

CS51 Project 1b: Aye, Sea Turtles

Due: Tuesday, 24 February 2009 at 11:59 PM

Total Points: 50 (including 5 style points)

1 Introduction to Pirate CFG

Ahoy me harties! You are going to teach your computer to speak pirate! Well, not really, but you are going to write a random sentence generator using a formalism called a Context Free Grammar (CFG). To do this, you will need to learn how to use Scheme's `random` function and learn how to work with a CFG.

For this part of the project, please use the Intermediate Student with Lambda language.

1.1 What is a Context Free Grammar?

A *grammar* is a formalism for representing constraints about a language. Grammars can, in fact, be considered a “theory” of language, since a grammar, by its very nature, contains a description of the constraints that determine whether a particular word sequence is a valid sentence or whether it is disordered garbage. For our purposes, a grammar will consist of a collection of rules (to be represented in Scheme as a dictionary). For this assignment, we will use a specific type of grammar, known as a *context free grammar*, where each rule consists of a single item mapping to one or more sets of items.

For this assignment we will have two different types of CFG rules:

1. *non-terminal* rules which map to a list of sequences of terminal or non-terminal rules
2. *terminal* rules map to a list of possible words

Here is an example of a CFG for a small subset of English¹:

¹If you are curious about the letters used in these rules, they are fairly standard abbreviations from Linguistics: S = sentence, NP = Noun Phrase, VP = Verb Phrase, Det = Determiner, N = Nouns, V = Verb, Name = Proper Name

- Non-terminal rules:
 - $S \rightarrow NP VP$
 - $NP \rightarrow Det N \mid Name$
 - $VP \rightarrow V NP$
- Terminal rules:
 - $Det \rightarrow \text{“a”} \mid \text{“the”}$
 - $N \rightarrow \text{“dog”} \mid \text{“cat”}$
 - $Name \rightarrow \text{“Greg”} \mid \text{“Ramin”}$
 - $V \rightarrow \text{“chases”} \mid \text{“bites”}$

In the above representation, \rightarrow means “can expand to one of the following” and \mid means “or.” Words in quotes (on the right-hand side of terminal rules) are the actual words that will appear in a sentence generated by the grammar. These are frequently referred to as the *terminal symbols* of the language. Words on the left hand side of rules represent syntactic categories in the language being defined. These are called the *non-terminal symbols* of the language since these symbols will not appear in a sentence generated by the grammar. Symbols on the left-hand side of terminal rules, such as N (noun) and V (verb) are often referred to as “parts of speech”.

In order to generate grammatical sentences that obey the specified rules, we must start with a top (or root) non-terminal symbol—in this case, **S**. We then choose one of the alternatives that appears on the right-hand side of the rule whose left-hand side is **S**. In this case, **S** has only one possible right-hand side expansion, the sequence (**NP VP**). When we expand **NP**, we have a choice of (**Det N**) or **Name**; **VP** maps to (**V NP**).

After choosing one of these alternatives, we then scan left to right along the symbols of the chosen alternative. If the symbol being considered is a non-terminal, then the rule associated with this non-terminal is chosen, and the above procedure is repeated recursively until only terminal symbols remain.

A sentence is considered “grammatically correct” if it can be generated from the grammar. The example grammar above can generate “grammatically correct” sentences such as:

- Greg chases the dog
- A cat bites Ramin
- The dog chases a cat
- Ramin bites Greg

1.2 Representing CFGs in Scheme

To represent our grammar rules in Scheme, we are going to use structs to represent non-terminal rules and terminal rules:

```
(define-struct nonterm (lhs rhs))  
(define-struct term (pos list-of-words))
```

The `nonterm` struct represents non-terminal rules. Its values are `lhs` (left-hand side), which is a string representing the syntactic category, and `rhs` (right-hand side), which is a list of possible expansions.

The `term` struct represents terminal rules. Its values are `pos` (part of speech), which is the part of speech of that category, and `list-of-words`, which is the list of words in that pos category.

2 Here thar be monsters!

(45 points total)

Exercise 1.

2 points

(a) [1 point] **Procrastination.** In preparation for teaching your computer to speak pirate, we thought it best that you review pirate for yourself. Please see the following educational video:

- <http://www.youtube.com/watch?v=fqMu6e5Dgtg>

(b) [1 point] **Randomize.**

In order to generate a *random* sentence, we need to be able to pull a random element from a list. Here are some helpful functions:

- `(random x)` returns a random integer between 0 and x-1 (inclusive)
- `(list-ref lst i)` returns the *i*th element of `lst` (with index starting at 0)
- `(length lst)` returns the length of a list

Write the function `pull-random` that takes a list and returns a random element from that list:

```
(define (pull-random lst) . . .
```

2.1 The Pirate CFG grammar

You can find the rules for the Pirate CFG in the .scm file. You don't need to worry about what exactly each rule stands for, but if you are curious...

The nonterminal rules are:

S (sentence)	VP (verb phrase)
NP-ac (active noun phrase)	NP-in (inactive noun phrase)

The terminal rules (parts of speech) are:

Int (interjection)	Det (determiner)	Dets (plural determiners)
N-ac (active noun)	Ns-ac (plural active noun)	PN (pronoun)
A-ac (active adjective)	A-in (inactive adjective)	N-in (inactive noun)
Ns-in (plural inactive noun)	IV (intransitive verb)	TV (transitive verb)
DV (ditransitive verb)	Adv (adverb)	Phr (phrase)
be (the verb "to be")		

Exercise 2.

