Programming Abstractions

Exam 1 Review

Stephen Checkoway

Exam Format

Take home exam

4 implementation problems ("Write a procedure to do *x*")

Write all of your solutions in DrRacket

Turn in your completed exam by pushing to GitHub

Exam will be released at midnight on Monday (you'll receive an email from Ed)

Your solutions are due by 23:59 on Monday (you have 24 hours)

Class time

During Monday's class, I will be in my office, feel free to stop by to ask any questions about the exam

So no normal lecture on Monday

Possible question topics

Basic Scheme/Racket functions and special forms

- `cons`, `first` (`car`), `rest` (`cdr`), `list`, `append`, `member`, `empty?`, `filter`, etc.
- `define`, `lambda`, `if`, `cond`, `let`, `letrec`, `and`, `or`, etc.

`map` and `apply`

`foldl` and `foldr` and how they differ

Recursion

- Tail recursion
- "Accumulator passing style"

Closures: how to create and use them

Given a list `lst` and an element `x`, how can we create a new list that consists of `x` prepended to `lst`? E.g., if `lst` is `'(1 2 3)` and `x` is 4, we want '(4 1 2 3)

A. `(prepend x lst)`

B. `(cons x lst)`

C. `(append x lst)`

D. It's not possible to modify `lst`

E. None of the above

Given a list `lst` and an element `x`, how can we create a new list that consists of `x` appended to `lst`? E.g., if `lst` is `'(1 2 3)` and `x` is 4, we want '(1 2 3 4)

A. `(cons lst x)`

B. `(append lst x)`

C. `(append lst '(x))`

D. `(append lst (list x))`

E. None of the above

If you were implementing the reverse function which takes a list as input and returns a list with the elements in reverse order, which implementation should you choose (and why)?

```
(define (reverse-1 lst)
  (foldr (λ (x acc) (append acc (list x)))
         empty
         lst))

(define (reverse-2 lst)
  (foldl (λ (x acc) (cons x acc))
         empty
         lst))
```

A. reverse-1

B. reverse-2

C. Either one is acceptable

Given a list of lists, lsts, how do you get a list containing the second element of each list, in order?

A. `(map second lsts)`

B. `(map rest lsts)`

C. `(apply second lsts)`

D. `(apply rest lsts)`

E. None of the above

Drop

Write a procedure (drop lst n) that takes a list and an integer and returns a list consisting of the elements of lst except for the first n elements

```
(drop '(1 2 3) 0) => '(1 2 3)

(drop '(1 2 3) 2) => '(3)

(drop '(1 2 3) 4) => (error 'drop "list too short")
```

Select

Represent a student as a three-element list (name year gpa), e.g., `'("Jane" 2 3.5)` represents Jane who is a second-year and has a 3.5 GPA

Write a procedure `(select lst)` that takes a list of students and returns the name of all second or third year students with a GPA that's at least `3.0`

Enumerate

Write a procedure `(enumerate lst)` that takes a list and returns a list of 2-element lists `(index elem)` where `elem` is in `lst` and `index` is its index, in order.

Implement `enumerate` by using `letrec` to implement a recursive procedure that performs the real work

E.g., `(enumerate '(a b c))` returns `'((0 a) (1 b) (2 c))`

Tail-recursive enumerate

Write a **tail-recursive** procedure `(enumerate2 lst)` that takes a list and returns a list of 2-element lists `(index elem)` where `elem` is in `lst` and `index` is its index, in order.

Use a `letrec` to implement the actual tail recursion

E.g., `(enumerate2 '(a b c))` returns `'((0 a) (1 b) (2 c))`

Flip

Write a procedure `(flip f)` that that takes a 2-argument procedure `f` and returns a 2-argument closure that, when called, calls `f` with its arguments in the opposite order. I.e., `((flip f) x y)` is the same as `(f y x)`

Write `(flip* f)` that takes any procedure f and returns a closure that, when called, calls f with all of its arguments reversed. E.g.,

- `((flip* f))` is `(f)`;
- `((flip* g) x)` is `(g x)`;
- `((flip* h) x y)` is `(h y x)`;
- `((flip* i) x y z)` is `(i z y x)`; and so forth

E.g., `((flip* -) 1 2 3 4 5)` returns `-5`

Reverse a structured (non-flat) list

Write a procedure `(reverse-all lst)` that takes a non-flat list and reverse it, including all contained lists

E.g., `(reverse-all '(1 () (2 3 (4 5)) 6))` returns
`'(6 ((5 4) 3 2) () 1)`