CS 96SI; Assignment 1; Word Salad

Orignal Idea by Mr. Alex Rylan, Adapted by Joseph Victor

April 18, 2012

Introduction

This assignment has two main parts. First, you will write a Scheme program which takes as its input a Context-Free Grammar (CFG) and outputs a random sentance in the language of that grammar. The second part will be to come up with a grammar for whatever you want. These will be shown in class a sort of competition, as the output can be quite amusing.

Just in case you have forgotten, a CFG is a set of symbols and rules for mapping between those symbols. For instance, here is a CFG for arithmetic expressions with numbers, + and *:

```
Exp  -> Exp + Exp  | Exp * Exp | Numb
Numb -> Dig | Numb Dig
Dig  -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The vertical bar | reads "or", so we read the above as an expression is either a either an expression plus an expression, an expression times an expression, or a number. A number is either a single digit or a number followed by a digit (this is syntactic. We really mean the number 83 looks like the number 8 followed by the digit 3). Of course a digit is what it is.

Here is an example of exploding this grammar into a sentance, starting with Exp, doing exactly one rewrite per arrow.

```
Exp -> Exp * Exp -> Numb * Exp -> Dig * Exp -> 4 * Exp -> 4 * Exp + Exp ->
4 * Numb + Exp -> 4 * Numb Dig + Exp -> 4 * Numb Dig Dig + Exp ->
4 * Dig Dig Dig + Exp -> 4 * 385 + Exp -> 4* 485 + 23
```

Where I skipped some steps in the last line, because I think you get the point. Of course, the cool thing about CFGs it that they are context free, so you are free to expand multiple branches at the same time. The expanded form can be represented by a Scheme expression, as follows, where we tag each node with the symbol that expanded to it:

```
(Exp (Numb (Dig 4)) *
     (Exp (Exp (Numb (Numb (Numb (Dig 3) (Dig 8)) (Dig 5)))  +
          (Exp (Numb (Numb (Dig 2)) (Dig 3))))))
```

If you know anything about parsing, this is sort of like you're parse tree. The actual thing to eventually be printed is of course

