Assigned: Tuesday 4 May 2010

Due: Friday 14 May 2010

Topics: Recursion, I/O, Collections

Collaboration: This homework assignment may be completed individually or in pairs.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Unit test what you are able; demonstrate correctness on other parts of your program as best as possible. Include a statement justifying the sufficiency of your tests and demonstration. One only one submission (paper and digital) per group is necessary.

1 Introduction

1.1 Grammars

A grammar is just a set of rules for some language, be it English, the C programming language, or a made-up language. If you go on to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce to you a particular kind of grammar called a Context Free Grammar (CFG), which is essentially a set of production rules of the form:

<sentence> → <noun-phrase><verb-phrase>
<noun-phrase> → <proper-noun>
<noun-phrase> → <determiner><common-noun>
<proper-noun> → Socrates
<proper-noun> → George (Socrates' evil twin)
etc...

Formally, a Context Free Grammar (CFG) is a quadruple $\langle V, T, P, S \rangle$ where

- V is a finite set of **variables** (**non-terminals**)
- T is a finite set of **terminal symbols**
- P is a finite set of **productions**. A production is a rewriting rule of the form: $V \rightarrow \alpha$ where V is a non-terminal and α is any string of terminals and nonterminals or ϵ, the empty string.
- S is a special symbol called the **start symbol**.

In the example above, all the non-terminals are surrounded by angle brackets, and the terminals are not. Each line is one production, and the start symbol is <sentence>. To read this grammar, one might say that to generate a sentence, one starts with a noun phrase followed by a verb phrase. A noun phrase may either be a proper noun or a determiner followed by a common noun. A proper noun may be the terminal "Socrates" or "George (Socrates' evil twin)". Then one would go on to describe the other non-terminals not yet specified. Context free grammars are a very powerful and very convenient tool for expressing the syntax of languages. A very common problem in computer science is to see if a particular string is recognized by a particular grammar. This is what a compiler must do before anything else when reading a Java program. This process is called parsing, and is a subject worthy of an entire course (CSC 362).

1.2 Derivations

The point of defining a language using a grammar is usually to be able to determine if strings are part of the language or not. A common question Java or C programmers ask of compilers is whether or not the sequence of characters they have just typed forms a valid program in the programming language of choice. If it is, the compiler goes and compiles it; if not, it tries to explain which syntax rules were violated (although it doesn't usually do a very good job of explaining this to the programmer). When checking for syntactic validity, we are really asking if there is a derivation from the start symbol to the text that makes up your program. For example, the grammar for the Java programming language contains:[1]

<Compilation-Unit> → [package <Qualified-Identifier>;] {<Import-Declaration>} {<Type-Declaration>}
<Type-Declaration> → <Class-Or-Interface-Declaration> ;
<Class-Or-Interface-Declaration> → <Modifiers-Opt> (<Class-Declaration> | <Interface-Declaration>)
<Class-Declaration> → class <Identifier> [extends <Type>] [implements <Type-List>] <Class-Body>
...

In this example, the brackets [] indicate optional parts of the production rule, the vertical bar separates different production possibilities, grouped by parentheses (). When we compile a Java program, we are asking, "Is it possible to start from the symbol <Compilation-Unit> and, using only the rules in the grammar, eventually produce the code in the Java program?"

1.3 Sample Grammar File

In our representation of grammars, the non-terminals that appeared to the left of the arrow now appear on two lines of the form

{
<non-terminal>
...
}

The non-terminal may expand to *any* of the productions listed directly below it, one per line. More precisely, each string in angle brackets '<' '>' is a non-terminal. A non-terminal is a placeholder that will expand to another sequence of words when generating a sound bite. In contrast, a terminal is a normal word that is not changed to anything else when expanding the grammar. The name "terminal" is supposed to conjure up the image that it is a dead-end: no further expansion is possible from here.

