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

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 + 2$</td>
<td>$(+ 1 2)$</td>
</tr>
<tr>
<td>$4 + 2 \times 3$</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

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

- Move open parenthesis in function calls

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

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

\[(f \ 0)\]
Evaluation is the Same as Before

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

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

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

\[
(f 0) \\
\rightarrow (+ (cos 0) 2) \\
\rightarrow (+ 1 2)
\]
Evaluation is the Same as Before

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

\[
(f \ 0)
\]
\[
\rightarrow (+ \ (\text{cos} \ 0) \ 2)
\]
\[
\rightarrow (+ \ 1 \ 2)
\]
\[
\rightarrow 3
\]
## 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>$1 &gt; 2 \rightarrow \text{false}$</td>
<td>$(&gt; 1 2) \rightarrow \text{false}$</td>
</tr>
<tr>
<td>$1 &gt; 2 \rightarrow \text{false}$</td>
<td>$(&gt; 1 2) \rightarrow \text{false}$</td>
</tr>
<tr>
<td>$2 \geq 2 \rightarrow \text{true}$</td>
<td>$(\geq 2 2) \rightarrow \text{true}$</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 $l = 1$</td>
<td>(or $(\leq 1 0)$ $(= 1 1)$)</td>
</tr>
<tr>
<td>$l \neq 0$</td>
<td>(not $(= 1 0)$)</td>
</tr>
</tbody>
</table>
Strings

(string=? "apple" "apple") \rightarrow true

(string=? "apple" "banana") \rightarrow false

(string-append "up" "on") \rightarrow "upon"

(string-append "a" "b" "c") \rightarrow "abc"

(string-length "hippopotamus") \rightarrow 12
Images

(image=? U U) → true

(overlay U u) →

(image-width u) → 88

(circle 10 "solid" "red") → ●

(overlay
  (circle 10 "solid" "red")
  (rectangle 30 40 "solid" "blue")) → ●
Functions on Images

\[
(\text{define } (\text{roll } \text{img})
  
  (\text{beside } \text{img}
    
    (\text{rotate} 90 \text{ img})
    (\text{rotate} 180 \text{ img})
    (\text{rotate} 270 \text{ img}))
  
(\text{roll } \text{U}) \rightarrow \text{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) \rightarrow \U
```
Conditionals
Conditionals

\((\text{maybe-wanted} \quad \rightarrow \quad \text{WANTED})\)

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

General format of conditionals in algebra:

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

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

\[
\text{(cond}
\hspace{1em} \text{[question answer]}
\hspace{1em} \ldots
\hspace{1em} \text{[question answer])}
\]

- Any number of \text{cond} “lines”
- Each line has one \text{question} expression and one \text{answer} expression

\[
\text{(define (absolute x)}
\hspace{1em} \text{(cond}
\hspace{2em} \text{[(> x 0) x]}
\hspace{2em} \text{[else (- x)])})
\hspace{1em} \text{(absolute 10)} \rightarrow 10
\hspace{1em} \text{(absolute -7)} \rightarrow 7
\]
Evaluation Rules for cond

First question is literally true or else

\[
\text{(cond}
  \begin{align*}
  &\text{[true answer]} \quad \rightarrow \quad \text{answer} \\
  &\ldots \\
  &\text{[question answer]})
\end{align*}
\]

• Keep only the first answer

Example:

\[
(* \ 1\ (\text{cond} \quad \rightarrow \quad (* \ 1 \ 0) \rightarrow \ 0 \\
  \text{[true} \ 0]))
\]
Evaluation Rules for cond

First question is literally \texttt{false}

$$(\text{cond}
\begin{array}{ll}
[\text{false} & \text{answer}] \\
[\text{question} & \text{answer}] & \rightarrow \\
\ldots & \\
[\text{question} & \text{answer}]\end{array})$$

- Throw away the first line

Example:

$$(+ 1 (\text{cond}
\begin{array}{ll}
[\text{false} 1] \\
[\text{true} 17]\end{array}) \rightarrow (+ 1 (\text{cond}
\begin{array}{ll}
[\text{true} 17]\end{array}))$$

$$\rightarrow (+ 1 17) \rightarrow 18$$
Evaluation Rules for cond

First question isn’t a value, yet

\[
\begin{align*}
&(\text{cond} \\
&\quad [\text{question} \newline \text{answer}]) \\
\quad \text{\ldots} \\
&\quad \quad [\text{question} \newline \text{answer}])
\end{align*}
\]

\[
\begin{align*}
&(\text{cond} \\
&\quad [\text{nextques} \newline \text{answer}]) \\
\text{\ldots}
\end{align*}
\]

where question \(\rightarrow\) nextques

- Evaluate first question as sub-expression

Example:

\[
\begin{align*}
(+1 \ (\text{cond} \\
&\quad [(<1 \ 2) \ 5] \\
&\quad [\text{else} \ 8]))) \\
\quad \rightarrow \ (+1 \ (\text{cond} \\
&\quad [\text{true} \ 5] \\
&\quad [\text{else} \ 8]))) \\
\quad \quad \rightarrow \ (+1 \ 5) \rightarrow 6
\end{align*}
\]
Evaluation Rules for cond

No true answers

\[(\text{cond}) \rightarrow \text{error}\]

Just an \texttt{else}

\[(\text{cond} \left[ \text{else} \ \texttt{answer} \right]) \rightarrow \texttt{answer}\]
(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 ) ➔
```
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

