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*:

\[
\begin{align*}
2 + 3 & \to 5 \\
4 \times 2 & \to 8 \\
\cos(0) & \to 1
\end{align*}
\]

• Rules for combining other rules:
  ○ Evaluate sub-expressions first

\[
4 \times (2 + 3) \to 4 \times 5 \to 20
\]

○ Precedence determines subexpressions:

\[
4 + 2 \times 3 \to 4 + 6 \to 10
\]
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 \textit{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>$4 + 2 \times 3$</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

<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = \cos(x) + 2$</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>$f(0)$</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

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

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

\[
\begin{align*}
(f \ x) &= (+ (\cos x) 2) \\
(f \ 0) &\rightarrow (+ (\cos 0) 2)
\end{align*}
\]
Evaluation is the Same as Before

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

\[
(f \ 0)
\]

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

\[
\rightarrow \ (\text{+} \ 1 \ 2)
\]
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>$(&lt; 1 \ 2) \rightarrow true$</td>
</tr>
<tr>
<td>$1 &gt; 2$ → true</td>
<td>$(&gt; 1 \ 2) \rightarrow false$</td>
</tr>
<tr>
<td>$1 &gt; 2$ → true</td>
<td>$(&gt; 1 \ 2) \rightarrow false$</td>
</tr>
<tr>
<td>$2 \geq 2$ → true</td>
<td>$(\geq 1 \ 2) \rightarrow true$</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

(symbol=? 'apple 'apple) → true
(symbol=? 'apple 'banana) → false
Beyond Numbers: Images

(rectangle 35 35 'filled 'red) → rectangle

(circle 13 'filled 'blue) → circle

(overlay circle rectangle) → overlay

(overlay/xy circle -5 -5 rectangle) → overlay/xy

(image=? (overlay circle rectangle) rectangle) → image=?

→ (image=? rectangle rectangle)

→ true
(define (anonymize i)
  (overlay/xy
   (circle (/ (image-height i) 3)
     'solid
     'blue)
   (* -1/6 (image-height i))
   (* -1/6 (image-width i))
   i))
Conditionals
Conditionals in Algebra

General format of conditionals in algebra:

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

Example:

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

abs(10) = 10

abs(-7) = 7
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{aligned}
  &\text{[true answer]} \\
  &\text{...}
  &\text{[question answer]}
  \end{aligned}
  \rightarrow \text{answer}
\]

- Keep only the first answer

Example:

\[(\times 1 \ (\text{cond} 
  \begin{aligned}
  &\rightarrow (\times 1 0) \rightarrow 0
  &\text{[true 0]})
  \end{aligned}
\]
Evaluation Rules for cond

First question is literally \texttt{false}

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

\cdot \text{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

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

...  

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

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

- Evaluate first question as sub-expression

Example:

\[
(+ 1 \quad (\text{cond} \quad \text{[(< 1 2) 5]} \quad \rightarrow \quad (+ 1 \quad (\text{cond} \quad \text{[true 5]} \quad \rightarrow \quad (+ 1 \quad 5) \rightarrow 6
\]

\[
\text{[else 8]})) \quad \rightarrow \quad \text{[else 8]}))
\]
Evaluation Rules for cond

Only queston is false answers

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

(image-inside? ) → true

(image-inside? ) → false
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]))

(detect-person

"should be"
Compound Data
Finding and Adjusting Images

Suppose we want to write *frame-person*:

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

"should be"

Need an operator that reports *where* an image exists
Finding an Image Position

\texttt{find-image : image image \rightarrow num num}

Must return a single value

Correct contract:

\texttt{find-image : image image \rightarrow posn}

- A \texttt{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

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 \texttt{X} and \texttt{Y}:

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

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

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 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

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

But \text{make-snake} is not built into DrScheme

We can tell DrScheme about \text{snake}:

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

Creates the following:

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