

# Section Notes 6

CS51—Spring 2009

Week of March 9, 2009

## Outline

1. Combinators and Parsing
2. Dijkstra's Algorithm
3. Impeccable Style

## 1 Goals for today:

At the end of today's section, you should be able to do the following:

1. Combine. Yep, it's a verb<sup>1</sup>.
2. Find the cheapest way to get to Miami for Spring break
3. Eliminate those pesky unnecessary lambdas from your programs.

## 2 Combinators and Parsing

We've already encountered the *parsing* problem in project 1: how to understand a string in terms of the grammar that generated it. Furthermore, in lecture, we saw a surprisingly elegant way to do this using higher-order parsing functions, or *parser combinators*.

In this example, we'll be applying those ideas to write a recognizer for valid arithmetic expressions.<sup>2</sup> Our recognizer will take expressions, like  $(4+1)*9-3$  or  $4+*/193$ , and tell us whether they are valid arithmetic expressions (true in the first case, false in the second).<sup>3</sup>

Before writing this recognizer, though, let's write out a BNF<sup>4</sup> grammar for our language of valid arithmetic expressions. We have to be somewhat careful in doing this. In particular, consider the following grammar.

```
exp ::= exp + exp
      | exp - exp
      | exp * exp
      | exp / exp
      | (exp)
      | num
num ::= digit
      | digit num
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

---

<sup>1</sup>In Italian.

<sup>2</sup>Later, we'll see how to turn this into something more useful, like a calculator for said expressions.

<sup>3</sup>And the expressions will be infix, rather than fully-parenthesized-prefix. Truly, this is the best of all possible worlds!

<sup>4</sup>Backus-Naur Form.

There are a few problems with this grammar. One problem is intrinsic to the grammar, and involves *ambiguity*. Can you come up with a string that can be parsed two or more ways according to this grammar?

Two other problems pertain to our usual style of implementation as applied to this grammar, and have to do with *left-recursion*<sup>5</sup> and *search order*, respectively. Can you identify them?

Which of those three issues are solved by the following grammar?

```
exp ::= num op exp
      | num
      | (exp)
op  ::= + | - | * | /
num ::= digit num
      | digit
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

And the following grammar?

```
exp ::= term + term
      | term - term
      | term
term ::= factor * factor
      | factor / factor
      | factor
factor ::= num
         | (exp)
num    ::= digit num
         | digit
digit  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

For simplicity, we'll use the second grammar above, though we'll want to use the third later, in lecture (when we set out to *parse* rather than just *recognize*). Now, how do we go about writing our recognizer? Recall from lecture that a pattern *p* is a function taking a list of characters and returning either a *result* or some non-list failure value (hackily taken to be 0), where a *result* is the remainder of the list after the pattern has matched as many of the initial characters as possible.<sup>6</sup>

As a warmup, let's write the **never** and **always** patterns: the former always fails to match, while the latter just matches the empty string (so will always succeed, even if the character list being matched against is empty).

```
(define (never l) ...)
(define (always l) ...)
```

A **parser combinator** is a higher-order function over patterns (i.e., taking pattern(s) and/or returning a pattern). For our grammar, we'll need the following combinators:

```
(define (alt p1 p2) ...) ; alternation (i.e., p1 or else p2)
(define (alts ps) ...)  ; alternation of a pattern list
(define (star p) ...)   ; Kleene star (zero or more p's)
(define (plus p) ...)   ; one or more p's
(define (cat p1 p2) ...) ; concatenation
(define (cats ps) ...)  ; concatenation of a pattern list
```

---

<sup>5</sup>As alluded to in lecture, *left-recursion* is when a nonterminal symbol occurs first in its own expansion, with no intervening terminal symbols matched in the string.

<sup>6</sup>Matching as many characters as possible means our parsers are *greedy*. Not all parsers must be greedy, and we should justify that this approach will work with our particular grammar. An alternative is exhaustive search – in this case, we return a list of possible results, rather than the result with the most characters immediately matched. This is inefficient, but can be sped up using a technique called *dynamic programming* in which we reuse results for substrings to parse larger strings. For more information, see, e.g., Sipser (*Introduction to the Theory of Computation*, Ch. 7).

And a couple of generally-useful patterns:

```
(define (char c) ...) ; only accepts the single character c
(define (charset s) ...) ; only accepts a single character that
                        ; is present in the string s
```

Now we can write the recognizer for our grammar:

