Computer Science I  
*Problem-Solving and Abstraction*  
Prof. Jonathan Gordon  
Lecture 16

Higher-order list processing:  
**map** revisited

So far, we’ve used **map** with functions that take only a single argument, e.g.,

```scheme
> (map even? '(1 2 3 4 5 6))
(#f #t #f #t #f #t)
```

However, **map** can be used with functions that take multiple arguments.

For a function that takes *n* arguments, you need to supply *n* lists, each of the same length.

**map** will first apply the function using the first element of each list, then using the second element of each list, and so on:

```scheme
> (map + '(1 2 3) '(3 2 1))
(4 4 4)
> (map * '(1 2 3) '(4 5 6) '(7 8 9))
(28 80 162)
```
zip and unzip using map

(define zip (lambda (lst1 lst2) ...?...))
(define unzip (lambda (lst) ...?...))

> (zip '(1 2 3) '(a b c))
((1 a) (2 b) (3 c))
> (unzip '((1 a) (2 b) (3 c))
((1 2 3) (a b c))

When we use map, it returns a list of the results of applying a function to each element of a list.

When we don’t care about the values that are returned, we can use for-each, e.g.,
> (for-each print '(1 2 3))
123

Higher-order list processing:
List comprehensions
List comprehensions

(define comprehension
  (lambda (lst pred?) ...?...))

> (comprehension '(5 -2 9 -4) positive?)
(5 9)
> (comprehension '(5 2 9 4) negative?)
()  
> (comprehension '(5 2 9 4)
    (lambda (x) (< x 5)))
(2 4)
> (comprehension '(5 2 9 4)
    (lambda (x)
      (and (< 2 x) (< x 9))))
(5 4)

The comprehension procedure

comprehension takes a list lst and a predicate pred? as inputs.

The predicate pred? should accept one argument.

comprehension returns a list containing the members of lst that satisfy pred.

List order and multiple occurrences are preserved.

Example: The partition procedure

(define partition (lambda (lst n) ...?...))

> (define nums '(5 2 9 4))
nums
> (partition nums 4)
((2 4) (5 9))
> (partition nums 9)
((5 2 9 4) ())
> (partition nums 0)
(() (5 2 9 4))
The `partition` procedure

`partition` takes a list `lst` of numbers and a single number `n` as inputs.

`partition` returns a list containing two other lists `low` and `high`:
- `low` is a list containing all members of `lst` that are less than or equal to `n`.
- `high` is a list containing all members of `lst` that are greater than `n`.

Definition of `partition`

```
(define partition
  (lambda (lst n)
    (list (comprehension lst
                  (lambda (x)
                    (<= x n))
          (comprehension lst
                  (lambda (y)
                    (> y n))))))
```

Example: Difference of two lists

```
(define difference
  (lambda (lst1 lst2) ...?...))
```

Return a list of the members of `lst1` that are not members of `lst2`:
- `(difference '(1 2 3 4) '(3 4 5 6))` returns `(1 2)`
- `(difference
    '(mon tue wed thu fri sat sun)
    'tue thu))` returns `(mon wed fri sat sun)`

```
(define difference
  (lambda (lst1 lst2)
    (comprehension
      lst1
      (lambda (x)
        (not (member x lst2))))))
```
Example: Intersection of two lists

(define intersection
  (lambda (lst1 lst2) ...?...))

Return a list of the values that are members of both lst1 and lst2.

(define intersection
  (lambda (lst1 lst2)
    (comprehension
      lst1
      (lambda (x)
        (member x lst2))))))

The built-in function equivalent to our comprehension function is filter:

> (filter even? '(1 2 3))
(2)

Higher-order list processing:
Reduction
Reduction

A pattern of computation.

Useful in a wide variety of situations.

Comes in two varieties:

- **Flat reduction**: (“List morphism”).
- **Deep reduction**: (“Tree morphism”).

Each is implemented by a higher-order procedure.

**List morphism**

```
(define reduce-flat
  (lambda (lst fun base)
    (if (null? lst)
        base
        (fun (first lst) (reduce-flat (rest lst) fun base)))))
```
When we give \texttt{cons} as \texttt{fun} and \texttt{'}()\texttt{ as base, we get back the input list!}

\[
\text{(define make-list (lambda (lst) (reduce-flat lst cons '())))}
\]

\[
> \text{(make-list '(1 2 3))}
\]

\[
(1 2 3)
\]

\[
\text{(define sum (lambda (lst) (if (null? lst) 0 (+ (first lst) (sum (rest lst))))))}
\]

\[
\text{(define sum (lambda (lst) (reduce-flat lst + 0))}]
\]
(define length
  (lambda (lst)
    (reduce-flate
     lst
     (lambda (x y) (+ 1 y))
     0)))

Flat reduction examples

(define for-all?  ; like ANDMAP
  (lambda (lst p?)
    (reduce-flat (map p? lst)
               (lambda (x y) (and x y))
               #t))

