Section Notes 4

CS51 — Spring, 2008

Week of February 24, 2008

1 Wisdom

"The Plug and Socket Model = many types of appliances, many sources of power, one interface!"

"An interface is too low level when it requires attention to the irrelevant."

"I object to doing things that computers can do."
— Olin Shivers

"One man's constant is another man's variable."
— Alan Perlis

2 Revisiting Higher-order functions

Last week we discussed *higher-order functions* (functions that takes functions as arguments). Here is a brief recap of one of the functions, `reduce`, that can be tricky to wrap your head around, but is hugely valuable. We usually think of `reduce` as combining the elements of a list to produce some result. This effect is achieved by providing a combining operation and a base value that will be used to process the input list. When we call `reduce` we provide three arguments: a binary functions we will call `new_cons`, a value we will call `base_case`, and our input list `lst` (i.e. `(reduce new_cons base_case lst)`). Consider:

```
(define lst '(1 2 3)) →(define lst (cons 1 (cons 2 (cons 3 '()))))
```

The function `reduce` simply replaces all instances of `cons` in `lst` with `new_cons`, and replaces the `'()` with `base_case`. How would we express the result of calling `(reduce new_cons base_case lst)`?

Now for a more concrete example, what do we get if we replace `new_cons` with `+` and `base_case` with 0 (as in `(reduce + 0 lst)`).

Can you think of another mathematical function that we can compute with `reduce`?

2.1 Custom reduce operations

We can also do fancier operations by varying `new_cons`. Rather than limiting ourselves to simple built-in functions like `+` and `*`, we can pass our own functions as created by either define or lambda. One thing to remember is that `new_cons` must *always* be a binary function. Why? (Hint: how many arguments does `cons` take?)

For our last example, let us look at interleaving a fixed value, say `0` into a list. Consider the following function:

```
(lambda (x y)
  (append (list x 0) y))
```

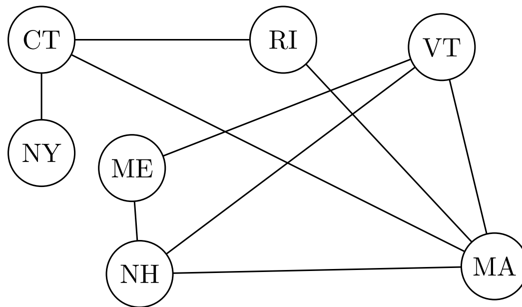
How would we write out what we would get if this lambda function is provided to `reduce` as `new_cons` and `'()` is provided as `base_case` along with `lst` from above?

```
(<lambda> 1 (<lambda> 2 (<lambda> 3 '())))
```

Can you manually expand and evaluate this expression?

3 Graphs

Graphs are general models useful for representing any situation involving entities with relationships or connections between them. A graph is defined as a set of *vertices* (or *nodes*) and *edges* connecting them. The classic real-world analogy would be that nodes are cities and edges are roads, train tracks, airline routes, etc. from one city to another. Consider the following graph:




```
N = {CT, MA, ME, NH, NY, RI, VT}

E = {(CT,MA),(CT,RI),(MA,NH),(MA,VT),(CT,NY),
     (NH,ME),(NH,VT),(RI,MA),(VT,ME)}
```

This particular graph is *undirected*, meaning that whenever A is connected to B it is implied that B is connected to A. What are some common situations that can be modeled by undirected graphs?

By contrast, in a *directed* graph, it matters which way vertices are listed in the ordered pair representing an edge (such as `(CT,NY)`). For nodes to be mutually connected in a directed graph, we have to specify, for example, that NY connects to CT and that CT connects to NY. One case where this is useful is when we want to label edges with weights (representing length, travel time, cost, etc.) and the directions between two places are unequal. Think of the George Washington Bridge, which connects New York and New Jersey. Nobody really wants to go to New Jersey, so that side of the road moves smoothly with little delay. The other side, filled with drivers desperate to get to Manhattan, is often more crowded. To model this scenario we would need two directed edges with different travel times, even though the two edges connect the same places.

4 State space search

The *state space* of a problem is the set of all potential solutions. This concept allows us to think of solving a problem as searching for a solution—making our way through the state space in some systematic way to discover an answer.

