Name: Ruben Vasconcelos

Programme: CASE4

Module Code: CA4003

Assignment Title: Semantic Analysis and Intermediate Representation for the basicL Language

Submission Date: 13-Dec-2015

I declare that this material, which I now submit for assessment, is solely my own work.

Ruben V.

Building the AST:

I set the option NODE\_DEFAULT\_VOID=true;

Then I went though all my grammar's rules in order to decide which ones should return something, I decided to add unique nodes for operators and cosnt assign as I thought this would make it easier to understand and come in andy during my semantic checks.

Here's an example of my AST:

Program

Var\_decl

Identifier (result)

Type (int)

Cont\_decl

Identifier (N)

Type (int)

Const\_assig (=)

Number (100)

Function (is\_prime)

I realised afterwards that there might have been a bit too much information, but I decided to keep it as this really simplified the semantic checking process.

Building the Symbol table:

Initially I wanted to use a Stack like data structure but after some thinking i thought it would become very awkward to use when implementing the semantic checks so decided to go with a hashmap. I also decided to create a symbol table for each function, main and global in order to keep track of variable's scope. All of the individual symbol tables were added to into one single symbol table.

I identified the locations where the scope changes, in our grammar and added the following code:

ST.put(scope, local\_ST);

local\_ST = new Hashtable();

the above inserts the current local ST into the general one and clears it so that it's ready for the next scope and set of symbols to be inserted.

The 3 locations that I applied this to are:

The first symbol table is added after the const declaration in my “program” rule, with the key/scope “global”. And it contains all the variables/consts declared at the start of the program.

I could have added the “functions” ST insertion to the program rule as well but i wanted to use the function's name as my key/scope for example “is\_prime”, i could have returned it from the function rule but I thought it looked a lot cleaner and easier to understand this way. The local symbol table is inserted into the general one after the token <END> is parsed. By doing this it guaranteed that all the required entries will be already in the local symbol table before it reaches the end token.

The third place that required this was in the main\_prog rule. And similarly to the ST for a function this could have been done in but as it was the exact same as before i decided to add it to my main\_prog rule for consistency.

For adding the actual entry into the in\_scope symbol tables, using the example given by the lecture I decided to first write and STC class object, with the following attributes.

**String type\_qualifier** for storing the values “const” or “var”

**String type** for holding the primary type “int”, “bool” or “void”

**String name** used to store the the variables ID

And the following booleans used in the semantic checks

**Boolean pre\_declared**

**Boolean written\_to**

**Boolean read\_from**

Even though I could have also added the functions id to the ST, I decided that there was no real need or use for it as all the functions are defined in the “global” scope.

So now that I had my ST being implemented with scope and my STC class defined I was ready to start adding entry into it.

So the entries were added into a local\_ST and this was then inserted into the general symbol table or “ST” at the appropriate time.

Adding const variables was straightforward and similar to the example given by the lecture using the constructor defined in the STC class the only difference was the amount of params being passed in:

local\_ST.put(id, new STC(type\_qualifier, id, type, local\_ST.containsKey(id)));

local\_ST.containsKey(id) returns the boolean value used for the check of previously declared variables within the cope. So If there is an entry with a matching id then **pre\_declared = true else = false.**

Entering vars into the symbol table was a bit more complex. Because there could be multiple var ids separated by a comma. This is taken care by the ident\_list rule and only afterwards we find out their type.

t = <VAR> ident\_list(t.image) <COLON> type = type()

so what i decided to do was call ident list and pass down “var” and insert them there.

So I had to create a new constructor in the STC class without type and also decided to add a small function called addType that was to be used afterwards once the type was known.

So we go into ident\_list the entries get added to the local\_ST using the new constructor:

local\_ST.put(id, new STC(t, id, local\_ST.containsKey(id)));

Nothing new so far until we return to var\_decl, then I iterate through the local\_ST and look for any entry with a null type(new entries added in ident\_list). And insert the value return from type() into each of them.

Enumeration en = local\_ST.keys();

while (en.hasMoreElements())

{

String tmp = (String)en.nextElement();

STC tmp2 = (STC)local\_ST.get(tmp);

if (tmp2.type == null)

tmp2.addType(type);

}

I also decided to add the params in the function definition as i thought this would be required for checking if an ID was defined or not before it was used. This was also pretty straightforward and similar to what i done for consts, except their are normal vars.

local\_ST.put(id, new STC("var", id, type, local\_ST.containsKey(id)));

Here's an example of my ST output.

Symbol Table:

Scope --> main

<---------------->

id = i

type qualifier = var

type = int

<---------------->

id = sum

type qualifier = var

type = int

Scope --> global

<---------------->

id = result

type qualifier = var

type = int

The Semantic Checks:

Once again using the Type check example supplied by the lecture I decided to create a new SematicVisitor to step through my AST and perform the checks. I also had to create a public enum DataType for the possible return types from visiting the individual nodes.

There's a few types that don't really serve much purpose but i decided to leave them in like the lecture did in his example as it would make the visitor file easier to ready afterwards instead of just returning null or the same thing afterwards plus i was unsure which one would actually become helpful.

The more useful ones that I want to point out are:

Bool\_val,

Number,

A number is used in arithmetic expressions and is any var of type “int” or an actual number.

A bool\_val is used for boolean expressions, and it could be either true or false.

Once I had all the operations returning the appropriate Data types , it became trivial performing the checks as most of them is simply making sure that the operations return the types or using the ST to make sure the correct type is being assigned to vars and consts.

I also used 2 hashtables and a linked list to keep track of function calls the number of expected arguments and the invoked id's without a function.

Instead of printing the all the correct message every time i decided to just print a message if they all passed or each individual failure.

Here's the output for the input file supplied by the lecturer:

PASS: Every identifier was declared within scope before it was used.

PASS: No identifier was declared more than once in the same scope.

PASS: All variables were assignment a variable of the correct type

PASS: All consts were assignment a variable of the correct type.

PASS: The arguments of all arithmetic operations were integers.

PASS: The arguments of all boolean operations were booleans.

PASS: The arguments of all relational operations were integers

PASS: The arguments of all equality operations were of the same type.

PASS: There is a function for every invoked identifier.

PASS: All functions were called with the correct number of arguments.

Warning: Never read from "result".

PASS: Every variable was written to.

PASS: All declared functions were called

The 3 address code.

I created another visitor to step through the AST and create the 3 address code.

I used the notes example to figure out how to parse the tree appropriately then i made sure every operation was being stored in a temp and returned appropriately onto its parent node.

The only complex part was implementing the goto jumps and labels correctly.

Here's a sample output:

N = 100

goto Main

func is\_prime

t1 := x = 1

t2 := x = 2

t3 := t1 or t2

ifFalse t3 goto LB1

res := true

goto LB2

LB1: