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***:

$$2 + 3 \quad \rightarrow \quad 5$$

$$4 \times 2 \quad \rightarrow \quad 8$$

$$\cos(0) \quad \rightarrow \quad 1$$

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

$$4 \times (2 + 3) \quad \rightarrow \quad 4 \times 5 \quad \rightarrow \quad 20$$

  - Precedence determines subexpressions:

$$4 + 2 \times 3 \quad \rightarrow \quad 4 + 6 \quad \rightarrow \quad 10$$

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

Old	**New**
$1 + 2$	`(+ 1 2)`
$4 + 2 \times 3$	`(+ 4 (* 2 3))`
$\cos(0) + 1$	`(+ (cos 0) 1)`

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))`

- Move open parenthesis in function calls

Old	New
f(0)	`(f 0)`
f(2+3)	`(f (+ 2 3))`

Evaluation is the Same as Before

```
(define (f x) (+ (cos x) 2))
(f 0)
```

Evaluation is the Same as Before

```
(define (f x) (+ (cos x) 2))
```

```
(f 0)
 → (+ (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:

Old		New	
$1 < 2$	$\rightarrow$ true	`(< 1 2)`	$\rightarrow$ `true`
$1 > 2$	$\rightarrow$ true	`(> 1 2)`	$\rightarrow$ `false`
$1 > 2$	$\rightarrow$ true	`(> 1 2)`	$\rightarrow$ `false`
$2 \geq 2$	$\rightarrow$ true	`(>= 1 2)`	$\rightarrow$ `true`

Beyond Numbers: Booleans

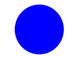
Old	New
true and false	`(and true false)`
true or false	`(or true false)`
$1 < 2$ and $2 > 3$	`(and (< 1 2) (> 2 3))`
$1 \leq 0$ and $1 = 1$	`(or (<= 1 0) (= 1 1))`
$1 \neq 0$	`(not (= 1 0))`

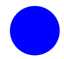
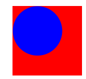
Beyond Numbers: Symbols

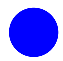
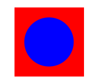
```
(symbol=? 'apple 'apple)  →  true
(symbol=? 'apple 'banana)  →  false
```

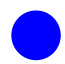
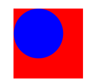
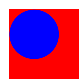
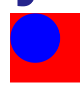
Beyond Numbers: Images

(rectangle 35 35 'filled 'red) → 

(circle 13 'filled 'blue) → 

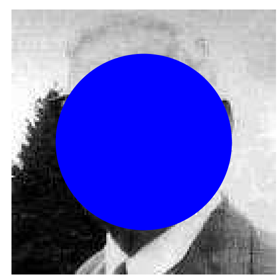
(overlay  **)** → 

(overlay/xy  **-5 -5** **)** → 

(image=? (overlay  **)** **)**
→ **(image=?**  **)**
→ **true**

Programming with Images

```
(define (anonymize i)
  (overlay/xy
   (circle (/ (image-height i) 3)
           'solid
           'blue)
   (* -1/6 (image-height i))
   (* -1/6 (image-width i))
   i))
```

(anonymize ) → ... → 

Conditionals

Conditionals in Algebra

General format of conditionals in algebra:

$$\begin{cases} \textit{answer} & \textit{question} \\ \quad \ldots & \\ \textit{answer} & \textit{question} \end{cases}$$

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`**

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

- Keep only the first answer

Example:

```
(* 1 (cond          →  (* 1 0)  →  0
       [true 0]))
```

Evaluation Rules for cond

First question is literally **false**

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

- Throw away the first line

Example:

```
(+ 1 (cond               →  (+ 1 (cond
       [false 1]                       [true 17]))
       [true 17]))

                              →  (+ 1 17)  →  18
```

Evaluation Rules for cond

First question isn't a value, yet

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

where *question* → *nextques*

- Evaluate first question as sub-expression

Example:

```
(+ 1 (cond               → (+ 1 (cond
       [(< 1 2) 5]                 [true 5]
       [else 8]))                  [else 8]))

                            → (+ 1 5) → 6
```

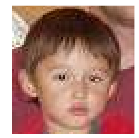
Evaluation Rules for cond

Only queston is false answers

```
(cond
 [false 10])
```

→ *error: all questions false*

Finding Images

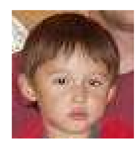
(image-inside?  ) → true

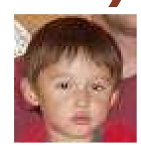
(image-inside?  ) → false

Image Tests in Conditionals

Now we can combine such operators with **cond**:

```
; detect-person : image image image -> image
; Returns a 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]))
```

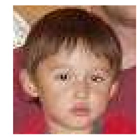
(detect-person    **)**

"should be" 

Compound Data

Finding and Adjusting Images

Suppose we want to write **`frame-person`**:

`(frame-person`   **`)`**

`"should be"` 

Need an operator that reports *where* an image exists

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***

Positions

- A **posn** is

  ```
  (make-posn X Y)
  ```

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

Examples:

```
(make-posn 1 2)
(make-posn 17 0)
```

A **posn** is a value, just like a number, symbol, or image

posn-x and posn-y

The **`posn-x`** and **`posn-y`** operators extract numbers from a **`posn`**:

```
(posn-x (make-posn 1 2))  →  1
(posn-y (make-posn 1 2))  →  2
```

- General evaluation rules for any **`X`** and **`Y`**:

```
(posn-x (make-posn X Y))  →  X
(posn-y (make-posn X Y))  →  Y
```

Positions and Values

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

Yes.

A `posn` is

```
(make-posn X Y)
```

where `X` is a `num` and `Y` is a `num`

Positions and Values

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

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

- Two more evaluation rules:

$$\texttt{(make-posn X Y)} \rightarrow \texttt{(make-posn Z Y)}$$
when $\texttt{X} \rightarrow \texttt{Z}$

$$\texttt{(make-posn X Y)} \rightarrow \texttt{(make-posn X Z)}$$
when $\texttt{Y} \rightarrow \texttt{Z}$

Example:

$$\texttt{(make-posn (+ 1 2) 200)} \rightarrow \texttt{(make-posn 3 200)}$$

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))
```

Copy

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

```
(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))  →  X
(snake-weight (make-snake X Y Z))  →  Y
(snake-food (make-snake X Y Z))  →  Z
```