

CS51 Project 1a: Words, Mere Words

Due: Tuesday, 17 February 2009 at 11:59 PM

Total Points: 65 (including 10 style points)

This is the first part of your first long-term project for the semester. These two projects are intended to get you thinking about issues of large-scale design and iteration of code through multiple versions and stages, as well as what complications and benefits arise as a result of working with a partner.

1 Project Overview

You will begin by understanding what an interface is and how it can help you write code that is extensible, maintainable, and able to be improved without a complete overhaul of the structure of your program. You will then build on the work you've done in the first part to create your own generative grammar, a relatively simple set of rules that defines a mini-language. You will use this grammar to make seemingly meaningful sentences, as well as determine whether a given string is a valid sentence within your language. You will then back up and move in a different direction by implementing a Markov chain babbling; this is a type of program which, rather than generating sentences based on a set of rules, takes a large text input and rearranges it with some randomness to form pseudo-sentences. (The output is usually much funnier than the input.) In the final stages, you will put together many of the earlier pieces to examine how Google and other search engines determine a website's page rank, and attempt to understand how this notion enables us to more effectively search the Internet.

2 Pair Programming

This project will be the first time we do pair programming. We've decided to allow you your choice of partners for this assignment. You may work with anyone else in CS51. Partners should work together (as in, both be physically present) when writing the code and understanding the assignment. You may also choose to be randomly assigned a partner. If you elect to have us assign you someone, you will probably end up working with someone

else in your section. We actually recommend random partner assignments, both because then you have the same TF and can easily ask joint questions after section, and because a big part of real-world software design is working with people you've never met before on projects that need to be finished quickly.

If you decide to pick your partner, you and your partner must **send an e-mail to both of your TFs before 11:59PM on Friday, February 13th** stating that you wish to work together. If you do not do so, you will have your partner assignment e-mailed to you by Saturday morning. You may also e-mail your TF ahead of time and state that you wish a random partner, in which case you will be matched as soon as someone else in your section also says they want to be assigned a partner.

2.1 Paired Submissions

Luckily, this is not very complicated, but you do need to be sure to do it right. You and your partner can use the same DrScheme submission tools you've used on the first two problem sets, with one important difference: the username you will be submitting under will be of the form `user1+user2`, where the "+" is a literal plus character. So, if Gideon and Victor were submitting a joint problem set, they would submit as `gwald+shnayder` or `shnayder+gwald`. The order of the usernames doesn't matter, and either partner's password can be used. (Although once you've submitted something with one ordering of the usernames, it won't let you resubmit using the other ordering.)

Once you've recorded a submission with your partner, you will *no longer be allowed* to submit individually or with a different person. So please be sure that when you submit into the project folder for the first time, you do it with the correct partner! (This constraint is per-folder, so you could have your own submission in the scratch space and still be able to submit with your partner in the official ps2 assignment.)

2.2 Why Pair Programming?

There are a lot of reasons why pair programming can make you a better programmer. You and your partner should be switching off at the keyboard, with whoever isn't currently typing watching carefully as the other one codes to check their work and give feedback and input. You should **not** have one partner doing all the typing, and nor should the person who isn't actively coding be spacing out, gchatting on their laptop, etc.

Take a look at the handout, "All I Really Need to Know I Learned in Kindergarten", linked to on the Course Policies page in the Partner Policy section:

(<http://cs51.seas.harvard.edu/docs/Kindergarten.pdf>).

You don't need to read the whole thing, but you and your partner should look at it together the first time you meet to get a sense of why we're asking you to program in pairs and how to make the experience as rewarding as possible.

3 Graphs, queues, and maps, oh my!

(5 points total)

This part of the project will focus on ADTs (Abstract Data Types), interfaces, and the ways in which these abstractions can help us become better coders. Two key ADTs for this project will be queues and dictionaries (maps). If you don't remember what these are, that's fine - we'll be going over them in a lot of detail in the sections to come.

3.1 BFS and Queues - explanation

This week, we'll be looking at an example of Breadth-First Search (BFS) on a graph. BFS is a widely-employed algorithmic technique used by programmers who have some data set that can be naturally represented as a graph, and who wish to be able to answer (efficiently) the question of whether any given element is in their graph.

