CS 6510
Practical Functional Programming

Spring 2012

Instructor: **Matthew Flatt**

Course Web Page

`http://www.eng.utah.edu/~cs6510/`

Subscribe to the mailing list!

Course Structure

```
(define (cs6510)
```

- You pick a functional programming language
- We pick a programming task
- You implement the task in your chosen language
- You show code in class

```
(cs6510))
```

Functional Programming Languages

Some options:

- Racket
- Haskell
- OCaml, F#, Standard ML
- Clojure

(This list is not exhaustive.)

Getting Started:

Arithmetic, Algebra, and Computing

Arithmetic is Computing

- Fixed, pre-defined rules for ***primitive operators***:

$$2 + 3 = 5$$

$$4 \times 2 = 8$$

$$\cos(0) = 1$$

Arithmetic is Computing

- Fixed, pre-defined rules for ***primitive operators***:

$$2 + 3 \rightarrow 5$$

$$4 \times 2 \rightarrow 8$$

$$\cos(0) \rightarrow 1$$

- Rules for combining other rules:
  - Evaluate sub-expressions first

$$4 \times (2 + 3) \rightarrow 4 \times 5 \rightarrow 20$$

  - Precedence determines subexpressions:

$$4 + 2 \times 3 \rightarrow 4 + 6 \rightarrow 10$$

Algebra as Computing

- Definition:

$$f(x) = \cos(x) + 2$$

- Expression:

$$f(0) \rightarrow \cos(0) + 2 \rightarrow 1 + 2 \rightarrow 3$$

First step uses the ***substitution*** rule for functions

Racket Expression Notation

- Put all operators at the front
- Start every operation with an open parenthesis
- Put a close parenthesis after the last argument
- Never add extra parentheses

Old	New
$1 + 2$	`(+ 1 2)`
$4 + 2 \times 3$	`(+ 4 (* 2 3))`
$\cos(0) + 1$	`(+ (cos 0) 1)`

Racket Definition Notation

- Use **`define`** instead of =
- Put **`define`** at the front, and group with parentheses
- Move open parenthesis from after function name to before

Old	New
f(x) = cos(x) + 2	`(define (f x) (+ (cos x) 2))`

- Move open parenthesis in function calls

Old	New
f(0)	`(f 0)`
f(2+3)	`(f (+ 2 3))`

Evaluation is the Same as Before

```
(define (f x) (+ (cos x) 2))

(f 0)
```

Evaluation is the Same as Before

```
(define (f x) (+ (cos x) 2))

(f 0)
→ (+ (cos 0) 2)
```

Evaluation is the Same as Before

```
(define (f x) (+ (cos x) 2))

(f 0)
→ (+ (cos 0) 2)
→ (+ 1 2)
```

Evaluation is the Same as Before

```
(define (f x) (+ (cos x) 2))

(f 0)
→ (+ (cos 0) 2)
→ (+ 1 2)
→ 3
```

Booleans

Numbers are not the only kind of value:

Old	**New**
$1 < 2 \rightarrow$ true	`(< 1 2)` → `true`
$1 > 2 \rightarrow$ false	`(> 1 2)` → `false`
$1 > 2 \rightarrow$ false	`(> 1 2)` → `false`
$2 \geq 2 \rightarrow$ true	`(>= 2 2)` → `true`

Booleans

Old	New
true and false	`(and true false)`
true or false	`(or true false)`
$1 < 2$ and $2 > 3$	`(and (< 1 2) (> 2 3))`
$1 \leq 0$ and $1 = 1$	`(or (<= 1 0) (= 1 1))`
$1 \neq 0$	`(not (= 1 0))`

Strings

```
(string=? "apple" "apple") → true
(string=? "apple" "banana") → false

(string-append "up" "on") → "upon"
(string-append "a" "b" "c") → "abc"

(string-length "hippopotamus") → 12
```

Images

```
(image=?        ) → true

(overlay        ) →

(image-width    ) → 88

(circle 10 "solid" "red") →

(overlay
 (circle 10 "solid" "red")
 (rectangle 30 40 "solid" "blue")) →
```

Functions on Images

```
(define (roll img)
  (beside img
          (rotate 90 img)
          (rotate 180 img)
          (rotate 270 img)))
```

(roll) →

Defining Constants

Use **`define`** and ***`name`*** without parentheses around ***`name`*** to define a constant:

```
(define upside-down-u
  (rotate 180 [image]))
```

Use the ***`name`*** without parentheses:

```
(beside upside-down-u
        upside-down-u) → [image]
```

Conditionals

Conditionals

WANTED

`(maybe-wanted` `)` →

`(maybe-wanted` `)` →

Conditionals in Algebra

General format of conditionals in algebra:

$$\begin{cases} \textit{answer} & \textit{question} \\ \ldots & \\ \textit{answer} & \textit{question} \end{cases}$$

Example:

$$\text{abs}(x) = \begin{cases} x & \text{if } x > 0 \\ -x & \text{otherwise} \end{cases}$$

$$\text{abs}(10) = 10$$

$$\text{abs}(-7) = 7$$

Conditionals in Racket

```
(cond
 [question answer]
 ...
 [question answer])
```

- Any number of **cond** "lines"
- Each line has one *question* expression and one *answer* expression

```
(define (absolute x)
  (cond
   [(> x 0) x]
   [else (- x)]))
```

```
(absolute 10) → 10

(absolute -7) → 7
```

