COSC 455
Programming Languages: Design and Implementation
Fall 2014

# Project #1 – LOLCODE Markdown Language Translation

> Any good software engineer will tell you that a compiler and an interpreter are interchangeable.
>
> -- Tim Berners-Lee

**Directions:** This project has two phases that will result in a small compiler/interpreter. Please read through the entire project description before starting it.

**Goals:** The intention of this project is to use and apply the concepts of programming languages discussed in class and in the textbook to design and develop a relatively complex interpreter/compiler. In doing so, you should develop a better understanding in the design of a high-level programming language and the need for studying the concepts behind programming languages. Additionally, this project will familiarize you with several software development tools (Eclipse, Subversion, etc.) and gain experience in software engineering and development.

This project will give you development experience in the following areas of programming languages that were covered in class:

- Writing BNF/EBNF grammar rules (Chapter 2)
- Lexical analysis (Chapter 2.6, 3)
- Syntactic analysis and parsing (Chapter 3)
- Semantics (Chapter 3)
- Static-scope analysis (Chapter 10)

Further, the implementation of these concepts will give you further experience in essential programming concepts including:

- Object-oriented design and development
- Method invocation
- Exception handling
- Recursion

**Description:** As described in class, the basic idea of a compiler is to take a high-level language and convert it to a low-level language that can be executed on a computer. In this project, you will design and develop a compiler/interpreter that translates my version of a LOLCODE Markdown language (see http://en.wikipedia.org/wiki/Markdown, http://daringfireball.net/projects/markdown/syntax). Briefly, a Markdown language allows for easy-to-read annotations within text that is then automatically converted to valid, well-formed HTML5. However, unlike a traditional Markdown language, our LOLCODE Markdown language was inspired by its esoteric programming language relative (see http://progopedia.com/language/lolcode/) and "created under the influence of a meme lolcat". As an extension to typical Markdown languages, our language will provide for statically scoped variables to be defined and used throughout the LOLCODE Markdown document.

Specifically, our LOLCODE Markdown language will support the following commands (bold is used to emphasize the syntax and differentiate it from the text):

- **#HAI** … **#KTHXBYE**
  The #HAI…#KTHXBYE annotations denote the beginning and ending of a valid source file in our LOLCODE Markdown language. All valid source files *must* start with #HAI and end with #KTHXBYE (i.e., there cannot be any text before or after). Between theses annotations, all other annotations (or none at all) may occur except for a repeating of the #HAI or #KTHXBYE annotations. In HTML5, these annotations correspond with the <html> and </html> tags, respectively.
- **#OBTW** … **#TLDR**
  The #OBTW…#TLDR annotations denote the beginning and ending of a comment in our LOLCODE Markdown language. The comment annotations are optional in any LOLCODE Markdown source file and may occur immediately after any legal annotation. Within the comment annotation, *only* plain text is possible (i.e., no other annotations) and may span several lines but may not contain any text. In HTML5, these annotations correspond with the <!-- and --> tags, respectively.
- **#MAEK HEAD** … **#OIC**
  The head annotations are a container for any head elements, in our LOLCODE Markdown language only the #GIMMEH TITLE annotation is allowed. The head tag is *not* required in a LOLCODE Markdown file, but if it is present it must be immediately following the #HAI annotation unless there are comments between the #HAI annotation and the #MAEK HEAD annotation. In HTML5, these annotations correspond with the <head> and </head> tags, respectively.
- **#GIMMEH TITLE** … **#MKAY**
  The title annotations denote the title of the resulting html page that shows up in the browser's toolbar. Within these annotations, *only* plain text is possible (i.e., no other annotations). Title annotations *must* occur within #MAEK HEAD annotations. In HTML5, these annotations correspond with the <title> and </title> tags, respectively.
- **#MAEK PARAGRAF** … **#OIC**
  The paragraph annotations denote the beginning and ending of a paragraph within a LOLCODE Markdown source file. Within these annotations, the bold, italics, list, item, sounds and video (described below) annotations are allowed but not required (note that you cannot have a #MAEK PARAGRAF annotation within another #MAEK PARAGRAF annotation). In HTML5, these annotations correspond with the  and  tags, respectively.
- **# GIMMEH BOLD** … **#MKAY**
  The bold annotations denote the beginning and ending of text within the LOLCODE Markdown source file that is in a bold font. Within these annotations, *only* plain text is possible (i.e., no other annotations). Bold annotations *do not* have to occur within paragraph annotations, they may occur on their own. In HTML5, these annotations correspond with the  and  tags, respectively.
- **# GIMMEH ITALICS** … **#MKAY**
  The italics annotations denote the beginning and ending of text within the LOLCODE Markdown source file that is in an italic font. Within these annotations, *only* plain text is possible (i.e., no other annotations). Italics annotations *do not* have to occur within paragraph annotations, they may occur on their own. In HTML5, these annotations correspond with the  and  tags, respectively.

