# Workshop 10 - Recursive Descent Parsing

### Recursive Descent Parsing

##### [Practical Marker](https://cs.adelaide.edu.au/services/pracmarker/)

To simplify the task of writing a recursive descent parser for a simple language, we have provided a precompiled tokeniser class, ***wktokens***, and a skeleton parser, **parser10.cpp**, both of which are included in the zip file attached below. To complete the workshop you just need to complete the bodies of the parseXXX() functions in the file **parser10.cpp**. There is one function for each term in the grammar being parsed.

#### Step 1 - Download, Compile the Skeleton Parser

The first exercise is to download and compile the skeleton parser.

First download the zip file attached below.

After expanding the file compile the main program using the command:

% make

This will compile the **parser10.cpp** files using the precompiled implementation of the **wktokens** class and name the executable **parser10**.

**Step 2 - Complete the Parser**

Write a recursive descent parser for the following language.

|  |  |  |
| --- | --- | --- |
| **Term** |  | **Definition** |
| program | ::= | declarations statement |
| declarations | ::= | ('var' identifier ';')\* |
| statement | ::= | whileStatement | ifStatement | letStatement | '{' statementSequence '}' |
| whileStatement | ::= | 'while' '(' condition ')' statement |
| ifStatement | ::= | 'if' '(' condition ')' statement ('else' statement)? |
| letStatement | ::= | 'let' identifier '=' expression ';' |
| statementSequence | ::= | statement\* |
| expression | ::= | term (op term)? |
| condition | ::= | term relop term |
| term | ::= | identifier | integerConstant |
| **Token Class** |  | **Token Value** |
| identifier | ::= | ('a'-'z'|'A'-'Z')('a'-'z'|'A'-'Z'|'0'-'9')\* |
| integerConstant | ::= | ('0'-'9')('0'-'9')\* |
| relop | ::= | '<' | '<=' | '==' | '!=' | '>' | >=' |
| op | ::= | '+' | '-' | '\*' | '/' |
| keyword | ::= | 'var' | 'while' | 'if' | 'else' | 'let' |
| symbol | ::= | '{' | '}' | '(' | ')' | ';' | '=' |

##### Program Structure

So that you do not have to worry about how to tokenise the input, we have provided three useful functions at the start of the skeleton parser, ***nextToken()***, ***mustbe()*** and ***have()***.

**nextToken()** calls the precompiled tokeniser and returns the next token in the input. It also saves the token, the token class and the token value in global variables for convenience. The token classes and token values are shown above. When an identifier or integer is read, the token returned is the token class, that is "**identifier**" or "**integerConstant**" respectively. For all other tokens, the token returned returned is the token value. For example, for the "**if**" keyword the token returned is the string "**if**" not the token class "**keyword**". If there are any errors constructing the next token or the end of the input is reached, the returned token, token class and token value will all be "**?**".

**mustbe()** is used when you know what the last symbol read **must be**.The function will check whether or not the last symbol read is the expected symbol and will terminate the program with an error if it is not. If no error occurred, it will then read the next symbol from the input.

**have()** is used when you want to know if a particular symbol has just been read. The function will check whether or not the last symbol read is the expected symbol. If the expected symbol was read, the function will read the next symbol from the input and then return true. If the expected symbol was not read, the function simply returns false with out reading any more symbols.

##### Parsing Functions

In a recursive descent parser we write a function for each term in the programming language's grammar which is responsible for parsing that term.  For example, if parsing the grammar shown above, we would have functions:

* void parseProgram() ;
* void parseStatement() ;
* void parseWhileStatement() ;
* void parseIfStatement() ;
* void parseLetStatement() ;
* void parseStatementSequence() ;
* void parseExpression() ;
* void parseCondition() ;
* void parseTerm() ;

To complete this workshop you need to complete the bodies of these parse functions.

**main()** creates a new tokeniser for you, initialises it with a call to ***nextToken(),*** calls the ***parseProgram()*** function and finally checks that the end of the input has been reached.

##### Example Parse Function

To illustrate how we use the ***have()*** and ***mustbe()*** functions, here is what the ***parseStatement()*** function might look like:

void parseStatement()

{

if ( have("while") ) parseWhileStatement() ; else

if ( have("if") ) parseIfStatement() ; else

if ( have("let") ) parseLetStatement() ; else

{

mustbe("{") ;

parseStatementSequence() ;

mustbe("}") ;

}

}

This indicates that if we have a "**while**", "**if**" or "**let**" we should call a particular parsing function or we must have the symbol "**{**" immediately followed by a statementSequence and the symbol "**}**". **Note**: if we have a "**while**" the call of ***have("while")*** will read the next token after the "**while**". Therefore, the first token that the ***parseWhileStatement()*** function should see is a "**(**" and not "**while**".

#### Step 3 - Testing

The completed parsing program reads from standard input and writes to standard output. However, it does not produce any output if the input program is correct. Next week, you will add code to a working parser so that it either produces VM code or an XML representation of the parse tree.

So that you can test your parser, we have provided two examples programs in the files ***eg1*** and ***eg2***. The first program should parse without error, whilst the second should fail because it contains more than one statement after the declarations. You can test your parser as follows:

% ./parser10 < eg1  
% ./parser10 < eg2

#### Step 4 - Error Handling

If you have completed steps 1, 2 and 3 it is instructive to consider how to report syntax errors. For example, what information goes into the error message, how do you represent the location of the error and do we attempt to continue parsing so we can identify as many syntax errors as possible?

Recovering from a syntax error may not always be possible because the parser may not be able to tell what the programmer was trying to write and therefore how to correct the error. There are number options available. The easiest option is to simply give up completely so the compiler never has to handle more than one error. If the compiler writer wants to look for other errors there are two main approaches. Either pretending an expected symbol was present after all and continuing to parse the program on that basis. Alternatively, assuming that some extra input is present and simply deleting all input until the expected symbol is finally found. Many compilers adopt a combination of both.

The way to implement the reporting and recovery is by modifying the behaviour of the **mustbe()** function. When the first error is discovered, the **mustbe()** function simply reports the error and returns. This in effect, simulates inserting the missing symbol into the program and continuing as if the program was correct. The second time an error is discovered by the **mustbe()** function, the error is reported but, before it returns, the **mustbe()** function reads and discards tokens until either it finds the expected token or the end of the input.

For this step, modify your **mustbe()** function so that it can recover from syntax errors as just described. If you now give your parser an incorrect program, what error messages would help you as a programmer work out what your mistake was?