WORKSHOP 7 – THE ABSTRACT SYNTAX

PURPOSE OF THE WORKSHOP

The purpose if this workshop is to introduce the Abstract Syntax tree as the first stage of a semantic analysis. As we saw in last week's lecture the abstract syntax tree (AST) is based on the language's Abstract Syntax. The Abstract Syntax is a cut down version of the language definition (the concrete syntax) which only describes the actual phrases and not the non-terminal in between.

The bulk of today's work will be adding the methods to create the AST. As part of today's lab you will also build a skeleton Symbol Table which you will use in your compiler.

PART 1: A WORKING SCANNER & PARSER

For today's labs and the future coursework submission I have provided you with a working version of the Scanner and Parser. You may use your own parser and scanner if you wish but I would advise you use the new version on Moodle.

You should download the source package zip on Moodle now. This source package contains a single folder, Triangle.Compiler, **which you should extract to your Triangle.NetCore.Student folder**. If you wish to keep your own compiler folder you should rename it first.

Source Package Contents –

The root folder of the package contains 4 files

StreamErrorReporter & ErrorReporter which provides an error reporting structure

StandardEnvironment which provides all the standard functions of the language

Compiler.cs – the entry point for the compiler

The Syntatic Analyzer Folder contains all the parser files; this should be familiar to you. Have a look now and compare how the parser compares to your own.

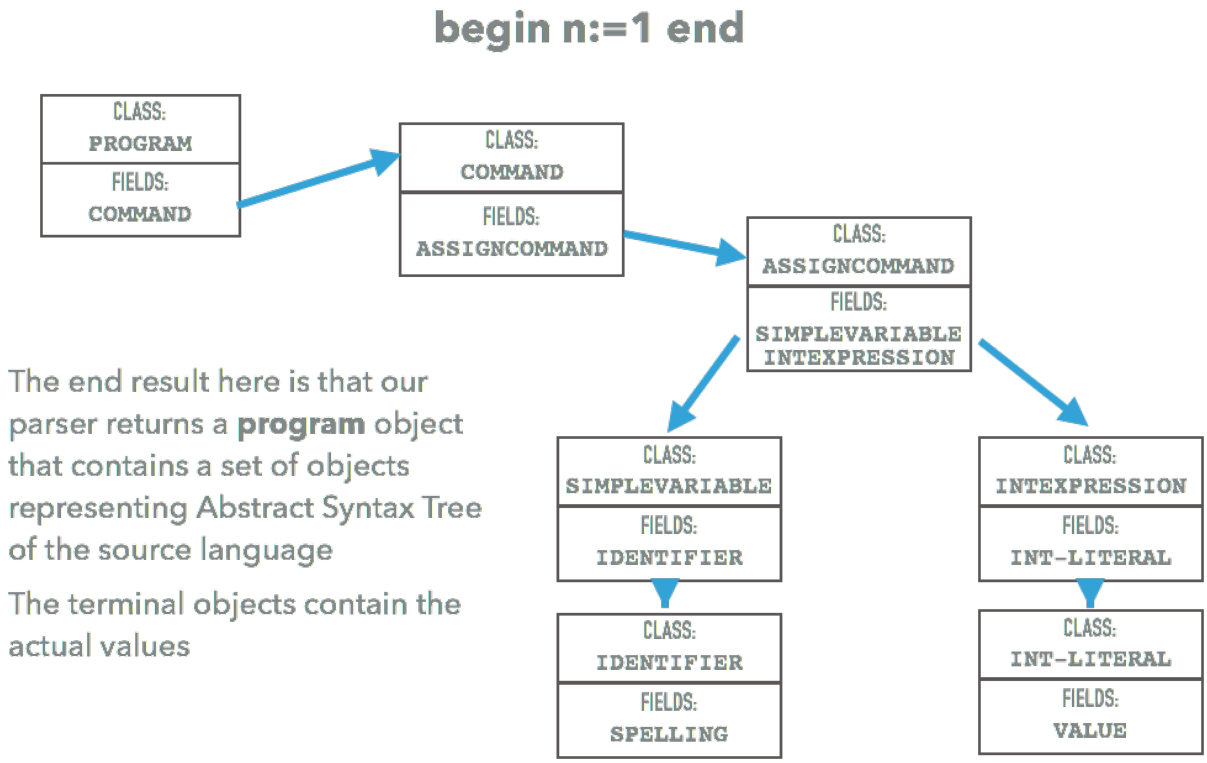
Finally, the SyntaxTrees folder contains all the files needed to represent the Abstract Syntax Tree of an application.

Once extracted type the command **dotnet restore** at the command prompt to ensure all the project dependencies are present. (do not type **dotnet run** the compiler is not complete and will not run)

PART 2: THE ABSTRACT SYNTAX

On Moodle you will find a copy of the Abstract Syntax for the reduced triangle language. This documents shows the reduced form of languages' definition. More importantly I have annotated this document for you adding a unique description for each type of statement (LetCommand, VnameExpression, etc)

As we say in the lecture last week, we can represent the Abstract Syntax Tree using an object orientated class structure.



