Allison Hansen
CS106X

Handout #53
May 24, 1993

Assignment #6:
Random Sentence Generator
Due: Wednesday, June 2, 5p.m.

Design of Data Structure Due in Section This Week
No Late Assignments accepted after Monday, June 7

This assignment is adapted from an assignment written by Mike Cleron.

Note: This assignment is substantial both in difficulty and length. Do not put it off for the night before!! In this handout there is a suggested step-by-step procedure for writing this assignment and suggested sub-deadlines for you to meet. You are to turn in the design of your data structure on paper in section this week.

Tactic #1: Wear the TA's patience

I need an extension because I used up all my paper and then
my dorm burned down and then I didn't know I was in this class
and then I lost my mind and then my karma wasn't good last week
and on top of that my dog ate my notes and as if that wasn't enough
I had to finish my doctoral thesis this week and then
I had to do laundry and on top of that my karma wasn't good last week
and on top of that I just didn't feel like working and then
I skied into a tree and then I got stuck in a blizzard at Tahoe
and on top of that
I had to make up a lot of documentation for the Navy in a big hurry
and as if that wasn't enough I thought I already graduated
and as if that wasn't enough I lost my mind and on top of that
I spent all weekend hung-over and then I had to go to
the Winter Olympics this week and on top of that all my pencils broke

Tactic #2: Plead innocence

I need an extension because I forgot it would require work

Tactic #3: Honesty

I need an extension because I just didn't feel like working

Running out of ideas for extension requests? Your troubles are solved! Presenting... The Random Sentence Generator! Feed it the grammar of your choice, and you too can derive random sentences at will! Choose from the variety of grammars which come with the assignment free of charge, and make your own extension requests, James Bond movie plots, or sound bites for your future in politics. Or, create your own grammar. Fun for the whole family!

What is a Grammar?

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 can be described as follows:

A **Context Free Grammar** (**CFG**) is a quadruple <V, T, P, S> 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 non-terminals or ε.
- 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 C program. This process is called parsing, and is a subject worthy of an entire course.

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 Pascal 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, a grammar for the Pascal programming language might begin as follows:

<*program*> → <*program heading*> **;** <*program block*>
<*program heading*> → **program** <*identifier*> **(**<*file identifier*> {**,** <*file identifier*>}**);**
...

When we compile the Pascal program, we are asking, "Is it possible to start from the symbol <*program*> and, using only the rules in the grammar, eventually produce the code in the Pascal program?"

A **derivation** of a string from a grammar is a sequence of **sentential forms**. A sentential form is a string composed of terminals and non-terminals. The derivation begins with the start symbol, and ends with the desired string of terminals. Each step of the sequence must be a transformation of the previous step based on the rules of the grammar.

Below is an example derivation from the start symbol <wff> (which is a non-terminal) to a line consisting of only terminals. In this representation of a grammar, the arrow shows that the non-terminal to the left of the arrow goes to the productions listed on the right of the arrow. The vertical bar separates the different productions for a particular nonterminal.

Example:
Using the following grammar
<*wff*> → <*proposition*> | **(~**<*wff*>**)** | **(**<*wff*> ∧ <*wff*>**)**
<*proposition*> → **A** | **B** | ... | **Z**
show that (~(A B)) is a wff.

<*wff*>
(~<*wff*>)
(~(<*wff*> <*wff*>))
(~(<*proposition*> <*wff*>))
(~(<*proposition*> <*proposition*>))
(~(A <*proposition*>))
(~(A B))

Sample grammar file

```
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. Here, the non-terminals that appeared to the left of the arrow now appear on a line of the form "<non-terminal> =". The non-terminal may expand to any of the productions listed one per line directly below it.

```
<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 nonterminal <impressive person> followed by a terminal consisting of just an exclamation point. You choose one of these productions at random, and then continue to 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 nonterminals 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 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.

What Is Provided For You?

Grammar Files

You have at your disposal 6 pre-made grammar files. Part of the fun is creating your own if you choose to enter the Grammar Contest described at the end of this handout. The files given to you are:

simple.g
soundbite.g
extension.g
bond.g
proverb.g (written by Chris Fang-Yen, Winter '93 CS106X student)
trek.g (written by Jeremy Henrickson, Winter '93 CS106X student)

The Symbol Table Abstraction

You are given the binary tree implementation of the symbol table. You are required to use a symbol table to store the non-terminals of your grammar. The key would be the name of the non-terminal, and the value would be the list of productions for that non-terminal.