```
(define digit ...)
(define num ...)
(define op ...)
(define exp ...)
(define (is-arith-exp s) ...)
```

As we can see, it's much easier to have higher-order functions write our code for us than to do it ourselves.

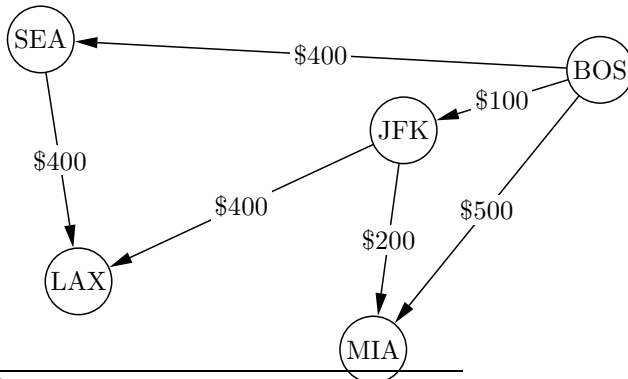
### 3 Dijkstra's Algorithm

As we have seen in lecture, *Dijkstra's algorithm* gives us a method for determining the cheapest path through a graph from a source node to the other nodes. Why do we care about this? Well, graphs are used to model information in many different contexts. Finding the shortest path from one point to another is a common problem. Applications vary from figuring out the best route between real-world points to routing packets across the globe over a hodge-podge of networks that we call the Internet.

In CS51, we study this algorithm for a couple additional reasons. First, this is a classic algorithm that you will want to be familiar with as you proceed through life<sup>7</sup>. Second, it's a great, non-trivial extension to topics we have already covered. In particular, this algorithm makes use of edge weights or costs that are associated with each edge. The algorithm itself is trying to find a path that minimizes the sum of these costs<sup>8</sup>.

#### 3.1 Motivating Example

Let's assume that you are a Harvard student<sup>9</sup>. Let's further assume that you have limited cashflow<sup>10</sup>. Lastly, let's assume that you have had enough of the cold weather and would like to spend Spring Break in a warmer climate<sup>11</sup>. Given these assumptions, we have come up with a graph that you might find worth studying:



---

<sup>7</sup>Really! While you are unlikely to apply it by hand, you will encounter problems that can be solved using the algorithm in many fields and applications

<sup>8</sup>Note that we must have non-negative costs on edges for this algorithm to work properly in all cases. Can you see why?

<sup>9</sup>Quite a stretch, I know.

<sup>10</sup>Yet another mind-bender, I'm sure.

<sup>11</sup>That's the last of our crazy assumptions.

The nodes in this graph represent airport codes for US cities. Edges between these nodes represent direct flights that can be taken to get from one airport to another. The labels on the edges represent the best prices that the CS51 Fare Finder could secure for each of these direct flights. We can now use Dijkstra's Algorithm to compute, for each airport, the cheapest price to get to it.

Dijkstra's Algorithm works by keeping track of a set of "final-cost" nodes, a set of "nonfinal-cost" nodes, and a "current" node. The algorithm proceeds as follows:

1. Associate a cost with each node in the graph. Set our initial node (e.g. BOS in our example) to a cost of \$0 and mark all the others as having an infinitely high cost.
2. Make our initial node the "current" node and add all other nodes to the "nonfinal-cost" set.
3. For "current" node, take its neighbors that are in the "nonfinal-cost" set and compute their cost from the initial node. For example, if "current" node JFK has cost of \$100, and its neighbor MIA is connected by an edge, \$200, the cost to get to MIA through JFK will be  $\$100 + \$200 = \$300$ . If this is cheaper than current cost associated with MIA, then associate the new cost with the node.
4. After computing costs and possibly updating for each neighbor of "current", remove it from the "nonfinal-cost" set and add it to the "final-cost" set.
5. Find the node in "nonfinal-cost" with the smallest cost and make it the next "current" node and continue from 3.

In lecture, Ramin referred to the "nonfinal-cost" set as  $Q$  and the "final-cost" set as  $S$ . Run this algorithm on the graph above by hand, maintaining current,  $Q$ , and  $S$  as you go. You may find it helpful to mark the current cost right next to the nodes on the graph above. You should find that you can reach the two "warm-weather" destinations on our map for no more than \$500, though in one case you should find that while running the algorithm you must update the current cost at least once.

## 3.2 Correctness

After walking through an example you may have an intuitive sense that this algorithm produces the desired results, but can we turn our intuition into a more compelling argument? Even if you do not need to formally prove the correctness of this algorithm, you will probably understand it better if you can identify the invariants precisely and provide a rigorous explanation for why it works.

Let's use the analytical tools that we have developed in writing inductive proofs to help us. The property that we would consider proving is our invariant that the set  $S$  only contains nodes for which we have identified the lowest cost route from the source node. From this start, can you provide a brief sketch of what the proof would look like? You will probably find it helpful to consider the recipe that we talked about last week. Also, rather than formally proving the inductive step, just provide the basic reasoning for why it holds.

### 3.3 Implementing in Scheme

While the algorithm stated earlier should make sense to you, it may not be immediately clear how we would implement this in a functional programming language like Scheme. In particular, we probably don't want to start mutating variables everywhere that we said "make foo the current node" and continue. Also, while it seemed useful to mark down the lowest cost to reach a node on the graph itself, this seems to imply that we will be constantly updating things in our graph. Note that we have avoided mutation (i.e. changing stored values) in our programs so far. Our decision to avoid mutability was not arbitrary, but in case you weren't sure, here are a few of the benefits:

1. Identical expressions evaluate to identical values (*referential transparency*) with no side-effects. This is really the alpha-and-omega benefit, from which many others follow.
  - (a) Shared data structures don't need to be copied.
  - (b) Our code is thread-safe (i.e. two instances running simultaneously would not trample on each other).
  - (c) Allows certain compiler optimizations (e.g. eliminating function calls when the return value is not used, since side-effects are impossible).

With all this at stake, you may be wondering if we have lost something with this algorithm. Never fear, however: we are not<sup>12</sup> ready to compromise the benefits of immutability that we have worked so hard to cultivate.

The key to implementing Dijkstra's algorithm in a functional language is to break down our state into its components and think recursively. So, instead of having a "current" variable that gets reassigned, we will have a let-bound variable that we compute each time we enter a recursive function to which we will assign the name of the node with the smallest cost. But where will we store these costs? Rather than worrying

---

<sup>12</sup>Yet.

about extending our graph interface, we will construct a new association list that associates node names with their costs. We can then pass this around along-side our graph. As for updating the costs, we won't mutate the contents of this structure, but instead produce a new structure with the updated costs as needed.

How will all this look in practice? Well, here's one possible interface for the "interface" to Dijkstra's algorithm:

