# Project 2 Writeup
# Simple Grammar Analyzer
# CS 311

Mike Lane

5 June 2015

## Overview

I chose to use Ruby for the programming language and YAML to encode the CFG. This turns out to have been a reasonably good choice. Ruby offers a wide variety of built-in functions on strings (which constitutes the basic building block of everything I was working with), offers built-in YAML encoding and decoding, and YAML is exquisitely readable. Additionally, I was pleased with the object-oriented nature of Ruby and the fact that I could simply test a class creating an object of the class right in the same file as the class and I could sample the methods right after they were created. Ultimately, that made for code that came together much more quickly and much easier than I could have done with a language like C++.

## Explanation of my approach to each step

### Encoding a Grammar

In Karla's classes I spent far too much time struggling with file I/O to want to be bothered with it anymore. Ever. Besides, someone (or some group) who is a much better programmer than I am has already made file I/O ridiculously easy for most programming languages - Ruby included.

For Ruby, it seems, the preferred method is YAML. When I investigated YAML, I found that it is highly human-readable language (which is, apparently, a superset of JSON) that is a bit more complex (potentially) than JSON. As with node.js and JSON, there is a built-in YAML parser in Ruby. This will write a Ruby object out as a perfectly-formatted YAML file and

it will read in a YAML file into a fully formed Ruby object. This is good since I just could not for the life of me figure out how to create the YAML file by hand. Once I got the brilliant idea to create an object in Ruby first and then dump that object into a YAML file, life became good again. As an added benefit of that approach, writing the code to do that also wrote a UI for creating new CFGs. Handy!

### Reading the Grammar

Since the first step of encoding a grammar was to first make it an object in Ruby, I naturally chose to read the grammar into an object of the same type. And since Ruby offers the same built-in YAML parser as well as encoder, this became nothing more than a simple function call. I did find that the process of loading the YAML file simply loaded a string into the buffer and then there was an extra step to parse that string into an object. Either this is the intended method or I'm not fully understanding some part of the intended process. Honestly, let's face it, it's probably the latter. Nevertheless, the end result is a simple object method.

I chose a command line interface because I have no idea how to do anything else. I should look into that. Ruby has plenty of simple built-in features to assist with that. I chose to create pretty menu boxes using UTF-8 corners and sides and what not because Ruby makes it easy and part of my personal goal was to learn a new language. Also, Ruby makes it easy to read the contents of a directory, so I chose to simply offer a menu of possible CFGs for the user to choose from when reading in so I could avoid having to deal with error handling. Of course, if the subdirectory I used gets changed or removed, this would be a problem that my program wouldn't be able to handle. Please don't do that!

### Parsing the String

Using a recursive function seemed like a natural choice for the string-parsing function. The algorithm is set up so that either the input string gets evaluated one character at a time until it is empty or it finds an error and quits. Hence, if it is written properly, there is no chance of an infinite loop. That said, I had to find and fix several infinite loops during the process of writing it.

Quite simply, I used the steps in the algorithm provided as the conditional statements in my recursive function. First, I printed out the contents of the stack and input tape. Next, I checked to see if the ACCEPT state

was satisfied. Then I checked to see if the stack was empty when the input wasn't (a REJECT condition). After that I popped an element off the stack and

1. if I popped a variable, I peeked at the next character in the input tape and I concatenated the two characters with an arrow between them (*e.g.* S→a – note LaTeX doesn't give me an easy way to print the *actual* arrow used). I used this concatenated string to compare with the first 3 characters of each of the strings in the relations array to find the next relation to use. If no matching rule was found, this was a REJECT state; otherwise, I pushed the right side of that string onto the stack and called the function again with the new stack and input.

2. If I popped a terminal, I popped the next character off the input string and if the character popped off the stack does not match the character popped off the input, then REJECT; otherwise, call the function again on the modified stack and input.

Finally, as a guard against a malformed CFG, I checked to make sure that the item popped off the stack was in the sigma array. If not then REJECT.

## Testing

For each class I identified as being required, I was able to write a test to go along with it right in the class definition file itself. So for each method written, I would write a simple test that would take good and bad input and would then verify that the output was correct. This definitely minimized the time required to build the project as a whole since I was able to catch simple errors long before they propagated into other methods in the same or other classes. The rest of my testing was focused around testing strings that should be accepted, strings that should be rejected, and corner cases.

An empty string is handled gracefully. I can't find a string in any of my grammars that is accepted and shouldn't be. I can't find a string that should be accepted and isn't. And there is a limit of 1024 characters on input (including the ¡enter¿ key, so 1023 characters, really). I don't know where that limit comes from; however, my program handles strings of that size just fine.

## Questions

### *Why were those particular restrictions placed on the grammars?*

1. The input string contains only terminals, so if a variable is popped off the stack, the way we find the next relation to use is to look at the relations and find the matching terminal on the right-hand side. If there was no rule that the right-hand side must start with a variable, this wouldn't work.

2. Similarly, if any rule had the same variable on the right-hand side, then it would be impossible to determine which rule to use next for the next input character.

### *If those restrictions weren't there, how would your program need to change?*

Maybe we could solve that by creating a new input stack each time an ambiguity arose. Make a clone of the current stack and push both possible rules onto it. Continue processing but now all actions should be taken on all of the stacks at each step. If any step requires further stacks, those stacks should get cloned and evaluation should continue. If there is more than one stack at any given time and if that stack enters a reject state, delete that stack; otherwise, if there is only one stack left that enters a reject state, reject everything. If any stack enters an accept state, accept the entire input.

### *What would happen if the grammar were ambiguous?*

I suspect that if the grammar were ambiguous, that this approach wouldn't work. Either you'd have to have a rule where the right hand side started with the same terminal or the rule would have to start with a variable.

### *What other approaches might be used to tell if a string can be generated by a given grammar?*

We might consider constructing an actual PDA in a similar manner that we encoded a DFA in Program 1. After all, a CFL is just a language that can be recognized by some PDA.