WORKSHOP 5 – THE RECURSIVE PARSER

PURPOSE OF THE WORKSHOP

The purpose of this week is to give you a chance to get acquainted with the language definition you will be using for the coursework. If you haven't already you should view this on the Moodle page now.

The steps below describe the implementation of our recursive descent parser from identifying the non-terminals to creating the code to deal with the production rules. The final part of the tutorial describes the process to create a robust error reporting structure. You should attempt to implement this for the coursework, but overall the correct parsing of the language is more important.

PART 1: LANGUAGE DEFINITION & PARSE METHODS

As we saw in the lecture, language definitions contain a finite set of non-terminals and productions.

I have given you the language definition with the non-terminals on the left and the productions in a list on the right. In a recursive descent parser each non-terminal must have a parse method associated with it, so for example, the Command non-terminal will be parsed by the parseCommand() method.

The code I have already given you follows a particular structure that ensures that the Parser code is laid out in a way that is understandable. You should ensure that you follow this structure when you add additional parts of the parser.

Parser - Common.cs :

> Contains the helper methods, Accept and AcceptIt. Accept allows you to check the current token against what you would expect at that point in the program and moves to the next token in the stream. AcceptIt just moves to the next token, this is helpful for accepting a token tin the predict set.

Parser - Programscs :

> Contains the code to deal with the productions related to the Start Symbol (root node), Program.

Parser - Commands.cs :

> Contains the code to deal with the productions related to Command non-terminals, e.g. Command and Single-Command

Parser - Expressions.cs :

> Contains the code to deal with the productions related to Expression non-terminals, e.g. Expression, Secondary-Expression and Primary-Expression

Parser - Vnames.cs :

Contains the code to deal with the productions related to the V-name non-terminal. This only needs the parseVname function.

Parser - Declarations.cs :

Contains the code to deal with the productions related to Declaration non-terminals, e.g. Declaration and Single Declaration

Parser - Parameters.cs :

Contains the code to deal with the productions related to Parameters e.g Actual Parameter and Actual Parameter Sequence

Parser - TypeDenoters.cs :

Contains the code to deal with the productions related to the TypeDenoters, again this only has the parserTypeDenoter method.

Parser - Terminals.cs :

Contains the code to deal with the terminals of the language, Identifier, Integer-Literal, Character-Literal, Operator.

You should make sure that your parser code contains **all the parse methods** for the set of non-terminals in our language. Even if you do not code in all the functionality now you should put the methods headers so your structure is correct and implementation can be added as you go on.

PART 2: PREDICT SETS

In the lecture I showed you how predict sets contain all the terminal symbols that allow your parser to determine which production rule to choose. In our parser code, each of our parse methods, e.g parseSingleExpression(), will have to identify what the next token is to choose what state to move into next. This is usually performed by an if, while or switch statement.

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

Our parseSingleCommand code will start off looking something like this

```
void ParseSingleCommand(){
switch (_currentToken.Kind){

       case TokenKind.Identifier: { break; }
       case TokenKind.If:{ break; }
       case TokenKind.While:{ break; }
       case TokenKind.Let:{ break; }
       case TokenKind.Begin:{ break; }
```

There is a case statement for each of the Terminal Tokens in the predict set for the SingleCommand non-terminal.

Again follow this process for you own parser, go through each of the non-terminals in turn and work out the predict set, ie the Terminals that allow your parser to choose a production rule. Again just create the case statement, you can fill in the actual production rule at the next stage.

PART 3: PRODUCTION RULES

Finally, we need to implement each production rule, to ensure our parser can cope with sentences written in our language. Let's look in detail at one of the Single Command productions

if Expression **then** Single-Command **else** Single-Command

The production states that an IF token should be followed by an Expression then a THEN token followed by a Single-Command, finally a ELSE token followed by a Single-Command.

When translated into code this gives us the following;

```
case TokenKind.If:

{
     AcceptIt();              //1
     ParseExpression();       //2
     Accept(TokenKind.Then);  //3
     ParseSingleCommand();    //4
     Accept(TokenKind.Else);  //5
     ParseSingleCommand();    //6
     break;

}
```

Line 1 uses the AcceptIt() method to clear the token just checked, i.e IF.

Line 2 calls the ParseExpression() method to deal with the expression that should appear here

Line 3 uses the Accept() method to look for the THEN token

Line 4 calls the ParseSingleCommand() method to deal with the SingleCommand that should appear here

Line 5 uses the Accept() method to look for the ELSE token

Line 6 calls the ParseSingleCommand() method to deal with the SingleCommand that should appear here

Finally, we break out of the statement.

For each production, for each non-terminal you need to build up the methods in the same way, calling the correct parse method to parse the sentence you are expecting.

PART 4: SOURCEPOSITION AND LOCATION

A number of you had issues creating the source position and location system that is needed to create meaningful error messages to the user. I have provided you with the two classes (Source Position and Location) needed to be able to implement this.

The parser really needs a sensible way of reporting errors to the user, the user really wants to know where and what has gone wrong with their code.

For your parser the error output should display a message which accurately represents what has gone wrong, but should continue to parse the document, even if an error has been detected.

e.g Token “IF” expected “END” found at line 4 index 6

Token “IF” cannot start a declaration between line 8 index 0 and line 8 index 1

To be able to do this correctly, your Token class constructor should receive an instance of SourcePosition which will maintain the position of that token within the source file.

public Token(TokenKind kind, string spelling, SourcePosition position)

The source file class I have already given you has fields (_index and _line) that you can use to get the start positions. When you create a token in your scanner you should use the Location within the sourcefile class to set the tokens position.

Finally, for a robust error message system you should create a class called ErrorReporter, this class should have the follow methods

ReportError – a method that receives a message (string) and a token which can then write a sensible error message to the console about an error.

HasErrors – a method that returns a Boolean defining if the parser found any errors

ErrorCount – a method that returns the total number of errors found in the sourceprogram

You should update your parser so that it takes an instance of ErrorReporter which can then be used to receive errors to when they occur.

Your Parser constructor should look like this

public Parser(Scanner scanner, ErrorReporter errorReporter)

You should then be able to call the error Reporter class with a statement such as

errorReporter.ReportError(message, currentToken, position);

when you wish a new error to be provided to the user.