Potential Exam 2 Topics

Nondeterminism \texttt{(random)}

Local variables \texttt{(let)}

Local procedures \texttt{(letrec)}

\texttt{†} Sorting / efficiency of algorithms

Procedures as parameters

Built-in higher-order functions \texttt{(apply, map, ormap, andmap, filter)}

Returning procedures

\texttt{†} Abstracting functions / design patterns

\texttt{†} Data structures

\begin{verbatim}
;; RANDOM (built-in)
;; -------
;; INPUTS: N, a positive integer
;; OUTPUT: A pseudo-randomly chosen integer from the set
;; \{0, 1, 2, ..., N-1\}

> (random 6)
0

> (random 6)
4

> (random 6)
1
\end{verbatim}
So, if we want to flip a coin?

;; FLIP-COIN
;; --------
;; INPUTS: None
;; OUTPUT: 'heads or 'tails
(define flip-coin
  (lambda ()
    (if (= (random 2) 0)
      'heads
      'tails))))
> (flip-coin)
heads
> (flip-coin)
tails

Toss a die?

;; TOSS-DIE
;; --------
;; INPUTS: None
;; OUTPUT: Random number in {1, 2, 3, 4, 5, 6}
(define toss-die
  (lambda ()
    ;; Note: (random 6) gives you one of six choices.
    ;; We add 1 to the result to get a value in
    ;; {1, 2, 3, 4, 5, 6} instead of something in
    ;; {0, 1, 2, 3, 4, 5}.
    (+ 1 (random 6))))
> (toss-die)
6
> (toss-die)
4

Toss a D20 (nerds)?

;; TOSS-D20
;; --------
;; INPUTS: None
;; OUTPUT: Random number in {1, 2, ..., 20}
(define toss-d20
  (lambda ()
    (+ 1 (random 20))))
> (toss-d20)
16
> (toss-d20)
2

Potential Exam 2 Topics

Nondeterminism (random)
Local variables (let)
Local procedures (letrec)
† Sorting / efficiency of algorithms
Procedures as parameters
Built-in higher-order functions (apply, map, ormap, andmap, filter)
Returning procedures
† Abstracting functions / design patterns
† Data structures
The let special form

The following let expression creates a local variable called toss whose value is determined by a single toss of a six-sided die. The expressions in the body of the let refer to toss several times.

(let ((toss (toss-die)))
  (printf "the toss: \~A\~%n toss)
  (printf "the toss squared: \~A\~%n (* toss toss))
  toss)

Dependencies among let values

We can nest let expressions when one value depends on another:

(let ((x 3))
  (let ((y (+ x x)))
   (* y)))

This lets us split a computation into parts with their own names.

How many seconds are in n years?

;;; NUM-SECONDS-IN-N-YEARS
;;; ----------------------
;;; INPUTS: N, a non-negative integer
;;; OUTPUT: the number of seconds in N years
(define num-seconds-in-n-years
  (lambda (n)
    ;; NUM-DAYS = the number of days in N years
    (let ((num-days (* 365 n)))
      ;; NUM-HOURS = the number of hours in N years
      (let ((num-hours (* 24 num-days)))
        ;; NUM-MINUTES = the number of minutes in N years
        (let ((num-minutes (* 60 num-hours)))
          ;; Finally, the number of seconds in N years
          (* 60 num-minutes))))))

We can make things much simpler by using the let* special form, which allows us to define a variable using the previous ones.

;;; NUM-SECONDS-IN-N-YEARS
;;; ----------------------
;;; INPUTS: N, a non-negative integer
;;; OUTPUT: the number of seconds in N years
(define num-seconds-in-n-years
  (lambda (n)
    (let* ((num-days (* 365 n))
      (num-hours (* 24 num-days))
      (num-minutes (* 60 num-hours)))
      (* 60 num-minutes))))
Potential Exam 2 Topics

Nondeterminism (random)
Local variables (let)
Local procedures (letrec)
† Sorting / efficiency of algorithms
Procedures as parameters
Built-in higher-order functions (apply, map, ormap, andmap, filter)
Returning procedures
† Abstracting functions / design patterns
† Data structures

We can use let or let* to define local functions, but they won’t work if that function is recursive.

For that, we use letrec, most often to define a helper function inside a wrapper.