```
; dijkstra: node graph -> association-list
;
; This function takes a graph with edge-weights and the name of a node in that
; graph that will be used as a source. This function returns an association
; list that maps node names to costs, where each cost represents the cheapest
; cost to get from the source node to the named node.
(define (dijkstra source graph) ...
```

Internally, this function will depend on helper functions. For example:

```
; update-neighbor-costs: node graph association-list -> association-list
;
; This function takes the name of a node, the graph that contains it, and an
; association list that maps nodes in the graph to their costs. For each
; neighbor of the provided node in the graph, the edge weight between node
; and the neighbor will be added to the current cost of node (found in
; the association list). If this sum is less than the value for the neighbor in
; the provided association list, then the new sum will be included in place of
; the old in the returned association list. Otherwise, for non-neighbor nodes
; or neighbors where the sum is greater, the original value is used in the
; returned association list.
(define (update-neighbor-costs node graph costs) ...
```

We might also want to replace association lists with something more efficient, like immutable hashes – but regardless, it should be clear how we can express a traditionally imperative algorithm like Dijkstra's in functional style.

## 4 Eta-Expansion (A Style Note)

When using higher-order functions, it's common to find oneself writing things like:

```
(foldr (lambda (x y) (+ x y)) 0 1)
```

But remember that `(lambda (x y) (+ x y))` is an expression that, when evaluated, yields *a function that takes two arguments and adds them*. We know of another expression like that, and it's much shorter:

```
(foldr + 0 1)
```

Replacing `F` by `(lambda (...) (F ...))` is known as *eta-expansion* of `F`. There are legitimate reasons to do eta-expansion in call-by-value<sup>13</sup> languages like Scheme. For instance, if we are writing a Kleene star combinator, as in Tuesday's lecture:

```
(define (star p)
  (lambda (cs) ((alt (cat p (star p)) always) cs)))
```

We might want to eta-reduce (the opposite of eta-expand), and write:

---

<sup>13</sup>I.e., "strict" or "non-lazy" languages: those in which arguments must be evaluated before the functions are called.

