CS135 Finals Notes

Structures

Templates:

```
(define struct point (x y))
;; A Point is a (make-point Num Num)

(define-struct rect (topleft w h))
;; A Rect is a (make-rect Point Num Num)
;; Requires: w, h >= 0

(define (point-template p)
  (... (point-x p)
       (point-y p)))

(define (rect-template-v1 r)
  (... (point-template (rect-topleft r))
       (rect-w r)
       (rect--h r)))

(define (rect-template-v2 r)
  (... (point-x (rect))
       (point-y (rect-topleft r))
       (rect-w r)
       (rect-h r)))
```

General idea

Basically the function should perform some operator on all the elements in a struct

Lists

Templates:

```
;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) (... (first lox)
                          (listof-X-template (rest lox)))]))
```

Sometimes, each `X` in `(listof X)` requires further processing. Indicate this as following:

```
;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) (... (X-template (first lox))
                          (listof-X-template (rest lox)))]))
```

General idea

Check for list being empty -> Some code shall be run

If list is not empty -> Operate on the first element using `X-template` and apply the list template on the rest of the list.

Natural Numbers

Formal Definition

```
;; A Nat is one of
;; * 0
;; * (add1 Nat)
```

Templates:

```
;; nat-template: Nat -> Any
(define (nat-template n)
  (cond [(zero? n) ...]
        [else (... n
                   (nat-template (sub1 n)))]))
```

More Lists

MergeSort

```
;; (mergesort lon) puts lon in increasing order
;; mergesort: (listof Num) -> (listof Num)
(define (mergesort lst)
  (cond [(empty? lst) empty]
        [(empyt? (rest lst)) lst]
        [else (merge (mergesort (keep-alternates lst))
                     (mergesort (drop-alternates lst)))]))

;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lon1 and lon2 are already in ascending order.
(define (merge lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) empty]
        [(and (empty? lon1) (cons? lon2)) lon2]
        [(and (cons? lon1) (empty? lon2)) lon1]
        [(and (cons? lon1) (cons? lon2))
         (cond [(< (first lon1) (first lon2))
                (cons (first lon1) (merge (rest lon1) lon2))]
               [else (cons (first lon2) (merge lon1 (rest lon2)))])]))
```

Here are the: keep-alternates and drop-alternates function definitions

Patterns of Recursion

max-list example

```
;; (max-list v1 lon)
;; Requires: lon is nonempty
(define (max-list-v1 lon)
  (cond [(empty? (rest lon)) (first lon)]
        [else (max (first lon) (max-list-v1 (rest lon)))]))
```

Inefficient way of "In-lining" max due to exponential blowup:

```
(define (max-list-v2 lon)
  (cond [(empty? (rest lon)) (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else (max-list-v2 (rest lon))]))
```

An accumulative and intuitive way:

```
(define (max-list-v3 lon)
  (max-list/acc (rest lon) (first lon)))

;; (max-list/acc lon max-so-far) produces the largest
;;     of the maximum element of lon and max-so-far

;; max-list/acc: (listof Num) Num -> Num
(define (max-list/acc lon max-so-far)
  (cond [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
         (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]))
```

Binary Trees

Data definition

```
(define-struct node (key left right))
;; A Node is a (make-node Nat BT BT)

;; A Binary Tree (BT) is one of;
;; * empty
;; * Node
```

Template

```
(define (bt-template bt)
  (cond [(empty? t) ...]
        [(node? t) (... (node-key t)
                        (bt-template (node-left t))
                        (bt-template (node-right t)))]))
```

Binary Search Tree (BST)

Data Definition

```
;; A Binary Search Tree (BST) is one of:
;; * empty
;; * a Node

(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; Requires: key > every key in left BST
;;           key < every key in right BST
```

Template

Pretty much the same as Binary Tree above

search-bt-path:

```
;; search-bt-path: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path k bt)
  (cond
    [(empty? bt) false]
    [(= k (node-key bt)) empty]
    [else
     (local [(define left-path (search-bt-path k (node-left bt)))]
       (cond [(list? left-path) (cons 'left left-path)]
             [else (local [(define right-path
                             (search-bt-path k (node-right bt)))]
                     (cond [(list? right-path) (cons 'right right-path)]
                           [else false]))]))]))
```

