WORKSHOP 8 – SEMANTIC ANALYSIS

PURPOSE OF THE WORKSHOP

The purpose of this workshop s to give you a chance to understand the semantic analysis process using a Visitor Pattern. As part of this week's work you will develop implement some of the semantic rules for the Triangle language that will enable your compiler to test the Abstract Syntax Tree created by your parser.

During this week you will also complete the creation of the Symbol Table enabling you to keep track of the types and declarations used by your source language.

YOU MUST COMPLETE LAST WEEK'S WORK BEFORE ATTEMPTING THIS WORKSHOP

Make sure you complete this week and last week's work so you can start the Code Generation next week.

PART 1: THE SEMANTIC ANALYSER FRAMEWORK

As we saw in the lecture for the visitor pattern to work we need to create a visitor that is able to visit and check all the nodes of the AST created by the parser. To ensure the consistency of this visitor we need to implement a set of interfaces which define the methods that an instance of the visitor class must have. To complement this each of the classes in our syntax tree must have a visit method, so they can be visited.

For today's labs and the future coursework submission I have provided you with a partly functional Semantic Analyser, which include the interface methods for the visitor class, a nearly complete implementation of the Checker (again discussed in the lecture) and the updated Syntax Tree classes (which will replace those you used last week)

You should download the source package zip on Moodle now. This source package contains two folders;

ContextualAnalyser – this folder contains a nearly complete implementation of the Checker Class.

SyntaxTrees – a set of syntax tree classes, which should replace those you used last week, containing the addition visit methods required for the visitor pattern

Delete your existing SyntaxTree folder and extract the new SyntaxTree folder and the ContextualAnalyser folder into your Triangle.Compiler Folder.

Once you have extracted the files to your compiler directory first have a look within the **SyntaxTrees/Visitors** folder, here you will see all the interfaces for any class which is required to act as a visitor. Each interface represents a set of phrases, eg Declarations, Expressions, Terminals. These interfaces are complete and you do not have to do any more work to them However you should, for your final report, understand how they work and what they are used for.

Next open up any of the **SyntaxTree** classes, you will see that I have added the visit method for each class after the code you used last week. Adding these classes will not affect how the parser you created last week works, but does now give us the ability to visit each node of the completed AST.

Finally look at the code in the **ContextualAnalysis** folder, you will see that the implementation of the Checker described in the lecture is stored here. The checker is composed on partial classes, just like our parser, and each partial class, implements the correct visitor interface for a specific class of phrases.

For example, Checker – Program.cs implements IProgramVisitor and Checker – Terminals implements the interface for ILiteralVisitor, IIdentifierVisitor and IOperatorVisitor i.e all the terminals. The checker also contains an extra partial class called common.cs. This file contains a selections of helper methods which cover the internal functions of the language, hold a reference to the Symbol Table and deals with generating error messages for this section of the compiler.

Now to get the new files working with the setup you already have you will need to make some changes to your compiler.cs file.

First add the declaration

```
Checker _checker;
```

Just below where your parser is declared.

Then in your compiler's constructor instantiate the checker with an instance of your error reporter.

```
_checker = new Checker(ErrorReporter);
```

Finally in your CompileProgram method you need to add our 2nd pass to the compiler.

```
// 2nd pass
ErrorReporter.ReportMessage("Contextual Analysis ...");
_checker.Check(program);

if (ErrorReporter.HasErrors) {
ErrorReporter.ReportMessage("Compilation was unsuccessful.");
                return false;
}
```

Once you have had a look at all the classes, and added the code above, type the command **dotnet restore** at the command prompt to ensure all the project dependencies are present.

PART 2: CREATING THE SYMBOL TABLE

In your compiler folder you should still have some test files, including the very basic hi.tri source code . Try running this against your compiler by typing **dotnet run tests/programs/hi.tri** What output do you get? Why is this the case?

