Language

This page provides a summary of the programming language used in this course: Beginning Student Language in How to Code: Simple Data, then progressing to Intermediate and Advanced in How to Code: Complex Data.

Values

Numbers:	`1, 3.5, 1/2, #i1.4142135623730951, ...`
Strings:	`"Marvolo", "Black", "carrot", ...`
Images:	▲, 🟥,...
Booleans:	`true, false`
Compound data:	`(make-person "Claude" "Monet"), ...`
Lists:	`empty, (cons 2 (cons 1 empty)), (cons "x" (cons "y" (cons "z" empty))), ...`

NOTE: The primitive types are: `Number, Integer, Natural` (Integers greater than or equal to 0), `String, Image and Boolean`
2htdp/image also provides a primitive `Color` type, and 2htdp/universe provides primitive `KeyEvent` and `MouseEvent` types.

Primitive Operations

```
+, -, *, / ...
string-append, string-length, substring ...
circle, square, overlay, above, beside...
not, =, <, >, string=?, string<?, cons, first, rest, empty?,
cons?
```

Forming Expressions

Form	Example
`<value>`	`3`
`<name-of-defined-constant>`	`WIDTH`
`(<name-of-primitive-operation> <expression> ...)`	`(+ 2 (* 3 6))`
`(<name-of-defined-function> <expression> ...)` A function call should have the same number of operands as parameters.	`(yell "hello")`

`(if <question>`
`    <true-answer>`
`    <false-answer>)`
 	
`<question>` must be an expression that evaluates to a boolean.
`<true-answer>` and `<false-answer>` must be expressions.
	`(if (> (string-length x) 3)`
	`    "long"`
	`    "short")`

`(cond [<question> <answer>]`
`      ...)`
 	
Each `<question>` must be either `else` or an expression that evaluates to a boolean.
Each `<answer>` must be an expression.
	`(cond [(> x y) "more"]`
	`      [(< x y) "less"]`
	`      [else "same"])`

`(and <question> ...)` Each `<question>` must be an expression that evaluates to a boolean.	`(and (< 0 x) (>= x 10))`

Form	Example
`(or <question> ...)` Each `<question>` must be an expression that evaluates to a boolean.	`(or (< x 0) (> x 10))`

`Intermediate Student Language`
 	
`(local [<definition>...]`
`  <expression>)`
 	
Any function or constant defined within the `local` is valid within the entire body of the local expression, but not outside of the local expression.
	`(local [(define DOT (circle 5 "solid" "red"))`
	`        (define (add-dot img)`
	`          (beside img DOT))]`
	`  (add-dot (square 20 "solid" "blue")))`

`Intermediate Student Language`
 	
`(shared [(<variable-name> <expression>)...]`
`  <expression>)`
 	
Shared is a special version of local that makes it possible to build circular structure. Any variable defined within the shared is valid within the entire body of the shared expression, but not outside of the shared expression. By convention the variable names that are used in circular references are given names of the form -1- -2- etc. But this is just a convention, any variable name can be used.
	`(shared [(-1- (list -2-))`
	`         (-2- (list -1-))]`
	`  -1-)`

Forming Definitions

```
(define SIZE (* 3 6))         ;a constant definition, the value of SIZE is 18
```

```
(define (bulb c)              ;defines a function named bulb, with parameter c
  (circle 30 "solid" c))      ;this is the body of the function
```

```
(define-struct wand (wood core length))  ;defines the
functions below:

; constructor: make-wand
; selectors:   wand-wood, wand-core, wand-length
; predicate:   wand?
```

Evaluation Rules

For a **constant reference**, such as `SIZE`:

- The constant reference evaluates to the defined value of the constant.

For a **call to a primitive** such as `(+ 2 (* 3 6))`:

- First reduce the operands to values: proceed left to right making sure all the operands are values, for any that are not, evaluate them. These values are called the arguments to the primitive.
- Apply the primitive to those arguments.

For a **call to a defined function** such as `(bulb (string-append "r" "ed"))`:

- First reduce the operands to values (as for a call to a primitive). These values are called the arguments to the function.
- Replace the function call expression with the body of the function in which every occurrence of the parameter(s) has been replaced by the corresponding argument(s).

For example:

```
(bulb (string-append "r" "ed"))
(bulb "red")
(circle 30 "solid" "red")
```

For an **if expression**:

- If the question is not a value, evaluate it and replace it with its value.
- If the question is `true`, replace the entire if expression with the true answer expression.
- If the question is `false`, replace the entire if expression with the false answer expression
- If the question is a value other than `true` or `false`, signal an error

For example:

```
(if (> (+ 1 2) 3)
    (* 2 3)
    (* 3 4))      ;since (> (+ 1 2) 3) is an expression, not
a value,
                  ;evaluate it left to right
(if (> 3 3)
    (* 2 3)
    (* 3 4))

(if false
    (* 2 3)
    (* 3 4))
                  ;replace entire if expression with the
false answer expression
(* 3 4)
                  ;evaluate false answer expression
12
```

For a **cond expression**:

- If there are no question/answer pairs, signal an error.
- If the first question is not a value, evaluate it and replace it with its value. That is, replace the entire `cond` with a new `cond` in which the

first question has been replaced by its value.