4.1 Example: N-Queens

The famous "N-Queens" problem asks us to place N queens on an N-by-N chessboard so that none of them could capture another (recall that a queen may attack anywhere on its row, column, or diagonal).

Try to place 4 queens on a 4-by-4 board. If you put the first queen in the top left corner, you will get stuck after placing 2 queens. However, this doesn't mean the problem is unsolvable, since you haven't exhausted all possibilities. You must *backtrack* and try placing the first queen somewhere else.

To make sure we exhaust all posibilities, we must employ some strategy.

4.2 Depth-first search

Depth-first search (DFS) is a popular strategy for searching a space. DFS is characterized by the idea of exhaustively pursuing the current path before considering alternate paths.

To do a DFS, begin with a *stack* containing the starting state. We then repeat two simple steps:

- Pop the top state S off the stack and process it
- Push states adjacent to S onto the stack

We decide which states to visit next based on the current state. Say we decide to visit states A, B, and C next. Because we're using a stack, state A will be considered next, followed by A's first child. State B will only be considered when the search of A's children terminates. This is the depth-first strategy.

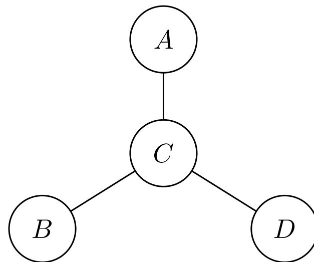
Since DFS uses a stack, it lends itself nicely to recursive implementation, where the interpreter's call stack can handle stack operations for you implicitly.

4.3 Searching for colorings

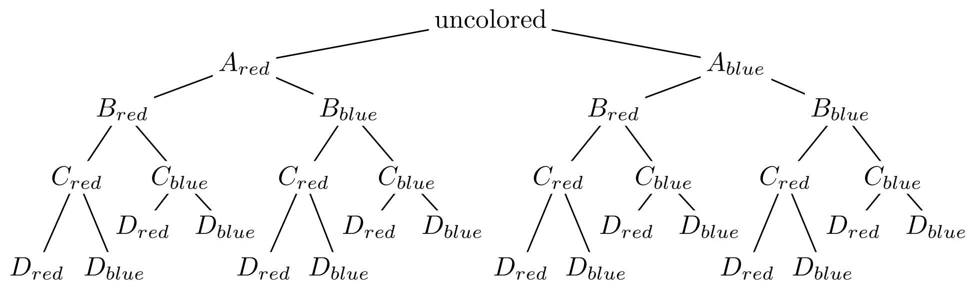
To use state space search, begin by describing what a particular state looks like. In the case of graph coloring, a state is an assignment of colors to nodes. For example, in Scheme we can represent a coloring using an association list such as `((A red) (B blue) (C red) ... )`.

To search for a coloring, we need to decide how to generate the "next states" to visit, given the current state. We are free to use any definition of adjacent states that we wish, as long as our definition is consistent and guaranteed to eventually reach the entire relevant space. In the case of graph coloring, moving to an adjacent state probably means changing the color of a single node.

Let's look at the state space for 2-coloring the following graph:



We are searching for a state in which all nodes are colored, and no two connected nodes have the same color. I will artfully define an "edge rule" that makes it easier to draw the state space graph: from a given state, we can move to states where the alphabetically first uncolored node has various colors. Convince yourself that this rule allows us to reach every state where all nodes are colored (some states where only some of the nodes are colored are *not* reachable, but that's ok since we are not searching for those states). Here is a visualization of this state space:



This search order makes the state space graph look like a tree.

Since the state space is a tree, there are no cycles and we know we will never accidentally try the same coloring twice. Therefore we don't need to explicitly check whether we've tried a certain coloring before. As you can probably imagine, this is a convenient property for a search space to have.

4.4 Effective graph coloring

To color a graph, you will be doing a DFS down a tree like this one. To put the next states on the stack, you make recursive calls with the current node's color set to each possibility in turn. One method would be to loop over every color in the palette. Then, in the base case when no nodes are left to be colored, check the graph to see if we've found the goal state, i.e. no two adjacent nodes have the same color.