We saw in week 7 the functions that a symbol table should have;

- Some way of storing the information
- An enter function to add variables
- A Retrieve function to retrieve existing variables
- A open function to start a new scope block
- A close function to remove a used code scope block

Let's look at how we can implement this now.
Open IdentificationTable.cs file within the contextualAnalysis folder, this file is going to contain the representation of our SymbolTable. Currently this class is simply a skeleton with no implementation.

The first thing we need to implement is a way of storing the information in the table. We saw last week how the symbol table needs to hold a reference to the declaration of each identifier.

We can do this using a Dictionary

```
Dictionary<string, Declaration>
```

Where the string (key) will be the identifier name and the Declaration will be the AST class representing that declaration.

But this isn't quite good enough, when dealing with scope we need multiple levels, so what we actually need is a List of Dictionaries, with each entry in the list representing a new sub level.
We don't want anything adding to this list outside our SymbolTable structure so we make it read only.

```
readonly List<Dictionary<string, Declaration>> _levelStack;
```

Now we need to look at creating the functionality with in the methods so our Symbol table can actually be used.
Let's deal with the constructor first.

```
public IdentificationTable()
{
    _levelStack = new List<Dictionary<string, Declaration>>();
    _levelStack.Add(new Dictionary<string, Declaration>());
}
```

The first line here instantiates the List to hold our dictionaries. The second creates a dictionary for our first level code block. If we never use a Let statement within our code this will be the only level we use. (kind of like a monolithic block structure – see week 7)

Let's look at the Retrieve method first. Remember that this is designed so that we can get the attributes for a declaration. So for example if we want to check the type of a variable we can ask the symbol table for the declaration and then get the type, it also allows us to determine if the id has been declared before.

```
public Declaration Retrieve(string id)
{
     foreach (var level in _levelStack)
     {
          Declaration attr = null;
```

With a nested block structure that Triangle uses for scope, any variable can be used as long as it is declared in its own scope or the scopes above. With this in mind we can simply use a foreach loop to run through the table from top to bottom. (more on this later)

```
          if (level.TryGetValue(id, out attr))
          {
               return attr;
          }
```

If an ID exists at this level return it's attribute, (i.e te declaration). If it doesn't we must have gone through the entire table so return null.

```
     }
     return null;
}
```

Now let's look at the Enter method. This method is designed to allow us to add new Identifiers to the symbol table and to check if these identifiers already exist. Identifiers are always terminal symbols so we use that as our first parameter, declarations are used to declare variable so we use that as our second.

```
public void Enter(Terminal terminal, Declaration attr)
{
      attr.Duplicated = _levelStack[0].ContainsKey(terminal.Spelling);
      _levelStack[0][terminal.Spelling] = attr;
}
```

The first line, sets the declarations duplicated attribute based on whether the current scope level (the scope level at the top of our list, i.e index 0) already contains the Identifier (key) we are trying to add.
Finally we add the new declaration to the List at index 0, using the terminal spelline (i.e the identifier) as the key.

Ok now let's look at the openScope method, this is designed to open a new scope, representing a nested code block.

```
public void OpenScope()
{
    _levelStack.Insert(0, new Dictionary<string, Declaration>());
}
```

this is pretty simply, but the main thing to note is that it uses the INSERT function to put the new dictionary representing the scope at the top of the list, moving the other scopes down. This means that we always know our current closest scope is at `_levelStack[0]` and when we run through the table with a for loop (as we did in the retrieve method above) that we are always finding the most local declaration first!

Finally the closeScope method, we need this to close and get rid of a scope once it has been used. In our language this is when the current LET-IN block is finished.

```
public void CloseScope()
{
    _levelStack.RemoveAt(0);
}
```

Because we know that our most local scope is always at index 0 this makes the closeScope method very easy. We just remove whatever is currently at Index 0 and the rest of the scope dictionaries will move up to replace it.

Now try running your compiler against the hi.tri file again and see what happens. Why are we getting the same answer when we have now completed the symbol table? In the next section you will see the importance of adding identifiers to the symbol table so that they can be used and checked.

