Function “Arithmetic” and the Lambda Calculus
Logistics

• HW 4 out
  • due: Sun 10/22 11:59 pm EST
Common List Function #1: map

;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Produces a list resulting from applying
;; a given fn to each element of a given lst

(define (map fn lst)
  (cond
   [(empty? lst) empty]
   [else (cons (fn (first lst))
               (map (rest lst)))]))
map in other high-level languages

Array.prototype.map()

The `map()` method of `Array` instances creates a new array populated with the results of calling a provided function on every element in the calling array.

JavaScript Demo: Array.map()

```javascript
const array1 = [1, 4, 9, 16];
// Pass a function to map
const map1 = array1.map(x => x * 2);
// Expected output: Array [2, 8, 18, 32]
console.log(map1);
```

Python3

```
# Add two lists using map and lambda
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```
Common List Function #2: foldl / foldr

;; foldr: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, last-element-first

(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))

;; foldl: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, first-element-first

(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))])))
fold (reduce) in other high-level languages

JavaScript Demo: Array.reduce()

```javascript
const array1 = [1, 2, 3, 4];
// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce((resultSoFar, x) => resultSoFar + x, initial);
console.log(sumWithInitial);
// Expected output: 10
```

JavaScript Demo: Array.reduceRight()

```javascript
const array1 = [
  [0, 1],
  [2, 3],
  [4, 5],
];
const result = array1.reduceRight((resultSoFar, x) => resultSoFar.concat(x));
console.log(result);
// Expected output: Array [4, 5, 2, 3, 0, 1]
```
Fold “dual”: build-list

(build-list n proc) → list?
  n : exact-nonnegative-integer?
  proc : (exact-nonnegative-integer? . -> . any)

Creates a list of n elements by applying proc to the integers from 0 to (sub1 n) in order. If lst is the resulting list, then (list-ref lst i) is the value produced by (proc i).

Examples:

> (build-list 10 values)
'(0 1 2 3 4 5 6 7 8 9)
> (build-list 5 (lambda (x) (* x x))
'(0 1 4 9 16)

(build-list 4 add1)
;; = (map add1 (list 0 1 2 3))
;; = (list 1 2 3 4)
Fold “alternative”: \texttt{apply} (and variable-arity fns)

\[
\text{foldl} + 0 \ (\text{list} \ 1 \ 2 \ 3 \ 4)) = (+ (+ (+ (+ 1 0) 2) 3) 4)) = 10
\]

\[
\text{apply} + \ (\text{list} \ 1 \ 2 \ 3 \ 4)) = (+ 1 \ 2 \ 3 \ 4) = 10
\]

\[
\text{apply} \ \text{string-append} \ (\text{list} \ “a” \ “b” \ “cd”)) = “abcd”
\]

- \texttt{apply} applies its function arg to the contents of its \texttt{list} arg
- function arg to apply must accept: 
  \# of arguments = \texttt{length} of \texttt{list} arg
Common list function #3: filter

;;; filter: Listof<X> (X -> Boolean) -> Listof<X>
;;; Returns a list containing elements of given list
;;; for which the given predicate returns true

(define (filter lst pred?)
  (cond
    [(empty? lst) empty]
    [else (if (pred? (first lst))
              (cons (first lst) (filter (rest lst)))
              (filter (rest lst))))])
**filter** in other high-level languages

JavaScript Demo: Array.filter()

```javascript
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter((word) => word.length > 6);
console.log(result);
// Expected output: Array ['exuberant', 'destruction', 'present']
```
Common list function #3: \texttt{filter}

\begin{verbatim}
;; filter: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of given list for which the given predicate returns true

(define (filter lst pred?)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
           (cons (first lst) (filter (rest lst)))))
   ))
\end{verbatim}

\textbf{lambda rules:}
- Can skip the design recipe steps, BUT
- \textbf{name, description, and signature} must be “obvious”
- \textbf{code} is arithmetic only
- otherwise, create standalone function define

\begin{verbatim}
;; smaller-than: Listof<Int> Int -> Listof<Int>
;; Returns a list containing elements of given list less than the given int

(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst)
)
\end{verbatim}

\textbf{lambda creates an anonymous “inline” function (expression)}
Functions as Values

• In high-level languages, functions are no different from other values (e.g., numbers)
• They can be passed around, or be the result of a function