```
(define (star p)
  (alt (cat p (star p)) always))
```

But this function will enter into an infinite loop when we try to build a parser! We use eta-expansion to "delay execution" until we actually have a string of characters to match on. In call-by-name languages, there is no need for this because execution will be delayed until necessary. This is in essence why such languages are called "lazy".

We might also legitimately want to use (implicit) eta-expansion to circumvent Scheme's restriction on using an identifier in its own definition:

```
(define (exp l) ((alt (cats (list num op exp)) num) l))
```

rather than:

```
(define exp (alt (cats (list num op exp)) num)) ; (error)
```

Still, functions should only be eta-expanded when necessary to work around language issues. An example of where eta-expansion can go astray came up in Problem Set 3. On this assignment you may have been tempted to write `my-compose list` using a `lambda`. One example out of many possible answers would look like:

```
(define (my-compose-list l)
  (lambda (x)
    ((foldr my-compose id l) x)))
```

Many students arrive at such an answer by the following line of reasoning:

1. I know that I want `my-compose-list` to return a function of one input.
2. I'll achieve the above by writing a `lambda`...
3. I'll futz with the body of the `lambda` until I get it to work.
4. I have a test case that passes, it must be time to submit my work!

The flaw with this logic is that it overlooks the fact that writing a function that returns a function of one input may be possible without an explicit `lambda`. In this case, `my-compose` returns to you such a function, so the extra `lambda` is unnecessary. In cases like this one, the code can be simplified to eliminate the extra indirection. As a general rule, you have probably gone astray if you find yourself writing code that looks like:

```
(lambda (x)
  (<some-expression> x))
```

without a very good reason.

## 5 Answers

### Combinators and Parsing

Problems with the first grammar:

- It is ambiguous.  $3+4*5$  can be parsed either as:

```
      exp
      +
exp   exp
num   *   exp
digit digit digit
  3       4   5
```

or as:

```
      exp
      *
exp   exp   exp
exp + exp   num
num   num   digit
digit digit   5
  3       4
```

- The `exp` nonterminal is left-recursive. If we had a pattern for `exp`, it would call itself to match the first `exp` in `exp + exp`, and so on *ad infinitum*.
- The `num` nonterminal matches on `digit` before `digit num`. Because we are deterministically taking the first successful match, we'll match, e.g., the 4 in 42 as a digit, rather than matching the whole thing as a number.

The second grammar solves the latter two problems, but it's still ambiguous. The third grammar solves all three problems, but for *recognizing* (rather than *parsing*) it doesn't matter if our grammar is ambiguous, so we just used the second one.

Solutions not already in the lecture notes:

```
(define digit (charset "0123456789"))
(define num (plus digit))
(define op (charset "+-*/"))
(define (exp 1) ((alt (cats (list num op exp)) num) 1))
(define (is-arith-exp s) ...)
```

### Dijkstra's Algorithm (Section 3.2)

Recipe for proving that Dijkstra's Algorithm is correct:

1. Variable is  $n$ , the size of the set  $S$ .
2.  $P[n]$  = after  $n - 1$  iterations, all  $n$  nodes in  $S$  are annotated with the cost of their cheapest path from the source node.
3. Base case: show that  $P[1]$  holds. After 0 iterations, at the beginning of the algorithm, the cost to the source is initialized to 0, and the source is the only vertex in  $S$ . This is the cheapest cost, since if there are no negative edges, any path to the source that follows an edge will be more expensive than the empty path.

4. Inductive hypothesis: assume  $P[n]$  holds for arbitrary  $n$ . Now prove  $P[n + 1]$  from  $P[n]$ .

In a full proof we would need to establish that we have appropriately updated the costs of reaching the nodes in “nonfinal-cost” set, so that any node in this set is annotated with the minimum cost of reaching it via a path that stays entirely in  $S$ . With that shown, we then show that we only add an element to  $S$  when we know the cheapest way to reach that node.

The algorithm adds to  $S$  the node  $v$  in “nonfinal-cost” with the smallest cost  $c(v)$ . We have that this cost is minimal among paths from the source that stay in  $S$  until they get to the given node; and we know the node cannot be reached more cheaply via any path going outside  $S$ , since such a path would first visit some *other* node in “nonfinal-cost”, already incurring a higher cost than  $c(v)$  (and never decreasing it while following additional edges, since there are no negative edge weights). So when we add  $v$  to  $S$ , we know we have annotated it with the minimum possible cost from the source overall.

By the inductive hypothesis we know that the  $n$  existing elements in  $S$  are annotated with the minimum cost from the source; we add a new element with this property, and do not change the existing ones, so the property again holds of our new  $S$  with  $n + 1$  elements.