- **#MAEK LIST** … **#OIC**
  The list annotations denote the beginning and ending of a bulleted list within the LOLCODE Markdown source file. This annotation must be immediately followed by the #GIMMEH ITEM annotation. In HTML5, these annotations correspond with the <ul> and </ul> tags, respectively.

- **#GIMMEH ITEM** … **#MKAY**
  The item annotations denote the beginning and ending of a list item within the LOLCODE Markdown source file. All list item annotations *must* occur within a #MAEK LIST annotation block. Within these annotations, the bold and italics annotations are allowed but not required (i.e., it can just be plain text). In HTML5, these annotations correspond with the <li> and </li> tags, respectively.

- **#GIMMEH NEWLINE**
  The #NEWLINE annotation, within the LOLCODE Markdown source file, may appear anywhere within a LOLCODE source document outside of the head and title annotations. In HTML5, this annotation corresponds with the <br> tag.

- **#GIMMEH SOUNDZ** *address* **#MKAY**
  The audio annotations denote an audio element (see http://www.w3schools.com/html/html5_audio.asp) within the Markdown source file. The #GIMMEH SOUNDZ annotations *must* contain some text (denoted by *address* above) giving the address of the MP3 file to link to. For example, the following Markdown annotation:

  ```
  #GIMMEH SOUNDZ http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3 #MKAY
  ```

  would correspond in HTML5 to:

  ```
  <audio controls>
    <source src="http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3">
  </audio>
  ```

  For this annotation, you do not need to validate the address – you may assume whatever address provided is valid. For simplicity, we will only use MP3 encoded files.

- **#GIMMEH VIDZ** *address* **#MKAY**
  The video annotations denote a YouTube video element (captured in an iframe tag) within the LOLCODE Markdown source file. The video annotations *must* contain some text (denoted by *address* above) giving the address of the YouTube file to link to. For example, the following LOLCODE Markdown annotation:

  ```
  #GIMMEH VIDZ http://www.youtube.com/embed/zoO0s1ukcqQ #MKAY
  ```

  would correspond in HTML5 to:

  ```
  <iframe src="http://www.youtube.com/embed/zoO0s1ukcqQ"/>
  ```

  For this annotation, you do not need to validate the address – you may assume whatever address provided is valid. For simplicity, we will only use YouTube links.

In addition to these LOLCODE Markdown annotations, our language will also include the capability to define and use statically-scoped variables, defined as follows:

- **#I HAS A** *variable name* **#ITZ** *value* **#MKAY**
  This annotation structure denotes the beginning and ending of a variable definition within a LOLCODE Markdown source file. The #I HAS A … #MKAY annotations contain a variable name following the #I HAS A annotation, that is some text (denoted by *variable name* above)

giving the name of the variable (i.e., a single word containing no spaces) and a #ITZ annotation that must be followed by some text (denoted by *value* above) giving the value of the variable. The #I HAS A annotation may occur within any other annotation block but, if it occurs, it must be the very first annotation to occur within that block (i.e., immediately following the GIMMEH or MAEK of another annotation). The scope of the variable definition starts after the #I HAS A tag in the block and continues to the end of its immediate enclosing block.

- **#VISIBLE** *variable name* **#MKAY**
  The use annotations denote the beginning and ending of the use of a variable within the LOLCODE Markdown source file. The #VISIBLE … #MKAY annotations *must* contain only text (denoted by *variable name* above) noting the variable value to use. Again, the variable name must only contain text and is a single word (i.e., no spaces). The #VISIBLE annotation may occur within *any* other annotation block.

***Note that all annotations are not case sensitive (i.e., #HAI and #hai are legal).***

Finally, in our LOLCODE Markdown documents, you may assume that whenever there is text (both in text and the cases when an address is provided) possible, the following are the only allowed characters:

- Upper and lower-case letters: A .. Z; a .. z
- Numbers: 0 .. 9
- Punctuation: commas (i.e., ‘,’), periods (i.e., ‘.’), quotes (i.e., ‘””), colons (i.e., ‘:’), question marks (i.e., ‘?’), exclamation points (i.e., ‘!’), percent sign (i.e., ‘%’) and forward slashes (i.e., ‘/’)
- Special characters: newline, tabs

Except for these characters, you may assume no other character is possible in the text and your grammar does not need to account for them (i.e., the “#” character will only be used to denote one of our Markdown annotations and will not be found in the text).