```
;; make< : Int -> (Int -> Bool)
;; makes a function that returns true
;; for values less than the given thresh value
(define (make< thresh)
  (lambda (x) (< x thresh)))
```

```
(define (smaller-than lst thresh)
  (filter (make< thresh)) lst)
```

• lambda is just one way to “make” functions
• We can also do “arithmetic” with functions
Currying

- A “curried” function is “partially” applied to some (but not all) args
- Result is another function

\[
\text{(curry } < 4) \]
\[
;; = \text{ a function that returns true when given a number less than 4}
\]

\[
\text{(define (smaller-than lst thresh)}
\]
\[
\text{ (filter (lambda (x) (< x thresh)) lst)}
\]

\[
\text{(define (smaller-than lst thresh)}
\]
\[
\text{ (filter (curry > thresh)) lst)}
\]
History Lesson: Haskell B. Curry

• Mathematician / Logician
• Born in Millis, MA, in year 1900

• “currying” functions is named after him
• and also, the “Haskell” (functional) programming language

• Invented “combinatory logic”, i.e., a system of function “arithmetic”
Currying

\[
\text{define (smaller-than lst thresh)} \\
\text{(filter (lambda (x) (< x thresh)) lst)}
\]

\[
\text{(define (smaller-than lst thresh)} \\
\text{(filter (curry > thresh)) lst)}
\]

\[
\text{(define (smaller-than lst thresh)} \\
\text{(filter (curry < thresh)) lst)}
\]

NOTE: First argument gets curried first
Composing Functions

• compose combines multiple functions into one function
  • last one is applied first

(\text{compose} \ \text{sqrt} \ \text{add1})
\text{;;} \ = \ a \ function \ that \ first \ applies \ \text{add1} \ to \ its \ argument, \ then \ \text{sqrt}

((\text{compose} \ \text{sqrt} \ \text{add1}) \ 8) \ \text{;;} \ = \ 3
Composing Functions

- compose combines multiple functions into one function
  - last one is applied first

```
apply above
(build-list 5
  (compose 4 (curryr square "solid" "blue")
    (curry * 20)
    add1))
)

; = (list 0 1 2 3 4)
; = (list 1 2 3 4 5)
; = (above (square 20 "solid" "blue")
  (square 40 "solid" "blue")
  (square 60 "solid" "blue")
  (square 80 "solid" "blue")
  (square 100 "solid" "blue"))

