WORKSHOP 4 – THE PARSER

PURPOSE OF THE WORKSHOP

The purpose of this week is to analyse the Triangle language and start to produce a recursive descent parser to check the syntax of our inputted code.

PART 1: MINI-TRIANGLE LANGUAGE DEFINITION

Just as last week we tested our scanner on mini-triangle, a shortened version of the Triangle language. This is the EBNF of mini triangle. Keywords and Significant characters are highlighted in RED remember these are part of the language and should be separate tokens.

Program	::=	Command
Command	::=	single-Command (**;**single-Command)*
Single-Command	::=	Identifier (:=Expression|**(**Expression**)**)
	|	**if** Expression **then** single-Command
		else single-Command
	|	**while** Expression **do** single-Command
	|	**let** Declaration **in** single-Command
	|	**begin** Command **end**
Expression	::=	primary-Expression
		(Operator primary-Expression)*
primary-Expression	::=	Integer-Literal
	|	Identifier
	|	Operator primary-Expression
	|	**(**Expression**)**
Declaration	:==	single-Declaration (**;** single-Declaration)*
single-Declaration	::=	**const** Identifier **~** Expression
	|	**var** Identifier **:** Type-denoter
Type-denoter	::=	Identifier

REMEMBER

In terms of EBNF () means optional and * means can repeat and | means or. These are NOT part of the language

PART 2: STARTING OUR PARSER.

On Moodle I have provided you with another zip file containing the skeleton for a very basic Parser. You will notice that these files have an odd class definition at the top, they all contain;

```
public partial class Parser
```

A partial class just a way of defining that separate code files all belong to the same class. At compile time any class marked as partial will be treated as one big single class. All methods of in a partial class are available to all other partial classes with the same name. Basically we are just splitting the definition of the class up into smaller pieces. (remember I said it wouldn't all fit into separate classes in the lecture).

Copy these files into your SyntaticAnalyzer folder from last week.

In your compiler.cs file change the constructor to look like this.

```
// Creates a compiler for the given source file.
Compiler(string sourceFileName)
{
      _source = new SourceFile(sourceFileName);
      _scanner = new Scanner(_source);
      _parser = new Parser(_scanner);
}
```

In your Compiler.cs file change the Main method to look like this

```
// Triangle compiler main program.
public static void Main(string[] args)
{
  var sourceFileName = args[0];
  if (sourceFileName != null)
  {
     var compiler = new Compiler(sourceFileName);
     //foreach (var token in compiler._scanner)
     //{
     // Console.WriteLine(token);
     //}
     //compiler._source.Reset(); //uncomment to reset source code.
     compiler._parser.ParseProgram();
  }
}
```

All this does is attach the scanner to the parser so the tokens discovered by your scanner can be passed on. The commented out code just removes the printouts we did last week, if you want to put these back in for testing go ahead, but remember to uncomment the _source.reset() method to go back to the start of your source for parsing.
Finally add `Parser _parser;` field just below the similar line for the scanner to declare the parser.
Now let's run this on a test file and see what happens. I have created a file for you called rubbish.tri, if you open this you will see it is a simple file with a single identifier in it.
Run your current compiler against this by typing **dotnet run rubbish.tri** in your compiler directory.

You should see a series of console printouts telling you what order your methods were called in.

PART 3: RECURSIVE DESCENT PARSING

Now open some of the parser files and have a look at the code contained in them.

Parser – Common.cs

This file contains the most basic components of the parser. It has our constructor that takes an instance of our Scanner from last week and creates an **IEnumerator called _tokens** from this which represents our token stream from the scanner. Remember that an IEnumerator is a very simple readonly array or list like object that can be Iterated through with the MoveNext() command. The Parser – Common also has a field **_currentToken** to store the current token being parsed.

Parser – Programs.cs

The Program.cs file deals with our root node (or start symbol) on our language definition, i.e program. From the lecture we saw that in a Recursive Descent Parser has a method to deal with each nonterminal symbol. Given our EBNF for Program is **Program ::= Command** You can see that all the parseProgram() method does is to set up the rest of the parse and call the method only production rule of program, which is parseCommand().

Parser – Commands.cs

The Commands.cs file deals with the command phrases. These are phrases that comply to the command production rules in our EBNF **Command ::= single-Command (;single-Command)***

From the lecture we saw that the Recursive Descent Compiler mimics the Syntax Tree through the order it calls the parsing methods, and the parsing methods themselves match the production rules.