BFS proceeds as follows:

1. Set your root node (whatever node you decide you want to start with) as the "next" node, and add it to a *queue*.
2. Look at the next node in the queue. (If there is no "next" node, you failed to find the item - return **false**.) Pull it off the queue and examine it.
3. Is it the thing you're looking for? If so, great, you've found it! Return your current node.
4. If not, put all unexplored nodes connected to your current node on your *queue*. Go back to step 2.

Many of you probably found yourselves thinking, "Gee, that's not so bad - except for the part about a queue. What's a queue again?" A queue is the first ADT we'll be exploring in this project.

A queue is basically a collection of elements arranged in line, waiting their turn. (If we were in England, the notion of a queue would be very intuitive.) A queue supports at least two operations: **push** and **pop**. (And sometimes others, such as **peek**, which we won't need here.) **Pushing** an element onto the queue puts it in the back of the line; **popping** an element off the queue gives you the next element and takes it off the queue. Your program can **pop** the next node off the queue and look at it, or it can tell new nodes that they need to go stand at the back of the line and wait their turn with **push**.

You'll notice that nowhere in the above description of a queue have I explained how to implement a queue. And yet, I was still able to talk about what a queue does! This is because a queue is a perfect example of an ADT. We can reason about what it does without actually knowing the details of how it's implemented, and write programs that use queues and will still function even if we change the specific queue implementation we're using. In fact, on this part of the project, we'll be doing just that.

So, back to BFS. What it's doing is searching a graph by recursively spreading out and checking all of the nodes that are a distance k away from the root node, for successive values of k . At the first step, it puts all of the nodes connected to the root node - i.e., those that are distance 1 from the root node - onto the queue. Then it proceeds to examine each of these in turn, **pushing** all of the nodes that are connected to these (and thus distance 2 from the root) onto the queue until it's determined that the value sought isn't at any of the distance 1 nodes. It then checks all of the distance 2 nodes, keeping track of the distance 3 nodes on the queue, etc.

3.2 Code Reading

Don't worry - we're not actually going to ask you to implement BFS from scratch (yet)¹. Instead, we've written for you a particularly silly BFS algorithm. You'll notice that it uses an *implicit queue*; it basically passes around a big long list of unexplored (as well as explored) nodes that it wants to get to, and then goes through those nodes one-by-one using **first** and **rest**. This implementation is correct in that it gives the desired result, but as we will soon see, we can be much smarter.

Your task: read and understand the code we have given you. You will see that it takes a graph as input. We are assuming for the purposes of this assignment that the graph is *undirected*; see the lecture notes (or ask on the BB/at OH/in section) if you need a refresher on undirected graphs. We've implemented undirected graphs for you as association lists. However, note that since we've given you the complete interface for our graphs, you don't actually need to understand how they're implemented! Your code can sit on top of our abstraction and use the functions we've provided, and it'll work just fine without you having to know the ins and outs. This is the principle that this part of Project 1 aims to illustrate.

Exercise 1.

5 points

UNDIRECTED-GRAPH

Methods:

`ugraph-new` – creates a new undirected graph. Returns the new empty graph.

`(ugraph-nodes graph)` – finds the list of nodes of a given graph. Returns the list of nodes.

`(ugraph-empty? graph)` – checks whether the graph is empty. Returns true if it's empty, false otherwise.

`(ugraph-neighbors node graph)` – finds the neighbors of the input node. Returns the list of neighbors.

`(ugraph-add-node node graph)` – adds a new node to the graph, initially unconnected to

¹Although if you want to try, you should stop reading after the graph code below and give it a shot. You have all of the coding tools you'll need.

any other nodes. Returns the graph with the new node added.

(`ugraph-add-edge node1 node2 graph`) – adds a new edge between nodes 1 and 2. If either node isn't in the graph, it adds the node to the graph. Returns the graph with the new edge included.

We've also given you a function `make-ugraph` to help you create graphs to test with. See the `.scm` file for more details.

(a) [5 points]

Write tests for `ugraph`.

You should do this by generating some graphs with `make-ugraph`, and naming them different things. Then use the other functions - `ugraph-nodes`, `ugraph-empty?`, etc. - and check that they return what you would expect. Don't break the abstraction barrier! You should be able to write your tests without ever assuming that we're using any particular graph representation.

4 BFS - Implementation and Testing *(13 points total)*

Exercise 2.