Exercises

- `(search-bst n t) produces true if n is in t; false otherwise.`
- `(bst-add n t) produces a transformed tree, t, with n in it.`
- `(bst-from-list lon) builds a BST from a list of keys`
- `(bst-min t) produces the minimum value in the non-empty tree`
- `(bst-max t) produces the maximum value in the non-empty tree'
- `(bst-from-lst/acc lon) produces a bst from lon with accumulative recursion`

Data Definition

```
(define-struct binode (op left right))
;; A Binary arithmetic expression Internal Node (BINode)
;;     is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)

;; A binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode
```

Template

```
;; binode-template: BINode -> Any
(define (binode-template node)
  (... (binode-op node)
       (binexp-template (binode-left node))
       (binexp-template (binode-right node))))

;; binexp-template: BinExp -> Any
(define (binexp-template ex)
  (cond [(number? ex) (... ex)]
        [(binode? ex) (binode-template ex)]))
```

Evaluating Expressions

```
;; (eval ex) evaluates the expression ex and produces its value

;; eval: BinExp -> Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode (binode-op ex)
                                   (eval (binode-leftr ex))
                                   (eval (binode-right ex)))]))

;; (eval-binnode node) evaluates the expression represented by node.
;; eval-binode: BINode -> Num
(define (eval-binode op left-val right-val)
  (cond [(symbol=? '* op) (* left-val right-val)]
        [(symbol=? '/ op) (/ left-val right-val)]
        [(symbol=? '+ op) (+ left-val right-val)]
        [(symbol=? '- op) (- left-val right-val)]))
```

Mutual Recursion

Example Functions

```
(define (keep-alternates lst)
  (cond [(empty? lst) empty]
        [else (cons (first lst) (drop-alternates (rest lst)))]))

(define (drop-alternates lst)
  (cond [(empty? lst) empty]
        [else (keep-alternates (rest lst))]))
```

(General Trees) Arithmetic Expression Trees

Data Definition

```
An Arithmetic Expression (AExp) is one of:
;; * Num
;; * OpNode

(define-struct opnode (op args))
;; An OpNode (operator node) is a
;; (make-opnode (anyof '* '+) (listof AExp))
```

Evaluating Expression

```
;; eval: AExp -> Num
(define (eval exp)
  (cond [(number? exp) exp]
        [(opnode? exp) (apply (opnode-op exp)
                              (opnode-args exp))]))

;; apply: (anyof '+ '*) (listof AExp) -> Num
(define (apply op args)
  (cond [(empty? args) (cond [(symbol=? op '+) 0]
                             [(symbol=? op '*) 1])]
        [(symbol=? op '+) (+ (eval (first args))
                             (apply op (rest args)))]
        [(symbol=? op '*) (+ (eval (first args))
                             (apply op (rest args)))]))
```

Exercises

```
(define-struct gnode (key children))
;; A GT (Generalized Tree) is a (make-gnode Nat (listof GT))
```

- Write a template
- `(reverse-gt t) consumes a generalized tree and produces it in reverse`
- `(most-populated-level t) produces the (n l) where n is the level with the most number of nodes, and l is the amount of nodes`

Local Definitions

Reasons to use Local

- Clarity: Naming subexpressions
- Efficiency: Avoid re-computation
- Encapsulation: Hiding stuff
- Scope: Reusing parameters

Note

Local functions require a design recipie

Insertion sort

```
;; Full Design Recipe for isort goes here...
(define (isort lon)
  (local [;; (insert n slon) inserts n into slon, preserving the order
          ;; insert: Num (listof Num) -> (listof Num)
          ;; Requires: slon is sorted in nondecreasing order
          (define (insert n slon)
            (cond [(empty? slon) (cons n empty)]
                  [(<= n (first slon)) (cons n slon)]
                  [else (cons (first slon) (insert n (rest slon)))]))]
    (cond [(empty? lon) empty]
          [else (insert (first lon) (isort (rest lon)))])))
```

Functions as Values

`filter`

```
;; filter: (X -> Bool) (listof X) -> (listof X)
(filter pred? lst)
```

Functions as first class values

We can do anything with a function that we can do with other values.

