

# Section Notes 7

CS51—Spring 2009

Week of March 30, 2009

## 1 Outline

1. Dijkstra's Algorithm review/implementation discussion
2. Parsing Combinators review/discussion
3. Midterm return/common mistakes

At the end of this section, you should be able to do the following:

1. Be able to understand/modify a BNF grammar
2. Understand how to use parsing combinators
3. Appreciate how combinators generate code that we'd otherwise have to write manually.
4. Understand how to implement Dijkstra's algorithm in Scheme

## 2 Parsing Combinators

For PS6, you'll need to understand how the parsers and parser combinators defined in lecture work, so we decided that the most useful thing to do in section is to go over the ones we gave you.

First, here's the example grammar from before break, just so we have some context:

```
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
```

And here are the contents of the parse.ss file:

```
; The main idea: a parser is a function that takes a list of chars,
; and tries to match some of them.  If it fails, it returns a fail
; struct (make-fail); if it succeeds, it returns an ans struct, with
; the first field being a result of the parse (a number, string,
; etc...), and the second being the unconsumed characters from the
```

```

; list.

(define-struct ans (result unconsumed))
(define-struct fail ())

; Try to parse string s with parser p. The parser must successfully
; consume all of s.
(define (parse p s)
  (let ([a (p (string->list s))])
    (if (or (fail? a) (not (empty? (ans-unconsumed a))))
        (error 'match "parse failed")
        ; Since the parser matched the whole string, can just return
        ; the result of the parse
        (ans-result a))))

; The parser that always succeeds without consuming any characters.
; Note that the result of the answer is the empty list. (This is
; useful in defining cats below)
(define always
  (lambda (cs) (make-ans empty cs)))

; The parser that always fails
(define never
  (lambda (cs) (make-fail)))

; (failwhen f p) is a parser that behaves the same as the parser p,
; except fails whenever p's result doesn't satisfy the predicate f.
; For example, we can pass a predicate like
;
; (lambda (x) (not (or (equal? x "name") (equal? x "age"))))
;
; and a general alphabetic string parser to reject any parses
; that don't return "name" or "age".
; (Note that our parsers are greedy by default, so if we used
; such a combinator to parse "ageforty-two", it would read
; "ageforty", conclude that it was neither "name" nor "age",
; and therefore fail.)
(define (failwhen f p)
  (lambda (cs)
    (let ([a (p cs)])
      (if (or (fail? a) (f (ans-result a))) (make-fail) a))))

;(char c) is a parser that parses the single character c, returning c
;as the result.
(define (char c)
  (lambda (cs)
    (cond [(empty? cs) (make-fail)]
          [(equal? (car cs) c) (make-ans c (cdr cs))]
          [else (make-fail)])))

; (cat p1 p2) is a parser that runs parsers p1 and p2 in succession,

```

```

; returning a cons of their results if both succeed.
(define (cat p1 p2)
  (lambda (cs)
    (let ([a1 (p1 cs)])
      (if (fail? a1) a1
          (let* ([r1 (ans-result a1)]
                 [cs1 (ans-unconsumed a1)]
                 [a2 (p2 cs1)])
            (if (fail? a2) a2
                (make-ans (cons r1 (ans-result a2))
                          (ans-unconsumed a2))))))))))

; (cats ps) is a parser that concatenates all the parsers in ps--runs
; them in order, and conses the results if they all succeed. Note
; that the definition of always (above) means that the result we get
; back on success is a proper list that ends in empty.
(define (cats ps) (foldr cat always ps))

; (alt p1 p2) is a parser that does alternation: tries running p1, and
; only if that fails tries running p2. (If p2 then fails, the alternation
; fails.)
(define (alt p1 p2)
  (lambda (cs)
    (let ([a1 (p1 cs)])
      (if (fail? a1) (p2 cs) a1))))

; (alts ps) is a parser that implements alternation over the list of
; parsers. It tries them in order, and stops after the first one succeeds.
; (If they all fail, it fails -- hence the zero of our foldr is never.)
(define (alts ps) (foldr alt never ps))

; (pmap f p) is a parser that runs p, then replaces the result with (f
; result).
(define (pmap f p)
  (lambda (cs)
    (let ([a (p cs)])
      (if (fail? a) a
          (make-ans (f (ans-result a)) (ans-unconsumed a))))))

; (str s) is a parser for the given string s. If it succeeds it
; returns the list of chars matched (not a string).
(define (str s)
  (cats (map char (string->list s))))

; A parser that tries to match p, but then returns success without
; consuming any chars if that fails. (I.e., a parser for zero or one p's.)
(define (optional p) (alt p always))

; Match p zero or more times, returning a list of the answers.
(define (star p)
  (lambda (cs)