5 points

We've defined BFS for you on `ugraphs`, as defined above (see the `.scm` file). There is, however, one twist. The values we're seeking aren't just the names of the nodes ("a", "b", "c", etc.); rather, the values are stored elsewhere. In this example, we're associating each node to its true value with the help of a struct specially designed for this purpose. As you'll see in the `.scm` file, `node-lookup` is a struct with two fields, a name and a value. The name will be the name of a node (single-letter strings in our case), and the value will be its value (you may assume that nodes have been given numeric values for the purposes of this exercise). Our BFS algorithm takes `node-values`, which is a list of `node-lookup` structs associating each node in the graph to its value.

(a) [5 points]

Using the graph and list of `node-lookups` given, write tests for `bfs` (and its helper functions!).

Note that `bfs` returns false if the value is not found, and the name of the node where it is found otherwise. Test with a variety of root inputs and values sought. Note that sometimes the answer for a certain sought value will depend on what node you use as the root; `bfs` will always return the node that has the value sought and is closest to the root node you pick.

4.1 Queues - Implementation and Testing

Again, we've written a silly queue implementation for you. The interface and implementation are below.

QUEUES

Methods:

`queue-new` – creates a new queue. Returns a new empty queue.

`(queue-empty? queue)` – checks if a queue is empty. Returns true if the queue is empty, false otherwise.

`(queue-push elt queue)` – pushes `elt` onto the back of the queue. Returns the queue with `elt` at the back of it.

`(queue-get-next queue)` – gets the element on top of the queue. Returns the next element.

`(queue-pop queue)` – removes the front element from the queue. Returns the queue without the first element on it.

You'll notice that we need two separate functions in place of the usual one `pop` function. This is because we are using Scheme as a purely functional language. Thus, we aren't allowed to run functions with side effects, such as mutating variables. Normally, `pop` returns the first element and simultaneously shrinks the queue, but here we need to have two separate steps for doing that, since we aren't allowed to mutate the queue in the process of returning its first element.

Exercise 3.*8 points***(a)** [3 points]

Write tests for `queue`.

Check that all of the given functions work correctly in a variety of scenarios.

(b) [5 points]

Now, using the queue interface we've given you, rewrite `bfs` in terms of queues. That is, change our code to use the queue functions defined above, wherever appropriate, instead of whatever it's doing now. The code has been copy/pasted in the Scheme file for your convenience.

What this means you need to do is find anywhere in the original BFS implementation where you could effectively be using a queue, and change the code so that you are doing so. A hint: in our reference solution, we use each one of the queue methods defined above at least once, but no more than five times.

After you're done rewriting the code, take the tests you wrote in the BFS section above, and **run your new BFS algorithm through those tests**. This will ensure that in changing the implementation of BFS, you haven't modified its functionality. Notice how important it is that you have good tests for this part! Otherwise you might have introduced a bug that could prove very hard to find.

5 Dictionaries

(23 points total)

So, we finally have a working BFS on our undirected graph implementation, using queues as an ADT. However, there's still something in our BFS implementation that practically begs to be abstracted out. Do you see it?

That's right – the silly hard-coded node-lookup struct mappings we were using should really be improved. Every time we want to look up an element, we have to scan the whole list! Isn't that slow? The answer is that yes, it's very slow, and we can do better.

However, first we should define an interface, so that we know exactly what we will need to do. You should by now be able to anticipate what's coming: we'll define an interface for dictionaries with all the functionality we need; write tests for our dictionaries; start with a simple implementation; and then go back and rewrite our BFS code in terms of our shiny new dictionary ADT.

(Don't worry: we'll get to efficiency improvements in a bit.)

DICTIONARIES

Methods:

`dict-new` – creates a new empty dictionary. Returns a new empty dictionary.

`(dict-empty? dict)` – checks whether a dictionary is empty. Returns true if it is, false otherwise.

`(dict-keys dict)` – finds the keys of the dictionary. Returns a list of all keys in dict.

`(dict-lookup key dict)` – looks up an element in our dictionary. Returns the value that the element is bound to in our dictionary, or empty if it's not bound.

`(dict-add-pair key value dict)` – adds the key/value pair to dict. If key is already bound to something in dict, it should overwrite the current binding; if not, it should add key to dict and bind it to value. Returns dict with the key/value pair included.

Exercise 4.

23 points

(a) [3 points]

Write tests for the dictionary interface above. Uncomment them when you're done implementing the functions below, but please do write these tests before doing part (b).

