Learning grammaR

Samantha L. Davis

August 19, 2015

1 Introduction

`grammaR` is a parser for context-free grammar. It is basically a go-between for several different scripts built by other people. `grammaR` should enable you to do things like spin articles and stories, build dictionaries of synonyms, generate cover art, and even build epubs for submission to places like Amazon Kindle and Smashwords.

In this manual, I will demonstrate some of the common functionalities of `grammaR` and hopefully get you on your way to creating masterpieces.

2 Dependencies

Unfortunately, you will need several pieces of software to run `grammaR`. I'll list them below then briefly describe how to get them in a few paragraphs.

- R, from `http://cran.r-project.org`
- grammaR, from `http://www.github.com/ecology-rocks/grammaR`
- ImageMagick, from `http://www.imagemagick.org`
- Pandoc, from `http://www.pandoc.org`
- LaTeX, as outlined in the pandoc installation instructions above
- Perl, which comes installed on Mac and Unix (`http://www.perl.org`)
- OCaml, from `http://www.ocaml.org`

Many of these can be downloaded using a package manager like Homebrew or Macports, or Windows alternatives. Feel free to contact me if you run into problems with installation.

3 How It Works

Here's how your dictionary gets built. The standard format for a dictionary is as follows:

- `key ::= value1 | value2 | value3`
- `key_ran ::= ran | skipped | jumped | hopped`

So if you have a script that looks like this:

- `top ::= <myCharacter> <keyRan>.`
- `myCharacter ::= Jane | John | Julie`
- `keyRan ::= ran | skipped | jumped | hopped`

It would generate one of the following combinations:

- Jane ran.
- Jane skipped.
- ...
- Jule hopped.

You can see how quickly, with just a few lines of code, you can generate many, many stories or possibilities. The recursive grammar also works through multiple levels, so if you call a key within a key, it will generate just fine, e.g.:

- `top ::= <myCharacter> <myAdjective> <keyRan>.`
- `myCharacter ::= Jane | John | Julie`
- `keyRan ::= ran | skipped | jumped | hopped | <myAdjective> <keyRan>`
- `myAdjective ::= quickly | strangely`

This script would generate with a possibility of "myAdjective" being inserted multiple times, e.g., "Jane quickly strangely ran."

So everything about this package is based on this premise, and helping you write great stories, or articles, or whatever.

4 Building Your Dictionary

You can build a dictionary that will be loaded into R from a text string in R, like so:

```
> ## load library
> library(grammaR)
> ## build dictionary from string
> myDic <- buildDictionary("rawr ::= space | magic", readfile=F, formatKey=T)
> ## show dictionary
> myDic

     key   val
1 <rawr> space
2 <rawr> magic
```

You can also build a dictionary from a text file in R:

```
> ## create a text file with 2 lines with dictionary values
>   cat("rawr ::= 1 | 2",
+       "boo ::= a | b",
+       file="ex.txt",
+       sep="\n")
> ## read them into R
> myDic <- buildDictionary("ex.txt", TRUE)
> ## show results
> myDic

     key val
1 <rawr>   1
2 <rawr>   2
3  <boo>   a
4  <boo>   b

> ## delete text file
> unlink("ex.txt")
```

There is also an interactive option that I can't compile automatically in this document, but you can access it using makeNewDictionary() – example output below.

```
> # store <- makeNewDictionary()
> #  > Do you need this to build a dataframe or a file? Answer d/f.
> #  d
> #  > What is the key for this set of values?
> #  rawr
> #  > How many values are you entering?
> #  2
> #  > What is your value?
> #  osaurus
> #  > What is your value?
> #  rex
> #  > Are you done?
> #  y
> #  store
> #   >    key     val
> #   > 1 rawr osaurus
> #   > 2 rawr     rex
```

5 Cleaning Your Dictionary

When you're loading from a text file, you can often have repeat values. This is exceptionally common in fiction. You might include "said" in "keySighed" and "keySmiled" as alternates for the words sighed and smiled. When this happens, if you're using the spinStory feature, it will only choose one to operate on, and you might get unexpected results.

