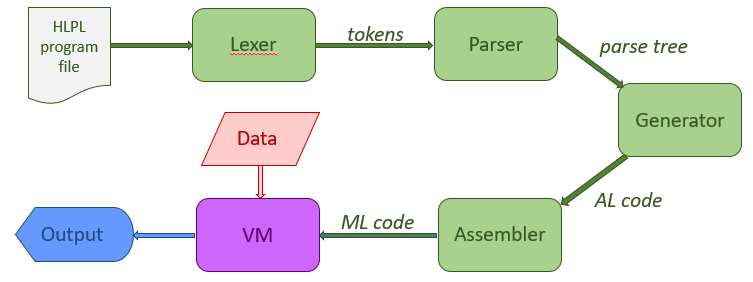
**PROJECT – Part 3 – due Wednesday, April 21 at 11:55 pm  
Lexical Analyzer**

* **Introduction**

In Part 1 of the project worked on the back end of your language translator, designing an assembly language (AL), developing an assembler to translate it to a kind of machine language (ML), and constructing and interpreter to execute the ML. This was the back end, enclosed in the green rectangle below. In Part 2 of the project, you looked at the other end, designing a (relatively) high-level programming language according to certain specifications, in which to write your program file (the orange rectangle in the figure below. The next step is to work on the Lexical Analyzer (Lexer or Scanner) for your HLPL.



**Note that, if you did not get the first part of your program fully working, or you want to improve your grade, this is an opportunity to do so. You can resubmit an improved version of Part 1 with this submission. If you do, make sure you say so and describe what has changed.** Even if your previous submission was working fine, it is possible that you might find that you need to modify a little in order to be able to properly translate the HLPL to it.

* **What the Lexer Does and How You Should Think about It**

The Lexer converts your text input file into a token stream, which becomes the input to your parser. In order to use the tokens in the CFG grammar (next phase of the project) and make the grammar legible, you will want to use named constants that have a unique value.

The Lexer also begins adding symbols to the Symbol Table, but you have to be a little careful here because the same identifier name, if it occurs in different scopes, will not correspond to the same object (variable, function, etc.) and you won’t really be able to determine that until the parsing stage, so Symbol Table entries at this stage are only partial and are likely to be modified downstream.

The Lexer also builds the Literal table, when it encounters literals in the code for the types contained in the language.

* **Implementation Choices**

You have some choices in how you build the Lexer.

1. You can use the JLex scanner generator (see below for information on getting it and using it).
2. You can keep working in C and use a regular expression library with it. Here are some options:
   * <https://sourceforge.net/projects/crx/>
   * <https://github.com/kokke/tiny-regex-c>
   * <https://www.gnu.org/software/regex/>
3. You can work in Python and use the regular expression facilities in the **re** module.

Regardless of which method you use, you have a choice of writing the Lexer output (the token stream) to standard output, so it can be piped into the next stage with ‘>’ or redirected to a file from which the parser will read. Alternatively, you can write the token stream directly to the file.

You should also be conscious of the fact that the ordering of the REs that match input to extract tokens may need to occur in a particular order because, in general, your REs need to match a maximum number of characters. E.g. in C, you would want to match ‘==’ before you match ‘=’.

* **Sample Lexer Output**

The Lexer output used by the parser is a stream of “tokens”, but the raw output of the Lexer will also contain additional information. This extra info will not be used by the parser to check the acceptability of a token sequence according to its grammar; will get stripped by the function called by the parser to get the next token. For example, given an input such as the following one, and assuming the language was C-like:

**if (x==0)  
return 1;**

The output could be:

Line 1 Token #34: if

Line 1 Token #10: (

Line 1 Token #2: x

Line 1 Token #37: ==

Line 1 Token #3: 0

Line 1 Token #11: )

Line 2 Token #27: return

Line 2 Token #3: 1

Line 2 Token #38: ;

The token numbers are the unique internal IDs for each token type, each of which is mapped to a named constant. Take a look at the code in the textbook and choose the exact format of the output, keeping in mind that, to help the user debug, it should provide the line number in case a problem occurs and, to help the parser do its work, it should provide the information that will appear in the grammar rules to indicate the token.

* **Work Distribution**

In this part of the project, you are probably better off working together on the entire project, at least the design of it.

**Team Captains**

For this delivery, the team captain will be the person indicated below. He/she has the following tasks:

* To make sure the team members are communicating and the project is advancing.
* To provide a brief report of advancement of the project by Sunday April 18 at midnight (11:55pm) in Jenzabar.
* To provide a summary of what was done by whom when.
* To submit by the deadline.