- **Consume**: Functions should be consumable
- **Produce**: Functions should be produce-able
- **Bind**: Functions should be bindable with other values
- **Store**: It should be able for functions to be stored

Functional Abstraction

`map`

```
;; my-map: (X -> Y) (listof X) -> (listof Y)
(my-map f lst)

;; Note: the built-in map function can take in n number of lists,
;;       and f should have n parameters
```

`foldr`

`foldr` consumes three arguments:

- a function which combines the first list item with the result of reducing on the rest of the list.
- a base value.
- a list on which to operate.

```
;; my-foldr: (X Y -> Y) Y (listof X) -> Y
(my-foldr combine base lst)

;; Note: Like map, the built-in foldr can take n number of lists,
;;       and combine should have n of X parameters
```

`foldl`

`foldl` consumes three arguments:

- a function that computes the new value of the accumulator, given the first list item and the old value of the accumulator.
- the initial value of the accumulator.
- a list on which to operate.

```
;; my-foldl: (X Y -> Y) Y (listof X) -> Y
(my-foldl combine base lst)

;; Note: the built-in foldr can take n number of lists,
;;       and combine should have n of X parameters
```

Personal understanding

`foldr` folds the list from right to left
`foldl` folds the list from left to right

Example

Negating a list:

```
(define (negate-list lst)
  (foldr (labmda (x rror) (cons (- x) rror)) empty lst))
```

Note

When using a lambda function in foldr/foldl, `rror` is used as a parameter to mean "Result of Recursing On the Rest of the list"

`build-list`

```
;; my-build-list: Nat (Nat -> X) -> (listof X)
(define my-build-list n f)

;; Note: There exists a built in build-list function
```

Generative Recursion

GCD

```
;; (euclid-gcd n m) computes gcd(n, m) using Euclidean algorithm
;; euclid-gcd: Nat Nat -> Nat

(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

QuickSort

```
;; (my-quicksort lon) sorts lon in non-decreasing order
;; my-quicksort: (listof Num) -> (listof Num)

(define (my-quicksort lon)
  (cond [(empty? lon) empty]
        [else (local [(define pivot (first lon))
                      (define less (filter (lambda (x) (< x pivot))
                                           (rest lon)))
                      (define greater (filter (lambda (x) (>= x pivot))
                                              (rest lon)))]
                (append (my-quicksort less)
                        (list pivot)
                        (my-quicksort greater)))]))

;; There also exists a built-in quicksort
;; (quicksort lst comparison_operator)
;; Example:
(check-expect (quicksort (list 5 4 3 2 1) <) (list 1 2 3 4 5))
(check-expect (quicksort (list 1 2 3 4 5) >) (list 5 4 3 2 1))
```

Graphs

Terminology

- **Directed** and **undirected** graphs
- Directed graphs have **in** and **out** neighbors
- **Cyclic** and **acyclic** graphs
- **DAGs** (directed acyclic graph)
- A sequence of nodes is a **path** or **route**
- Graphs consist of **nodes** (aka **vertices**) and **edges**

Graphs will be represented by an adjacency list in this course

Data Definition

```
;; A Node is a Sym

;; A Graph is one of;
;; * empty
;; * (cons (list v (list w_1 ... w_n)) g)
;;   where g is a Graph
;;         v, w_1, ..., w_n are Nodes
;;         w_1, ..., w_n are out-neighbours of v in the Graph
;;         v does not appear as an in-neighbour in g
```

Template

```
;; graph-template: Graph -> Any
(define (graph-template g)
  (cond
    [(empty? g) ...]
    [(cons? g)
     (... (first (first g))            ;; first node in graph list
          (listof-node-template
              (second (first g)))      ;; list of adjacent nodes
          (graph-template (rest g)))]))
```

Neighbours

```
;; (neighbours v g) produces list of neighbours of v in g
;; neighbours: Node Graph -> (anyof (listof Node) false)
;; Requires: v is a node in g
(define (neighbours v g)
  (cond
   [(empty? g) false]
   [(symbol=? v (first (first g))) (second (first g))]
   [else (neighbours v (rest g))]))
```

Find path

A shitty approach (can't handle cycles and slow):

```
;; (find-path orig dest g) finds path from orig to dest in g if it exists
;; find-path: Node Node Graph -> (anyof (ne-listof Node) false)
(define (find-path orig dest g)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig g))
                      (define ?path (find-path/list nbrs dest g))]
                (cond [(false? ?path) false]
                      [else (cons orig ?path)]))]))