(define there-exists? ; like ORMAP
  (lambda (lst p?)
    (reduce-flat (map p? lst)
               (lambda (x y) (or x y))
               #f)))

for-all? as list morphism (map p? lst)

for-all? as list morphism (reduce-flat ...)

for-all? as list morphism (map p? lst)
Deep reduction

Tree morphism

(define reduce-deep
  (lambda (exp ind-fun base-fun null-value)
    (cond ((null? exp)
           null-value)
           ((not (pair? exp))
            (base-fun exp))
           (else
            (ind-fun (reduce-deep
                      (first exp)
                      ind-fun base-fun null-value)
                     (reduce-deep
                      (rest exp)
                      ind-fun base-fun null-value))))))
Deep reduction examples

```
(define item-count*
  (lambda (lst)
    (reduce-deep lst +
      (lambda (x) (if (null? x) 0 1))
          0)))

(define item-sum*
  (lambda (lst)
    (reduce-deep lst +
      (lambda (x) (if (null? x) 0 x))
          0)))

> (item-count* '(1 2 (3)))
3
> (item-sum* '(1 2 (3)))
6
```

```
(define maxx
  (lambda (x y)
    (cond ((null? x) y)
          ((null? y) x)
          (else (max x y)))))

(define maximum*
  (lambda (lst)
    (reduce-deep lst maxx identity null)))

> (maximum* '(1 2 (3)))
3
```

maximum* as tree morphism

```
Procedures as return values
```
The procedures we’ve written have returned numbers, lists, strings, true/false. But we can also write functions that write functions.

General form of definition for a procedure returning a procedure

\[
\text{(define } \langle \text{name} \rangle \text{)}
\begin{align*}
\text{(lambda } \langle \text{args1} \rangle \\
\text{(lambda } \langle \text{args2} \rangle \langle \text{body} \rangle))
\end{align*}
\]

Expression defining the function \( \langle \text{name} \rangle \)

Expression defining the return value of the function \( \langle \text{name} \rangle \)

Basic example of returning a procedure

\[
\text{;; \text{MULT-BY-X}}
\]

\[
\text{;; \text{INPUTS: } X, \text{ a number}}
\]

\[
\text{;; \text{OUTPUT: } a \text{ procedure that multiples its}}
\]

\[
\text{;; \text{input by } X}
\]

\[
\text{(define mul-by-x}
\begin{align*}
\text{(lambda (x)} \\
\text{(lambda (n) (* n x)))}
\end{align*}
\]

\[
\text{(define mul-by-5 (mul-by-x 5))}
\]

\[
\text{> (mul-by-5 10)}
\]

\[
50
\]

Example: \text{map-function}

Converts a function with signature:

\[
\langle \text{type A} \rangle \rightarrow \langle \text{type B} \rangle
\]

into a function with signature:

\[
\langle \text{list of type A items} \rangle \rightarrow \langle \text{list of type B items} \rangle
\]

E.g.,

\[
\begin{align*}
\text{> (define double (lambda (x) (* x 2)))}
\end{align*}
\]

\[
\begin{align*}
\text{> (define double-all (map-function double))}
\end{align*}
\]

\[
\begin{align*}
\text{> (double-all '(1 2 3 4 5))}
\text{(2 4 6 8 10)}
\end{align*}
\]

\[
\begin{align*}
\text{> (map double '(1 2 3 4 5))}
\text{(2 4 6 8 10)}
\end{align*}
\]
Given an input function \( F \), return a new function that maps \( F \) across its input list \( LST \):

\[
(\text{define \textit{map-function}}
\begin{align*}
\text{lambda} (f) \\
(\text{lambda} (\text{lst}) \\
(\text{map} ~ f ~ \text{lst})))
\end{align*}
\]

**Example:** \textit{flat-reduce-function}

Converts a function with signature:
\[
\langle \text{type} \rangle \times \langle \text{type} \rangle \rightarrow \langle \text{type} \rangle
\]
into a function with signature
\[
\langle \text{list of type items} \rangle \rightarrow \langle \text{type} \rangle
\]

E.g.,
\[
\begin{align*}
> (~ + ~ 1 ~ 2) \\
3
\end{align*}
\]
\[
\begin{align*}
> (~ \text{(define \textit{sum} (flat-reduce-function} + \text{0)})) \\
> (~ \text{(sum} '((1 ~ 2 ~ 3)))
6
\end{align*}
\]

Given an input function \( F \), return a new function that repeatedly applies \( F \) to pairs of items in \( LST \), using ID as the base case.

\[
(\text{define \textit{flat-reduce-function}}
\begin{align*}
\text{lambda} (f ~ \text{id}) \\
(\text{lambda} (\text{lst}) \\
(\text{reduce-flat} ~ f ~ \text{lst} ~ \text{id})))
\end{align*}
\]