Example: Count the numbers on a list between two bounds

(define count-between
  (lambda (lst low high) ...
  ...
  )))
> (count-between '(5 1 2 3 4) 2 4)
  3
> (count-between '(1 2 3 4 5) 7 9)
  0
> (count-between '() 2 4)
  0
Potential Exam 2 Topics

- Nondeterminism (random)
- Local variables (let)
- Local procedures (letrec)
- Sorting / efficiency of algorithms

Procedures as parameters
- Built-in higher-order functions (apply, map, ormap, andmap, filter)
- Returning procedures
- Abstracting functions / design patterns
- Data structures

Example: Do at least $n$ elements of a list satisfy a predicate?

```scheme
(define at-least-n?
  (lambda (lst n p?)
    (cond
      ((<= n 0) #t)
      ((null? lst) #f)
      ((p? (first lst))
       (at-least-n? (rest lst) (- n 1) p?))
      (else
       (at-least-n? (rest lst) n p?))))))
```

```
> (at-least-n? '(1 2 3 4 5) 3 even?)
#f
> (at-least-n? '(1 2 3 4 5) 3 odd?)
#t
> (at-least-n? '(1 2 3 4 5) 0 pair?)
#t
```
Example: Are at least $m$ elements of a list greater than $n$?

```
(define at-least-n-above-m?
  (lambda (lst n m) ...?...))
```

```
> (at-least-n-above-m? '(1 2 3 4 5) 3 2)
#t
> (at-least-n-above-m? '(1 2 3 4 5) 2 4)
#f
> (at-least-n-above-m? '(1 2 3 4 5) 0 6)
#t
```

Potential Exam 2 Topics

- Nondeterminism (random)
- Local variables (let)
- Local procedures (letrec)
- ∗ Sorting / efficiency of algorithms
- Procedures as parameters

**Built-in higher-order functions** (apply, map, ormap, andmap, filter)
- Returning procedures
- ∗ Abstracting functions / design patterns
- ∗ Data structures

Example: Add up the absolute values of numbers on a list

```
(define sum-absolute-values
  (lambda (lst) ...?...))
```

```
> (sum-absolute-values '(-5 1 2 -3 4))
15
> (sum-absolute-values '(5 1 2 3 4))
15
> (sum-absolute-values '())
0
```
(define sum-absolute-values
  (lambda (lst)
    (apply + (map abs lst))))

(define abs
  (lambda (x)
    (if (< x 0)
        (- x)
        x)))

Example: Select the numbers on a list within low and high bounds

(define select-within?
  (lambda (lst low high)
    ...?...))

> (select-within '(1 2 3 4 5) 2 4)
'(2 3 4)

> (select-within '(1 2 3 4 5) 4 4)
'(4)

> (select-within '(1 2 3 4 5) 6 8)
'()
Make a predicate that checks if at least one of a list of predicates is true

\[
(\text{define} \textit{\textbf{satisfies-one-predicate}} \\
(\lambda (\textit{plist}) \\
\ldots?\ldots))
\]

\[
> (\text{define} \textit{\textbf{pe?}} \\
(\textit{satisfies-one-predicate} \\
(\textit{list} \text{positive?} \text{even?})))
\]

\[
> (\textit{pe?} -4) \\
\#t ;; \text{It's even (but not positive)}
\]

\[
> (\textit{pe?} 3) \\
\#t ;; \text{It's positive (but not even)}
\]

\[
> (\textit{pe?} -7) \\
\#f ;; \text{It's neither}
\]

Recall that we wrote a higher-order function \textit{\textbf{there-exists?}}, which is equivalent to the built-in function \textit{\textbf{ormap}}.

\[
(\text{define} \textit{\textbf{satisfies-one-predicate}} \\
(\lambda (\textit{plist}) \\
\ldots?\ldots))
\]

\[
> (\text{define} \textit{\textbf{pe?}} \\
(\text{there-exists?} \\
(\text{list} \text{positive?} \text{even?})))
\]

\[
> (\textit{pe?} -4) \\
\#t ;; \text{It's even (but not positive)}
\]

\[
> (\textit{pe?} 3) \\
\#t ;; \text{It's positive (but not even)}
\]

\[
> (\textit{pe?} -7) \\
\#f ;; \text{It's neither}
\]

If we wanted to write it without using a higher-order function, we could do so:

\[
(\text{define} \textit{\textbf{satisfies-one-predicate}} \\
(\lambda (\textit{plist}) \\
\ldots?\ldots))
\]

\[
> (\text{define} \textit{\textbf{pe?}} \\
(\text{there-exists?} \\
(\text{list} \text{positive?} \text{even?})))
\]

\[
> (\textit{pe?} -4) \\
\#t ;; \text{It's even (but not positive)}
\]

\[
> (\textit{pe?} 3) \\
\#t ;; \text{It's positive (but not even)}
\]

\[
> (\textit{pe?} -7) \\
\#f ;; \text{It's neither}
\]
Acknowledgments

This lecture incorporates material from:

Tom Ellman
Luke Hunsberger