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*:

\[
\begin{align*}
2 + 3 & \rightarrow 5 \\
4 \times 2 & \rightarrow 8 \\
\cos(0) & \rightarrow 1
\end{align*}
\]
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:
  
  o Evaluate sub-expressions first

  \[
  4 \times (2 + 3) \rightarrow 4 \times 5 \rightarrow 20
  \]
Arithmetic is Computing

- Fixed, pre-defined rules for **primitive operators**:
  
  \[
  \begin{align*}
  2 + 3 & \rightarrow 5 \\
  4 \times 2 & \rightarrow 8 \\
  \cos(0) & \rightarrow 1 
  \end{align*}
  \]

- 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
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>
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

  Old                                      New

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

**Old**

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

**New**

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

- Move open parenthesis in function calls

**Old**

\[ f(0) \]

\[ f(2+3) \]

**New**

\[ (f \ 0) \]

\[ (f \ (+ \ 2 \ 3)) \]
Evaluation is the Same as Before

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

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

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

\[ (f 0) \quad \rightarrow \quad (+ (\cos 0) 2) \]
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
### 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>$2 \geq 2$ → true</td>
<td>$(\geq 1 2)$ → true</td>
</tr>
</tbody>
</table>
**Beyond Numbers: 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>$1 &lt; 2$ and $2 &gt; 3$</td>
<td>(and ($&lt;$ 1 2) ($&gt;$ 2 3))</td>
</tr>
<tr>
<td>$1 \leq 0$ and $1 = 1$</td>
<td>(or ($\leq$ 1 0) ($=$ 1 1))</td>
</tr>
<tr>
<td>$1 \neq 0$</td>
<td>(not ($=$ 1 0))</td>
</tr>
</tbody>
</table>
Beyond Numbers: Symbols

\[(\text{symbol}=? \text{ 'apple 'apple}) \rightarrow \text{true}\]

\[(\text{symbol}=? \text{ 'apple 'banana}) \rightarrow \text{false}\]
Beyond Numbers: Images

(filled-rect 35 35 'red) → □

(filled-circle 25 25 'blue) → ●
Beyond Numbers: Images

\[(\text{filled-rect} \ 35 \ 35 \ 'red) \rightarrow \square\]

\[(\text{filled-circle} \ 25 \ 25 \ 'blue) \rightarrow \circ\]

\[(\text{image+} \ \square \ \circ) \rightarrow \square\circ\]
Beyond Numbers: Images

\[(\text{filled-rect 35 35 'red}) \rightarrow \text{□}\]

\[(\text{filled-circle 25 25 'blue}) \rightarrow \text{●}\]

\[(\text{image+ □ ●}) \rightarrow \text{●□}\]

\[(\text{offset-image+ □ 5 5 ●}) \rightarrow \text{●□}\]
Beyond Numbers: Images

(filled-rect 35 35 'red) → □

(filled-circle 25 25 'blue) → ●

(image+ □ ●) → □●

(offset-image+ □ 5 5 ●) → □●

(image=? (image+ □ ●) □●) → (image=? ● ●) → true
Programming with Images

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

(anonymize )  →  ...  →
Conditionals
Conditionals in Algebra

General format of conditionals in algebra:

\[
\begin{array}{ll}
\text{answer} & \text{question} \\
\vdots & \\
\text{answer} & \text{question}
\end{array}
\]

Example:

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

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

General syntax of `cond` in Scheme:

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

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

General syntax of `cond` in Scheme:

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

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

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

```
(abs 10) "should be" 10
(abs -7) "should be" 7
```
Completing max-image

• Use cond to complete max-image

(define (max-image a b)
  (cond
    [(bigger-image? a b) a]
    [else b])))
Evaluation Rules for cond

