temp);
  return tempName;
}
```

```
public Object visit(ASTConstDecl node, Object data)
{
  Vector<Quadruple> temp = addrCode.get(label);
  if(temp == null)
    temp = new Vector<Quadruple>();
  int numChildren = node.jjtGetNumChildren();
  for(int i = 0; i < numChildren; i=i+3)
  {
    Quadruple decl = new Quadruple(
        "=",
        (String)node.jjtGetChild(i).jjtAccept(this,null),
        (String)node.jjtGetChild(i+2).jjtAccept(this,null)
      );
    temp.add(decl);
  }
  addrCode.put(label,temp);
  return null;
```

The labels in this representation are generated when a different scope has been entered we also then add this label to a map where the key is the scopes node name This allows us to use the correct label in our GOTO instructions:

```
public Object visit(ASTFunction_Decl node, Object data)
{
  prevLabel = label;
  label = "L"+(labelCount+1);

  //add the lable to the map so we can get funcions label when using a goto function
  jumpLabelMap.put((String)node.jjtGetChild(1).jjtAccept(this,null) , label);
```

Along with this at the end of every function deceleration visit we add a return instruction for

```
Vector<Quadruple> temp = addrCode.get(label);
if(temp == null)
  temp = new Vector<Quadruple>();

Quadruple retrun = new Quadruple(
    "return",
    ""
  );
temp.add(retrun);
addrCode.put(label,temp);

label = prevLabel;
labelCount++;
```

When we visit a functionCall node we generate a call instuction with the paramters idname generated in arg_list:

```
public Object visit(ASTFunctionCall node, Object data)
{
  Vector<Quadruple> temp = addrCode.get(label);
  if(temp == null)
    temp = new Vector<Quadruple>();

  Quadruple call = new Quadruple(
      "call",
      "",
      (String)node.jjtGetChild(0).jjtAccept(this,null),
      (String)node.jjtGetChild(1).jjtAccept(this,null)
    );

  temp.add(call);
```

And then create our goto insturction using the function name to fin the label in our map:

```
  Quadruple gt = new Quadruple(
      "goto",
      jumpLabelMap.get((String)node.jjtGetChild(0).jjtAccept(this,null)),
      ""
    );
  temp.add(gt);

  addrCode.put(label,temp);
  return null;
}
```

The only problem with my 3-address code is it doesn't handle instructions for While and if statements this was because I simply didn't have time to get it implemented.

Example 3-address code for sum_primes.bl:

```
-------IR 3-address code-------

L1
[ = , 100 ,  , N ]
L2
[ = , true ,  , res ]
[ = , 2 ,  , i ]
[ = , true ,  , res ]
[ / , x , i , t3 ]
[ * , t3 , i , t2 ]
[ - , x , t2 , t1 ]
[ = , false ,  , res ]
[ * , i , 1 , t4 ]
[ = , t4 ,  , i ]
[ + , i , 1 , t5 ]
[ = , t5 ,  , i ]
[ return ,  ,  ,  ]
L3
[ = , 0 ,  , sum ]
[ = , sum ,  , i ]
[ param1 , i ,  ,  ]
[ call , is_prime , param1 ,  ]
[ goto ,  ,  , L2 ]
[ + , sum , 1 , t6 ]
[ = , t6 ,  , sum ]
[ = , sum ,  , result ]

C:\Users\Eoin\Documents\GitHub\Complier_Construction_CA4003\Assignment_two>
```