**Examples:** This section presents some basic examples of our LOLCODE Markdown language and its "compiled" HTML5 code (indented for readability).

The LOLCODE Markdown source code

```
#HAI
    #OBTW This is a LOLCODE Markdown source file #TLDR
    #MAEK HEAD
        #GIMMEH TITLE The Simpsons #MKAY
    #OIC
    #MAEK PARAGRAF
        The Simpsons! #GIMMEH NEWLINE
        #GIMMEH SOUNDZ
            http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3
        #MKAY
        #GIMMEH NEWLINE

        The members of the #GIMMEH BOLD Simpson #MKAY family are:
        #MAEK LIST
            #GIMMEH ITEM Homer Simpson #MKAY
            #GIMMEH ITEM Marge Simpson #MKAY
            #GIMMEH ITEM Bart Simpson #MKAY
            #GIMMEH ITEM Lisa Simpson #MKAY
            #GIMMEH ITEM Maggie Simpson #MKAY
        #OIC
        #GIMMEH NEWLINE
        Lets watch now: #GIMMEH NEWLINE
        #GIMMEH VIDZ http://www.youtube.com/embed/zoO0s1ukcqQ #MKAY
    #OIC
#KTHXBYE
```

would compile to the HTML5 code

```
<html>
    <!-- This is a LOLCODE Markdown source file -->
    <head>
        <title> The Simpsons </title>
    </head>
    <p> The Simpsons! <br>
    <audio controls>
         <source src="http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3">
    </audio> <br>
      The members of the <b> Simpson</b> family are:
    <ul>
        <li> Homer Simpson</li>
        <li> Marge Simpson</li>
        <li> Bart Simpson</li>
        <li> Lisa Simpson</li>
        <li> Marge Simpson</li>
    </ul> <br>
    Lets watch now: <br>
    <iframe src="http://www.youtube.com/embed/zoO0s1ukcqQ"/>
    </p>
</html>
```

Note that your code does not need to preserve the spacing and tabs as shown above.

Using a definition of a variable in our LOLCODE Markdown language, we could provide the source code:

```
#HAI
    #I HAS A lastname #ITZ Simpson #MKAY
    #MAEK PARAGRAF
        The members of the #GIMMEH BOLD #VISIBLE lastname #MKAY #MKAY are:
        #MAEK LIST
            #GIMMEH ITEM Homer #VISIBLE lastname #MKAY #MKAY
            #GIMMEH ITEM Marge #VISIBLE lastname #MKAY #MKAY
            # GIMMEH ITEM Bart #VISIBLE lastname #MKAY #MKAY
            # GIMMEH ITEM Lisa #VISIBLE lastname #MKAY #MKAY
            # GIMMEH ITEM Maggie #VISIBLE lastname #MKAY #MKAY
        #OIC
    #OIC
#KTHXBYE
```

would compile to the HTML5 code

```
<html>
    <p> The members of the <b> Simpson</b> family are:
    <ul>
        <li> Homer Simpson</li>
        <li> Marge Simpson</li>
        <li> Bart Simpson</li>
        <li> Lisa Simpson</li>
        <li> Marge Simpson</li>
    </ul>
    </p>
</html>
```

Your compiler should take the value of the *lastname* variable and replace it in the compiled code whenever the variable is used for its statically determined scope. That is, the definition of the *lastname* variable in this example is essentially global since it is defined immediately following the #HAI block. To fully illustrate the scoping, consider the following example with two defined variables:

```
#HAI
    #I HAS A myname #ITZ Josh #MKAY
    Hi, my name is #VISIBLE myname #MKAY .

    #MAEK PARAGRAF
        #I HAS A myname #ITZ Jon #MKAY
        Inside the paragraph block, my name is #VISIBLE myname #MKAY
    #OIC
    Now my name is #VISIBLE myname #MKAY again.
#KTHXBYE
```

should correctly compile into HTML5 as

```
<html>
    Hi, my name is Josh.
    <p> Inside the paragraph block, my name is Jon. </p>
    Now, my name is Josh again.
</html>
```

The scoping used here is the same as you are used to in most programming languages. If a variable is used without first being defined, this should be an error.