; = (list (square 20 "solid" "blue")
  (square 40 "solid" "blue")
  (square 60 "solid" "blue")
  (square 80 "solid" "blue")
  (square 100 "solid" "blue"))
```
The Lambda ($\lambda$) Calculus

• A “programming language” consisting of only:
  • Lambda
  • Function application

• Equivalent in “computational power” to
  • Turing Machines
  • Your favorite programming language!
History Lesson: Alonzo Church

- Mathematician, logician, computer scientist

- Invented the lambda calculus

- And (half of) Church-Turing Thesis
  - Any function that can be “computed” has an equivalent Turing Machine
  - And an equivalent program in the lambda calculus
  - So, a Turing Machine = a lambda
Church Numerals

;;;; A ChurchNum is a function with two arguments:
;;;; “fn” : a function to apply
;;;; “base” : a base ("zero") value to apply to
;;;; For a specific number, its "Church" representation
;;;; applies the given function that number of times

(define czero
  (lambda (f base) base))  Function applied zero times

(define cone
  (lambda (f base) (f base)))  Function applied one times

(define ctwo
  (lambda (f base) (f (f base))))  Function applied two times

(define cthree
  (lambda (f base) (f (f (f base)))))  Function applied three times
Church “Add1”

;; cplus1 : ChurchNum -> ChurchNum
;; “Adds” 1 to the given Church num

(define cplus1
  (lambda (n)
    (lambda (f base)
      (f (n f base))))))

(define czero
  (lambda (f base) base))

(define cone
  (lambda (f base) (f base)))

(define ctwo
  (lambda (f base) (f (f base))))

(define cthree
  (lambda (f base) (f (f (f base)))))
Church Addition

;;; cplus : ChurchNum ChurchNum -> ChurchNum
;;; “Adds” the given ChurchNums together

(define cplus
  (lambda (m n)
    (lambda (f base)
      (m f (n f base))))))

(define czero
  (lambda (f base) base))

(define cone
  (lambda (f base) (f base)))

(define ctwo
  (lambda (f base) (f (f base))))

(define cthree
  (lambda (f base) (f (f (f base)))))

(we know “n” will apply f n times)

Input ChurchNums

Returns a ChurchNum ...

... that adds “m” extra application of f
Code Demo 1
Church Booleans

;;; A ChurchBool is a function with two arguments, where the representation of: 
;;; “true” returns the first arg, and 
;;; “false” returns the second arg

(define ctrue 
  (lambda (a b) a))
Returns first arg
(define cfalse 
  (lambda (a b) b))
Returns second arg
Review: “And”

The truth table of $A \land B$:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \land B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

When $A = True$, then $\text{And}(A, B) = B$

When $A = False$, then $\text{And}(A, B) = A$
# Church “And”

The **truth table** of \( A \land B \):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( A \land B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

**Note:**
- When \( A = True \), want \( \text{And}(A, B) = B \)
- When \( A = False \), want \( \text{And}(A, B) = A \)

## ChurchBool: ChurchBool -> ChurchBool

`; cand: ChurchBool ChurchBool-> ChurchBool

; “ands” the given ChurchBools together

\[
(\text{define cand} \\
(\lambda (A B) \\
\quad (A B A)))
\]

\[
(\text{define ctrue} \\
(\lambda (a b) \\
\quad a))
\]

\[
(\text{define cfalse} \\
(\lambda (a b) \\
\quad b))
\]

; If \( A = \text{ctrue} \), want \( \text{cand A B} = B \)

; If \( A = \text{cfalse} \), want \( \text{cand A B} = A \)

(Returns first arg)

(Returns second arg)
Church “Or”

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \lor B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

When $A = \text{true}$, want $\text{or}(A, B) = A$

When $A = \text{false}$, want $\text{or}(A, B) = B$

;; cor: ChurchBool ChurchBool -> ChurchBool
;; “or” the given ChurchBools together

(define cor
  (lambda (A B)
    (A A B)))

(define ctrue
  (lambda (a b) a))

(define cfalse
  (lambda (a b) b))

;; if $A = \text{ctrue}$
;; then $(A A B) = A$
;; want $(\text{cor} A B) = A$

;; if $A = \text{cfalse}$
;; then $(A A B) = B$
;; want $(\text{cor} A B) = B$

(Retruns first arg)

(Retruns second arg)
Code Demo 2
Church Pairs (Lists)

;;; A ChurchPair<X,Y> 1-arg function, where
;;; the arg fn is applied to (i.e., "selects") the X and Y data values

;;; ccons: X Y -> ChurchPair<X,Y>
(define ccons
  (lambda (x y)
    (lambda (get)
      (get x y)))))

;;; Gets" the first item
(define cfirst
  (lambda (cc)
    (cc (lambda (x y) x)))))

;;; Gets" the second item
(define csecond
  (lambda (cc)
    (cc (lambda (x y) y)))))
Code Demo 3
The Lambda Calculus

• A “programming language” consisting of only:
  • Lambda
  • Function application

• “Language” has:
  • Numbers
  • Booleans and conditionals
  • Lists
  • ...
  • Recursion?
Check-In Quiz 10/18
on gradescope

(due 1 minute before midnight)