CS 1410 — Computer Science I
Section 20

Fall 2010

Instructor: **Matthew Flatt**

Course Details

- Everything is in the course web page:

  `http://www.eng.utah.edu/~cs1410-20/`

- The starting book is online:

  How to Design Programs, Second Edition
  Felleisen, Findler, Flatt, Krishnamurthi
  `http://www.ccs.neu.edu/home/matthias/HtDP2e/index.html`

- Assignments use DrRacket:

  `http://racket-lang.org/`

Things You Need to Do

- Read the course syllabus
- Subscribe to **`cs1410-20@list.eng.utah.edu`**
    - See the course web page for instructions
- Go to lab on Thursday
- Complete HW 0
    - On the course schedule page
    - Maybe mostly in lab

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***:

$$2 + 3 \rightarrow 5$$

$$4 \times 2 \rightarrow 8$$

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

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

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

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:

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

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 × 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))`

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`
$1 > 2 \rightarrow$ true	`(> 1 2)` → `false`
$1 > 2 \rightarrow$ true	`(> 1 2)` → `false`
$2 \geq 2 \rightarrow$ true	`(>= 1 2)` → `true`

Booleans

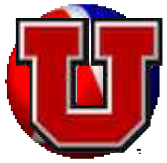
Old	**New**
true and false	`(and true false)`
true or false	`(or true false)`
$1 < 2$ and $2 > 3$	`(and (< 1 2) (> 2 3))`
$1 \leq 0$ and $1 = 1$	`(or (<= 1 0) (= 1 1))`
$1 \neq 0$	`(not (= 1 0))`

Strings

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

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

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

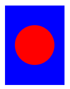
Images

`(image=?` `)` → `true`

`(overlay` `)` → 

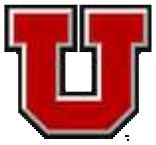
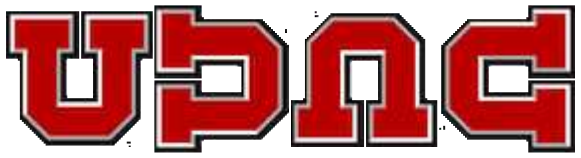
`(image-width` `)` → `88`

`(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 ) → 

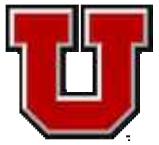
Defining Constants

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

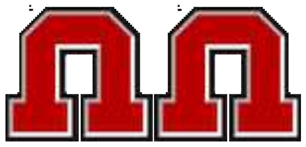
```
(define upside-down-u
  (rotate 180 ))
```

Defining Constants

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

```
(define upside-down-u
  (rotate 180 
```

```
  ))
```

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

```
(beside upside-down-u
        upside-down-u) →
```


Conditionals

WANTED

`(maybe-wanted` `)` →

`(maybe-wanted` `)` →

Conditionals in Algebra

General format of conditionals in algebra:

$$\begin{cases} \textit{answer} & \textit{question} \\ \quad \dots & \\ \textit{answer} & \textit{question} \end{cases}$$

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

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)]))
```

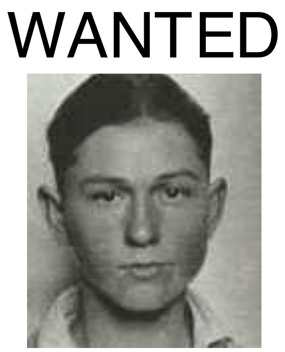
```
(absolute 10) → 10

(absolute -7) → 7
```

Conditionals

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

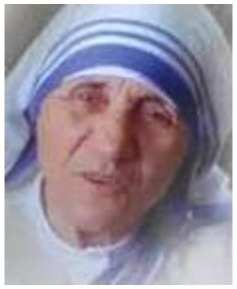
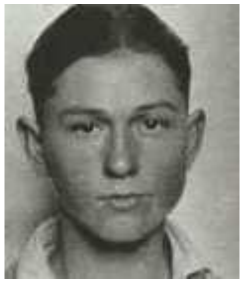
Conditionals

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

(maybe-wanted) → 

Conditionals

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

(maybe-wanted  ) →