- If the first question is `true` or `else`, replace the entire `cond` expression with the first answer.
- If the first question is `false` drop the first question/answer pair; that is, replace the `cond` with a new `cond` that does not have the first question/answer pair
- Since the first question is a value other than `else`, `true` or `false`, signal an error.

For example:

```
(cond [(> 3 3) "more"]
      [(< 3 3) "less"]
      [else "same"])
                        ;the first question is not a value,
the expression
                        ;(> 3 3) is evaluated and replaced
with a value
(cond [false "more"]
      [(< 3 3) "less"]
      [else "same"])
                        ;the first question is false, so the
first
                        ;question/answer pair is dropped
(cond [(< 3 3) "less"]
      [else "same"])
                        ;the first question is not a value,
so (< 3 3)
                        ;is evaluated and replaced with its
value
(cond [false "less"]
      [else "same"])
                        ;the first question is false, so the
first
                        ;question/answer pair is dropped
(cond [else "same"])
                        ;since the question is else, the
entire cond expression
                        ;is replaced by the answer
"same"
```

For an **and expression**:

- If there are fewer than 2 operands an error is signalled.
- Evaluate the operands one at a time, left to right, replacing each operand with its value.
- Except that as soon as an operand evaluates to `false` then

immediately produce `false` from the entire `and` expression.

- If all operands evaluate to `true` then produce `true` from the entire `and` expression.

For example:

```
(and (< 0 3)
     (< 3 10))    ;since (< 0 3) is an expression, not a
value,
                  ;evaluate it
(and true
     (< 3 10))

(and true
     (< 3 10))    ;now evaluate (< 3 10)

(and true
     true)

true              ;entire and produces true
```

For an **or expression**:

- If there are fewer than two operands an error is signalled.
- Evaluate the operands one at a time, left to right, replacing each operand with its value.
- Except that as soon as an operand evaluates to `true` then immediately produce `true` from the entire `or` expression.
- If all operands evaluate to `false` then produce `false` from the entire `or` expression.

For example:

```
(or (< 14 0)      ;evaluate (< 14 0)
    (> 14 10))
(or false
    (> 14 10))    ;now evaluate (> 14 10)

(or false
    true)         ;operand produced true

true              ;so entire or produces true
```

Intermediate Student Language

For a **local expression**:

- For each locally defined function or constant, rename it and all references to it to a globally unique name, and
- **in the same step** lift the local definition(s) to the top level with any existing global definitions, and
- **in the same step** replace the local expression with the body of the local in which all references to the defined functions and constants have been renamed.

For example:

```
(define b 1)
(+ b
   (local [(define b 2)]
     (* b b))
   b)
                        ;b evaluates to its defined value, 1



(define b 1)
(+ 1
   (local [(define b 2)]
     (* b b))
   b)
                        ; since b is a locally-defined
constant,
                        ;it is renamed to a globally unique
name b_0
                        ;the local definition of b_0 is
lifted to
                        ;the top level and the entire local
expression
                        ;is replaced by its body

(define b 1)
(define b_0 2)          ;---this renamed define was lifted

(+ 1
  (* b_0 b_0)           ;---entire local replaced by renamed
body
  b)

                        ;evaluation continues normally from
this point
```

For a **shared expression**:

The evaluation of shared is truly one of the deep mysteries of the universe. There are two very different ways to describe the evaluation rules.

One cannot be described exactly in terms of BSL (or ISL), but informally goes as follows:

- Each of the variables is defined as a constant with a special unique dummy value.
- Then the expressions are evaluated.
- Then each of the constants has its value change to the corresponding value. This is the part we have no words for in BSL or ISL, it is called mutation.
- Finally each of those values is traversed and each time one of the dummy values appears it is replaced in-situ by the corresponding value. (A different form of mutation.)

This evaluation rule for shared is cumbersome but goes beyond what we know.

The other evaluation rule is truly profound, see The Why of Y for an explanation.

Built-In Abstract Functions

ISL and ASL have the following built-in abstract functions.

```
(@signature Natural (Natural -> X) -> (listof X))
;; produces (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

(@signature (X -> boolean) (listof X) -> (listof X))
;; produce a list from all those items on lox for which p
holds
(define (filter p lox) ...)

(@signature (X -> Y) (listof X) -> (listof Y))
;; produce a list by applying f to each item on lox
;; that is, (map f (list x-1 ... x-n)) = (list (f x-1) ... (f
x-n))
(define (map f lox) ...)

(@signature (X -> boolean) (listof X) -> Boolean)
;; produce true if p produces true for every element of lox
(define (andmap p lox) ...)

(@signature (X -> boolean) (listof X) -> Boolean)
;; produce true if p produces true for some element of lox
(define (ormap p lox) ...)

(@signature (X Y -> Y) Y (listof X) -> Y)
;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n
base))
(define (foldr f base lox) ...)

(@signature (X Y -> Y) Y (listof X) -> Y)
;; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1
base))
(define (foldl f base lox) ...)
```