A definition consists of a non-terminal and its set of "productions," each of which is terminated by a semi-colon ';'. There will always be at least one and potentially several productions that are expansions for the non-terminal. A production is just a sequence of words, some of which may be non-terminals. A production can be empty (i.e., just consist of the terminating semi-colon) which makes it possible for a non-terminal to expand to nothing. The entire definition is enclosed in curly braces '{' '}'. Comments and other irrelevant text may be outside the curly braces and should be ignored. All the components of the input file: braces, words, and semi-colons will be separated from each other by some sort of white space (spaces, tabs, newlines), so you will be able to use those as delimiters when parsing the grammar. And you can discard the white-space delimiter tokens since they are not important.

> A sound bite, as every news junkie and couch potato knows, is a snippet of film that catches the rhetorical highlight of a speech, a quotation that is bright, snappy and memorable, and never mind the boring profundity.
>
> – William Safire

The following is the Soundbite.g grammar file.

[1] `http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html`

```
{
<sound-bite>
<catch-phrase> <vague-platitude> ;
You're no <impressive-person> ! ;
}

{
<catch-phrase>
Read my lips: ;
As <impressive-person> always said: ;
}

{
<vague-platitude>
no taxes! ;
fix the economy! ;
nuke <someone>! ;
reduce the deficit! ;
}

{
<someone>
<impressive-person> ;
the <description> whales ;
the Russians ;
}
```

```
{
<description>
communist ;
liberal ;
middle of the road ;
conservative ;
reactionary ;
}

{
<impressive-person>
<first-name> Kennedy ;
Ross Perot ;
}

{
<first-name>
Don ;
Ted ;
Jack ;
}
```

For example, this grammar starts with the start symbol <sound-bite>, and from there one can derive one of two productions:

- a <catch phrase> followed by a space followed by a <vague-platitude>, or
- the terminal "You're no " followed by the non-terminal <impressive-person> followed by a terminal consisting of just an exclamation point.

You choose one of these productions at random, and then continue to (recursively) expand the nonterminals in that production. You go about expanding each nonterminal in the same way you went about expanding the start symbol, i.e., by choosing a random production to replace that non-terminal. By expanding all the non-terminals in this way, you can derive a sentence corresponding to a sound bite. For example, running a completed assignment on the above grammar might produce the following output:

```
<sound bite>
    You're no
    <impressive person>
        <first name>
            Don
        Kennedy
    !

You're no Don Kennedy!
```

Since we are choosing productions at random, doing the derivation a second time might produce a different output:

```
<sound bite>
    <catch phrase>
        As
        <impressive person>
            Ross Perot
        always said:
    <vague platitude>
        fix the economy!

As Ross Perot always said: fix the economy!
```

As you can see, the output above includes the derivation as well as the final sentence. Each time a non-terminal is expanded into one of its productions, the components of that production are each printed one indentation in from the non-terminal itself. In the first output example, the second production for <sound-bite> was chosen, so each of the terminals and non-terminals in that production are listed below <sound-bite> with one tab inward. Likewise, when <impressive-person> was printed, it too was a non-terminal that had to be expanded, and the production chosen was "<first name> Kennedy", which appears below <impressive-person>, again indented one tab to the right of the non-terminal from which it was derived.

2 Assignment

Since there are several aspects of this assignment, you should find it helpful to decompose it into a few major components, which we will outline here. Although you have almost two weeks to complete this, you should under no circumstances wait to begin your work. Many pitfalls await the student who expects to create an implementation linearly with no breaks for mental rest, discovering bugs, or seeking advice.

You should think carefully about all aspects of the assignment given below, as they are mutually dependent on one another. Perhaps the parsing process will influence your representation. Maybe the representation will drive your generation algorithm (or make you realize the generation algorithm for your representation is horrible and thus you should re-think your representation). In short, think about all of them carefully, plan out your approach, and only then should you try to implement them sequentially, getting the pieces working one at a time.

