Text generation from grammars

G Michaelson

The generation of text from grammar rules using the Navel interpreter-interpreter language is discussed.

programming, functional languages, rules, semantics

I have been intrigued by computer-generated text since reading *Cybernetic serendipity*[1], which contains a wide range of material on computers and art from 1968. My first program, written in IMP under EMAS on an English Electric 4/75, produced rather laconic and somewhat '60s prose from a set of rules developed with Paul Cockshott. Subsequently, I rewrote the program in Basic and Jack Campin at Glasgow University changed the rules to generate Vogon poetry in the spirit of the *Hitch hiker's guide to the galaxy*[2].

The program is based on a numeric representation of simple rules. Each rule is made up of references to one or more word classes. Each word class is made up of one or more strings. The rules and word classes are held in arrays and the generation program works through random rules, randomly selecting from word classes. The main disadvantages of this approach are the lack of options on the right-hand sides of rules and the need to hand translate rules into the numeric representation.

More recently, I have been developing the Navel[3] interpreter-interpreter system, which is based on a functional language with integrated context-free grammar rules. Navel is used to build interpreters from formal definitions written in a denotational semantics[4] style.

In Navel, rules are objects in their own right. They may be passed to and returned from functions and may be generalized through abstraction over nonterminal and terminal symbols. Rules may be used for text generation as well as parsing, for example, to generate test sequences from rules.

Here the generation of paper titles, rhyming gibberish, and limericks using Navel is considered. Note that Navel runs in an interactive environment and prompts for an expression or command with:

```
ok
```

PAPER TITLES

Consider the problem of generating titles for learned papers on computing. Titles might be described in Backus-Naur Form (BNF). Given a list of topics:

```
<topic> ::= COBOL | VDM | hash tables | data bases |
            networks | LISP | gotos
```

then plausible titles might have the form:

```
<conjunction> ::= and | or | with | through
<title> ::= <topic> <conjunction> <topic>
```

From these titles could be generated, for example:

```
networks through hash tables
VDM with COBOL
. . .
```

Starting with a <title>, generate a random <topic> followed by a random <conjunction> followed by another random <topic>.

Navel rules are, effectively, BNF right-hand sides, enclosed in {}s. Nonterminals are variables and terminals are strings. Thus the above example is:

```
ok
def topic = {"COBOL" "VDM" | "hash tables" | "data bases" |
             "networks" | "LISP" | "gotos"};
ok
def conjunction = {"and" | "or" | "with" | "through"};
ok
def title = {topic " " conjunction " " topic};
```

Here:

```
def <name> = <expression>;
```

defines the global variable <name> and associates it with the value of <expression>. Subsequent references to <name> will return the associated value. Thus the reference to conjunction in title will pick up the rule:

```
{"and" "or" | "with" | "through"}
```

In Navel, the operator ? generates random numbers, selects random elements from lists, and generates random strings from rules. For example:

```
ok
? title;
"data bases with gotos"
```

The system command 'repeat' will repeat endlessly the evaluation of an expression, for example:

```
ok
repeat ? title;
"COBOL or VDM"
"LISP through networks"
"gotos and hash tables"
. . .
```

Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK.
Paper submitted: 28 January 1990.
Revised version received: 19 March 1990.

PENGUIN SPEAK

Steve Bell's eponymous penguin[5] says "Yibble!" in periods of stress. "Yibble" might be abstracted over and sequences of words generated that end in a double consonant and "le". First of all, "yibble" may be generalized by abstraction over the initial consonant:

```
ok
def consonant = {"b" | "c" "d" | "f" | "g" | "h" "j" "k"
                 "l" | "m" | "n" | "p" | "r" | "s" | "t" | "v" |
                 "w" | "y" | "z"};
ok
def penguin = {consonant "ibble"};
ok
repeat ? penguin;
"fibble"
"mibble"
"libble"
. . .
```

Next abstract over the vowel:

```
ok
def vowel = {"a" | "e" "i" "o" | "u"};
ok
def penguin = {consonant vowel "bble"};
ok
repeat ? penguin;
"yabble"
"dobble"
"jibble"
. . .
```

Next abstract over the double consonant:

```
ok
def double = {"b" "d" "f" "g" "m" "n" "p" "t" "z"};
ok
def penguin d = {consonant vowel d d "le"};
```

Here 'penguin' is a function which, when applied to a letter, returns a rule with that letter doubled:

```
ok
? (penguin "b");
"babble"
```

Now generate a random 'double' for the parameter 'd':

```
ok
repeat ? (penguin (? double));
"gipple"
"juddle"
"pebble"
. . .
```

Alternatively, construct a rule to generate rhyming pairs of penguin words. First of all, abstract over the vowel and the double consonant:

```
ok
def penguin v d = {consonant v d d "le"};
```

Here 'penguin' is a function which, given a vowel for parameter 'v' and a double consonant for parameter d, will return a rule in which only the first consonant may change.