```
4 * 485 + 23
```

You're job is to generate the Scheme expression above, and then walk the tree, printing out the thing you need.

Getting Grammatical

We havn't talked much about structs in Scheme, but you can imagine how they would go, and anyways, this has been implemented for you the low tech way. The avaiable functions are pretty self explainatory, but I'll explain them anyways.

There are two structs, an "expansion" struct and a "rule" struct. An "expansion" contains basically one thing between a set of verticle bars in a grammar, but with some extra information. In general, some rules could have a higher probability of firing than others, and some rules could be more "expensive" in terms of how big they make the tree, so each rule gets an associated probability and weight. You can make, test, update and acess your data with the available functions in the first chunk grammar_structs.scm. I think this should be clear, but if not, shoot me an email.

The second struct is a "rule" struct. This struct contains a symbol (that is, the LHS of each line in the grammar), a list of expansions (the RHS), and a function which prints the expanded thing, given a function which can print any expansion.

Now, we also have functions to make rules and expansions in less generality, since most of the time all the probabilities are the same, the weights are just a simple function of expansions and the print function is just print

each symbol. The file "examples.scm" contains two examples of grammars in this format.

JUST BECAUSE THE STRUCTS ARE LISTS DOESN'T MEAN YOU SHOULD USE THEM WITHOUT THE INTERFACE.

Consider the structs as black boxes, only accesable through the interface. This is of course good programming practice, even if the language in question does not enforce this requirement. In fact, we will see later how to make this requirement enforced, for now just consider it the polite thing to do. However, if you think you need a different struct, use the same strategy I used to implement them by hand.

Expanding the Grammar

Once you've gathered all the rules you need for a specific symbol, each with their associated probability and weight, we need to write the expansion function. The expansion function for a given symbol takes a single argument, the allowed weight, and returns one of the rewrites, chosen randomly perportional to the probabilities, subject to the constraint that the weight of the chosen rewrite is less than the weight of allowed weight. This means that, for each rule with an allowable weight, the probability of that rule being chosen must expand a bit. As a concrete example, suppose you have 4 rules with probability .25 each, and one rule has weight 2 and the rest have weight 0, and suppose that allowable weight is 1. Then we need each of the 3 allowed rules to be selected between with probability $1/3$. Mathematically, the if you have a bunch of rules R_1 through R_n with associated probabilities P_1 through P_n and weights W_1 through W_n, the probability that you choose rule k when the allowable weight is w is given

$$Pr(R_k \text{ is picked}) = \begin{cases} 0 & W_k > w \\ \frac{P_k}{\sum_{i|W_i \leq w} P_i} & W_k \leq w \end{cases}$$

There is a simple way to organize you're data so this just happens without too much pain.

Try to practice really using functions. There are many ways to do this, but a "functional" way might be to write a function called make-expansion-function which takes a list of rules and returns a function which takes the max-allowed weight and returns the rule. Practice using let-over-lambda style to set up you're data when you create the function.

That was the hard part, and not so bad. Next, you need to write a recursive function which fully expands the grammar. This just calls the appropriate expansion function on an expandable symbol and then recursively

expands out the new nonterminals created by that expansion, subtracting appropriately from the weight as you go. How do you select which expansion function to use? Again, I reccomend organizing you're data using your new favorite data container type, the function. Congratulations, you now have built a tree representing a sentance in you're grammar.

But I Don't Want a Tree

QUIT WHINING!!

Actually, yes, the tree is hard to read (for a human!) and not really any fun. Now, each expansion rule has a print function which takes as arguments a "master-print-fun" function and a the gramatical tree you want to print as a string. You're job is to write the master-print-fun function in terms of the print functions for each symbol, that is, you have to write a make-master-print-fun function which takes as input the print function for each symbol and outputs a function that returns a string representing the gramatical tree without the annotations. For instance, it should turn

```
(Exp (Numb (Dig 1)) + (Numb (Dig 2)))
```

to $1 + 2$. The print function for each symbol has one argument being the master-print-fun, so when writing the master-print-fun you will have to pass the function you are writing into the arguments you are passing in. Oh, what a functional recursive rock n roll hell we have gotten ourselves into.

Applied Programming

Now is the fun part, you get to design you're own grammar! I don't care if its for general english sentances, limericks (or other poetry), mathematical proofs (see theproofistrivial.com, except with this technology you could probably make something better than that), context-free C++ (which is useless), arithmetic expressions (not so useless actually, you can build a random smooth function generator), or anything else you can think of. One of the example files included has a simple english grammar with a small ad-hoc dictionary, but I think a more complicated grammar could generate more realistic (and hilarius) sentances. In fact, we will make a competition out of it :)

I have included an english dictionary that gives parts of speech and stuff. This dictionary includes words, parts of speech, syllable counts and phonetic pronounciations of english words. I will be adding a key shortly, as well as a parser which parses the dict.txt file so you don't have to figure that out.

Deliveriables

Since everybody is making their own grammar, everybody should send me, via email:

1) Their code, nicely commented, working, tested, etc

2) Top 4 or 5 outputs from the grammar they wrote, which will be shown in class for the amusement of all.

3) A short report on how you did it, what you learned

4) Any comments on the assignment itself. This is the first time I have ever written my own assignment, and any feedback is HUUUGELY appriciated. In particular, I want to know what you thought of the difficulty, clarity and programming fun of the project. What was the hardest part? What was the most fun part? How long did you spend?

Hints and other comments

Think about using the stuff we have learned in class to make you're programs pretty. If a solution is getting cumbersome, do the things programmers do to make their programs pretty, and USE the idiom.

Do NOT use begin, set!, or other imperitive stuff for this assignment.

Also, try to put type constraints in the comments by functions. Scheme doesn't care, but you do!

If you have any comments or questions, or if anything is at all unclear, please send me an email, and I will do my best to clarify.

Best of luck :)