You should not have to make any changes to either symboltable.h nor symboltable.c unless you finish early and decide to do the optional part of the assignment, which asks you to implement the command which displays the grammar. Until that point, you can use the symbol table as an abstraction, which allows you to think about the random sentence generator program and leave the details of the symbol table implementation to its implementor.

The Queue Abstraction

How are you going to store the productions for a particular non-terminal? How are you going to store the terminals and non-terminals that go with a particular production? As you can imagine, there are many places where queues would be the appropriate way to store information. Fortunately, you have already written a queue package for the previous assignment, and you may start with your own queue.c and queue.h. Since you wrote your queue package to be "polymorphic," you can store anything in your queue. You can even have a queue of queues!

In order to pick a random production from the list, you will need to a function which returns a pointer to the Nth element in the list. The standard queue functions Enq and Deq are insufficient to accomplish this (if you Deq items, you destroy the list, which you want to remain intact). So you should modify the *queue* abstraction to be a *list* abstraction, simply modifying the names of the types and functions to reflect this change, and adding a function with a prototype such as the following:

```
listElementType GetNthElement(listADT list, int N);
```

Other

Don't forget that you need to include the course library random.lib in your project as well as include "random.h" in your program in order to have access to the functions RandomInteger and Randomize. See the course libraries handout for descriptions of these functions.

Where Do I Start?

The following is a suggested step-by-step procedure for writing this large program. Beside each step, there is a suggested date by which you should complete that step.

Run The sample applications (Monday, May 24)

There are two sample applications for you to play with. They are essentially the same except for one feature which is included in RSGPlus but not the other. Your task is to emulate the behavior of RSG, which implements all the functionality required in this assingment. If you have time, you can pursue the optional section, which includes writing the function to display the grammar. Displaying the grammar requires using functions as data, and therefore is not required for the assignment. RSGPlus is the application which demonstrates the visible optional improvement, namely the ability to display the grammar.

Design the Data Structure (Tuesday-Wednesday, May 25-26, **due in section**)

Once again, you are required to use an unaltered version of the symboltable abstract data type as your main data structure for storing the grammar. Each non-terminal will be the key of an entry in the symbol table, and the value associated with that key should be the list of productions which that non-terminal can expand to. Draw out what one of the grammars would look like in full -- showing the non-terminals stored in the binary tree symbol table structure, the production list that extends from the value field of each symbol table entry, the list of terminals and non-terminals that constitute one production. Turn this data structure sketch in to your section leader in section this week.

Modify the Queue Abstraction to be a List Abstraction (Wednesday, May 26)

You should take your queue abstraction from the previous assignment, and modify it as described on the previous page. Once modified, you should have a list abstraction that is like the queue abstraction but also provides the GetNthElement functionality. Remember to test the list package before using it in your program.

A Word on Global Variables

So far, we haven't talked very much about the use of global variables, partly to shield you from temptation to abuse them. A global variable is a variable that is declared outside the main program, usually between the typedef section and the local function declaration setion. All functions, including main as well as the local function declarations, can "see" the global variables, and therefore these variables do not need to be explicitly passed to the functions. Within a program of this size, it makes sense for a few of the variables to be declared globally, since they have global meaning throughout the program. For example, you would want to declare the variable which contains the symbol table of non-terminals as a global variable; thereafter, the functions which read the file into this symbol table and derive a sentence based on this symbol table can all access that variable directly.

Choosing which variables to declare globally should be done sparingly. In my solution, I only used four global varialbes. You may declare up to but no more than FIVE global variables in your program.

Read in the Grammar File (Friday, May 28)

Believe it or not, properly reading the file into the data structure is a good chunk of the work. This is a good milestone to try to reach early on.

You may assume the following about the grammar file:

• The first line in the file contains the start symbol (the non-terminal with which the derivation will begin).
• Every non-terminal used in the file will be eventually defined as one or more productions to which it can expand.
• The non-terminal definitions are separated by a blank line.
• The last non-terminal definition is not followed by a blank line.
• Each non-terminal definition consists of a line containing the nonterminal in angle brackets followed by a space and an equal sign, and this line is in turn succeeded by one or more lines corresponding to the one or more productions for that non-terminal.
• Each production consists of terminals and/or non-terminals. (It is easiest not to store each separate word as a terminal, but to consider everything up to the first open angle bracket as the first terminal, everything from the first close angle bracket to the second open angle braket as the second terminal, etc. Of course, you have to consider how to handle the boundary conditions.)
• You may assume that there are no extraneous spaces within the angle brackets surrounding a non-terminal. For example, you won't find `<person>` declared but the `<person >` used.
• You should be able to replace a non-terminal like `<person>` directly with the production it goes to without introducing any spacing problems.
• Error-checking on the input file is not required.

