Getting Started:
Arithmetic, Algebra, and Computing

Arithmetic is Computing

- Fixed, pre-defined rules for *primitive operators*:
  - $2 + 3 = 5$
  - $4 \times 2 = 8$
  - $\cos(0) = 1$

- 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
Scheme 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>

Evaluation is the Same as Before

[Code]
```scheme
(define (f x) (+ (cos x) 2))
(f 0)
```

Scheme Notation

- Use the keyword `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>(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

[Code]
```scheme
(define (f x) (+ (cos x) 2))
(f 0)
→ (+ (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

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

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

Beyond Numbers: Booleans

Numbers are not the only kind of values:

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt; 2 → true</td>
<td>(&lt; 1 2) → true</td>
</tr>
<tr>
<td>1 &gt; 2 → true</td>
<td>(&gt; 1 2) → false</td>
</tr>
<tr>
<td>1 &gt; 2 → true</td>
<td>(&gt; 1 2) → false</td>
</tr>
<tr>
<td>2 ≥ 2 → true</td>
<td>(≥ 1 2) → true</td>
</tr>
</tbody>
</table>

<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>1 &lt; 2 and 2 &gt; 3</td>
<td>(and (&lt; 1 2) (&gt; 2 3))</td>
</tr>
<tr>
<td>1 ≤ 0 and 1 = 1</td>
<td>(or (&lt;= 1 0) (= 1 1))</td>
</tr>
<tr>
<td>1 ≠ 0</td>
<td>(not (= 1 0))</td>
</tr>
</tbody>
</table>
Beyond Numbers: Symbols

(symbol=? 'apple 'apple)  →  true
(symbol=? 'apple 'banana)  →  false

Beyond Numbers: Images

(filled-rect 35 35 'red)  →  
(filled-circle 25 25 'blue)  →  
(image+ 10 10 10)  →  
(offset-image+ 5 5 10)  →  
(image=? (image+ 10 10 10)  
  (image=? 10 10 10))  
  →  true

Programming with Images

(define (anonymize i)
  (offset-image+  
    i  
    0 0  
    (filled-circle (image-width i)  
      (image-height i)  
      'blue)))

(anonymize  
  )  →  ...

Conditionals
Conditionals in Algebra

General format of conditionals in algebra:

\[
\begin{align*}
\{ & \text{answer} \quad \text{question} \\
& \ldots \\
& \text{answer} \quad \text{question}
\end{align*}
\]

Example:

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

\[
\begin{aligned}
\text{abs}(10) &= 10 \\
\text{abs}(-7) &= 7
\end{aligned}
\]

Completing max-image

Use \texttt{cond} to complete \texttt{max-image}

\[
\begin{align*}
\text{(define (max-image a b)} \\
\text{(cond \\
  [(bigger-image? a b) a] \\
  [else b]))}
\end{align*}
\]

Evaluation Rules for \texttt{cond}

First question is literally \texttt{true} or \texttt{else}

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

Keep only the first answer

Example:

\[
\begin{aligned}
(* 1 \text{(cond) \rightarrow } (* 1 0) \rightarrow 0 \\
\text{[true 0])}
\end{aligned}
\]

Conditionals

General syntax of \texttt{cond} in Scheme:

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

\begin{itemize}
  \item Any number of \texttt{cond} lines
  \item Each line has one \texttt{question} expression and one \texttt{answer} expression
\end{itemize}

\[
\begin{align*}
\text{(define (abs x) \\
  \text{(cond } \\
  [(> x 0) x] \\
  [else (- x)])})
\end{align*}
\]

\[
\begin{aligned}
\text{(abs 10) "should be" 10} \\
\text{(abs -7) "should be" 7}
\end{aligned}
\]
Evaluation Rules for cond

First question is literally false

\[
\text{(cond [false answer] [question answer] \ldots [question answer])}
\]

- Throw away the first line

Example:

\[
(+ 1 \text{(cond [false 1] [true 17])})
\]
\[
\rightarrow (+ 1 17) \rightarrow 18
\]

Finding Images

\[
\text{(image-inside? [])} \rightarrow \text{true}
\]
\[
\text{(image-inside? [])} \rightarrow \text{false}
\]
Image Tests in Conditionals

Now we can combine such operators with `cond:

; detect-person : image image image -> image
; Returns a or b, depending on which is in i
(define (detect-person i a b)
  (cond
    [(image-inside? i a) a]
    [(image-inside? i b) b]))

(detect-person

"should be"

Compound Data

Finding and Adjusting Images

Suppose we want to write `frame-person:

(frame-person

"should be"

Finding an Image Position

`find-image : image image -> num num

Must return a single value

Correct contract:

`find-image : image image -> posn

• A `posn is a `compound value

Finding an Image Position

Need an operator that reports `where an image exists
Positions

- A posn is
  \[(\text{make-posn } X \ Y)\]
  where \(X\) is a \text{num} and \(Y\) is a \text{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 \text{posn-x} and \text{posn-y} operators extract numbers from a posn:

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

- General evaluation rules for any \(X\) and \(Y\):
  - \((\text{posn-x } (\text{make-posn } X \ Y)) \rightarrow X\)
  - \((\text{posn-y } (\text{make-posn } X \ Y)) \rightarrow Y\)

Positions and Values

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

Yes.

A posn is

- \((\text{make-posn } X \ Y)\)
  where \(X\) is a \text{num} and \(Y\) is a \text{num}

Positions and Values

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

No. \((+ \ 1 \ 2)\) is not a \text{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:

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

(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))

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...

Programmer-Defined Compound Data

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 DrScheme

We can tell DrScheme 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

\[(\text{make-snake } \text{sym } \text{num } \text{sym})\]

But make-snake is not built into DrScheme

We can tell DrScheme about snake:

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

Creates the following:

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