CSE 341
Lecture 17

Higher-order procedures; lists and pairs

slides created by Marty Stepp

http://www.cs.washington.edu/341/

Higher-order procedures

```
; apply procedure f to each element of lst
(map f lst)

; retain only elements where p returns #t
(filter p lst)

; reduce list; f takes 2 elements -> 1
(foldl f initialValue lst)
(foldr f initialValue lst)
```

- equivalent to ML's `map`/`List.filter`/`fold*`
- each takes a procedure (or "predicate") to apply to a list

Higher-order exercise

- Implement our own versions of `map` and `filter`, named `mapx` and `filterx`.
  - e.g. `(mapx f '(1 2 3 4 5))`
  - e.g. `(filterx p '(1 2 3 4 5))`

Higher-order solutions

```
; Applies procedure f to every element of lst.
(define (mapx f lst)
    (if (null? lst)
        ()
        (cons (f (car lst)) (mapx f (cdr lst)))))

; Uses predicate p to keep/exclude elements of lst.
(define (filterx p lst)
    (cond ((null? lst) ())
          ((p (car lst)) (cons (car lst)
                               (filterx p (cdr lst))))
          (else (filterx p (cdr lst)))))
```

Anonymous procedures ("lambdas")

```
(lambda (param1 ... paramN) expr)
```

- defines an anonymous local procedure
  - you can pass a lambda to a higher-order function, etc.
  - analogous to ML's: `fn(params) => expr`

  - Example (retain only the even elements of a list):

```
(filter (lambda (n) (= 0 (modulo n 2)))
        (range 0 100))
```

Lambda exercise

- Using higher-order procedures and lambdas, find the sum of the factors of 24.
  - Hint: First get all the factors in a list, then add them.

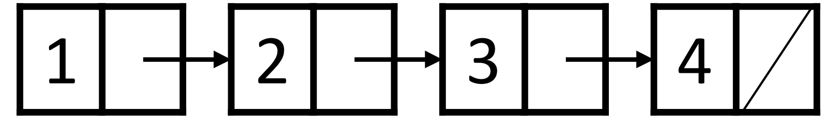
- Solution:

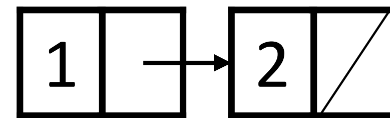
```
(foldl + 0
    (filter (lambda (n)
                (= 0 (modulo 24 n)))
            (range 1 24)))
```

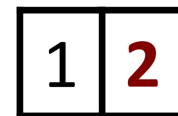
Improper lists (pairs)

Improper lists (pairs)

```
> (cons 1 '(2 3 4))
(1 2 3 4)
> (cons 1 '(2))
(1 2)
> (cons 1 2)
(1 . 2)
```







- if you cons two non-list values together, you get a *pair*
  - a list node whose `data` field stores the first value, and whose `next` field stores the second value

Working with improper lists

```
> (define p (cons 1 2))
> (car p)
1
> (cdr p)
2
> (cons p 3)
((1 . 2) . 3)
> (cons 3 p)
(3 1 . 2)
> (cons p '(3 4))
((1 . 2) 3 4)
> (cons p (cons 3 4))
((1 . 2) 3 . 4)
> (length p)
expects argument of type <proper list>
```

data	next
1	2

Why improper lists?

- a consequence of Scheme's relaxed dynamic typing
  - list nodes ("pairs") usually store a list as their "next"
  - but if the "next" is anything other than another pair or null, the list is improper

- an improper list is Scheme's closest analog to ML's **tuple**
  - used for storing short sequences of values that must be of a certain length (don't want to handle arbitrary length lists)

Var-args

- **variadic procedure**: can take a varying number of params
  - we have already seen this: `+`, `*`, `and`, `or`, `list`, etc.

- Three ways to define a Scheme procedure's parameters:
  - *list* of parameters: exactly that many must be passed
  - *single* parameter: any number may be passed
  - *improper* list: at least a given number must be passed

Fixed args vs. var-args

```
(define (name param1 … paramN) expr)
```

- a procedure with exactly ***N*** required parameters

```
(define (bigger a b) (if (> a b) a b))
```

```
(define name (lambda param expr))
```

- a procedure that accepts any number of parameters
- must be defined with the `lambda` syntax

```
(define sum-all (lambda L (foldl + 0 L)))
```

Var-args via improper lists

(define (***name param1*** … ***paramN*** . ***rest***) ***expr***)

- a procedure with ***param1-N*** required parameters, and a list of varying length to represent additional params passed
- allows passing a variable number of arguments (≥ N)

- Example:

```
(define (multiply-all-by n . args)
  (define (f k) (* n k))
  (map f args))

> (multiply-all-by 5 2 3 -1 7)
(10 15 -5 35)
```

Associative lists (maps) with pairs

- Recall: a **map** associates *keys* with *values*
  - can retrieve a value later by supplying the key
- in Scheme, a map is stored as a list of key/value **pairs**:

```
(define phonebook (list '(Marty 6852181)
     '(Stuart 6859138) '(Jenny 8675309)))
```

- look things up in a map using the `assoc` procedure:

```
> (assoc 'Stuart phonebook)
(Stuart 6859138)
> (cdr (assoc 'Jenny phonebook))  ; get value
8675309
```