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

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + 2</td>
<td>(+ 1 2)</td>
</tr>
<tr>
<td>4 + 2 × 3</td>
<td>(+ 4 (* 2 3))</td>
</tr>
<tr>
<td>(\cos(0) + 1)</td>
<td>(+ (cos 0) 1)</td>
</tr>
</tbody>
</table>
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**

\[
f(x) = \cos(x) + 2\
\]

**New**

\[
(define (f x) (+ (cos x) 2))\
\]
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

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) = \cos(x) + 2 )</td>
<td>( (\text{define} \ (f \ x) \ (+ \ (\cos \ x) \ 2)) )</td>
</tr>
</tbody>
</table>

- Move open parenthesis in function calls

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(0) )</td>
<td>( (f \ 0) )</td>
</tr>
<tr>
<td>( f(2+3) )</td>
<td>( (f \ (+ \ 2 \ 3)) )</td>
</tr>
</tbody>
</table>
Evaluation is the Same as Before

\[(\text{define } \mathbf{f} \mathbf{x} (\mathbf{+} \mathbf{(cos} \mathbf{x}) 2))\]

\[\mathbf{f} \mathbf{0}\]
Evaluation is the Same as Before

\[\text{(define } f \ x \ (\text{+ } (\text{cos } x) \ 2))\]

\[f \ 0\]
\[\rightarrow \ (\text{+ } (\text{cos } 0) \ 2)\]
Evaluation is the Same as Before

\[(\text{define } (f \ x) \ (\text{+} \ (\text{cos} \ x) \ 2))\]

\[(f \ 0)\]

\[\rightarrow (\text{+} \ (\text{cos} \ 0) \ 2)\]

\[\rightarrow (\text{+} \ 1 \ 2)\]
Evaluation is the Same as Before

\[
\text{(define (f x) (+ (cos x) 2))}
\]

\[
(f 0)
\]
\[
\rightarrow (+ (\cos 0) 2)
\]
\[
\rightarrow (+ 1 2)
\]
\[
\rightarrow 3
\]
Booleans

Numbers are not the only kind of value:

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 &lt; 2 \rightarrow \text{true})</td>
<td>((&lt; 1 2) \rightarrow \text{true})</td>
</tr>
<tr>
<td>(1 &gt; 2 \rightarrow \text{true})</td>
<td>((&gt; 1 2) \rightarrow \text{false})</td>
</tr>
<tr>
<td>(1 &gt; 2 \rightarrow \text{true})</td>
<td>((&gt; 1 2) \rightarrow \text{false})</td>
</tr>
<tr>
<td>(2 \geq 2 \rightarrow \text{true})</td>
<td>((\geq 1 2) \rightarrow \text{true})</td>
</tr>
</tbody>
</table>
# Booleans

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>true and false</td>
<td>(and true false)</td>
</tr>
<tr>
<td>true or false</td>
<td>(or true false)</td>
</tr>
<tr>
<td>( l &lt; 2 ) and ( 2 &gt; 3 )</td>
<td>(and (&lt; 1\ 2) ( &gt; 2\ 3))</td>
</tr>
<tr>
<td>( l \leq 0 ) and ( l = l )</td>
<td>(or (&lt;= 1\ 0) ( = 1\ 1))</td>
</tr>
<tr>
<td>( l \neq 0 )</td>
<td>(not ( = 1\ 0))</td>
</tr>
</tbody>
</table>
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=?) \rightarrow true

(overlay) \rightarrow

(image-width) \rightarrow 88

(circle 10 "solid" "red") \rightarrow ⬤

(overlay
 (circle 10 "solid" "red")
 (rectangle 30 40 "solid" "blue")) \rightarrow 🟡
Functions on Images

\[
\text{(define \ (roll \ img)} \\
\text{\ (beside \ img)} \\
\text{\ \ \ (rotate \ 90 \ img)} \\
\text{\ \ \ (rotate \ 180 \ img)} \\
\text{\ \ \ (rotate \ 270 \ img)))}
\]

\[
\text{(roll \ U)} \rightarrow \text{\ducn}
\]
DefiningConstants

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

```
(define upside-down-u
  (rotate 180 U))
```
Defining Constants

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

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

Use the `name` without parentheses:

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

\[(\text{maybe-wanted}) \rightarrow \text{WANTED}\]

\[(\text{maybe-wanted}) \rightarrow \text{WANTED}\]
Conditionals in Algebra

General format of conditionals in algebra:

\[
\begin{cases}
\text{answer} & \text{question} \\
\text{...} & \\
\text{answer} & \text{question}
\end{cases}
\]

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

\[
\text{(cond}
\quad \text{[question answer]}
\quad \text{...}
\quad \text{[question answer])}
\]

- Any number of `cond` “lines”
- Each line has one `question` expression and one `answer` expression

\[
\text{(define (absolute x)} \\
\quad \text{(cond}
\quad \quad \text{[ (> x 0) x]}
\quad \quad \text{[else (- x)])})
\]

\[
\text{(absolute 10) \rightarrow 10}
\]

\[
\text{(absolute -7) \rightarrow 7}
\]
(define (maybe-wanted who wanted-who)
  (cond
    [(image=? who wanted-who)
     (above (text "WANTED" 32 "black") who)]
    [else
     who]])
(define (maybe-wanted who wanted-who)
  (cond
    [(image=? who wanted-who)
     (above (text "WANTED" 32 "black") who)]
    [else who]))

(WANTED)

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