```

```

; We have to eta-expand this definition to prevent the recursive
; call to (star p) from expanding forever at definition time.
; This way, it only gets expanded when actually called with a list
; of chars, so as long as p consumes at least one char, it
; eventually stops. (Fun note: (parse (star always) "oh oh") just
; keeps expanding forever and causes an infinite loop.)
((alt (cat p (star p)) always) cs)))

; Match p one or more times
(define (plus p) (cat p (star p)))

; Match any single character in the string s
(define (charset s) (alts (map char (string->list s))))

; Convert a digit char like #\7 to the corresponding number (7 in this case)
(define (digitchar->num c)
  (- (char->integer c) (char->integer #\0)))

; parses a digit
(define digit (charset "0123456789"))

; Convert a list of digit chars to a number: (list #\4 #\2) -> 42
(define (digits->num ds)
  (foldl (lambda (d a) (+ (digitchar->num d) (* 10 a))) 0 ds))

; Parse numbers, and convert the result from a list of chars to an
; actual number. For example:
;> (parse (plus digit) "123")
;(#\1 #\2 #\3)
;> (parse number "123")
;123
(define number (pmap digits->num (plus digit)))

; Parse a single whitespace character (space, tab, newline)
(define white (alts (map char (list #\space #\tab #\newline))))

; (const v p) runs the parser p, and always returns the constant value
; v if p succeeds. (useful if you want to make sure some syntactic
; element is present, but don't care about the actual
; contents--whitespace is a good example)
(define (const v p) (pmap (lambda (r) v) p))

; match zero or more whitespace chars
(define ws (star white))

; Run p1 and p2 as in a cat, but only return the result of p2
(define (catr p1 p2) (pmap cdr (cat p1 p2)))
; Run p1 and p2 as in a cat, but only return the result of p1
(define (catl p1 p2) (pmap car (cat p1 p2)))

; Parse whitespace followed by a number, and ignore the whitespace in

```

```

; the result.
; Example:
;> (parse number " 123")
; match: parse failed
;
;> (parse (catr ws number) " 123")
;((#\space) . 123)
;
;> (parse num " 123")
;123
(define num (catr ws number))

(define alpha (charset "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNopqrstuvwxyz"))
(define sym (charset "+-*/=?<>"))

; You're supposed to explain what this does on PS6. Make sure you
; understand catr and catl first.
(define (cats-n n lst)
  (if (= n 0)
      (catl (car lst) (cats (cdr lst)))
      (catr (car lst) (cats-n (- n 1) (cdr lst)))))

; Explained in ps6, but here's another example:

;(parse ((pat-k num)
;        (lambda (n)
;          (cats (list ws (plus alpha) ws
;                    (failwhen (lambda (x) (not (equal? x n))) number))))))
;"123 abc 123"
;((#\space) (#\a #\b #\c) (#\space) 123)

;(parse ((pat-k num)
;        (lambda (n)
;          (cats (list ws (plus alpha) ws
;                    (failwhen (lambda (x) (not (equal? x n))) number))))))
;"123 abc 124"
; match: parse failed
(define (pat-k p)
  (lambda (k) ; (a continuation)
    (lambda (cs) ; We return a parser combinator, i.e. a function taking
      ; a list of characters cs...
      (let ([r (p cs)] ; which first applies the pattern p to cs,
            (if (fail? r)
                r
                ((k (ans-result r)) ; then gives p's result to the
                    ; continuation k, yielding
                    ; another parser combinator...
                    (ans-unconsumed r)))))) ; which we finally apply
      ; to the remainder of
      ; the string.
    )
  )
(define (pat-k-str p)

```

```
(lambda (k) ; (a continuation expecting a string)
  ((pat-k p) (lambda (cs) (k (list->string cs))))))
```

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

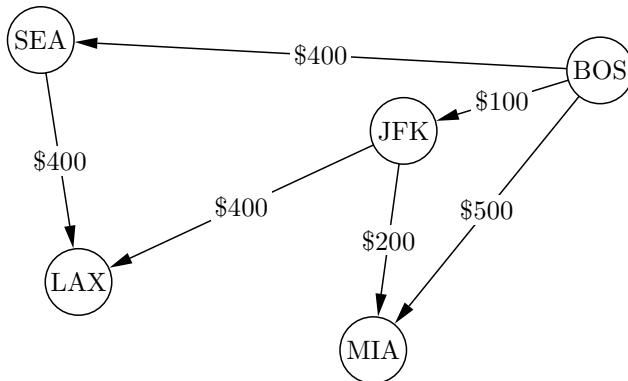
### 3 Dijkstra's Algorithm

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>1</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>2</sup>.

#### 3.1 Motivating Example

Let's assume that you are a Harvard student<sup>3</sup>. Let's further assume that you have limited cashflow<sup>4</sup>. Lastly, let's assume that you have had enough of the cold weather and would like to spend your summer in a warmer climate<sup>5</sup>. Given these assumptions, we have come up with a graph that you might find worth studying:



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.

---

<sup>1</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>2</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>3</sup>Quite a stretch, I know.

<sup>4</sup>Yet another mind-bender, I'm sure.

<sup>5</sup>That's the last of our crazy assumptions.

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 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>6</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? Luckily for you, we've implemented a **dictionary** module. A dictionary is just a way of associating keys with values. So, using our dictionary interface, you will be able to associate each node in the graph with its cost at each step. However, since our dictionaries are immutable, you won't actually be carrying around one dictionary with changing values; instead, you'll be creating a whole new dictionary at each step, with the one entry that you wanted to change updated to its new value.