42 points

(a) [20 points] **Generator.** In this exercise you are going to write a function that randomly generates a sentence in pirate. We have given you the main function:

```
(define (generate starting-rule)
  (to-string (generate-helper starting-rule)))
```

The `generate` method takes a starting rule, and will return a random grammatically correct phrase of the type of the starting rule (as a string). That is, if you call `(generate S)`, you should get a grammatically correct pirate sentence. To `to-string` function takes a (possibly nested) list of strings, flattens the list and appends all of the strings together (with a space in between each), resulting in one large string. For example:

```
> (to-string (list "hello," (list "my" "name" (list "is")) "Arrrrr!"))
"hello, my name is Arrrrr!"
```

Your assignment is to write the `generate-helper` function, which takes as its only input a rule and should return a randomly chosen expansion of that rule until there are only terminal symbols (i.e., actual words in Pirate) left:

- `generate-helper` applied to a terminal rule will return a random word from that part of speech.
- `generate-helper` applied to a nonterminal rule will return a randomly chosen expansion of that nonterminal rule (and recurse on that expansion).

You may find the mini-S grammar useful for testing.

Some examples of this function:

```
> (generate-helper N-ac)
"lass"
> (generate-helper N-ac)
"landlubber"
> (generate-helper NP-ac)
(list "thy" "half-wit" "landlubber")
> (generate-helper VP)
(list "bin" "orderin" (list "those" "saucy" "corsairs") "to Davy Jones' locker")
> (generate-helper S)
(list "arrrr!" (list "these" "landlubbers") "gunna be" "scurvy")
```

(b) [20 points] Parsing.

Now that your computer can generate sentences, you need to be able to parse a sentence, that is, figure out if a sentence is “grammatically correct” in pirate.

We have defined a `word struct` and a part-of-speech tagger, `POStagger`, for you. The `word struct` has two values:

- `token` - the actual word (i.e. the string)
- `tag` - the part of speech tag (i.e. “N-ac”, “TV”, “Det”, etc)

`POStagger` will input a string and return a list of word structs for each word or multi-word phrase in that string. Be careful with your input – `POStagger` was not written defensively so it will throw an error if you input a word that is not in the lexicon (the set of all words in Pirate). Some examples of `POStagger` in action:

```
> (POStagger "arrr!")
(list (make-word "arrr!" "Int"))
> (POStagger "shiver me timbers!")
(list (make-word "shiver me timbers!" "Int"))
> (POStagger "avast! yon landlubber be drinkin me grog")
(list
  (make-word "avast!" "Int")
  (make-word "yon" "Det")
  (make-word "landlubber" "N-ac")
  (make-word "be" "be")
  (make-word "drinkin" "IV")
  (make-word "me" "Det")
  (make-word "grog" "N-in"))
> (POStagger "ponies! bunnies! flowers!")
POStagger-helper: Wordlist ran past end - Unknown word?
```

Your job is to write a function `valid-parse?`, which inputs a list of term and/or non-term structs (called rules) and a list of word-structs (i.e. a `POStaggered` word or sentence) and checks if there is some way to replace the non-terminals (recursively) with terminals so that you have terminals that match the part of speech tags on the list of words.

Some examples:

```
> ; N-ac is a terminal that matches the tag on "cap'n"
> (valid-parse? (list N-ac) (list (make-word "cap'n" "N-ac")))
true
> ; N-ac is a terminal that doesn't match the tag on "ship"
> (valid-parse? (list N-ac) (list (make-word "ship" "N-in")))
false
> (valid-parse? (list NP-ac) (POStagger "me bonny lads"))
true
> (valid-parse? (list NP-ac) (POStagger "I bin drinkin"))
false
> (valid-parse? (list NP-ac DV) (POStagger "those corsairs sendin"))
true
> (valid-parse? (list Int NP-ac VP)
  (POStagger "ahoy! the salty sea-dog be plunderin yon dangerous island"))
true
> (valid-parse? (list S)
  (POStagger "ahoy! the salty sea-dog be plunderin yon dangerous island"))
true
```

Some hints for writing `valid-parse?`:

- What are the base cases?
- What if your list starts with a terminal rule?
- What if it starts with a non-terminal rule?
- You might find it helpful to write an `expander` function which iterates through the possible expansions of a non-terminal node.

We've added a handy predicate for you, `pirate-sentence?`, which will check if a sentence is grammatically correct in pirate:

```
(define (pirate-sentence? sent) (valid-parse? (list S) (POStagger sent)))}
```

Make sure your code passes the following `check-expect` at least 5 times:

```
(check-expect (pirate-sentence? (generate S)) #t)
```

(c) [2 points] **“I gunna be sendin ye to Davy Jones’ locker - savvy?”**

List in a comment your three favorite sentences that your generator produced (or more!). You can see sentences by typing (generate S) into the Scheme evaluator.

Exercise 3.

1 point

(a) [1 point]

Tell us how long you spent on this assignment.