If you look at this file you can see that the first method **ParseCommand()** represents exactly what our Command rule states, i.e that a command should contain a single command followed by 0 or more commands separated by a semi colon..

```
void ParseCommand()
{
        System.Console.WriteLine("parsing command");
        ParseSingleCommand();                                   single-Command
        while (_currentToken.Kind == TokenKind.Semicolon)
        {
                AcceptIt();                                     (;single-Command)*
                ParseSingleCommand();
        }
}
```

Parser – Vnames.cs

The vnames.cs parser contains a simple method that deals with the only production rule for Vnames, i.e identifier. While this may seem redundant, you will see why it starts to become important next week.

Parser – Terminals.cs

Finally, our terminal.cs file contains all the code required to deal with our terminal symbols. The parser only needs to deal with the very last level before the actual terminals. Remember the actual values were stored and

dealt with by the scanner, the parser only deals with the terminal Token type. Looking at the Terminal.cs file you will see that it currently contains only one method parseIdentifier.

PART 4: COMPLETING THE PARSER - COMMANDS

Within the commands.cs file's ParseSingleCommand() you will see that the parser is currently dealing with only two of the production rules for the single-Command non-terminal. The language definition clearly shows that we should be dealing with 3 other types of single-command. I.e **if**, **while** and **let** commands.

Single-Command	::=	Identifier (:=Expression | (Expression))
	|	**if** Expression **then** single-Command
		else single-Command
	|	**while** Expression **do** single-Command
	|	**let** Declaration **in** single-Command
	|	**begin** Command **end**

Follow the format I have given you in the skeleton to implement the parsing methods for these commands. Follow in particular the pattern used to recognise the BEGIN command phrase.

You will see that when you start to complete this file you will notice that some of the methods you are required to call, such as parseExpression() and parseDeclaration(), do not yet exist. You can put the calls in but obviously your code will not yet work. Show me your complete commands file before continuing and I'll give you feedback on how you are getting on.

PART 4: COMPLETING THE PARSER - EXPRESSIONS

Once you have completed the Commands.cs file you will need to implement the Expressions.cs partial class.

Follow the same structure as the commands partial class to implement the Expressions defined in the EBNF

Expression	::=	primary-Expression
		(Operator primary-Expression)*
primary-Expression	::=	Integer-Literal
	|	Identifier
	|	Operator primary-Expression
	|	(Expression)

You will need to implement the parseExpression() and parsePrimaryExpression() methods to deal with these rules. If you find it easier, you can create the method and fill in the functionality bit by bit.

PART 4: COMPLETING THE PARSER - DECLARATIONS

Again the declarations partial class needs to be created to deal with the Declaration phrases.

Declaration	:==	single-Declaration (**;** single-Declaration)*
single-Declaration	::=	**const** Identifier **~** Expression
	|	**var** Identifier **:** Type-denoter

Use the same approach as the Expressions partial class to create the required recursive methods parseDeclaration() and parseSingleDeclaration(). Remember that the * symbol in EBNF means 0 or more so you will need to use a loop here to adhere to the syntax.

PART 5: COMPLETING THE PARSER – TERMINALS

As seen above the terminals.cs partial class does not yet represent all the terminal symbols in our language. It currently only deals with the identifiers with the parseIdentifier() method.

```
Identifier ::= Letter (Letter|Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= + | - | * | / | < | > | =
```

We have already dealt with the values in the scanner, so we only have to deal with the actual tokenKinds. Again follow the same pattern used in the ParseIdentifier method to create the methods to deal with **Operators** and **Integer-Literals**.

PART 6: TESTING & ERRORS

Currently your parser should print out, simply using a Console.WriteLine statement which type of phrase is being parsed, eg Declaration, Command etc. This will allow you to see the call stack, which represents the syntax tree of the source program.

You do need, however to add default cases to each part of your parser to ensure that errors are shown when the code has code does not match the syntax.

I have created three test files for you test-mini.tri which is a correctly written file and should not produce any errors. You can test this on your current compiler to determine the call stack.

Work though the test-mini.tri testfile yourself on paper, using the language definition and you should be able to determine the call stack you should get from this program. (if you want to know as me and I'll give you the solution)

Finally try the two error programmes and make sure that your parser is identifying errors in the syntax. Next week we will look at actually creating the SyntaxTree.