Now, we will ask you to implement the above interface as *association structs*. An association struct implementation of a dictionary first defines a struct with two fields, a key and a value. This struct represents a node in the dictionary. Then the dictionary itself **is simply a list of such structs**. In other words, you define a node as a struct with a key and a value, and then your dictionary just keeps track of all the nodes you have. You will need to define the struct you'll use as well as the dictionaries themselves. We've started you off.

(b) [1 point]

(define-struct node (SOME-VALUES-HERE))

(c) [1 point]

Write dict-new.

(d) [1 point]

Write dict-empty?.

(e) [2 points]

Write dict-keys. For this and all future parts of this assignment, feel free to define any helper functions you need.

(f) [3 points]

Write dict-lookup.

(g) [6 points]

Write dict-add-pair. Be careful! If the node is already in our dictionary, we should change its value, not simply create another node where it's bound to another value. Also, don't reinvent the wheel - You might find at least one of the above functions useful. You will also likely need to define at least one helper function. Note: you may find the `append` function useful in making some lists look the way you want them to.

Now, uncomment the tests you wrote in part (a) and run them against your new dictionary implementation.

(h) [6 points]

As before, we'd now like to implement BFS in terms of dictionaries. Take your implementation of `bfs` from the end of exercise 3 and rewrite it using dictionaries where appropriate. In particular, you should no longer need the `node-values` struct. You might want to copy over your `bfs-queue` function below rather than starting from scratch again on `bfs`, because we want to be using both dicts and queues.

Be sure to again apply the tests you wrote for `bfs-queue` in Exercise 2, to check that you haven't introduced new bugs! In order to do so, you'll need to create some dictionaries to test with. Again, we've provided a function `make-dict` to make that a bit easier for you. It won't compile until you've defined the `node` struct above, so uncomment it once you have.

6 Queues - efficiency improvements

(14 points total)

6.1 Theory

It's okay if you don't follow every single bit of this section. However, in order to be able to do the next part of the exercise, you will need to understand generally how this type of queue works. So, read this section through a few times until you get the gist of it; if you and your partner are both having a lot of trouble with it, ask at OH or on the BB. Don't be scared off by the fact that there's a proof in this section; proofs are a common tool in theoretical computer science, and you will see much more of them, both later in this class and in any future computer science classes you might take (particularly CS121 and CS124).

You might have noticed that our queue implementation is $O(n)$. What does this mean, and why is it true? We're using something called "Big-O" notation here. Big-O notation is concerned with the *asymptotic* run-time of a function. What this means is that you imagine running the function on a really huge input, of size n for some gigantic n , and you think about how long it would take it to run. Does it just do a constant number of operations regardless of how big our input is? This would be a constant-time function, or $O(1)$. Does it need to do something (or a few somethings) for each element in the list? This would be a linear function, or $O(n)$. How about if it needs to do some number of operations on each element of the input, for each other element of the input? One example would be if we wanted to compare every element to every other element of the input to find the biggest one (note that this would be a particularly bad way to find the biggest element of an input). This is $O(n^2)$, since we do some number of operations proportional to the square of the input.

It's okay if you're not completely clear on Big-O notation; we'll be practicing with it a bit more later on in the semester. For now, it's only important that you understand why our queue implementation is $O(n)$. Do you see it?

The culprit is the call to **append** in **push**. In Scheme, **append** is really slow. It basically goes through the whole list until it finds the very end of the list, and then sticks our element onto the end of it. This is $O(n)$, because it needs to examine each element of the list to check if it's the last one, until it actually does get to the end. This means that our queue implementation could turn out to be pretty slow on a really big input.

It turns out that there's a really nice way to do queues, which is *amortized* $O(1)$. What does this mean? This means that occasionally, you do need to do operations that are bigger than $O(1)$, but you do them infrequently enough that your implementation is "on average" $O(1)$. How does this work?

It essentially works by maintaining two queues concurrently: one forward queue and one backwards queue. Whenever we want to push an element onto our queue, we stick it onto the *front* of our backwards queue; whenever we want to pop an element off our queue, we take it off the front of the forwards queue. Whenever we notice the forwards queue is now empty, we **reverse** the backwards queue and *switch* the forwards and backwards queues.

