Assignment 4

CS 311 Data Structures
Spring Quarter, 2006

Due: Wednesday 5/17/06, 11:59pm

Brief description

You will write a program which, given a context-free grammar, generates a random derivation. This assignment is adapted from the one given at Stanford University's programming class. For a Java demo, see `http://www-cs-faculty.stanford.edu/~zelenski/rsg/`

Context-free grammars

A context-free grammar (CFG) is a system that describes a language by specifying how a legal "sentence" of the language can be derived. A CFG consists of

- A collection of symbols, called *terminals*, that are the basic units ("words") of the language.
- A collection of symbols, called *non-terminals* that can expand to sequences of terminals and non-terminals.
- A set of *productions*, rules that specify the sequences that the non-terminals can expand to.
  A production is written as follows:

  $$< non-terminal > \rightarrow sequence\ of\ terminals\ and\ non-terminals$$

  The non-terminal on the left-hand side of the arrow can expand to the sequence on the right-hand side.
- A special symbol, called *the start symbol*, that is the initial non-terminal.

A sequence of expansions is called a *derivation*.

Example:
Consider the grammar:

```
<start> -> I <verb> <object>
<object> -> <noun_phrase>
          | <adjective> <noun_phrase>
<noun_phrase> -> <noun phrase> and <noun_phrase>
          | apples
          | pears
```

```
<verb> -> like
       | dislike
<adjective> -> green
            | yellow
```

The symbol | means OR. For example, $< verb >$ can expand to *like* or to *dislike*.

The non-terminals are `start, verb, object, noun phrase, adjective`. There is at least one production for every non-terminal. Some, such as `noun_phrase` have more than one productions. Non-terminals are surrounded in angle brackets to distinguish them from terminals.

All the other symbols are terminals.

A legal derivation is "I dislike yellow apples and pears". It is produced from left to right (i.e. expand leftmost non-terminal first) as follows:

```
<start> ==> I <verb> <object>
        ==> I dislike <object>
        ==> I dislike <adjective> <noun_phrase>
        ==> I dislike yellow <noun_phrase>
        ==> I dislike yellow <noun_phrase> and <noun_phrase>
        ==> I dislike yellow apples and <noun_phrase>
        ==> I dislike yellow apples and pears
```

Format of input grammar

The first stage of your program involves reading in the grammar. The file containing the grammar will be specified in the command line. You can find some sample grammars at `http://www-cs-faculty.stanford.edu/~zelensk` As you can see from these files,

- Non-terminals are enclosed in angle brackets
- The productions for each non-terminal are enclosed in curly braces
  - the first line after the opening { contains the non-terminal
  - each subsequent line up to the closing } contains a production for that non-terminal, terminating in a semi-colon. The semi-colon is not part of the production.
- The very first set of productions is for the start symbol, which is always called <`start`>.

Some of the grammars at that site may not satisfy these formatting requirements. Ignore those grammars.

Storing the productions

Store the production info in a hash table. The keys are the non-terminals. You do not need to store the terminals separately.

The hash table

You may implement either open addressing or chaining. Document your choice. You do not have to implement a Delete operation; just Search and Insert. You must implement the hash table from scratch.

As mentioned above, the keys (strings) are the nonterminals.

You must decide what hash function to use (pick a good one to avoid rehashing) and what is a good initial size for your hash table. You do not need to come up with your own hash function. You may use one from any source.

The load factor should be 0.5 for open addressing or 2 for chaining. If the load factor is exceeded, then the data must be rehashed. Do not assume that you'll never need to rehash.

Storing productions

Each element of the in the hash table should contain the key and the corresponding productions.

How are you going to store the productions for a particular nonterminal? Keep in mind that every time you expand a non-terminal you must pick a production at random. This means that the structure you use here must allow random access. Also consider the fact that each non-terminal has a varying number of productions.

How are you going to store the individual terminals and non-terminals of a production? Keep in mind that you will need to examine each symbol in a production in order from left to right.

The process of filling out the table is as follows:

- Read a non-terminal
- Compute its hash value
- Read the productions for this non-terminal
  - Read each production separately and store its symbols in an appropriate structure. Each production will be on a different line in the input.
  - Insert each production in an appropriate structure.
- Create a new element for this non-terminal and insert it in the table.

Producing text

Once you have stored the productions in the hash table, you can produce text. At each intermediate step, you will need to keep track of the current *sentential form*. This is the current sequence of terminals and non-terminals. For example,

```
I dislike yellow <nouns> and <nouns>
```

is a sentential form.

How should the sentential form be stored? You must find an appropriate structure for it. The following thoughts should help you get started:

At each step you expand a non-terminal. Looking at the example shown above, at the first step, $< start >$ expands to $I \ < verb > < object >$. Go through these symbols from left to right. I is a terminal so it can

be placed in an output buffer, or printed directly on the screen. $< verb >$ is a non-terminal, so it must be expanded. Find it in the hash table. Use a random number generator to select one of its productions. Remove $< verb >$ and insert the symbols of this production. Continue this process until no more non-terminals are left to expand.

The output

The output must be formatted in paragraphs rather than a single long line. A reasonable paragraph width is 60-70 characters. In addition, make sure that there is appropriate spacing between words (no stray newlines or too much space between words). You won't be penalized for having one extra space character between two words, or between a word and punctuation.

Every time your program is run with a particular grammar as input, it should generate a different story.

Implementation notes

Hashing and rehashing

You'll have to choose an initial table size as well as a good hash function. Keep track of the load factor. If it becomes greater than 2 (in chaining), or greater than 0.5 (in open addressing), the table size should be doubled and the elements rehashed.

Design

In addition to correctness, your code will be graded on modularity, readability, efficiency and clarity. Plan out your program before you start writing code. Your code should be easy to maintain. For example, the hash table implementation should be in a separate file. The hash function should be clearly defined and NOT hidden in an `insert()` function.

Submitting your code

As usual, email your tarball to `cs311@cs.northwestern.edu` and to yourself.