generating sentences given a grammar (part 1)

by RIK0 • FEB. 24, 2011

Introduction

The idea of this post is writing a simple sentence generator based on a given (context free) grammar. The idea and the program come from PAIP.

However, this is essentially only a starting point to present some philosophical differences of programming languages and to show how Clojure is every bit as versatile as Common Lisp (on this kind of problems) and perhaps some new built-in features make it an even more interesting choice for this problem.

Description of the problem

The problem here is simple: generate sentences in a small subset of english. Small is an overstatement; perhaps diminutive is more appropriate. At the cost of looking snob, I believe that many mainstream programmers may already be cowling in fear or, if skilled enough, they are looking for a library doing the task.

There is nothing wrong with the library approach. Moreover, it is probably the right choice for a "real" project, because it is certainly more versatile than the few lines we are going to write, more tested [well, actually not, as our piece of code is just a very close translation of some 25 years old program or so]. Perhaps, more efficient.

Now suppose we are not meant to use external libraries. Unrealistic as it may be, consider that we want to learn something beside a new bunch of API's. Or maybe we just want a lightweight solution. Different approaches are favored for different programming languages.

I jump the stage in which the programmer writes a program specifically translating the example grammar. I believe it is clear from the specs that we want to generate sentences for *every* grammar with some prerequisites. As a consequence, we have just to represent the grammar in some format, read it, understand it and generate the sentences.

This is essentially a language interpretation problem: we have a "program" written in a given language (the grammar) and we want to interpret it (and generate the sentence). The language is not turing-complete, of course [as we have decided that it should be a CFG, and consequently out interpreter machine needs not to be a turing machine]. I read somewhere (which I mistakenly believed to be the Unix Programming Environment) that the authors strongly advised to transform generic problems into language interpretation problems. The idea is that this way it is easier to think at a higher level of abstraction (the one given by the interpreted language, not C); efficiency is probably not an issue either, as a specific domain language is likely to be more efficient than a general purpose interpreted language.

An example grammar is:

```
Sentence : Noun-Phrase + Verb-Phrase
Noun-Phrase : Article + Noun
Verb-Phrase : Verb + Noun-Phrase
Article : the, a, ...
Noun : man, ball, woman, table, ...
Verb : hit, took, saw, liked, ...
```

Implementation details

The question is how to represent the grammar. I believe that most Java programmers here are already mentally checking the API's of their favourite XML parser. Although in Clojure there are more appropriate data structures, I chose to represent it as a list of lists, as with the Lisp tradition. This has the advantage that examples prepared for PAIP program can be directly used without modification. Moreover, the size of the grammar is modest enough not to make it an issue. Besides, switching from lists to vectors does not lead to changes in the code.

I belive that here we can spot a very evident cultural difference. In the Java/OOP world the problem is two-fold. First we have to translate the grammar in data structures manipulable by Java, then we have to actually interpret them. The first part is essentially the task of a parser: for XML there are many parsers (but the syntax becomes very clumsy, IMHO); on the other hand it is possible to use parser generators such as Yacc or Antlr. The second part is also considered a serious issue: in fact in GOF's Design Patterns the Interpreter pattern exactly deals with these kind of problem. Their suggestion is to create a class for every terminal and non terminal. Please notice that we are not talking about the terminals and non terminals of the grammar (such as Sentence or Noun Phrase) but about the terminals and non terminals of the language in which we express the grammar (such as Rule, Alternative, Head and things like that).

It is interesting to notice how both problems in Lisp are non-problems. Lists and atoms provide a nice representation for the grammar itself:

```
(def *simple-grammar*
'((sentence -> (noun-phrase verb-phrase))
(noun-phrase -> (Article Noun))
(verb-phrase -> (Verb noun-phrase))
(Article -> the a)
(Noun -> ball man woman table)
(Verb -> hit took saw liked)))
```

The parser problem is completely solved. And the interpreter pattern just becomes some list processing. Less fancy for sure, far easier nonetheless. It is **just** list processing. Nothing more, nothing less.

```
(ns org.enrico_franchi.paip.simplegrammar.grammar
  (:use clojure.core))


(def *simple-grammar*
     '((sentence -> (noun-phrase verb-phrase))
       (noun-phrase -> (Article Noun))
       (verb-phrase -> (Verb noun-phrase))
       (Article -> the a)
       (Noun -> ball man woman table)
       (Verb -> hit took saw liked)))


(def *grammar* *simple-grammar*)


(defn rule-lhs [rule]
  "The left hand side of a rule."
  (first rule))


(defn rule-rhs [rule]
  "The right hand side of a rule."
  (rest (rest rule)))
```

```
(defn get-rule [grammar category]
  "Get the selected rule from the grammar"
  (letfn [(key [rule]
             (let [lhs (rule-lhs rule)]
               (when (= lhs category) rule)))]
  (some key grammar)))

(defn rewrites [category]
  "Return a list of the possible rewrites for this
  (rule-rhs (get-rule *grammar* category)))

(defn generate [phrase]
  "Generate a random sentence or phrase."
  (cond
   (list? phrase) (mapcat generate phrase)
   (seq (rewrites phrase)) (recur (rand-nth (rewri
```

This is essentially a straightforward translation of the original common-lisp variant. However, does the job. You can also experiment with different grammars.

Morale

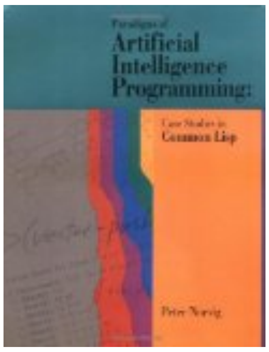
Notice that here the key was *not* lisp homoiconicity. It is a feature which has not been used. A similar approach could have been used also in C (though grammars would have been impossible to share, unless a specific file based representation was developed). Here the key was just thinking in terms of a DSL.

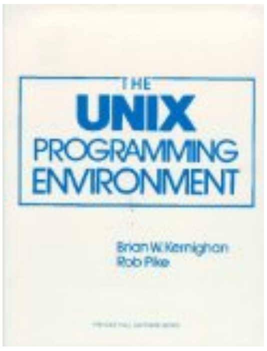
This is not obvious. Some times ago I developed a toy Prolog program which generates imprecations and similar stuff. Unfortunately enough, this example did not immediately come to my mind. Instead I assumed that Prolog ease of development would be (alone) a time safer. It was *not*. Using the full language power to generate sentences proved to be far more complex than switching to a simple grammar representation + a bunch of procedures.

Future work

Right now, I'm trying to adapt the generate-all to Clojure. Indeed, although the straightforward translation actually works, it is not lazy, and consequently cannot be used with grammars generating an infinite language.

Books & Amazon associates





"Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp" (Peter Norvig)

"Unix Programming Environment (Prentice-Hall Software Series)" (Brian W. Kernighan, Rob Pike)

Technorati Tags: Clojure, Lisp, Programming