To get around this, I created a function called getUniques, which removes any duplicated value in the dictionary. So if "said" is in there twice, both instances will be removed. Then you can go back through your story and choose different words, or manually add an entry to your dictionary to make a rule for "said."

```
> ## create duplicates in values in myDic
>  myDic[,2] <- c("Arbys", "McDonalds", "McDonalds", "Burger King")
> ## show dictionary
> myDic

     key         val
1 <rawr>       Arbys
2 <rawr>   McDonalds
3  <boo>   McDonalds
4  <boo> Burger King

> ## show getUniques function
> getUniques(myDic)

     key         val
1 <rawr>       Arbys
4  <boo> Burger King
```

If you're having problems playing with getUniques(), you may have values that are too short. getUniques() automatically weeds out any values that are only one character long.

One other thing that you'll need to do if you're going to use the spinStory() function described later is to remove any keys in the values column. This just has to do with the current state of grammaR, and may be supported later. To remove those, you can do something like:

```
> ## reassign for example
>   myDic[,2] <- c("apples", "<rawr>", "bananas", "broccoli")
> ## find the first character of "val" column
> ## and put it in a new column named "clean"
>   myDic$clean <- substr(myDic$val,1,1)
> ## remove any rows where the "clean" column contains <
>   myDic <- myDic[-which(myDic$clean=="<"),]
> ## delete the clean column
>   myDic <- myDic[,-3]
> ## show results
>   myDic

     key      val
1 <rawr>   apples
3  <boo>  bananas
4  <boo> broccoli
```

6 Spinning a Story

Okay, you have a dictionary: now what? The spinStory() function in this package takes normal text, e.g., ”A man walks into a bar” and spins it ”up” based on your dictionary. So, if you have a dictionary with the values ”man” and ”woman” in a key labeled ”character” and you run it through, you would get a story output that looked more like, ”A <character> walks into a bar.”

```
>   ## story file
> cat("A man walks into a bar and steps on a bug.", "",
+     file="a.txt", sep="\n")
>   ## dictionary file
>  myDic <- buildDictionary("character ::= man | woman | alien | horse", F)
>  ## bind the old mDic to a new myDic entry
>  myDic <- rbind(myDic, buildDictionary("place ::= bar | restaurant | home | lake", F))
>  myDic

          key        val
1 <character>        man
2 <character>      woman
3 <character>      alien
4 <character>      horse
5     <place>        bar
6     <place> restaurant
7     <place>       home
8     <place>       lake
```

Ok, so there are our needed components, now, we should just be able to use spinStory() to generate the new file. Here, "exec" is set to false, and the program prints the output to our console in a list format (all of those brackets), but if you set exec=T, it will write to the file name where "test.txt" is in the function.

```
>   ## make list of file names
>   spinStory("a.txt", myDic, "test.txt", exec=F)

[[1]]
[1] "A <character> walks into a <place> and steps on a bug."

[[2]]
[1] ""
```

7 Generating Characters

Another neat thing you can do with this program is automatically generate character names. To do this, we rely on another package in R called `randomNames`. If it didn't install automatically when you installed the grammaR package, you can do so with the following code, just uncomment it:

```
> # install.packages("randomNames")
>  library(randomNames)
```

After you load the library, you can use the default function to generate some names:

```
> generateCharacters("mainChar", "F")

[1] "mainChar ::= Holly \n "

> generateCharacters(c("char1", "char2"), c("f", "m"), writefile=T, outfilename="char.txt")

[1] "char1 ::= Marisa \n char2 ::= Justin \n "
```

These names are meant to be put into a file, which you can specify by setting writefile to true and outfilename to your intended filename. When I'm generating in bulk, I create a character file each time I run my script, and it gives my characters new names.

8 Putting It All Together

Finally, let's say you have a story file (with your spun story), a top file that starts it all, a dictionary file, and your character file. You need to combine these into a final file that will then be read by the Perl/OCaml files and turned into a story. Let's do a simple example here.

```
> ## your first file, read in alphabetically,
> ## will be the starting point of the program
> cat("start ::= My Story <linebreak> Sam Davis <linebreak> <story>",
+     "linebreak ::= \\n",
+     file="a.txt", sep="\n")
> cat("story ::= Once upon a time, <char1> kissed <char2>",
+     file="b.txt", sep="\n")
> ## char.txt already got generated with
> ## the values of <char1> and <char2> inside.
> myFiles <- list.files(pattern=".txt")
> myFiles

[1] "a.txt"    "b.txt"    "char.txt"

> combineFiles(myFiles, "myStory.gram")
> ## show the file
> readLines("myStory.gram")

[1] "start ::= My Story <linebreak> Sam Davis <linebreak> <story>"
[2] "linebreak ::= \\n"
[3] "story ::= Once upon a time, <char1> kissed <char2>"
[4] "char1 ::= Marisa "
[5] " char2 ::= Justin "
[6] " "
```

