Higher-order procedures

A procedure is called *first-order* if none of its arguments is itself a procedure.

A procedure is called *higher-order* if one or more of its arguments is a procedure.

The importance of higher-order procedures

They allow us to write very general procedures that we can use over and over.

They allow us to precisely define *patterns of computation*, such as flat recursion, deep recursion, and the accumulator method.
The apply procedure

> (apply cons '(a ()))
'(a)
> (cons 'a '())
'(a)

> (apply - '(30 15 10))
5
> (- 30 15 10)
5

The apply procedure

apply takes a procedure p and a list lst as inputs.
If the length of lst is n, then p should accept n arguments.

apply applies p to the members of lst.
Suppose the value of lst is: (a₁ a₂ ... aₙ).
Then (apply p lst) returns the same value as: (p 'a₁ 'a₂ ... 'aₙ).

The apply procedure

(apply <proc>
(list <arg₁> <arg₂> ... <argₙ>))

downward

is equivalent to:

downward

(⟨proc⟩ ⟨arg₁⟩ ⟨arg₂⟩ ... ⟨argₙ⟩)

The apply procedure

(apply ⟨proc⟩ '(⟨arg₁⟩ ⟨arg₂⟩ ... ⟨argₙ⟩))

downward

is equivalent to:

downward

(⟨proc⟩ '⟨arg₁⟩ '⟨arg₂⟩ ... '⟨argₙ⟩)
The apply procedure

\[ \text{(apply } \langle \text{proc} \rangle \langle \text{lst} \rangle) \]

\[ \downarrow \]

is equivalent to:

\[ \langle \text{proc} \rangle \text{(list-ref } \langle \text{lst} \rangle 0) \]
\[ \text{(list-ref } \langle \text{lst} \rangle 1) \]
\[ \ldots \]
\[ \text{(list-ref } \langle \text{lst} \rangle n-1) \]

Using apply to implement sum

\[ \text{(define sum } \text{(lambda } (\text{lst}))(\text{apply } + \text{lst})) \]

> (sum '(1 2 3 4 5))
15
> (sum '())
0

Notice that \(\text{(apply } + \text{') evaluates to zero. Why?\)

Sum numbers from 1 to \(n\)

\[ \text{(define sum-to } \text{(lambda } (\text{n}))(\ldots?\ldots)) \]

> (sum-to 5)
15
> (+ 1 2 3 4 5)
15
> (sum-to 1)
1
> (sum-to 0)
0

Sub-problem:
List numbers from \(m\) to \(n\)

\[ \text{(define from-to } \text{(lambda } (\text{m n}))(\ldots?\ldots)) \]

> (from-to 1 5)
'(1 2 3 4 5)
> (from-to 1 1)
'(1)
> (from-to 1 0)
'()
from-to

(define from-to
  (lambda (low high)
    (if (> low high)
        '()
        (cons low
            (from-to (+ low 1) high)))))

sum-to

(define sum-to
  (lambda (n)
    (apply + (from-to 1 n))))

Using apply to implement factorial

(define factorial
  (lambda (n) ...?...))

> (factorial 5)
120
> (factorial 0)
1

Using apply to implement factorial

(define factorial
  (lambda (n)
    (apply * (from-to 1 n))))

Notice that (apply * '()) evaluates to one. Why?
So, what’s apply really?

(define apply
  (lambda (func lst)
    (eval (cons func lst))))

*Caution:* It’s very rare that we need to call the built-in eval function. We use in the tester function and we would use it here, but if you find yourself using the eval function a lot, you should probably rethink things.

The map procedure

(define map (lambda (fun lst) ...?...))

> (map list '(a b c))
  ((a) (b) (c))

> (map first '((a b c) (d e) (f)))
  (a d f)

> (map rest '((a b c) (d e) (f)))
  ((b c) (e) ()))

The map procedure

(define map (lambda (fun lst) ...?...))

> (map (lambda (x) (+ x 1))
    '(1 2 3))
  (2 3 4)

> (map (lambda (x) (* x x))
    '(1 2 3))
  (1 4 9)
The map procedure

map is a procedure that takes a procedure \( p \) and a list \( \text{lst} \) as inputs.

The procedure \( p \) should accept one argument.

map returns a list of the results of applying \( p \) to each member of \( \text{lst} \):

Suppose the value of \( \text{lst} \) is: \((a_1 \ a_2 \ \ldots \ a_n)\).

Then \((\text{map} \ p \ \text{lst})\) returns the same value as \((\text{list} \ (p \ a_1) \ (p \ a_2) \ \ldots \ (p \ a_n))\).

The map procedure

\[
(\text{map} \ \langle \text{proc} \rangle \ (\text{list} \ \langle \text{arg}_1 \rangle \ \langle \text{arg}_2 \rangle \ \ldots \ \langle \text{arg}_n \rangle))
\]

\[
\downarrow
\]

is equivalent to:

\[
(\text{list} \ (\langle \text{proc} \rangle \ \langle \text{arg}_1 \rangle))
\]

\[
(\langle \text{proc} \rangle \ \langle \text{arg}_2 \rangle)
\]

\[
\ldots
\]

\[
(\langle \text{proc} \rangle \ \langle \text{arg}_n \rangle))
\]

The map procedure

\[
(\text{map} \ \langle \text{proc} \rangle \ \langle \text{lst} \rangle)
\]

\[
\downarrow
\]

is equivalent to:

\[
(\text{list} \ (\langle \text{proc} \rangle \ \langle \text{list-ref} \ \langle \text{lst} \rangle \ 0 \rangle))
\]

\[
(\langle \text{proc} \rangle \ \langle \text{list-ref} \ \langle \text{lst} \rangle \ 1 \rangle)
\]

\[
\ldots
\]

\[
(\langle \text{proc} \rangle \ \langle \text{list-ref} \ \langle \text{lst} \rangle \ n-1 \rangle)
\]
Example: double-all

(define double-all (lambda (lst) ...?...))

> (double-all '(5 10 15))
(10 20 30)
> (double-all '(5))
(10)
> (double-all '())
()

Example: double-all

(define double-all
  (lambda (lst)
    (map double lst)))

(define double
  (lambda (x)
    (* 2 x)))

Definition of map procedure

(define map
  (lambda (p lst)
    (if (null? lst)
        ;; Applying p to each element of
        ;; the empty list yields...
        ;; the empty list.
        '()
      ;; Apply p to the first element
      ;; and to the rest recursively.
      (cons (p (first lst))
            (map p (rest lst)))))

But it’s also built-in!
Example: `sum-squares-to`

Sum of squares of numbers from 1 to n

```
(define sum-squares-to
  (lambda (n)
    ...
  ))
```

> (sum-squares-to 3)
14
> (+ (* 1 1) (* 2 2) (* 3 3))
14

We can do this easily using `from-to`, `map`, and `apply`!

Acknowledgments

This lecture incorporates material from:

Tom Ellman
Luke Hunsberger