**Captains**: You will personally get up to 2 points off your score (not your team members’ score) if you do not submit the report by April 18 and/or are not monitoring progress advancement. It’s your responsibility to keep the project advancing and on time for this phase.

|  |  |
| --- | --- |
| **TEAM** | **CAPTAIN** |
| **MMA** | AALABOU, Mariem |
| **SIFO** | LAMIRI, Fatima Zahrae |
| **OMA** | CHOUKRI, Anass |
| **DACY** | CHTAINI, Yassine |
| **MEBB** | EL GHAYATE, Salma |
| **TRNWRCK** | BOUDRA, Imane |
| **Cup\_of\_Java** | LAMRINI, Walid |
| **2As 2Bs** | BOUNAJMA, Nada |
| **ABN** | BENZINA, Hadil |
| **MQ** | MOURAFIK, Saad |
| **AB** | BENKHALIFA, AIS |

* **Submission and Grading**

The entire project counts for 15% of your final grade. This delivery counts for 3% of your final grade, or 3/15of the project grade.

The contents of the submission are specified in the table below.

**The team captain should submit to Jenzabar, on or before the due date, a zip or rar file.**

|  |  |
| --- | --- |
| **CONTENTS** | **POINTS** |
| **Cover document** | **5** |
| Team members | 0.5 |
| Team captain for this delivery | 0.5 |
| Paragraph reflecting on team dynamics (successes, challenged, issues, etc.) for this delivery | 2 |
| Team captain mid-term report (by Sunday April 18) | 2 |
| **Lexer Documentation** | **25** |
| Justify choice of implementation among alternatives provided | 2 |
| General description of how Lexer is designed and functions | 8 |
| ***Lexer User Manual***: what you need to do in order to run it (needed downloads, assumptions about location of files, format of input and output) | 5 |
| ***Requirements Check***: describe in a tabular fashion how each element of your language design is handled by the lexer. | 10 |
| **Lexer Implementation** | **20** |
| Completeness | 10 |
| Correctness | 10 |
| **Testing** | **10** |
| Input was provided for testing every element (operator, punctuation, resreved word, user-defined IDs, and literals/constant values, comments, and whitespace). This can be done in a single program or in multiple programs, but they should be legal programs (though you won't be checking them yet until the next phase). The programs don't have to make sense, they just need to collectively cover all the elements the Lexer needs to deal with. | 10 |
| **TOTAL** | **60** |

* **JLex Option**

JLex (<https://www.cs.princeton.edu/~appel/modern/java/JLex/>) is Lexer construction tool implemented in Java, that you can use to build a Lexer for your language. Assume that you will call your Lexer **MyLexer**. It will read and process a stream of tokens from a filename given as the first argument to the **MyLexer** command or from standard input, e.g.

java MyLexer *filename*

or from standard input if the <filename> argument is not present:

java MyLexer < *filename*

In the second case, the ‘<’ is indicating that standard input is being redirected to *filename* instead of being the keyboard.

Your implementation with JLex will consist of the following elements, which mostly require adaptation to your HLPL design of the sample specification file, sample.lex, found on the JLex web page.

* The **lexer specification**. Compiled with JLex, it must produce the file Lexer.java which contains the lexical analyzer. In the picture below, the **lexer specification** corresponds to the file *xxx.jlex* and Lexer.java corresponds to *xxx.jlex.java*.
* **class MyLexer**: The command java MyLexer *testfile*.*ext* should enable reading the *testfile*.*ext* file, breaking it into tokens, and successively calling the next token method of the generated Lexer to display the file as a series of tokens, one token per line as in the sample out shown earlier. The output must include the token identifier, the value of the token, and the line number for that token (but you can adapt it to your own design). Note that class **MyLexer** corresponds to the Sample class given in file sample.lex and to ***P.main*** in the picture below.
* **class Token**: The scanner returns an object of this class for each token. The Token class, which corresponds to the Yytoken class in sample.lex and must contain at least the following information:
  + **id**: an integer identifier for the token;
  + **value**: an arbitrary object holding the specific value of the token (e.g., the character string, or the numeric value);
  + **line**: an integer representing the line number where the token occurs in the input file.

The numeric identifiers for all tokens should be placed in a file sym.java. A sample of such class, which you can modify, can be downloaded together with this assignment.

* **class LexicalError**: The Lexer should detect and report any lexical analysis errors. An exception class for lexical errors must be implemented which contains at least the line number where the error occurred and an error message. Whenever the program encounters a lexical error, the Lexer must throw a LexicalError exception and the main method must catch it and terminate the execution. The program must always report the first lexical error in the file.

