**PROJECT Part 2 – Lexical Analyzer**

In Part 1 of the project, you designed a high-level programming language, the Green World Language (GWL) for the Green World application, whose goal is to allow a programmer to construct and change a world constructed of blocks, tubs and plants planted in the tubs.

The next step is to work on the Lexical Analyzer (Lexer or Scanner) for GWL.

Diagram

Description automatically generated

My responsibility

**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 (Part 3 of the project) and make the grammar legible, you will want to use named constants that correspond to a unique numerical ID. See section 4.2 of your textbook and below in **Lexer Output** for examples of that.

As it is tokenizing the text, the lexer may need to build the literal table, when it encounters literals in the code for the types contained in the language. Normally, a literal table contains string constants that may be used in different places in the program. It is up to you to decide if and how much you will need to use a literal table.

The lexer also begins adding symbols to the symbol table. The first draft of your symbol table will be very simple. It should be initialized to contain all reserved words. Then, as the lexer scans the input file, it will look up identifiers and, if they are not reserved words, it will create new entries in the symbol table. Just to look ahead to Part 3 a little bit, you should think that, when you write the analyzer, you will want to distinguish between local variables and global variables in some way. You don’t need to handle that right away, but chances are that, in Part 3, you will need to modify the contents of the symbol table a little.

**Lexer Implementation Choices**

You have some choices in how you build the lexer. Basically, you can follow one of two general approaches and then choose a particular implementation.

**Approach 1:** Use a lexical analyzer generator:

* You can work with Java and use the JLex scanner generator (see below for information on getting it and using it).
* You can work use ***lex*** or ***flex*** (see <http://dinosaur.compilertools.net/> for pointers to this info).

**Approach 2:** Build your own lexical analyzer from scratch:

* You can work in Python and use the regular expression facilities in the **re** module.
* You can work in C and use a regular expression library with it. *This is not really recommended*. Those who have tried this in the past have not been successful in getting them to work, but I’m giving you the choice of trying for yourselves. Here are some options:
  + <https://sourceforge.net/projects/crx/>
  + <https://github.com/kokke/tiny-regex-c>
  + <https://www.gnu.org/software/regex/>
* Work in some other language that has a good regular expression package.

If you choose the second approach, i.e., building your ow lexical analyzer, keep in mind that you can think of this program as consisting of an ordered set of “rules” that:

1. Test a regular expression against the input, and
2. If a match is made, execute the action that creates the token and adds whatever information is needed to the literal or symbol table (or, alternatively, throws away the matched information, as in the case of comments or whitespace—except in Python, where indentation counts).

If some of your tokens are substrings of other tokens, (e.g., “=” is a substring of “<=”), then you will want to order your “rules” such that you test for the longer matches first.

Regardless of which of the approaches you use, you have a choice of writing the lexer output (the token stream) to standard output (**stdout** in C), 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.

**Lexer-Parser Interaction**

Looking forward again to Part 3, where you will build the parser, we have seen another mode of interaction between parser and lexer in the recursive descent parser: the parser calls the lexer to get it the next token. This means that, once you write the parser, you could have the parser control the lexical analysis process by calling the lexer directly (easiest if you use the same language for the parser and the lexer). As an alternative, or you can have the parser call a “lexer” function that just reads from the file or stream of tokens where a separate lexer process has placed the token (this is the model where each process runs separately communicating via the standard output stream or a file).

**Lexer Output**

The lexer output used by the parser is fundamentally a token stream, 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 but will get stripped by the function called by the parser to get the next token. However, it is used for error reporting. 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 (see also Section 4.2 of the textbook for another example):

|  |  |
| --- | --- |
| To the parser (token stream or file) | To the screen (for debugging purposes) |
| 1 34 IF  1 10 LPAREN  1 2 ID x  1 37 EQ  1 3 INTCON 0  1 11 RPAREN  2 27 RETURN  2 3 INTCON 1  2 38 SEMICOLON | 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: ; |

Of the information shown above in the left column, the parser will use only the 2nd and 4th columns for syntax checking. The token numbers are the unique internal IDs for each token type, each of which is mapped to a named constant (e.g., IF, LPAREN, ID, INTCON, RPAREN, RETURN, SEMICOLON) and this mapping should be the same by the lexer and the compiler. The line number is helpful for error reporting. Thenamed constants are often used in writing the grammar rules instead of the actual lexeme they correspond to and instead of the unique ID which is hard for a human to process.

**Test Suite**

Start your work on the test suite by constructing a small set of programs to test that your lexer handles correctly all expected lexemes for your GWL design. The tests can be done in a single program or spread across multiple programs. At this point, your programs don’t need to make sense (i.e., be particularly meaningful), but they should be syntactically correct programs. You will be able to reuse them for your parser tests in Part 3.

**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 & Team Captain Report**

Team captains for Part 2 are indicated with C2 in [this file](https://alakhawayn365.sharepoint.com/:x:/s/FA21SSECSC331501/ERydhtiYJLdKsPN86y5N5xQBiPuLJbsUr26H0M6PPwhKWg?e=aURq1D) on Teams. The report should indicate the team members and contain a paragraph (or more, if necessary) reflecting on team dynamics (successes, challenges, issues, etc.) for this delivery.

**Submission & Grading**

The Team Captain submits to Jenzabar for the Team with a separate report on teamwork. Use the following filename: *TeamName*\_ProjectPart2 and *TeamName*\_ProjectPart2\_CapRep.

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.**

Late penalties, as specified in the Syllabus, apply.

|  |  |
| --- | --- |
| **CONTENTS** | **POINTS** |
| **Lexer Documentation** - A brief report including: | **13** |
| Why you chose this specific implementation among the alternatives provided | 1 |
| General description of how Lexer is designed and functions | 2 |
| ***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) | 1 |
| ***Requirements Check***: Describe, using a table format, how each element of your language design is handled by the lexer.You can use the same categories as for Testing (below). | 9 |
| **Lexer Implementation -** Completeness and Correctness | **8** |
| **Testing -** Input was provided for testing every element (see list below) | **9** |
| Punctuation | 2 |
| Operators | 2 |
| Reserved words | 3 |
| User-defined IDs | 0.5 |
| Numeric literals & String literals (if applicable) | 0.5 |
| Comments | 0.5 |
| Whitespace | 0.5 |
| **TOTAL** | **30** |

**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 language 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.

Diagram

Description automatically generated