**Details and Deliverables:** I *strongly* suggest using the standard software engineering approach for developing the compiler for our LOLCODE Markdown language. You will be required to meet two milestones during this project.

**Phase 1: Grammar Design** (15 points) **Deadline**: October 10, 2014, 11:59pm (Blackboard)
Write and submit a BNF grammar for our LOLCODE Markdown language. Your grammar should be parsable using a recursive-descent parser (as described in class and illustrated in Labs #1-3) using a one token lookahead. Note that this grammar is strictly to be written in BNF, not EBNF! Additionally, develop and submit the ANTLR-based grammar definition for this language. The submission of this phase should be in a single zip file containing your BNF in a text document (e.g., .odt, .doc, .docx, .text) as well as the ANTLR grammar file (e.g., .g) submitted to Blackboard.

**Phase 2: Implementation** (50 points) **Deadline:** November 2, 2014, 11:59pm (Blackboard/Google Code)
To complete this project successfully, you will need to implement a lexical analyzer, a syntax analyzer and a small semantic analyzer. To do so, I suggest the following steps:

Task 1. Implement a *character-by-character* lexical analyzer that partitions the lexemes of a source file in our LOLCODE Markdown language into tokens. To do this, you must use (i.e., `implements`) my Lexical Analyzer interface (to be posted to Blackboard). You may assume that troublesome HTML characters, such as "<", ">" and "&", do not appear in any of the text in the source file. You may also assume that the "#" character only appears prior to one of our LOLCODE Markdown annotations (e.g., #HAI). In a good object-oriented design, it may be helpful to have separate Java classes to represent token objects such as: *StartDocument*, *EndDocument*, *StartParagraph*, *EndParagraph*, etc. Any lexical errors encountered (e.g., #HEY) should be reported as output to the console with as much error information as possible (i.e., similar to what a Java compiler provides). Your compiler may exit after the first error is encountered. If an error is encountered, no output file should be created.

Task 2. Implement a recursive-decent parser (i.e., syntax analyzer) that builds an abstract syntax tree (parse tree). To do this, you must use (i.e., `implements`) my Syntax Analyzer interface (to be posted to Blackboard). It may be helpful to have parse tree nodes such as: *VarDefNode*, *VarUseNode*, *BoldNode*, etc. The implementation of the abstract syntax tree may best be done using a stack(s) or an array list/vector(s). Any syntax errors encountered should be reported as output to the console with as much error information as possible (i.e., similar to what the Java compiler provides). Your compiler may exit after the first error is encountered. If an error is encountered, no output file should be created.

Task 3. Implement the variable resolution for our LOLCODE Markdown language as described above. A good way of doing this may be to include print methods in the nodes from Task 2 and also lookup methods. These print methods should only be used for development/debugging and should not be included in the final deliverable (or at least turned off). Any static semantic errors encountered (i.e., a variable being used before it is defined) should be reported as output to the console with as much error information as possible (i.e., similar to what the Java compiler provides). Your compiler may exit after the first error is encountered. If an error is encountered, no output file should be created.

Task 4. Implement the semantic analyzer that takes the abstract syntax tree and translates it to a lower-level language – HTML5 in our case.

**Phase 3: Execution**
The final program should take *only* one command-line argument provided by the user: an input file name in our LOLCODE Markdown language. We will require and use the convention that *all* LOLCODE Markdown source files in our language will have an *lol* extension (i.e., any file that does not have a lol extension should not be accepted by the compiler). The compiler should then generate an "executable"

output file (saved in the same directory as the input file) with the same name but an *html* extension and be viewable in the Google Chrome 37+ browser. To run a compiler that has been packed as an executable Java jar file, named compiler.jar, a LOLCODE Markdown input file, named input.lol, I would issue the following at a command prompt:

```
C:\> java –jar compiler.jar input.lol
```

Your compiler *should not* output anything to the console unless there is an error. If there is no error, your compiler should create an HTML5 file (with the same name) and return to the command prompt. For example, the above command should create an input.html file.

For

```
C:\> java –jar compiler.jar Test1.lol
```

your compiler should create a Test1.html file, assuming that is has no lexical, syntax or semantic errors. ***Any projects not following this format will not be graded.***

**Phase 4: Testing**

A significant portion of the grading is dedicated to correctly processing my test case input files. Thus, you are responsible for ensuring that your compiler behaves as expected through your own testing. A key component of testing is to come up with test cases. The test cases should exercise the major paths through your code. For example, you should have tests that use each of the constructs in our LOLCODE Markdown language, including various kinds of blocks (paragraphs, lists, etc.). In addition, you should have test cases that check scope lookup for variables, and maybe even some tests with illegal input just to make sure you are recognizing only correctly formed programs. If your program passes each of these tests then you will know that your program is likely to be able to handle at least simple programs with each of the constructs.