However, this method would be rather inefficient (please *do not* implement it!). There is no reason to pursue colorings that *local* information tells us are certain to fail. For example, if we color nodes A and B red, there is no need to search the branch of the tree where C is red, since we can tell immediately that there is a conflict. This way, we only ever generate colorings that satisfy the constraint, so as soon as all nodes are colored we know we have succeeded. Similarly, if we try to color a node and no locally unique colors are available, we know we have failed and can immediately return some value like `#f` to indicate this.

Here is an overview of our graph coloring algorithm:

1. If no locally unique color exists or the current node has tried all possible colors without finding a coloring that works, return a value that signifies failed coloring.
2. Otherwise, choose the next locally unique color.
3. If all nodes have been colored, return the coloring.
4. If not, pick an uncolored node and color it recursively
   - If coloring that node fails, go back to step 1 (i.e. try the next color for this node).
   - If coloring that node succeeds, go back to step 3 (i.e. see if we have finished coloring the graph).

The way you pick the next uncolored node is up to you, but it isn't terribly important. Picking the next node randomly is fine.

4.5 Testing and backtracking

In the problem set we ask you to develop a test case that demonstrates backtracking. Which graphs require backtracking to k-color greatly depends on the particular algorithm you use.

Finding an interesting test graph can be challenging, because the *local search* heuristic our algorithm uses often gets "lucky" and ends up picking good colors right away on simple graphs.

To show backtracking, and for debugging purposes, you will want to instrument your code to display what it is doing. With good data abstraction, it should be easy to modify your program to provide useful verbose output.

5 Abstraction

Data abstraction and procedural abstraction are critical in this assignment. Scheme does not allow us to create genuinely new types; everything must be built out of pairs, numbers, and symbols. Therefore when implementing and using abstract data types in Scheme, responsibility lies with you to 1) make sure you access your data in such a way that the underlying representation could easily change, and 2) make sure you do not end up with invalid data structures.

5.1 Respecting boundaries

Part of creating an abstraction is defining what operations on it are valid. In order to observe the two rules given above, you should only use these supported operations on an object. For example, even if you know that a dictionary is implemented as a list, you should not use `car` and `cdr` on it, but rather the provided lookup functions. As another example, if graphs are implemented using dictionaries, then graph operations

are allowed to call dictionary functions to do their work, but "outsiders" should only use the provided graph functions on graphs.

If these rules seem strict, remember that not all data has to be abstract. Graph colorings, for instance, are *not* abstract. They are defined to be association lists, which are just Scheme lists of the form `((key1 value1) (key2 value2) ... )`. Because they are just lists, you are free to use `car`, `cdr`, etc. on them. Even though dictionaries *might* have this exact representation as well, we are not allowed to make that assumption. Be sure you understand the distinction.

For practice, name an abstraction and ask what operations it needs to support. Think of the data types and structures you're familiar with, such as: queue, stack, tree, list, string, set.

6 Implementing graph coloring

6.1 Representing graphs

Let's look at some graphs represented using neighbor lists. Notice that in order to represent undirected graphs this way, membership in neighbor lists must be symmetric. This way I always get the same answer to "Is A a neighbor of B?" and "Is B a neighbor of A?".

Try to color these graphs using the palette `(red green blue)`:

Graph 1: `((n1 n2 n3 n4) (n2 n1 n3 n4) (n3 n1 n2 n4) (n4 n1 n2 3))`

Graph 2: `((n1 n2 n6) (n2 n1 n3 n6) (n3 n2 n4 n5) (n4 n3 n5) (n5 n3 n4 n6) (n6 n1 n2 n5))`

How would you color the graph `((n1) (n2))`?

6.2 Advice

- Get to know the Scheme function `assq` for working with colorings.
- Expect spork-schedule to have three to five helper functions.
- Expect color-graph to have about two helper functions, and be aware that those helper functions might themselves require helper functions.
- Try to name functions as usefully as possible, instead of `colorgraph-helper-helper`. If there doesn't seem to be a sensible name for some function, you might want to factor the problem differently.
- Be aware of corner cases like disconnected nodes.