(Debugging Note: If you want to use the debugger to see the value of a variable whose type is equivalent to `void *`, then when you specify that variable in the data window you need to typecast it to its actual type. Doing this typecast allows you to see the actual structure to which that variable is pointing.)

Add the Command-Line Interface (Monday, May 31)

This step is relatively easy and straightforward. You should refer to the handout on Command Dispatch Tables. You should use the Case Dispatch method described in Method 2 on the handout. Even if you are not yet ready to fill in the functions in the command dispatch table, you can write the "stubs" for these functions. A stub is just a function without a body filled in -- it doesn't really do anything but serves as a placeholder as you test out your framework.

So far, the commands you could fill in easily are "set", "show", hide", "help", and "quit" (the C instruction exit(0) quits your program). You could also fill in the introduction at this point, and set up your main program to use your CommandDispatch function.

Note: If we were being conscientious programmers, we would free the symbol table between reading in different grammars and we'd free the list of terminals between derivations so as not to waste memory. However, freeing a list or a symbol table involves some knowledge of what is being stored in that structure. Implementors of these abstractions, would need the client to say how to free their client data. To do this, we need the ability to pass a function as a parameter. You will practice sending functions as data when implementing 'display grammar,' but due to the length of the assignment, deallocating memory is optional.

Implement the Derive Command (Monday, May 31)

It's time to derive a sentence based on the grammar. You will probably want to write the derive command function as a wrapper for the real recursive function. You'll probably want to do 80% thinking and 20% coding -- just jumping in and writing code will be less effective in most cases than thinking about the problem carefully and letting the code fall out of a well-thought-out solution. As a hint, one of the parameters you need to pass along in the recursion is the level of depth you're at, which you need in order to print out the appropriate number of leading tabs as you show the derivation. Also, you need to be storing away all the terminals you reach in your derivation so that you can print the resulting sentence at the end.

Clean up the Output (Wednesday, June 2)

Chances are that your output doesn't look very good, or it runs off the right edge. There are several strategies for dealing with this. One of the more straightforward and easy-to-implement strategies is to do the following when printing the sentence. Keep a temporary string to which you keep concatenating terminals until the terminal you are about to add would put the length of that string over some maximum line length. At this point, you can print out the temporary string as a line, and continue with the temporary string starting out as the terminal you wanted to add. There are other niceties you can add to make the output look proper, but these will be up to you to discover and implement once you have fully completed the assignment. Be sure to turn in sample output from various files in addition to all your code files.

Optional

Implement the 'Display Grammar' Command

At this point, you should have a working program, except for the implementation of the function which displays the grammar. It is a good idea to save a copy of your working program and its accompanying sample output files to this point in a safe and secure place. In this way, you could turn in this copy in case your additions to the program break it.

Your challenge here is to implement the command which displays the grammar. In order to do this, you have to map a function over the symbol table. In order to do this, you have to modify the symbol table package to add a function MapSymbolTable. You'll also want to modify the list package to provide a function such as MapList. You should use the "Functions as Data" handout as a guide. After adding this part, you should do sample runs that show off the functionality of all the commands, and turn these in along with all your code files.

Deallocate Memory

Another task which involves mapping functions is the task of freeing memory. Try writing the functions which deallocate memory which is no longer needed.

Use Function Pointers To Accomplish Command Dispatching

The handout on command dispatch tables describes another method (Method 3) of implementing dispatch tables using function pointers. You can modify your implementation of the command dispatching mechanism to use function pointers.

The CS106B Grammar Contest:

Entries Due by Monday, May 31 at 5pm under Allison's door.

This is optional and fun. Everyone can enter the grammar contest. The idea here is to design and write your own grammar. Use your creativity! See what outlandish or frighteningly realistic derivations you can achieve with your grammar.

Rules:

- All entries must be your own original work.
- Each person can submit any number of entries.
- For each entry, you must hand in
    - a hard copy of the grammar
    - a soft copy of the grammar on disk
    - a hard copy of your favorite few derivations from the grammar.
- If you have several entries, you may put all the grammars on the same disk.
- Label all sheets of paper you turn in, the disk, and the enclosing envelope with your student number rather than your name. Also put on the envelope label "CS106B Grammar Contest."
- Entries are due by 5pm on Monday, May 31.
- Winners will be announced in class on Wednesday, June 2.
- Quality and quantity of prizes will depend on the quality and quantity of the entries received. (Last quarter, the prizes were gift certificates to Tower.)