I strongly suggest you test your code at the end of each task in Phase 2. That is, make sure that your lexical analyzer is correctly working (i.e., providing only the next, valid token and recognizing illegal tokens). Only after you are 100% convinced that it is working correctly that you should move on to Task 2. If you wait until after implementing all four tasks, the bugs that you will likely have will be *much* harder to find (i.e, is it in the lexical analyzer, the syntax analyzer or the semantic analyzer).

I will make available the test cases I will use to grade your project on October 24, 2014 so that you can test your code against my test cases.

**Phase 5: Documentation**

Your code must be well commented – this includes documenting each class, method and complex parts of the code. Students that choose to use Javadoc and generate the html files will receive up to 5 points extra credit.

**Version Control**

All students will be required to maintain and update their source code using one of Google Code's version control system (https://code.google.com/).

**Project Timeline:** To summarize, the timeline for the project is as follows:

| | |
|---|---|
| 09/30 | Project assigned |
| 10/10 | BNF & ANTLR grammar deadline (Blackboard) |
| 10/12 | BNF Grammar solution provided |
| 10/24 | Test cases provided |
| 11/02 | Implementation deadline (Blackboard / Google Code) |

**Language Requirements:** It is *strongly* recommended that you use Java as the programming language to complete this project since this assignment was designed for, described below and solved by me using Java. If however, after reading the project requirements, you would like to use an alternative object-oriented language (e.g., C++, C#, Python, etc.) you **may do so with my permission**. While I may be able to offer limited help with programming-specific questions and difficulties to students who opt to use a language other than Java, you will ultimately be responsible for programming-specific questions.

**Environment Suggestions:** I have arraigned to have Eclipse and Subclipse installed in the lab classroom and will guide you through setting up Subversion accounts via Google Code. Thus, I ***strongly*** suggest that you set up the same development environment on your main computer/laptop. You can install Eclipse from http://www.eclipse.org/downloads/. I recommend installing Eclipse IDE for Java Developers. Once you have installed Eclipse, you may want to read through a tutorial to familiarize yourself with the tool. There are many good tutorials online, but I suggest http://www.vogella.de/articles/Eclipse/article.html. In addition to the Eclipse IDE, we will be using the Subclipse plugin to Eclipse as our configuration management tool. Subclipse (http://subclipse.tigris.org/) can be installed directly through Eclipse from the update site http://subclipse.tigris.org/update_1.6.x by following the Update and Install Plugins directions found at the Eclipse tutorial listed above. You can become familiar with Subclipse by going through the tutorial found at http://agile.csc.ncsu.edu/SEMaterials/tutorials/subclipse/. Finally, for documentation, we will use Javadoc. You should read http://java.sun.com/j2se/javadoc/ to become familiar with the use of Javadoc in Java programs as well as http://www.eclipse-blog.org/eclipse-ide/generating-javadoc-in-eclipse-ide.html to see how to configure and use Javadoc within Eclipse.

Configuration management is essential to keep track of different versions of software components. We will be using Eclipse and Subversion (you should have already read through the tutorials at http://www.vogella.de/articles/Eclipse/article.html and http://agile.csc.ncsu.edu/SEMaterials/tutorials/subclipse/).

**Grading:** Phase 2 of your project will be graded as follows: 25 points allocated for following tasks 1-4 and configuration management use; 20 points for handling my test cases (which will likely include variations of the above test cases); and, 5 points for quality of your code (i.e., design, structure, comments, etc.). As mentioned in Phase 5, up to 5 points extra credit will be awarded to those students that use Javadoc and generate the html files for it.

**Submission:** You will submit the following in a zipped file via Blackboard:

- All developed source code in a directory named *src*.
- An executable file of your developed compiler in a directory named *bin*. If you are developing your compiler in Java, you should create an executable jar file. You should also include any non-standard libraries within the *bin* directory.
- A directory of the test case files you used for input in a *test* directory. Ideally , this should include more than just the test files I will provide for grading basis.
- A readme.txt file in the root directory that provides your Google Code repository address. Additionally, you will need to specifically detailing how your project can be compiled and executed. These details should be specific enough so that I can compile and execute your source code on my own (i.e., you should describe the IDE you used for development and any non-standard libraries used).

***Any submitted projects not following this format will not be graded.***