C-moji: A C Variant Transplied With Flex and Bison

Palmer, Tyler palmert0147@my.uwstout.edu
Strickland, Karl stricklandk0163@my.uwstout.edu
Zimmer, Mitch zimmerm0674@my.uwstout.edu

April 29, 2017

Abstract

This paper defines provides an overview a c-moji a c variant defined and transpiled using lex and yacc. In addition, this paper will provide a brief introduction to using Lex and Yacc in a Linux environment.

In the first section, the paper will introduce Lex/Flex a language tokenizer. The second section describes Yacc/ Bison, a tool used in conjunction with Lex/Flex to define the grammar. Yacc/Bison is also used to write the output C file. The third section will discuss using Lex/Flex in conjunction with Yacc/Bison. The fourth section will overview c-moji and document how to use it. In the final section several example programs will be provided.

This project utilized Ubuntu variant of Linux, but having access to Yacc/Bison tooling is the only hard requirement. It is possible follow along with another operating system, but note that some files will need to be changed and the setup process will be different.

It is also worth noting that the examples given describe how *we* used lex/ yacc to meet our goals. Both of these tools are incredibly powerful in their own right, and contain many features not discussed in this paper.

Additional Note: We started this project with the intention of making this a C-variant, but due to difficulty in implementing templated strings for print statements we opted to use iostream from C++. Therefore when we mention transpiling C-moji code to C we are really transpiling to C++.

1 Lex/ Flex

1.1 What is Lex/ Flex?

Lex and its open source counterpart Flex are forms of transpilers. Both perform the task of converting a specially defined .l file into a .c file. This .c is in turn responsible for tokenizing user input. Tokenizers perform the task of taking some input stream or file and breaking it down into parts called tokens. There are some differences between Lex and Flex. When installing the open source version, both commands **lex** and **flex** will call the same open source software depending on installation. For simplicity, we will refer to the term **lex** to talk about either the use of either lex or flex.

A token is a meaningful subset of text within a larger text. For example in the programming language c, you may have the line of code "int x = 5;". Flex will take this line of code and map each word into tokens which are meaningful to the language. A sample tokenization of this code may lead to the following mapping.

- int -> INTTYPE
- x -> VARNAME
- = -> ASSIGNOP
- 5 -> NUM

1.2 How to Use

In Ubuntu Linux you can use the command **sudo apt-get install byacc flex** to download and install flex.

Once installed, a file with the extension .l will need to be created. This file is how we will define the tokenizer. In order for the .l file to be properly transpiled to c so it can be run, it must follow a very specific structure. This structure is as follows.

Definitions
%%
Rules
%%
Sub Routines

1. Definition Section
   (a) Defines regular expressions which can be used in the rules section
   (b) Defines c code which will be imported directly into lex.yy.c
      i. Allows user to import c libraries
      ii. Allows user to declare functions defined in the sub routines section
2. Rules Section
   (a) Defines mapping between regular expressions and token output
   (b) Allows user to assign values to tokens for use in Yacc
3. Sub Routine Section
   (a) Allows user to implement functions declared in definition section
   (b) Functions defined in this section can be used in rules section

Once we have the defined the .l file, we will pass it to lex to create a new .c file called lex.yy.c. To do this, we will use the command **lex <fileName>**. lex.yy.c is what will be used to tokenize our language input. The general work flow of flex is diagrammed in figure 1.

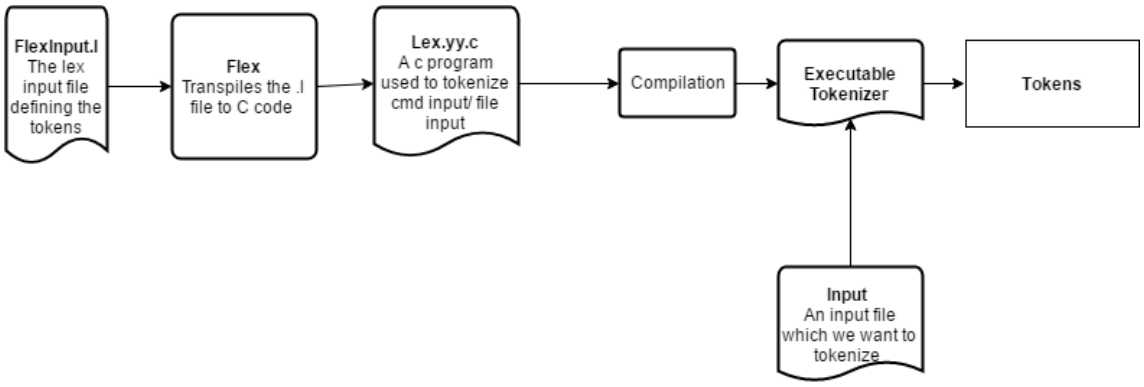

Figure 1: Lex Flow Diagram

1.3 Additional Resources

This was just a brief introduction to how lex / flex work. In addition to the features described there is much more that can be done with these tools. For more information use the following resources.

- https://www.youtube.com/watch?v=54bo1qaHAfk A video tutorial on how to use lex without yacc. Provides a good introduction to the syntax and features of lex.[1]
- http://dinosaur.compilertools.net/lex/ An introduction to using lex with information on the syntax of the .l file[4]
- https://github.com/zimm56/C-moji/blob/master/emoji.l The .l file from the c-moji language

2 Yacc/ Bison

2.1 What is Yacc/ Bison

Yacc and Bison are both transpilers similar in operation to Lex/ Flex. The job of Yacc/ Bison is to convert a .y file into a .c file (y.tab.c) (and potentially a .h file called y.tab.h). The .y file is used to define the grammar of the language. The .y file also defines functions to be called in when certain productions are used.

2.2 How to Use

In Ubuntu Linux you can use the command **sudo apt-get install bison** to download and install bison.

Once bison is installed a file with the extension .y will need to be created. This file is how we will define the grammar and functions related to productions. In order to properly transpile the .y file into c code it must follow a specific structure this structure is as follows.

Definitions
%%
Grammar
%%
Sub Routines

1. Definition Section
   (a) Defines the tokens for use in the grammar and in lex along with the token types
   (b) Defines the non-terminal symbol types in the grammar
   (c) Defines c code which will be imported directly into y.tab.c
      i. Allows user to import c libraries
      ii. Allows user to declare functions defined in the sub routines section
2. Grammar Section
   (a) Defines a grammar for the language
   (b) Defines functions related to the various productions in the language
      i. User can retrieve values from the various tokens/ terminal symbols
      ii. User can retrieve values from the various non-terminal symbols in the grammar
      iii. User can assign values to the non terminal symbols on the left hand side of each production for use in higher level productions
      iv. User can call any imported c function including print statements
3. Sub Routine Section
   (a) Allows user to implement functions declared in definition section
   (b) Functions defined in this section can be used in the grammar section

Once we have defined the .y file, yacc is used to tranpsile it into 2 files, y.tab.c and y.tab.h. y.tab.c will contain the c code needed to apply our grammar to the tokenized output from yacc. y.tab.c is also responsible for responding to each production with the functions defined in the .y file. y.tab.h will containe definitions for the terminal symbols defined in the definition section of the .y file. To run yacc and generate both y.tab.h and y.tab.c the command **yacc -d <fileName>** where the -d flag signifies generation of y.tab.h. The general work flow of yacc is diagrammed in figure 2.

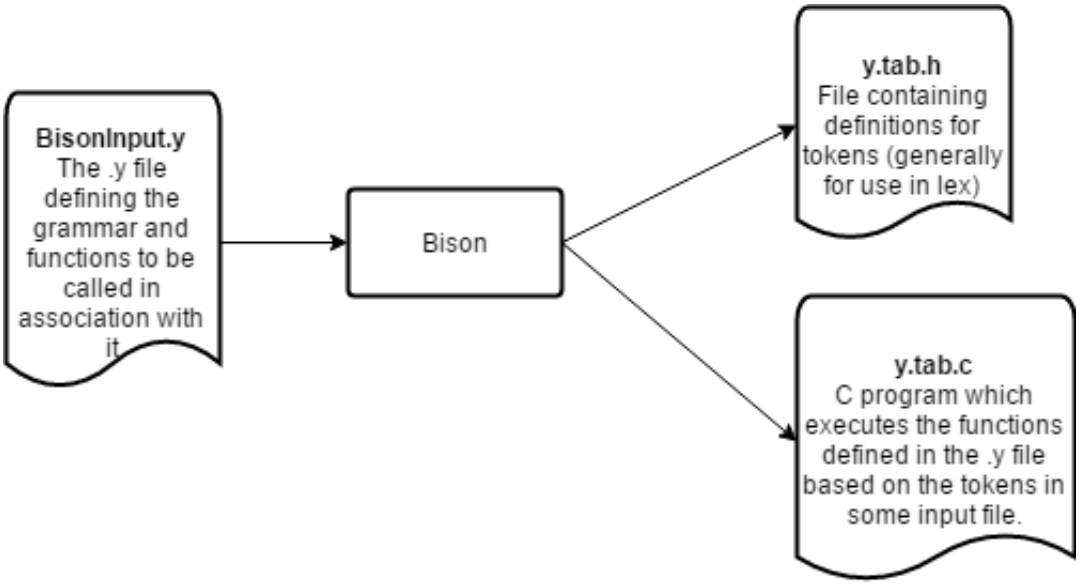

Figure 2: Bison Flow Diagram

2.3 Additional Resources

This was just a brief introduction to how yacc/ bison work. In addition to the features described there is much more that can be done with these tools. For more information use the following resources.

- https://www.youtube.com/watch?v=__-wUHG2rfM A video tutorial on how to use yacc together with lex. Provides a good introduction to the syntax and features of lex and shows how to write an interpreter. [2]
- http://dinosaur.compilertools.net/yacc/index.html An introduction to using yacc with information on the syntax of the .y file [3]
- https://github.com/zimm56/C-moji/blob/master/emoji.y The .y file from the c-moji language

3 Using Lex and Yacc Together

3.1 Why Use Lex and Yacc Together

Lex and Yacc are both incredibly powerful tools by themselves and can be used in standalone configurations. However, the real power from these tools comes when they are used together. In most configurations, when used together, our lex output will be used to tokenize our input file. The tokens found by lex will then be matched against our grammar using our output from yacc. The grammar matching process will then produce some form of output. In the case of c-moji this output is transpiled c code.

3.2 How to Use Lex and Yacc Together

In order The first step in running lex and yacc together is to generate the lex.yy.c, y.tab.h, and y.tab.c files. This process is described in detail for lex and yacc in sections 1.2 and 2.2 respectively. Since y.tab.h defines the tokens for lex, it is important that it is included in the top of the .l file prior to running lex on it. Once lex.yy.c, y.tab.c, and y.tab.h are all generated, we can compile them together using the command **gcc lex.yy.c y.tab.c -o output** (gcc command may vary by c compiler used) . This will produce an exectable called output which can be used to tokenize your input and run your grammar on it. The workflow of using lex and yacc together is shown in figure 3.

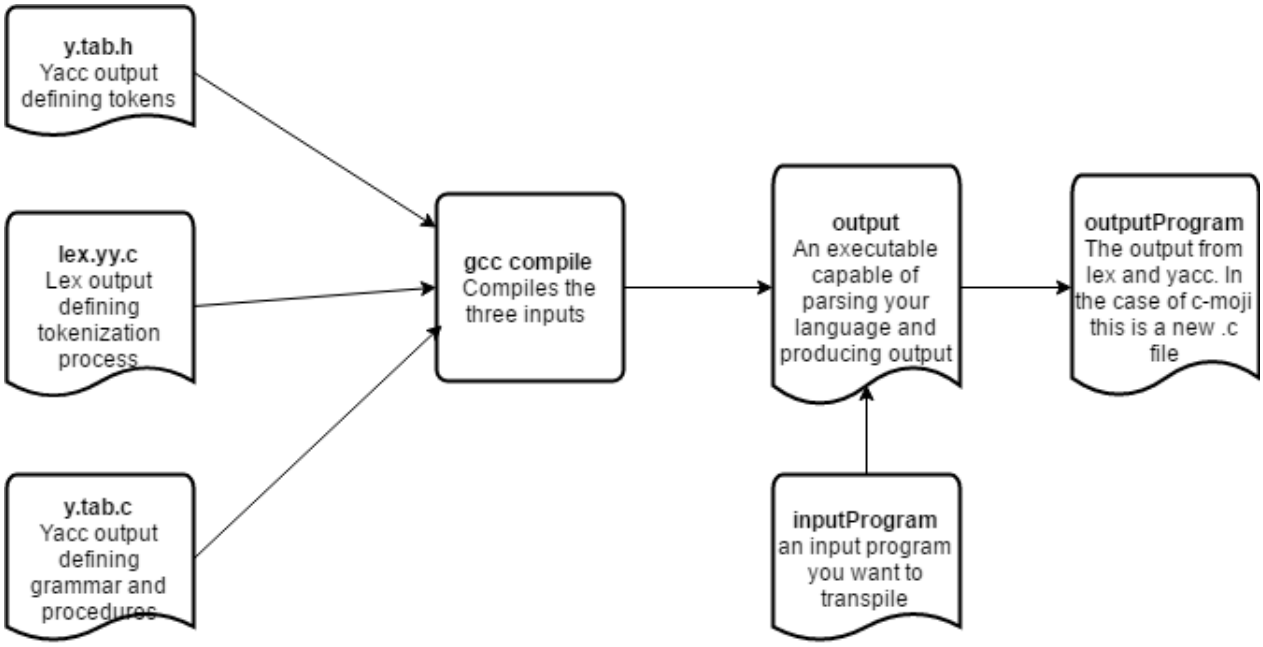

Figure 3: Lex and Yacc Flow Diagram

3.3 Additional Resources

- https://www.youtube.com/watch?v=__-wUHG2rfM A video tutorial on how to use yacc together with lex. Provides a good introduction to the syntax and features of lex and shows how to write an interpreter.[2]

4 C-moji Documentation

C-moji is a programming language that is as complicated to use as traditional C, but with the compactness of texting. C-moji is very similar to C in most aspects. In fact, the majority of the key words and expressions have a one to one mapping to the key words in C. Source files for transpiling, including the lex and yacc files is located on GitHub[5].

4.1 Transpiling and Compiling

Currently C-moji source files can be transpiled into C, with the 'compile.sh'. After being converted into C, using an optimized C compiler, for example, gcc, to compile that into a binary.

1. Building Compiler Files
   (a) Running **./buildTranspiler.sh** will run yacc on the emoji.y file as well as lex on the emoji.l file. The product of those operations is compiled to form an output file, **transpiler**. This file is used to take text input usually in the form of a file and output transpiled code to a file.
2. Transpiling .oji files to .c
   (a) Running **./transpile.sh [input.oji]** will run the contents of the provided input.oji file through the lexical analyzer and use the yacc grammer definitions to convert .oji syntax into traditional c code in an output file, **input.cpp** .
3. Compiling
   (a) When it is the desire to compile the program, executing **./compile.sh [input.oji]** will perform this. First the operations of running ./transpile.sh are executed, followed by compiling the resulting c code into a **input** file.

4. Compiling and Running

   (a) When it is the desire to transpile and run the program all in one set, executing **./compileAndRun.sh [input.oji]** will perform this. First the operations of running ./compile.sh are executed, followed by compiling the resulting c code into a **input** file, which is then executed.

Note that this compilation process requires lex, yacc, and a c compiler to be installed. To transpile/compile the emoji.l and emoji.y files must also be present.

4.2 Reserved Words

The following are resereved symbols in the langauge. the '-' symbol is used to seperate c-moji symbol from a C version and a definition. Symboles with direct C translations are included in parenthesis after the definition to prevent symbol confusion.

- ˜8} - main
- :(- Opening parenthesis
- :) - Closing parenthesis
- 8) - Quotation Mark (")
- :} - Closing curly brace
- :{ - Opening curly brace
- ;) - semi-colon
- :-) - returning void for functions
- @:-) - int: integer for describing both return values of function and initilizing integer variables.
- @@:-) - char: Used as the return value of functions and as type definitions for variables.
- @@@:-) - char*: Used to define return types on function of a char *.
- (>'.')> - kirby can print value that are passed in on the right (»).
- <('.'<) - kirby assigns values from the right value to the left element («).
- V̂ new line character.
- C(_) - while: with coffee much work can get done inside this loop.
- O.o - if
- o.O - else
- (c) - continue: with more coffee, any computer can continue for days.
- :*(- Multiplication sign (*)
- :-(- Subtraction sign (-)
- :+(- Addition sign (+)
- ! =:] - Not equal (!=)
- >=:] - Greater than or equal (>=)
- <=:] - Less than or equal (<=)
- <:] - Less than (<)
- >:] - Greater than (>)
- ==:] - Equal comparision (==)
- | :] - Or comparision (||)
- & :] - And comparision (&&)
- </3 - broken hearts are sad (break)

4.3 Number

Integers can be used similar to in C and are entered as binary. For c-moji,

- : - represents a one
- . - represents a zero

ex.