2.1 Objects

Create a simple class hierarchy with `Token` objects to represent the first two components of the grammar quadruple. The classes `NonTerminal` and `Terminal` should be derived from `Token` to represent V and T, respectively. Though they may not contain much by way of data or methods, these three names should help you clearly organize the underlying data structure for representing the grammar and (ultimately) deriving sentences from it.

2.2 Grammar Representation

Think about the data structures you will use to represent the grammar, once you have it parsed. Keep in mind that you will need to select randomly from among each non-terminal's productions, and that you want fast random access to each non-terminal. Create a class called `Grammar` that stores an appropriate representation of your grammar. At this point, it will be worthwhile to revisit the `Token` objects you created and ensure that any object methods depended on by the collection(s) you chose are correctly and efficiently implemented. (*Hint*: A non-terminal expands to one of a set of productions, each of which is a sequence of tokens.)

2.3 Parsing

Implement an algorithm to populate your grammar representation by parsing the grammar files as outlined above. Be sure to decompose the various steps in reading the file into small reasonable routines that you can develop and test in stages. (For instance, write several small methods that each handle only one aspect of the parsing.)

You should do all of the parsing work during the file-reading phase—your goal is to put the grammar into a format that makes it very easy to traverse and print expansions later without doing any further manipulations on the grammar.

2.4 Expanding the grammar

Once the grammar is loaded up, begin with the <start> production and expand it to generate a random sentence. Note that the algorithm to traverse the data structure and print the terminals is extremely recursive. The grammar will always contain a <start> non-terminal to begin the expansion. It will not

necessarily be the first definition in the file, but it will always be defined eventually. Your code can assume that the grammar files are syntactically correct (i.e. have a start definition, have the correct punctuation and format as described above, don't have some sort of endless recursive cycle in the expansion, etc.).

The one error condition you should catch reasonably is the case where a non-terminal is used but not defined. It is fine to catch this when expanding the grammar and encountering the undefined non-terminal rather than attempting to check the consistency of the entire grammar while reading it.

The names of non-terminals should be considered case-insensitively, <NOUN> matches <Noun> and <noun>, for example.

2.5 Printing the result

When generating the output, you can print the derivation as you expand, storing the complete expansion in an intermediate form until the expansion process is finished.

Each terminal should be preceded by a space when printed except the terminals that begin with punctuation like periods, comma, dashes, etc. which look dumb with leading spaces.[2]

2.6 Choosing the grammar file

Your program should take one argument, which is the name of the grammar file to read. For example, to read from the `dump.g` grammar file which is the `grammars` subdirectory of the current directory:

```
$ java Grammar grammars/dump.g
```

Your program should print a derivation, followed by a blank line and the complete expansion from the grammar and exit.

2.7 Getting Started

You will find four example grammar files on the MathLAN in the directory

```
~weinman/courses/CSC207/code/cfg/grammars
```

in order of increasing complexity: `Poem.g`, `Soundbite.g`, `Extension-request.g`, `Bond-movie.g`. You can find these and more examples online at `http://www-cs-faculty.stanford.edu/~zelenski/rsg/grammars/`.

2.8 Helpful Hints

- You may use the `Scanner` class if you wish, though it is by no means required. `String` methods and a `BufferedReader` may be just as useful
- A regular expression that matches any non-empty amount of whitespace is "`\\s+`" (the double backslash escapes the `\s` pattern, which matches one whitespace character, and the plus symbol matches one or more instances).
- Remember that `StringBuffer` is more efficient than `String` for repeated concatenation

Acknowledgments

This assignment was adapted (and borrows heavily) from materials provided by Julie Zelinski of Stanford University at `http://www-cs-faculty.stanford.edu/~zelenski/rsg`. Used by permission.

[2] This rule about leading spaces is just a rough heuristic, because some punctuation (quotes for example) might look better with spaces. Don't stress about the minor details, I am looking for something simple that is right most of the time and it is okay if it is a little off for some cases.