  \[(\text{make-posn } X \ Y)\]

  where \(X\) is a `num` and \(Y\) is a `num`

Examples:

  \[(\text{make-posn } 1 \ 2)\]

  \[(\text{make-posn } 17 \ 0)\]

A `posn` is a value, just like a number, symbol, or image
posn-x and posn-y

The \texttt{posn-x} and \texttt{posn-y} operators extract numbers from a \texttt{posn}:

\[
\text{(posn-x (make-posn 1 2)) } \rightarrow 1 \\
\text{(posn-y (make-posn 1 2)) } \rightarrow 2
\]

- General evaluation rules for any values \texttt{X} and \texttt{Y}:

\[
\text{(posn-x (make-posn X Y)) } \rightarrow X \\
\text{(posn-y (make-posn X Y)) } \rightarrow Y
\]
Positions and Values

Is \((\text{make-posn} \ 100 \ 200)\) a value?

Yes.

A \textit{posn} is

\((\text{make-posn} \ X \ Y)\)

where \(X\) is a \textit{num} and \(Y\) is a \textit{num}
Positions and Values

Is \((\text{make-posn} \ (+ \ 1 \ 2) \ 200)\) a value?

**No.** \((+ \ 1 \ 2)\) is not a \texttt{num}, yet.

- Two more evaluation rules:

\[
(\text{make-posn} \ X \ Y) \rightarrow (\text{make-posn} \ Z \ Y)
\]
when \(X \rightarrow Z\)

\[
(\text{make-posn} \ X \ Y) \rightarrow (\text{make-posn} \ X \ Z)
\]
when \(Y \rightarrow Z\)

Example:

\[
(\text{make-posn} \ (+ \ 1 \ 2) \ 200) \rightarrow
(\text{make-posn} \ 3 \ 200)
\]
More Examples

Try these in DrRacket’s stepper:

\[(\text{make-posn } (+ 1 2) (+ 3 4))\]

\[(\text{posn-x } (\text{make-posn } (+ 1 2) (+ 3 4)))\]

;;; 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

$(\text{make-snake } \text{sym num sym})$

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

We can tell DrRacket about snake:

$(\text{define-struct snake } (\text{name weight food}))$

Creates the following:

$(\text{snake-name } (\text{make-snake } X Y Z)) \rightarrow X$
$(\text{snake-weight } (\text{make-snake } X Y Z)) \rightarrow Y$
$(\text{snake-food } (\text{make-snake } X Y Z)) \rightarrow Z$