Now construct another function that builds a rule for a pair of rhyming words:

```
ok
def penguins v d = {(penguin v d) " " (penguin v d )};
```

Here both words have the same vowel and double consonant but random initial consonants, which may, of course, be the same. For example:

```
ok
repeat ? (penguins (? vowel) (? double));
"yabble nabble"
"goggle toggle"
"beddle heddle"
. . .
```

LIMERICKS

Consider the generation of limericks using the grammar:

```
<adjective1> ::= young tall | short large small
<person> ::= girl boy | lass | lad
<noun> ::= pig | wig fig cat | mat | hat dog log | cog
<place> ::= beach park pub
<pronoun> ::= she he
<verb> ::= sat on | ran to slept by ate up
<possessive> ::= her his
<adjective2> ::= silly foolish cunning clever
<limerick> ::= there was a <adjective1> <person> with a <noun>
               who went to the <place> with a <noun>
               <pronoun> <verb> <possessive> <noun>
               and <verb> <possessive> <noun>
               did that <adjective2> <adjective1> <person> with a
                   <noun>
```

This grammar may be used to generate 'limericks' like:

```
there was a young lass with a pig
who went to the pub with a cog
he sat on his fig
and ate up her log
did that clever large lad with a cat
```

A number of problems are immediately apparent:

- the <noun>s in lines 1, 2, and 5 should rhyme
- the <adjective1> <person> and <noun> in lines 1 and 5 should be the same
- the <noun>s in lines 3 and 4 should rhyme
- the genders of the <person>, <pronoun> and <possessive> should match

Consider, first of all, the problem of ensuring rhyming <noun>s at the ends of lines. Break the <noun>s up into rhyming classes:

```
ok
def ig = {"pig" | "wig" | "fig"};
ok
def og = {"dog" | "log" | "cog"};
ok
def at = {"cat" | "mat" | "hat"};
ok
def nouns = ig:og:at:;
```

Here 'nouns' is a list of noun classes joined together with the concatenation operator. Now select an arbitrary noun class from nouns, for example:

```
ok
? nouns;
{"dog" | "log" | "cog"}
```

Consider next the problem of ensuring gender matches. Group gendered people in a list with the appropriate pronouns:

```
ok
def female = {"lass" | "girl"}:"she":"her";
ok
def male = {"lad" | "lad"}:"he":"his";
ok
def people = female:male:;
```

and then select an arbitrary gender class:

```
ok
? people;
{"lass" | "girl"}:"she":"her"
```

Next define the remaining word classes:

```
ok
def adjective1 = {"young" | "tall" "short" | "large" | "small"};
ok
def place = {"beach" "park" "pub"};
ok
def verb = {"sat on" "ran to" "slept by" | "ate up"};
ok
def adjective2 = {"silly" "foolish" | "cunning" "clever"};
```

Now, to generate a limerick, select a <noun> class for lines 1, 2, and 5, select a <person> class, construct a rule that ensures that lines 1 and 5 have the same <adjective1>, <person>, and <noun>, and then generate the limerick:

```
ok
def limerick n =
let nouns1 = ? nouns
and nouns2 = ? nouns
and persons:pronoun:possessive = ? people
and adj1 = ? adjective1
and person = ? persons
and noun1 = ? nouns1
and phrase = {adj1 " " person " with a " noun1}
in
? {"there was a " phrase "\n"
   "who went to the " place " with a " nouns1 "\n"
   pronoun " " verb " " possessive " " nouns2 "\n"
   "and " verb " " possessive " " nouns2 "\n"
   "did that " adjective 2 " " phrase};
```

Here the construct:

```
let <name> = <expression>
and <name> = <expression>
. . .
in <expression>
```

introduces local <name>s with associated values. The <name>s may be referred to in any other <expression> in the construct, enabling local mutual recursion.

The construct:

```
. . .
and persons:pronoun:possessive = ? people
. . .
```

is used to match the gender list returned from 'people' and sets 'persons' to the rule for gendered people, 'pronoun' to the gendered pronoun, and 'possessive' to the gendered possessive pronoun.

Note that:

```
\n
```

is used as an explicit newline symbol in the final rule: pretty printing a string containing it generates the 'non-printable' ASCII newline sequence.

This function will generate limericks that satisfy the four criteria above:

```
ok
limerick();
"there was a large boy with a hat
who went to the beach with a cat
he sat on his pig
and ate up his wig
did that foolish large boy with a hat"
```

The construction of more varied word classes enables the generation of even more and even less plausible limericks.

POSTSCRIPT

Navel is written in C and runs under Unix. The system is available at no charge from the author.

REFERENCES

1 **Reichardt, J** *Cybernetic serendipity* Studio International (1968)
2 **Adams, D** *The hitch hiker's guide to the galaxy* Pan (1979)
3 **Michaelson, G** 'Interpreters from functions and grammars' *Comput. Lang.* Vol 11 No 2 (1986) pp 85–104
4 **Schmidt, D** *Denotational semantics: a methodology for language development* Allyn & Bacon (1986)
5 **Bell, S** *If ... chronicles* Methuen (1983)