COSC 455
Programming Languages: Design and Implementation
Fall 2017

Project #1 – 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 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 (e.g., IntelliJ, GitHub, Slack, etc.), Scala programming language tools (e.g., Scaladocs, Scala tests, ScalaCheck, 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 1)
- Lexical analysis (Chapters 1.61 & 2.2)
- Syntactic analysis and parsing (Chapter 2.3)
- Semantics (Chapter 1 & 2)
- Static-scope analysis (Chapter 3)

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

- Object-oriented program design and development
- Method invocation
- Exception handling
- Recursion
- Configuration management
- Unit testing

Finally, this project will require you to develop new programming skills in the Scala programming language to enable you to gain further confidence in your ability to quickly learn and make use of new programming languages.

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 an interpreter/compiler that translates my version of a Markdown language. Briefly, a Markdown language allows for easy-to-read annotations within text that is then automatically converted to valid, well-formed HTML5, and they have become quite popular with use in GitHub and Wikipedia. This semester, the Markdown language we will design will be *Gittex Markdown* (i.e., **Git**Hub – La**tex**), a hybrid markdown language mimicked after GitHub Flavored Markdown and Latex; Gittex Markdown will additionally provide for statically scoped variables to be defined and used throughout the Gittex Markdown document.

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

- DOCUMENT **\BEGIN** … **\END**
  The document annotations denote the beginning and ending of a valid source file in our Gittex Markdown language. All valid source files *must* start with \BEGIN and end with \END (i.e., there cannot be any text before or after). Between theses annotations, all other annotations (or none at all, other than the title annotation) may occur except for a repeating of the \BEGIN...\END annotations. You may assume that these annotations occur on their own lines (i.e., no other annotations on the same line) and that there is a newline character (i.e., "\n") after both \BEGIN and \END annotations. In HTML5, these annotations correspond with the <html> and </html> tags, respectively.
- TITLE **\TITLE[*text*]**
  The title annotation denotes the title of the resulting html page that shows up in the browser tab. Within the square brackets of this annotation, *only* plain text is possible (i.e., no other annotations). Title annotations are required for any valid Gittex file. You may assume that these annotations occur on their own lines (i.e., no other annotations on the same line) and that there is a newline character (i.e., "\n") after the \TITLE[*text*]. In HTML5, this annotation must be enclosed in <head> … </head> tags and correspond with the <title> and </title> tags, respectively.
- HEADINGS ***# text***
  The heading annotation denotes a first-level heading of the resulting html page. Heading annotations are followed by a space and then *text*. Within the text of this annotation, *only* plain text is possible (i.e., no other annotations). You may assume that this annotation occurs on its own line (i.e., no other annotations on the same line), that there is a newline character (i.e., "\n") after the *text* annotation. Heading annotations may not occur inside any other annotation other than the \BEGIN … \END document annotation required in any Gittex file and may not contain any other annotations (e.g., bold, italics, etc.). In HTML5, these annotations correspond to <h1> … </h1>.
- PARAGRAPH **\PARB** … **\PARE**
  The \PARB … \PARE (i.e., paragraph begin and paragraph end) annotation denotes a paragraph within the Gittex source file. You may assume that this annotation occurs on its own line (i.e., no other annotations on the same line), that there is a newline character (i.e., "\n") after the \PARB and \PARE annotation. Following the \PARB annotation can either be a variable definition (see below) and/or the *text* of the paragraph. The paragraph *text* is terminated by \PARE. Within a paragraph, only the bold, italics and link annotations are allowed, but not required (note that you cannot have a paragraph annotation within another paragraph annotation). In HTML5, this annotation corresponds with the and tags, respectively.

- BOLD *** *text* ***
  The bold annotation, signaled by one asterisk and a space before/after plain text, denotes the beginning and ending of text within the Gittex 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 where *text* is allowed other than title, heading, image, link and italics tags. In HTML5, these annotations correspond with the and tags, respectively.