What's going on here? Well, we're making **push** $O(1)$ by putting it into the front of the

backwards queue; `pop` is still $O(1)$ because we take the first element off the forwards queue. The only $O(n)$ operation we do is `reverse`, which happens sufficiently infrequently that the overall runtime is still $O(1)$ (amortized). (If you think about it, you'll realize that doing `reverse` in $O(n)$ instead of $O(n^2)$ is itself pretty tricky! Luckily, the Intermediate Student language in Scheme has a built-in `reverse` function that's $O(n)$.)

Don't believe me? I like your skepticism! Take a look at the proof given below. It's not essential that you understand all of it, but it will help to motivate the following exercise, and will give you a better understanding of how to go about doing it.

Proof of amortized $O(1)$ run-time of double-queues:

We examine `pop` and `push` and attempt to show that both operations are amortized $O(1)$.

1. `pop`

Case 1: The forwards queue is not empty. Then we simply take its first element, which is one operation regardless of the length of the list.

Case 2: The forwards queue is empty. Then we need to `reverse` the backwards queue and begin popping off of that. The `reverse` operation is $O(n)$ in the number of elements currently on the backwards queue. So, we need to know *how often* we'll need to do this operation. Well, once we reverse the backwards queue, we're left with n elements on our new forwards queue. That means it won't be empty until we've done another n `pop` operations. Thus, for every $O(n)$ `reverse` operation we do, we've bought ourselves another n $O(1)$ `pop` operations. This means that the overall cost of each `pop` operation is essentially increased by an $O(1)$ factor, since to get n of them we need one $O(n)$ operation. Thus, over the course of a long run of our queue program, `pop` is amortized $O(1)$, as desired.

2. `push`

Case 1: Luckily, there's only one case here. We always just push an element onto the front of our backwards queue, which is $O(1)$. Thus, both operations are amortized $O(1)$ or $O(1)$, which means that the overall program is amortized $O(1)$, as desired.

6.2 Implementation

What's all this theory good for, and how does it translate into implementation? (This part you will need to understand in order to get points!)

We want to implement queues using two different lists. One is our "forward" list and one is our "backward" list. Whenever we want to push an element onto the back of our queue, what we do is to put it on the front of our backward list. When we want to pop an element off the front of our queue, there are two possibilities. If the forward list isn't empty, we just take the first element off of it. If the forward list is empty, we need to `reverse` the backward list, make the backward list the new forward list (and vice versa), and then take the first element from the new forward list.

Before beginning the exercises below, you should convince yourself that this works like a queue should and try to understand why it's faster than the simple one-list queue implementation, even if you didn't follow all of the details of the efficiency proof in the theory part.

Exercise 5.*14 points*

Now then: let's actually implement this fancy thing! Note that you'll need to comment out our original queue implementation to be able to run the new one. Again, structs will be useful here. You will probably find yourself creating and returning new structs based on the values in the old ones a lot of the time; that's fine. Remember that Scheme is a functional language, meaning that we don't want to think of our function as passing around this variable that we're changing, but rather as performing one long computation.

You might find that some of your old queue tests will fail after you implement the functions below. This **might** be okay; it's possible that the actual queue you're getting is effectively the same as the queue you say you're expecting, in that the actual queue has the forwards and backwards queues reversed from the expected queue. In this case, you can either get clever or comment out those tests. But be sure that you aren't ignoring tests that point up a legitimate error in your code!

(a) [1 point]

(define-struct double-queue SOME-FIELDS-HERE)

(b) [2 points]

Rewrite queue-new to conform to the two-list queue specifications.

(c) [2 points]

Rewrite queue-empty? to conform to the two-list queue specifications.

(d) [2 points]

Rewrite queue-push to conform to the two-list queue specifications.

(e) [3 points]

Rewrite queue-get-next to conform to the two-list queue specifications. (Recall that queue-get-next gets the next element we'll want; don't worry about modifying the queue! That's what queue-pop is for.)

(f) [4 points]

Rewrite queue-pop to conform to the two-list queue specifications.

7 Dictionaries more efficiently

Even more luckily for you than the original problem set made out, we've decided that this exercise is sufficiently complicated that we're pushing it back to next week's part of the

project. I know you're all waiting with bated breath to see how we're going to go about improving our silly dictionary implementation, but don't worry - taking our time with this one and doing it in more depth will turn out to be well worth the wait.

Until next week!