Have a look at the Abstract Syntax document, particularly the statement descriptions, you will see that these descriptions correspond to classes within the SyntaxTree folder in the new source package. Because creating each of these classes is a tedious process I have done it for you (please remember me in your Christmas list!)

Have a look at each of these classes now and try to understand how they are laid out; you should see that each class fully represents the production rule.

For example, the IfCommand class has an expression and two commands, one for the true command and one for the false command.

You will also see I have included the following line in each of the classes

```
if (Compiler.debug) { System.Console.WriteLine(this.GetType().Name); }
```

this line is part of a debugging system that will print out the type of statement when it is created, allowing you to keep track of the AST just like you did with the parser.

PART 3: CREATING A SYNTAX TREE

Open the SyntaticAnalyzer folder, you should recognise the structure, it is the same as the structure you used in your submission.

First look at the Program.cs file. You will initially see that there is quite a big difference with the parser you submitted.

The ParseProgram() method now has a return type of Program.

```
public Program ParseProgram()
```

Instead of simply calling ParseCommand() we now create a Command object from the value returned from the ParseCommand() method

```
var command = ParseCommand();
```

and, importantly, we use this value to create a Program Object.

```
var program = new Program(command, pos);
```

Finally, we return this program object.

```
return program;
```

This is the start of a process where as the parser is going through the source code tokens it will create a set of Objects representing the AST, just like the diagram in section 2 above.

Now if you open Parser - Commands.cs you will see that it follows the same pattern. Instead of the methods being void, each parse methods will return an instance of the class representing that statement. Eg ParseSingleCommand now return an instance of the Command class.

The structure within the switch statement is the same, however we are again creating instances of the classes that represent our tree. You will see in the case for Tokenkind.Identifier, I have left a todo for you, lets walk through this now.

We know that to parse an identifier we need to call the parseIdentifier method, but when we do this we want an Identifier to be created.

```
var identifier = ParseIdentifier();
```

This won't work yet, as your ParseIdentifier won't yet return a value , but you can fix this later. The next bit of code is exactly what we have before, we check for the '(' to see if we have a CallCommand.

```
if (_currentToken.Kind == TokenKind.LeftParen)
{
     AcceptIt();
```

Now we need to deal with the actual parameters, again we want an instance of the class to be created so again we create an instance form the return value of the parseActualParameterSequence method.

```
var actuals = ParseActualParameterSequence();
```

The next bit of code is the same as before, just accept the token.

```
Accept(TokenKind.RightParen);
```

Now build the position for error checking later, again the same as before.

```
var commandPosition = new SourcePosition(startLocation, _curre
ntToken.Finish);
```

Finally create and return a call command (the type of call this is) using the identifier, the parameters and the position.

```
return new CallCommand(identifier, actuals, commandPosition);
}
```

We can now deal with the rest of the sequence, if we have an assignment statement.

```
else
{
```

We have already tried to parse the identifier so we now pass this to the Vname. (have a look at the parser-vname.cs to follow this through)

```
var vname = ParseRestOfVname(identifier);
```

accept the token

```
Accept(TokenKind.Becomes);
```

Parse the expression, creating a new instance of expression.

```
var expression = ParseExpression();
```

Again, let's build the location for any error reports,

```
var commandPosition = new SourcePosition(startLocation, _curre
ntToken.Finish);
```

Finally create and return an AssignmentCommand with the variable name, the expression and the position.

```
return new AssignCommand(vname, expression, commandPosition);
}
```

PART 4: OVER TO YOU

We have walked through one example for the commands section in the parser, you will see that I have already done this for you for the other commands. However you now need to follow the same process for all the other statement types in the AbstractSyntax. I have created the classes for you, all you need to do is create them in the right order. When you run your compiler on the same text programs you used before you should get the statement definitions being written to the console, describing the Abstract Syntax Tree of that program.

You should concentrate on this section in the lab where I am here to help you.

PART 5: THE SYMBOL TABLE

If you get finished and while you are away on assessment week you should start to develop a skeleton symbol table. As shown in the lecture you should create a skeleton class that has the following properties,

- Some way of storing the identifiers – I would suggest that you look at the C# implementations of List or Dictionaries. Remember you are using .net core when investigating this
- You need a constructor that creates an empty table
- An openScope() method that creates a new scope
- A closeScope() method that closes (removes) a used scope
- An Entry method that receives a value (instance of Terminal) and an attribute (instance of Declaration)
- And a retrieve method that receives an identifier names as a string and returns the Declaration stored if one exists.

You do not need to make this fully functional yet, it is just a skeleton that you can fill in later on.