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

```
; dijkstra: node graph -> dictionary
;
; 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 a dictionary
; 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 dictionary -> dictionary
;
; This function takes the name of a node, the graph that contains it, and a
; dictionary 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 dictionary). If this sum is less than the value for the neighbor in
; the provided dictionary, then the new sum will be included in place of
; the old in the returned dictionary. Otherwise, for non-neighbor nodes
; or neighbors where the sum is greater, the original value is used in the
; returned dictionary.
(define (update-neighbor-costs node graph costs) ...
```

There are other difficulties that you will encounter that we have not discussed explicitly here. For example, in Project2a, it will no longer be enough to simply return the lowest cost required to get to each node: we now actually do need to know the path taken to get there. You will have to think about what data structure you will add to the ones discussed above in order to keep track of this information, and how you will update it at each timestep.

---

<sup>6</sup>Yet.

## 4 Exam review

### 4.1 1d

If you recall, `foldr` takes a list and replaces all of the `cons` constructs with the function given to `foldr`. In this case then, `foldr` takes the list:

```
(cons 2 (cons 3 (cons 5 '())))
```

and replaces it with `'` to obtain the following: (

```
(- 2 (- 3 (- 5 0)))
```

as you may notice, the empty list at the end was replaced with 0, the base case given to `foldr`. Then simply this new expression is evaluated, which yields the value 4, because  $(2 - (3 - (5 - 0))) = 4$ .

### 4.2 2c

This expression cannot be made to return 42. Let's examine why. We had:

```
((car gregage) (gregage "car"))
```

Looking at the first part, `(car gregage)`, we see that `gregage` must be a list, because `car` must take a list as an argument. However, when we look at the second part, `(gregage "car")`, we see that `gregage` must be a function, because it is being applied to the argument `"car"`. We can't have something be a list and a function simultaneously, so this won't ever work.

### 4.3 2d

This was a little tricky. First we know that the last argument of `foldr` must be a list, so `(gregage 5)` must evaluate to some list. Also, in the lambda function, we see that `(gregage 2)` or `(gregage 3)` must evaluate to some number, because each might be added to `x`. This should be a clue that we need some sort of lambda function that evaluates to a list if given 5 as an argument, but a number if given 2 or 3 as an argument. Finally, `(gregage 4)` should similarly return a number, because as the base case it will be applied as an argument to the lambda function given to `foldr` too.

You could make a more convoluted solution, but a simple one was: (

```
(lambda (x) (if (= x 5) (list 42) 0))
```

When we think about how this is applied we see that when we replace the `gregages`, the `foldr` becomes (

```
(foldr (lambda (x r) (+ x (if (> r 0) 0 0))) 0 (cons 42 '()))
```

Which is the same as:

```
((foldr (lambda (x r) (+ x 0)) 0 (cons 42 '()))
```

Which becomes:

```
((lambda (x r) (+ x 0)) 42 0)
```

Which evaluates to 42.

### 4.4 3a

You could have used a structural induction on the length of the list or induction on the natural numbers here. The proof is well described in the exam solutions.

#### 4.5 3b

Each time mycopy is recursively called, it finds the length of the list, which is clearly an  $O(n)$  operation, thus this should appear in your recurrence relation.

#### 4.6 3c

This is  $O(n^2)$  because mycopy is called recursively once for each element of the list, so mycopy is called  $n$  times. Each time mycopy is called, mylength runs in  $O(n)$  time, because it must traverse the list to determine its length. So the running time is  $n * n = n^2$

#### 4.7 4a

This problem was relatively straightforward. Some of you tripped up because you didn't use struct specific functions.

#### 4.8 4b

The same thing applies here. You can't use car and cdr to get the elements of a pair, you have to use the automatic methods pair-x and pair-y.

#### 4.9 4c

First, think about types to get (lambda (x y) (lambda (z) ...)). Then, think about what type each of the arguments is. How are you using them? Then think about small cases.

#### 4.10 5a

The key here was that Dijkstra's algorithm examines all of the nodes touching the current node, and then picks the node with the lowest distance to examine next. The correctness of the algorithm rests on the assumption that once a node has a distance that distance can only increase for unvisited nodes connected to it. When negative edges are introduced, we are now allowing distances to decrease also and so when Dijkstra's algorithm picks the "shortest" path, it might have stopped too early. This is what happens in the example provided in the solution set.

Some of you didn't realize that shortest path means the path with the smallest length, not the number of edges you have to traverse. Thus if I can get from point A to point B by traversing a single edge with weight 100, or 99 edges with weight 1, the shortest path is to traverse the 99 edges, because the sum of the weights is  $99 < 100$ . For these types of questions, usually the simpler the example you can construct, the better. You could construct an example to show the algorithm fails using 1000 nodes, but an example with only 3 nodes is much easier to understand.

Note: when drawing a graph you had to give all of the edges weights, and the nodes unique names.

#### 4.11 5b

In order to prove that a graph algorithm works or fails, you must either show that the algorithm works correctly for any graph or, to disprove, find an example of a single graph where the algorithm fails to produce the correct answer. In this question, there were many simple examples to show that this algorithm fails on certain graphs .