And there we have it, a .gram file generated! ...Oh wait, we need to actually run the script and get the recursive story!

```
> createStory("myStory")
> readLines("myStory-1.tex")

[1] "My Story \\n Sam Davis \\n Once upon a time, Marisa kissed Justin "
```

So if you loop these say, 10 times, you'll get 10 different stories! I can show you the power of that here:

```
> for(i in 1:10){
+   generateCharacters(c("char1", "char2"),
+                      c("f", "m"),
+                      writefile=T,
+                      outfilename="char.txt")
+   combineFiles(list.files(pattern=".txt"), "myStory.gram")
+   createStory("myStory", 1)
+  print(readLines("myStory-1.tex"))
+ }

[1] "My Story \\n Sam Davis \\n Once upon a time, Sara kissed Jared "
[1] "My Story \\n Sam Davis \\n Once upon a time, Carly kissed Jonathan "
```

```
[1] "My Story \\n Sam Davis \\n Once upon a time, Nouha kissed Steven "
[1] "My Story \\n Sam Davis \\n Once upon a time, Kelsey kissed Sean "
[1] "My Story \\n Sam Davis \\n Once upon a time, Bethany kissed Christopher "
[1] "My Story \\n Sam Davis \\n Once upon a time, Hannah kissed Connor "
[1] "My Story \\n Sam Davis \\n Once upon a time, Autumn kissed Matthew "
[1] "My Story \\n Sam Davis \\n Once upon a time, Sierra kissed Nathan "
[1] "My Story \\n Sam Davis \\n Once upon a time, Kelsie kissed Dustin "
[1] "My Story \\n Sam Davis \\n Once upon a time, Brianna kissed Brett "
```

And you can modify as needed, of course, for when you actually want to save your outputs.

9 Experimental Features: Ebook Covers

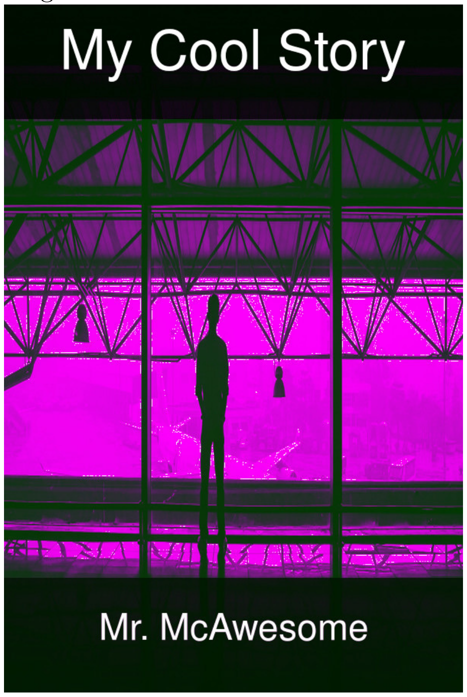
These features are currently supported for *my* workflow, which is: `file.gram` to `file.tex` (LaTex) to `file.pdf` or `file.epub`, using pandoc. If you're producing your ebooks in other formatting styles, then you'll need to extend this package to support these experimental functions.

To demonstrate, let's get a legal LaTeX file ready, something a little more complex than the .tex file that we generated above.

```
> cat("\\begin{document}",
+     "\\title{My Cool Story}",
+     "\\author{Mr. McAwesome}",
+     "\\maketile",
+     "\\chapter{The Start}",
+     "Once upon a time, there was a girl, and she was awesome.",
+     "\\end{document}",
+     file="test.tex", sep="\n"
+     )
> authordf <- getAuthorTitles(filename="test.tex")
> authordf

          title        author
1 My Cool Story Mr. McAwesome

> str(authordf)

'data.frame':	1 obs. of  2 variables:
 $ title : chr "My Cool Story"
 $ author: chr "Mr. McAwesome"
```

So the code above will generate a tex file and then extract the author and title from it. Then, you can use that author data.frame (or build your own to use) to generate covers.

Figure 1: Our converted ebook cover.



```
> makeCommandsCovers(authordf, stockdir="./", exec=T)
>
> ## use ?makeCommandsCovers to check out where your files belong.
```

And finally, we can generate an epub file using this command:

```
> store <- makePandocCommand(authordf, filename="test.tex")
>
> ## execute store using the system command to generate an epub.
> #   system(store)
>
```