;; (find-path/list nbrs dest g) produces path from
;;    an element of nbrs to dest in g, if one exists
;; find-path/list: (listof Node) Node Graph -> (anyof (ne-listof Node) false)
(define (find-path/list nbrs dest g)
  (cond [(empty? nbrs) false]
        [else (local [(define ?path (find-path (first nbrs) dest g))]
                (cond [(false? ?path)
                       (find-path/list (rest nbrs) dest g)]
                      [else ?path]))]))
```

A turtle approach (slow):

```
;; find-path/list: (listof Node) Node Graph (listof Node) ->
;;                 (anyof (listof Node) false)
(define (find-path/list nbrs dest g visited)
  (cond [(empty? nbrs) false]
        [(member? (first nbrs) visited)
         (find-path/list (rest nbrs) dest g visited)]
        [else (local [(define ?path (find-path/acc (first nbrs)
                                                    dest g visited))]
                (cond [(false? ?path)
                       (find-path/list (rest nbrs) dest g visited)]
                      [else ?path]))]))

;; find-path/acc: Node Node Graph (listof Node) -> (anyof (listof Node) false)
(define (find-path/acc orig dest g visited)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig g))
                      (define ?path (find-path/list nbrs dest g
                                                          (cons  orig visited)))]
                (cond [(false? ?path) false]
                      [else (cons orig ?path)]))]))

(define (find-path orig dest g)          ;; new wrapper function
  (find-path/acc orig dest g empty))
```

The (somehow) sane approach:

```
;; find-path/list: (listof Node) Node Graph (listof Node) -> Result
(define (find-path/list nbrs dest g visited)
  (cond [(empty? nbrs) (make-failure visited)]
        [(member? (first nbrs) visited)
         (find-path/list (rest nbrs) dest g visited)]
        [else (local [(define result (find-path/acc (first nbrs)
                                                     dest g visited))]
                (cond [(failure? result)
                       (find-path/list (rest nbrs) dest g
                                       (failure-visited result))]
                      [(success? result) result]))]))


;; find-path/acc: Node Node Graph (listof Node) -> Result
(define (find-path/acc orig dest g visited)
  (cond [(symbol=? orig dest) (make-success (list dest))]
        [else (local [(define nbrs (neighbours orig g))
                      (define result (find-path/list nbrs dest g
                                                     (cons orig visited)))]
                (cond [(failure? result) result]
                      [(success? result)
                         (make-success (cons orig
                                             (success-path result)))]))]))


(define (find-path orig dest g)
  (local [(define result (find-path/acc orig dest g empty))]
      (cond [(success? result) (success-path result)]
            [(failure? result) false])))
```

Personal understanding

The differences are:

- shitty version doesn't do shit.
- turtle version passes down a visited list.
- sane version also passes up the failed visited list (to avoid recalculating a failed path).

General

Tips for building templates

Follow the data definition. For each part of the data definition, if it

- says "one of", include a `cond` to distinguish the cases
- is a simple type `(Int, Str, Sym, empty, etc)`, do nothing.
- is a defined data type, apply that type's template.
- is compound data (a structure, fixed-length list), extract each of the fields, in order.
- is a `listof`, extract the first and rest of the list.
  Add ellipses `(...)` around each of the above.
  Apply the above recursively

Note that:

- `(list? empty)` produces `true`
- `(cons? empty)` produces `false`

Watch out for:

- Misusing length -> length is O(n)
- Exponential blowup like max-list-v2

Built-in Functions

```
*
+
-
...
/
<
<=
=
>
>=
abs
add1
and
append
boolean?
build-list
ceiling
char-alphabetic?
char-downcase
char-lower-case?
char-numeric?
char-upcase
char-upper-case?
char-whitespace?
char<=?
char<?
char=?
char>=?
char>?
char?
check-error
check-expect
check-within
cond
cons
cons?
cos
define
define-struct
define/trace
e
eighth
else
```

```
empty?
equal?
error
even?
exp
expt
false?
fifth
filter
first
floor
foldl
foldr
fourth
integer?
lambda
length
list
list->string
list?
local
log
map
max
member?
min
modulo
negative?
not
number->string
number?
odd?
or
pi
positive?
quicksort
quotient
remainder
rest
reverse
round
second
seventh
sgn
sin
sixth
```

```
sqr
sqrt
string->list
string-append
string-downcase
string-length
string-lower-case?
string-numeric?
string-upcase
string-upper-case?
string<=?
string<?
string=?
string>=?
string>?
string?
sub1
substring
symbol=?
symbol?
tan
third
zero?
```