: . :: .. : . :::: . :: ..... : . : . :

Becomes 23456789.

4.4 Characters

Characters are entered as ascii values, base 2 where 1 is Z, and 0 is z.

ex. $(-.-)zZzzZzzz(>'.') > (-.-)zZZzzZzZ(>'.') > (-.-)zZZzZZzz(>'.') > (-.-)zZZzZZzz(>'.') > (-.-)zZZzZZZZ(>'.') > (-.-)zzZzzzzz(>'.') > (-.-)zZzZzZZZ(>'.') > (-.-)zZZzZZZZ(>'.') > (-.-)zZZZzzZz(>'.') > (-.-)zZZzZZzz(>'.') > (-.-)zZZzzZzz(>'.') > (-.-)zzZzzzzZ$
Becomes 'Hello World!'.

4.5 Functions

Functions are named with the following pattern: ˜ ^{n}8}, where $n \in N$, and $n \geq 1$. Everyone else is going to be amazing with all the catepillers and snakes running through all the code.
ex.

˜˜˜˜˜˜8}

4.6 Variables

Variables are labeled as Caterpillars, (: @$^n D$, with increasing @ symbols.
ex. (: @@@@@@@@@@@D

5 Example Programs

We included some example programs to help get started with learning c-moji. They also demonstrate the readability and compactness of c-moji.

5.1 Power Function

5pow5.oji is an example proram that cacluated 5 rasied to the 5th power in c-moji.

```
˜˜˜˜˜˜˜˜˜8} @:-) :( @:-) (:@@@D :,) @:-) (:@@@@D :)
:{
	@:-) (:@@@@@D <('.'<) .... ;)
	@:-) (:@@@@@@D <('.'<) ...: ;)
	C(_) (:@@@@@D <:] (:@@@@D
	:{
		(:@@@@@@D <('.'<) (:@@@@D :*( (:@@@@@@D ;)
	(:@@@@@D <('.'<) (:@@@@@D :+( ...: ;)
	:}
	}:) (:@@@@@@D
:}

˜8}
:{
	@:-) (:@D <('.'<) :.: ;)
	@:-) (:@@D <('.'<) :.: ;)
	(>'.')> textasciitilde˜˜˜˜˜˜˜˜˜8} :( (:@D :,) (:@@D :) ;)
:}
```

5.2 Is Prime

The follow is an example program calculating if a number is prime.

```
    ~8} :{
       @:-)(:@@D<('.'<).;)
       @:-)(:@@@D<('.'<).;)
       @:-)(:@@@@D<('.'<):.:;)
       @:-)(:@@@@@D<('.'<).;)

       C(_) (:@@D <:] (:@@@@@D :{
              C(_) (:@@@D <=:] (:@@D :{
                     O.o (:@@D :*( (:@@@D ==:] (:@@@@D :{
                            (:@@@@@D<('.'<):;)
                            </3
                     :}
                     o.O :{
                     :}
                     (:@@@D <('.'<) (:@@@D :+( : ;)
              :}
              (:@@D <('.'<) (:@@D :+( : ;)
              (:@@@D <('.'<) . ;)
       :}
       O.o (:@@@@@D ==:] . :{
              (>'.')> (-.-)zZzZzzzz ;)
              (>'.')> (-.-)zZZZzzZz ;)
              (>'.')> (-.-)zZZzZzzZ ;)
              (>'.')> (-.-)zZZzZZzZ ;)
              (>'.')> (-.-)zZZzzZzZ ;)
       :}
       o.O :{
       :}
:}
```

References

[1] Jonathan Engelsma. Part 01: Tutorial on lex/yacc. https://www.youtube.com/watch?v=54bo1qaHAfk.

[2] Jonathan Engelsma. Part 02: Tutorial on lex/yacc. https://www.youtube.com/watch?v=__-wUHG2rfM.

[3] Stephen C. Johnson. Yacc: Yet another compiler-compiler.

[4] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator.

[5] Mitchell Zimmer Tyler Palmer, Karl Strickland. C-moji source code. https://github.com/zimm56/C-moji.