First question is literally true or else

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

- Keep only the first answer
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:

\[
(+ 1 \text{(cond} \\
\begin{align*}
&\text{[true 1]} \\
&\text{[false 0]}) \quad \rightarrow \quad (+ 1 1) \rightarrow 2
\end{align*}
\]
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
    [true 0]
    [(< 10 12) 10]
    [(>= 10 12) 12]))
```
Evaluation Rules for cond

First question is literally true or else

\[
\text{(cond}
\begin{array}{l}
\text{[true answer]} \rightarrow \text{answer} \\
\ldots
\end{array}
\text{[question answer])}
\]

• Keep only the first answer

Example:

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

First question is literally false

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

• Throw away the first line
Evaluation Rules for cond

First question is literally false

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

• Throw away the first line

Example:

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

First question isn’t a value, yet

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

where \text{question} \to \text{nextques}

- Evaluate first question as sub-expression
Evaluation Rules for `cond`

First question isn’t a value, yet

\[
\text{(cond}\begin{array}{c}
\text{[question answer]} \\
\ldots \\
\text{[question answer]}\end{array}\text{)} \rightarrow \text{(cond}\begin{array}{c}
\text{[nextques answer]} \\
\ldots \\
\text{[question answer]}\end{array}\text{)}
\]

where \text{question} \rightarrow \text{nextques}

- Evaluate first question as sub-expression

Example:

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

Only question is false answers

\[
(\text{cond} \newline
\quad [\text{false} \ 10])
\rightarrow \text{error: all questions false}
\]
Finding Images

(image-inside?) → true
Finding Images

(image-inside? true) \rightarrow \text{true}

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

Now we can combine such operators with `cond`:

; detect-person : image 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]))

"should be"
Compound Data
Finding and Adjusting Images

Suppose we want to write \texttt{frame-person}:

\begin{center}
\texttt{(frame-person)}
\end{center}

"should be"
Finding and Adjusting Images

Suppose we want to write \texttt{frame-person}:

\[\texttt{(frame-person)}\]

"should be"

Need an operator that reports \textit{where} an image exists
Finding an Image Position

\texttt{find-image : image image \rightarrow num num}
Finding an Image Position

\[ \text{find-image : image image --> num num} \]

Must return a single value

Correct contract:

\[ \text{find-image : image image --> posn} \]

- A \textit{posn} is a \textit{compound value}
Positions

• A **posn** is

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

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

• A posn is

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

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

Examples:

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

\((\text{make-posn } 17 \ 0)\)
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.
The **posn-x** and **posn-y** operators extract numbers from a **posn**: 

\[
\begin{align*}
\text{(posn-x (make-posn 1 2))} & \rightarrow 1 \\
\text{(posn-y (make-posn 1 2))} & \rightarrow 2
\end{align*}
\]
posn-x and posn-y

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

\[
\begin{align*}
(\texttt{posn-x (make-posn 1 2)}) & \rightarrow 1 \\
(\texttt{posn-y (make-posn 1 2)}) & \rightarrow 2 
\end{align*}
\]

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

\[
\begin{align*}
(\texttt{posn-x (make-posn X Y)}) & \rightarrow X \\
(\texttt{posn-y (make-posn X Y)}) & \rightarrow Y 
\end{align*}
\]
Positions and Values

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

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

Yes.

A posn is

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

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

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

\textbf{No.} \texttt{(+ 1 2)} is not a \texttt{num}, yet.
Positions and Values

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

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

- Two more evaluation rules:

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

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

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

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

• Two more evaluation rules:

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

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

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))
Programmer-Defined Compound Data
Other Kinds of Data

Suppose we want to represent snakes:

- name
- weight
- favorite food

What kind of data is appropriate?
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)
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
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 DrScheme

We can tell DrScheme about snake:

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

Here’s what we’d like:

A snake is

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

But make-snake is not built into DrScheme

We can tell DrScheme about snake:

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

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 DrScheme

We can tell DrScheme about snake:

(define-struct snake (name weight food))

Creates the following:

(snake-name (make-snake X Y Z)) \rightarrow X
(snake-weight (make-snake X Y Z)) \rightarrow Y
(snake-food (make-snake X Y Z)) \rightarrow Z