CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 14

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.

`apply`

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: $(\mathtt{a}_1\ \mathtt{a}_2\ \ldots\ \mathtt{a}_n)$.

Then `(apply p lst)` returns the same value as: $(\mathtt{p}\ \mathtt{'a}_1\ \mathtt{'a}_2\ \ldots\ \mathtt{'a}_n)$.

The **apply** procedure

$$(\mathtt{apply}\ \langle proc\rangle\ (\mathtt{list}\ \langle arg_1\rangle\ \langle arg_2\rangle\ \ldots\ \langle arg_n\rangle))$$

is equivalent to:

$$(\langle proc\rangle\ \langle arg_1\rangle\ \langle arg_2\rangle\ \ldots\ \langle arg_n\rangle)$$

The **apply** procedure

$$(\mathtt{apply}\ \langle proc\rangle\ \mathtt{'}(\langle arg_1\rangle\ \langle arg_2\rangle\ \ldots\ \langle arg_n\rangle))$$

is equivalent to:

$$(\langle proc\rangle\ \mathtt{'}\langle arg_1\rangle\ \mathtt{'}\langle arg_2\rangle\ \ldots\ \mathtt{'}\langle arg_n\rangle)$$

The **apply** procedure

```
(apply ⟨proc⟩ ⟨lst⟩)
```

is equivalent to:

```
(⟨proc⟩ (list-ref ⟨lst⟩ 0)
        (list-ref ⟨lst⟩ 1)
        ...
        (list-ref ⟨lst⟩ n-1))
```

Using **apply** to implement **sum**

```
(define sum (lambda (lst) (apply + lst)))
```

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

Notice that `(apply + '())` evaluates to zero. Why?

Sum numbers from 1 to *n*

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

```
> (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*

```
(define from-to (lambda (m n) ...?...))
```

```
> (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.

map

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 *lst* as inputs.

The procedure *p* should accept one argument.

map returns a list of the results of applying *p* to each member of *lst:*

Suppose the value of *lst* is: `(`a_1 a_2 … a_n`)`.

Then `(map p lst)` returns the same value as `(list (p` a_1`) (p` a_2`)` … `(p` a_n`))`.

The map procedure

(map ⟨*proc*⟩ (list ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ … ⟨*arg*$_n$⟩))

⬇

is equivalent to:

⬇

```
(list (⟨proc⟩ ⟨arg1⟩)
      (⟨proc⟩ ⟨arg2⟩)
      ...
      (⟨proc⟩ ⟨argn⟩))
```

The map procedure

(map ⟨*proc*⟩ '(⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ … ⟨*arg*$_n$⟩))

⬇

is equivalent to:

⬇

```
(list (⟨proc⟩ '⟨arg1⟩)
      (⟨proc⟩ '⟨arg2⟩)
      ...
      (⟨proc⟩ '⟨argn⟩))
```

The map procedure

(map ⟨*proc*⟩ ⟨*lst*⟩)

⬇

is equivalent to:

⬇

```
(list (⟨proc⟩ (list-ref ⟨lst⟩ 0))
      (⟨proc⟩ (list-ref ⟨lst⟩ 1))
      ...
      (⟨proc⟩ (list-ref ⟨lst⟩ n−1)))
```

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)))
```

Example: **double-all**

```
(define double-all
  (lambda (lst)
    (map (lambda (x) (* 2 x))
         lst)))
```

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`**!

Sum of squares of numbers from 1 to *n*

```
(define sum-squares-to
  (lambda (n)
    (apply + (map (lambda (x) (* x x))
                  (from-to 1 n)))))
```


Acknowledgments

This lecture incorporates material from:

- Tom Ellman
- Luke Hunsberger