**Example:** \textit{deep-reduce-function}

Converts a function with signature:
\[
\langle \text{type} \rangle \times \langle \text{type} \rangle \rightarrow \langle \text{type} \rangle
\]
into a function with signature
\[
\langle \text{tree of type items} \rangle \rightarrow \langle \text{type} \rangle
\]

E.g.,
\[
\begin{align*}
> (~ + ~ 1 ~ 2) \\
3
\end{align*}
\]
\[
\begin{align*}
> (~ \text{(define \textit{sum*} (deep-reduce-function} + \text{0)})) \\
> (~ \text{(sum*} '((1 ~ 2 ~ 3)))
6
\end{align*}
\]
Given an input function $F$, return a new function that repeatedly applies $F$ to pairs of items in the potentially nested list $LST$, using ID as the base case.

\[
\text{(define deep-reduce-function)} \quad \text{(lambda (f id)} \quad \text{(lambda (lst)}} \quad \text{(reduce-deep f lst id))})
\]

A function $f$ that takes two arguments is equivalent to another function $c$:

\[
((c f \langle\text{arg1}\rangle \langle\text{arg2}\rangle) = (f \langle\text{arg1}\rangle \langle\text{arg2}\rangle))
\]

$c f$ is a function of one argument that returns a function of one argument.

$c f$ is called the “Curried” version of $f$, after the mathematician Haskell Curry.

\[
\text{(define curry-binary-function)} \quad \text{(lambda (f)} \quad \text{(lambda (x)}} \quad \text{(lambda (y)}} \quad \text{(f x y))})
\]

\[
\text{(define uncurry-binary-function)} \quad \text{(lambda (f)} \quad \text{(lambda (x y)}} \quad \text{((f x) y))))
\]
There are a variety of types of Scheme objects: numbers, Booleans, symbols, lists, procedures.

Given any such object, no matter its type, we can ask Scheme to *evaluate* it.

- Numbers, Booleans, and procedures evaluate to themselves.
- Symbols are always evaluated by looking in some environment.
- Lists are evaluated (in the default case) by applying the first element (which had better be a function) to the rest of the elements.
- The key thing is that *any* Scheme object may be evaluated.

Similarly, the arguments that are passed to a function can be *any* type of Scheme object, including procedures.

Also, the return value of a function call can be *any* type of Scheme object, including a procedure.

In many other programming languages, the so-called “first-class objects” include things like numbers and symbols, but *not* functions!

- In those languages, an expression cannot evaluate to a function.
- In those languages, you cannot pass functions as arguments to other functions.

This is very limiting.

Scheme is not subject to these limitations. In Scheme, the “first-class objects” include procedures.

What we’ve seen is that this is a good thing!

**Appendix: Defining procedures with an arbitrary number of parameters**
So far, whenever we use a lambda form to specify a procedure, we explicitly specify how many arguments the procedure takes.

E.g., the squaring function \( \lambda (x) (* x x) \) takes exactly one argument. If we try calling it with some other number of arguments, we'll get an error.

However, we've seen examples of built-in functions that take any number of arguments, e.g., the arithmetic functions +, -, *, and /.

Can we do the same?

**Unrestricted lambda form**

Scheme provides an easy way to specify functions that, like these built-in functions, can be applied to arbitrarily many arguments:

\[
\text{(lambda \( args \) \( \langle\text{body}\rangle \))}
\]

Notice that \( args \) has no parentheses.

Expressions in the body of the \textit{lambda} form may refer to the \textit{args} symbol, which will evaluate to the list of arguments supplies to the function when it was called.

**Defining procedures with an arbitrary number of parameters**

\[
\text{(define max-\(\text{arb}\) (lambda \( values \) ...?...))}
\]

> (max-\(\text{arb}\) 3 6 5)
6
> (max-\(\text{arb}\) 5 3)
5
> (max-\(\text{arb}\) 7)
7

\[
\text{(define max-2}
\text{(lambda (x y)}
\text{(if (>= x y) x y))}
\text{)}
\]

\[
\text{(define max-\(\text{arb}\)}
\text{(lambda \( values \)}
\text{(cond}
\text{;; Base case: Exactly one argument}
\text{((null? \( \text{rest values} \))}
\text{(first \( values \))})
\text{;; Recursive case: More than one argument}
\text{(else}
\text{(max-2 \( \text{first \( \text{args} \)} \))})
\text{))}
\]
The built-in **and** and **or** keywords aren’t functions, so we can’t use them as inputs to **apply** – but we can define versions that are functions!

```scheme
(define and-proc
  (lambda values
    (reduce-flat values
      (lambda (x y) (and x y))
      #t)))

(define or-proc
  (lambda values
    (reduce-flat values
      (lambda (x y) (or x y))
      #f)))
```

---

**for-all? and there-exists? (again)**

Now we can very simply redefine our **for-all?** and **there-exists?** functions using **apply** and **map**:

```scheme
(define for-all?
  (lambda (lst p?)
    (apply and-proc (map p? lst)))))

(define there-exists?
  (lambda (lst p?)
    (apply or-proc (map p? lst)))))
```