Evaluation Rules for cond

First question is literally `true` or `else`

```
(cond
 [true answer]                → answer
 ...
 [question answer])
```

- Keep only the first answer

Example:

```
(* 1 (cond          → (* 1 0) → 0
       [true 0]))
```

Evaluation Rules for cond

First question is literally `false`

```
(cond
 [false answer]
 [question answer]       →    (cond
 ...                            [question answer]
 [question answer])             ...
                                [question answer])
```

- Throw away the first line

Example:

```
(+ 1 (cond            → (+ 1 (cond
       [false 1]                 [true 17]))
       [true 17]))

                        → (+ 1 17) → 18
```

Evaluation Rules for cond

First question isn't a value, yet

```
(cond                              (cond
 [question answer]       →          [nextques answer]
 ...                                ...
 [question answer])                 [question answer])
```

where *question* → *nextques*

- Evaluate first question as sub-expression

Example:

```
(+ 1 (cond               → (+ 1 (cond
      [(< 1 2) 5]                [true 5]
      [else 8]))                 [else 8]))

                         → (+ 1 5) → 6
```

Evaluation Rules for cond

No true answers

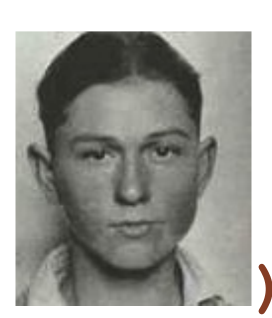
```
(cond)    →    error
```

Just an **else**

```
(cond
 [else answer])    →    answer
```

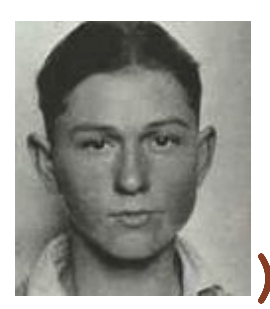
Conditionals

```
(define (maybe-wanted who wanted-who)
  (cond
   [(image=? who wanted-who)
    (above (text "WANTED" 32 "black") who)]
   [else
    who]))
```

(maybe-wanted ) → WANTED

Conditionals

```
(define (maybe-wanted who wanted-who)
  (cond
   [(image=? who wanted-who)
    (above (text "WANTED" 32 "black") who)]
   [else
    who]))
```

(maybe-wanted ) →

Compound Data

Transforming a Point

Convert Avenues corners to SLC coordinates

~~`; ave->slc : string num -> num num`~~

Must return a single value

Correct contract:

```
; ave->slc : string num -> posn
```

A `posn` is a ***compound value***

Positions

- A **posn** is

  ```
  (make-posn X Y)
  ```

  where **X** is a **num** and **Y** is a **num**

Examples:

```
(make-posn 1 2)
(make-posn 17 0)
```

A **posn** is a value, just like a number, symbol, or image

posn-x and posn-y

The **`posn-x`** and **`posn-y`** operators extract numbers from a **`posn`**:

```
(posn-x (make-posn 1 2)) → 1
(posn-y (make-posn 1 2)) → 2
```

- General evaluation rules for any values **`X`** and **`Y`**:

```
(posn-x (make-posn X Y)) → X
(posn-y (make-posn X Y)) → Y
```

Positions and Values

Is `(make-posn 100 200)` a value?

Yes.

A `posn` is

```
(make-posn X Y)
```

where `X` is a `num` and `Y` is a `num`

Positions and Values

Is `(make-posn (+ 1 2) 200)` a value?

No. `(+ 1 2)` is not a `num`, yet.

- Two more evaluation rules:

```
(make-posn X Y) → (make-posn Z Y)
                        when X → Z

(make-posn X Y) → (make-posn X Z)
                        when Y → Z
```

Example:

```
(make-posn (+ 1 2) 200) →
(make-posn 3 200)
```

More Examples

Try these in DrRacket's stepper:

```
(make-posn (+ 1 2) (+ 3 4))

(posn-x (make-posn (+ 1 2) (+ 3 4)))

; pixels-from-corner : posn -> num
(define (pixels-from-corner p)
  (+ (posn-x p) (posn-y p)))
(pixels-from-corner (make-posn 1 2))

; flip : posn -> posn
(define (flip p)
  (make-posn (posn-y p) (posn-x p)))
(flip (make-posn 1 2))
```

Programmer-Defined Compound Data

Other Kinds of Data

Suppose we want to represent snakes:

- name
- weight
- favorite food

What kind of data is appropriate?

Not `num`, `bool`, `sym`, `image`, or `posn`...

Data Definitions and define-struct

Here's what we'd like:

A **snake** is

```
(make-snake sym num sym)
```

... but **make-snake** is not built into DrRacket

We can tell DrRacket about **snake**:

```
(define-struct snake (name weight food))
```

Creates the following:

- **make-snake**
- **snake-name**
- **snake-weight**
- **snake-food**

Data Definitions and define-struct

Here's what we'd like:

A **snake** is

```
(make-snake sym num sym)
```

... but **make-snake** is not built into DrRacket

We can tell DrRacket about **snake**:

```
(define-struct snake (name weight food))
```

Creates the following:

```
(snake-name (make-snake X Y Z)) → X
(snake-weight (make-snake X Y Z)) → Y
(snake-food (make-snake X Y Z)) → Z
```