- UNORDERED LISTS **+ *list item***
  The unordered list annotation denotes a bulleted list item within the Gittex source file. A list item, which must start with a "+", be followed by a space and end with a newline (i.e., "\n") character may contain only the bold, italics and link annotations but not required (i.e., it can just be plain *text*). In HTML5, these annotations correspond with the <li> and tags, respectively.

- NEW LINE \\
  The new line annotation, within the Gittex source file, may appear anywhere within a document outside of the title annotations. In HTML5, this annotation corresponds with the
 tag.

- LINKS **[*text*](*address*)**
  The link annotation, within the Gittex source file, denotes a link element (see http://www.w3schools.com/html/html_links.asp). The [] and () annotations *must* contain some text (denoted by *text* and *address* above) giving the address of the page to link to. For example, the following Gittex annotation:

  ```
  [The Simpsons](http://www.simpsonsworld.com/)
  ```

  would correspond in HTML5 to:

  ```
  <a href" http://www.simpsonsworld.com/"> The Simpsons </a>
  ```

  For this annotation, you do not need to validate the address – you may assume whatever address provided is valid.

- IMAGES **![*text*](*address*)**
  The image annotation denotes an image element (see http://www.w3schools.com/tags/tag_img.asp). The ![] and () annotations *must* contain some text (denoted by *text* and *address* above) giving the address of the image. For example, the following Gittex annotation:

  ```
  ![Try your best]( http://bit.do/tryyourbest)
  ```

  would correspond in HTML5 to:

  ```
  <img src="http://bit.do/tryyourbest" alt="Try your best">
  ```

  For this annotation, you do not need to validate the address – you may assume whatever address provided is a valid image address.

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

- VARIABLE DEFINITIONS **\DEF[*variable name = value*]**
  The define annotations structure denotes the beginning and ending of a variable definition within the Gittex source file. The \DEF[] annotations *must* contain some text (denoted by *variable name* above) giving the name of the variable, an "=" annotation that must be followed by some text (denoted by *value* above) giving the value of the variable. The \DEF annotation may occur within the \BEGIN … \END and \PARB …\PARE annotation blocks but, if it occurs, it must be the very first annotation to occur within that block (i.e., immediately following the start of another annotation). The scope of the variable definition starts after the \DEF[] tag in the block and continues to the end of its immediate enclosing block.
- VARIABLE USAGE **\USE[*variable name*]**
  The use annotations denotes the use of a variable within the Gittex source file. The \USE[] annotations *must* contain only text (denoted by *variable name* above) noting the variable value to use. The #\USE[] annotation may occur within *any* other annotation block except title, links and images.

Note that all annotations are not case sensitive (i.e., \BEGIN and \begin are equivalent and legal).

Finally, in our Gittex 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., ‘?’), underscore (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 "#", "!", "*", etc. characters will only be used to denote one of our Gittex annotations and will not be found in the text).

Examples: This section presents some basic examples of our Markdown languages and its "compiled" HTML code (indented for readability).

The Gittex Markdown source code:

```
\BEGIN
\TITLE[The Simpsons]
# The Simpsons
\PARB
The members of the [The Simpsons](https://en.wikipedia.org/wiki/The_Simpsons) are:
\PARE
+ Homer Simpson
+ Marge Simpson
+ Bart Simpson
+ Lisa Simpson
+ Maggie Simpson
Here is a picture:
\\
![The Simpsons] (https://upload.wikimedia.org/wikipedia/en/0/0d/Simpsons_FamilyPicture.png)
\END
```

would compile to the HTML5 code (tabs added for readability):

```
<html>
    <head>
        <title> The Simpsons </title>
    </head>
    <h1> The Simpsons </h1>
    <p> The members of the <a href = “https://en.wikipedia.org/wiki/The_Simpsons”> The Simpsons</a> are: </p>
            <li> Homer Simpson</li>
            <li> Marge Simpson</li>
            <li> Bart Simpson</li>
            <li> Lisa Simpson</li>
            <li> Marge Simpson</li>
    Here is a picture:
    <br>
    <img src=" https://upload.wikimedia.org/wikipedia/en/0/0d/Simpsons_FamilyPicture.png" alt="The Simpsons">
</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 Gittex Markdown language, we could provide the source code:

```
\BEGIN
\TITLE[Variables]
\DEF[lastname = Simpson]
\PARB
The members of the \USE[lastname] family are:
\PARE
+ Homer \USE[lastname]
+ Marge \USE[lastname]
+ Bart \USE[lastname]
+ Lisa \USE[lastname]
+ Maggie \USE[lastname]
\END
```

would also compile to the HTML5 code

```
<html>
    <head>
        <title> Variables </title>
    </head>
    <p> The members of the Simpson family are: </p>
        <li> Homer Simpson</li>
        <li> Marge Simpson</li>
        <li> Bart Simpson</li>
        <li> Lisa Simpson</li>
        <li> Marge Simpson</li>
</html>
```

As you can see, the 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 \BEGIN block. To fully illustrate the scoping desired in our Markdown language, consider the following example in our Gittex Markdown language with two defined variables:

```
\BEGIN
\TITLE[Scoped Variables]
\DEF[name = Josh]
My name is \USE[name].
\PARB
\DEF[name = Jon]
Inside the paragraph block, my name is \USE[name].
\PARE
Now, my name is \USE[name] again.
\END
```

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 Gittex Markdown language. You will be required to meet two milestones during this project.

Phase 1: Grammar Design (10 points) **Deadline**: October 12, 2017, 11:59pm (Blackboard)
Write and submit a BNF grammar for our Gittex Markdown language. Your grammar should be parsable using a recursive-descent parser (as described in class and illustrated in Labs #1-4) 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 (60 points) **Deadline:** November 10, 2017, 11:59pm (Blackboard & GitHub)
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 Gittex Markdown language into tokens. To do this, you must use (i.e., `extends`) the Lexical Analyzer Scala trait (similar to a Java Interface) that I have provided (to be posted to Blackboard). In a good object-oriented design, it may be helpful to have separate Scala classes to represent token objects such as: *StartDocument*, *EndDocument*, *StartParagraph*, *EndParagraph*, etc. Any lexical errors encountered (e.g., \JOSH) should be reported as output to the console with as much error information as possible (i.e., similar to what a Java/Scala compiler provides). Your compiler may exit after the first error is encountered (i.e., System.exit(1)). 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., `extends`) my Syntax Analyzer Scala trait (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 Gittex 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/Scala 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 Gittex Markdown language. We will require and use the convention that *all* Gittex Markdown source files in our language will have an *gtx* extension (i.e., any file that does not have a gtx 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 60+ browser. To run a compiler that has been packed as an executable jar file, named compiler.jar, a Gittex Markdown input file, named input.gtx, I would issue the following at a command prompt:

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

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), return to the command prompt and open the generated html file in the users default browser (I will provide the code for this last requirement). For example, the above command should create an input.html file.

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

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 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 31, 2017 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 Scaladocs 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 GitHub. You should add me (cosc455fall2017) as a collaborator to your repository.

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

10/04	Project assigned
10/12	BNF & ANTLR grammar deadline (Blackboard)
10/13	BNF Grammar solution provided
10/31	Test cases provided
11/10	Implementation deadline

Environment: GitHub and the IntelliJ IDEA Community Edition IDE using Scala and SBT must be used in this Project along with Slack (#project1 channel) to communicate and submit portions of the lab. No other environment, programming language or submission will be allowed.

Grading: Phase 2 of your project will be graded as follows: 30 points allocated for following tasks 1-4; 5 points for good configuration management use (i.e., consistent updates, good commit comments); 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 Scaladoc (with good Scaladoc comments!) and generate the html files for it.

Submission: You must also submit the following in a zipped file via Blackboard:

- All developed source code in a directory named *src*.
- An executable jar file of your developed compiler in a directory named *bin*.
- A directory of the test case files you used for input in a *test* directory.
- A readme.txt file in the root directory that provides your GitHub repository address.

Any submitted projects not following these instructions will not be graded.