PART 3: ADDING TERMINAL TO THE SYMBOL TABLE

The PUT function used by triangle is defined in the StandardEnvironment, and it is the standard environment class that creates the function and adds it to the symbol table so we can actually use it. The problem is all the standard environment file actually does is call the visit identifier method that adds the actual function identifier to the symbol table. We haven't implemented this yet!
If you open Checker – Terminals.cs you will see that again I have given you the skeleton of the class and that the main functionality is not quite there yet.

Let's look at the VisitIdentifier method, this method will be fired whenever an identifier is reached during the pass of the Abstract Syntax tree.

```
public Declaration VisitIdentifier(Identifier identifier, Void arg)
{
     var binding = _idTable.Retrieve(identifier.Spelling);
```

The first line of our new method creates a variable called binding and asks the SymbolTable for the declaration that matches the spelling of this identifier. I.e this will be all the details required about this identifier.

```
      if (binding != null)
     {
      identifier.Declaration = binding;
     }
     return binding;
}
```

If the binding isn't null then we set this identifiers declaration to that returned from the symbol table. This means that the identifier node we have reached now contains all the details about that identifier that were made when the identifier was declared.
Finally we return the binding, if it was null we will deal with it later.

Follow the same patter to complete the VisitOperator method, it follows exactly the same pattern and should return the binding containing the info about the operator's declaration. (as discussed in the lecture)

```
public Declaration VisitOperator(Operator op, Void arg)
```

Once you have completed this try compiling the hi.tri file again.
You should now see that it compiles without errors!

PART 4: THE DECLARATIONS

While we have dealt with the terminals in the symboltable but we haven't yet implemented all the declarations. If you open the Checker – Declarations.cs file you will see that I have completed the last three declarations for you. But that to be complete the Const, Var and Sequential declarations need some more work.

Looking at the VisitConstDeclration method, we need to put the code in so that this visitor method checks the declaration in the correct way and add the right parts to the symbol table.

From the language definition the first part of a Const declaration is an Expression so we need to visit the expression first, ad like all the vist methods we pass the current point in the AST using the keyword "this". We let the expression method deal with any errors it finds while doing this.

```
ast.Expression.Visit(this);
```

Next, because it is a declaration we need to add the Identifier being used to the symbol table ,using the symbol tables Enter method.

```
_idTable.Enter(ast.Identifier, ast);
```

Next we check that the identifier is not duplicated, we have an inbuilt method in the common.cs file that helps us do this. We just need to pass it the AST.Duplicated parameter that we implemented in the previous section.

```
CheckAndReportError(!ast.Duplicated, "identifier \"%\" already
 declared",ast.Identifier, ast);
```

Finally we return null (there is nothing else to do) as if we get to this point there must have been no errors.
`return null;`

Follow the same pattern for the Var and Sequential declarations. Remember to go back and check the Abstract Syntax Rules on Moodle so that you know what nodes you should be visiting. The methods are very short and should only late a few minutes to work out what to visitor methods to call and when to add to the symbol table.

PART 5: THE LET COMMAND AND EXPRESSION

Finally, we need to deal with scope in the semantic analyser. In the triangle language we are using there are only two ways to change scope. The first is using the LET command the second is using the LET expression. In each case the scope opening and closing is defined by the Let and In commands respectively. Look at the VisitLetCommand method in the Checker –Commands.cs file. You will see that this is also incomplete.

```
public Void VisitLetCommand(LetCommand ast, Void arg)
{
      _idTable.OpenScope();

      ast.Declaration.Visit(this);
      ast.Command.Visit(this);

      _idTable.CloseScope();
      return null;
}
```

The methods themselves are very simple, all we do is open the scope using the OpenScope method at the start, then deal with the visiting the components in-between.
Finally when we are finished we close the scope. Working in this way with our symbol table ensures that we can only use the variables within the correct scope.

You shoud now be able to complete the LetExpression in the Checker – Expressions.cs file. It has a very similar structure however you will notice that it should return a TYPE. You can do this by using the
`ast.Type;` as the return statement.
Remember to refer to the Abstract Syntax so you can follow what node should be visited next.

Make sure you complete this week and last week's work so you can start the Code Generation next week.