In the last class, we covered the primitive data types. Primitive means this data is considered to be in simplest form and is to be evaluated no further – a valid input or output.

By most definitions, atomic primitive types are composed of only 1 piece of data that cannot be separated into pieces: numbers, booleans, characters, void, empty list, and quoted symbols.

Compound primitive types are those that can be broken down into smaller primitives: strings, non-empty lists, functions, images, and vectors.

When I introduced the lambda special form in the last lecture, I neglected to mention some important requirements:

A lambda expression has the following syntax

\[
\text{(lambda } (C_1 \ C_2 \ldots \ C_n) \ B) \ \text{where:}
\]

- each \( C_i \) is a character sequence denoting some Scheme parameter name,
- the names, \( C_1 \ C_2 \ldots \ C_n \), are distinct (i.e., there are no duplicate parameter names); and
- \( B \) is a character sequence denoting a Scheme datum, \( D \), of any kind (the body). Not evaluated until the function is called.

Thus, \( C_1, C_2, \ldots, C_n \) specify \( n \) distinct input parameters for the lambda expression, and \( B \) specifies the body of the lambda expression.

The following are well-formed lambda expressions:

- (lambda () 44)
- (lambda (x) (* x x))
- (lambda (w h) (* w h))
- (lambda (r h) (* 1/3 3.14159 r r h))
- (lambda (x y z) (* x (- y z)))

The body of a function is the return value.

The body of a lambda expression is evaluated only when the function is called on some inputs.

The following are malformed lambda expressions (see if you can describe why they are malformed):

- (lambda (x y x) (* x y))
- (lambda (x 10) (* x 10))
- (lambda x)

Calling a function creates a local environment in which parameter names and values are stored.

The result of evaluating ((lambda (x) (* x x)) 4)
The quote special form

The quote special form is used to shield data from default evaluation. It is used like this:
(quote abc) which is equivalent to 'abc
(the ' is the apostrophe).
quote is very useful for specifying non-empty lists: '3 42 9 is the same as (list 3 42 9)

Comparison functions

Many functions generate booleans and some primitive types have their own pre-defined comparison operators:
numbers: = <= < => > (all take n inputs)
      char-ci>=? char-ci?> char-ci>?
      char>=? char>? (ci is for case-insensitive)
booleans: boolean=?
symbols: symbol=?

strings: string=? string<? string<=?
        string>? string=>?
        string-ci=? string-ci<?
        string-ci<=? string-ci>?
        string-ci>=?
(ci is for case-insensitive)

The only comparison operators that are followed by a ? are started with alphabet characters. There is no
It is important to use the right comparison operator for the type of data you are using.

Predicate Functions

The convention used in Scheme is that every predicate function name ends in a ? (huh).

Predicates that compare values take a particular primitive type and return a boolean. Others, the logical operators, take any number of boolean expressions and return a boolean.
**Predicate Functions**

A function that always returns a boolean is known as a predicate. We have seen type-checker predicates and comparison predicates. There are also special forms that are applied to booleans and that generate a boolean. These are the special form logical operators AND and OR, and the function NOT.

AND requires both conditions to be true. OR requires at least one condition is true. NOT is a denial that a condition is true.

---

**The AND and OR special forms**

AND and OR are known as logical operators, but they do not follow the default evaluation rule.

AND returns true iff all its arguments are true. The AND special form is used as follows:

\[
\text{AND} \ (< \ 4 \text{ x} \ 3) (\geq \ 10 \text{ 100})
\]

OR returns false iff all its arguments are false. The OR special form is used as follows:

\[
\text{OR} \ (< \ 4 \text{ x} \ 3) (\geq \ 10 \text{ 100})
\]

---

**What are the results of evaluating**

\[
\begin{align*}
& (> \text{ x} \ 3) \\
& (\text{and} (> \text{ x} \ 3) (> \text{ x} \ 3)) \\
& (= (> \text{ x} \ 3) \text{ x})
\end{align*}
\]

for (a) x = 4, (b) x = 2, and (c) x = 7/2

Define a function that takes one argument, \( n \), and returns true if \( n = 5 \).

Define a function that takes argument \( n \) and returns true if \( n \) is greater than or equal to 5 and less than 6.

---

**Printing functions**

The first lab includes a function to write text as purple side-effect printing. This is done to display your lab output neatly.

The DISPLAY function takes a string as an argument and writes it as side-effect printing to the output window. It returns void. The function allows you to enter "escape characters" into the string. We use the \( \text{\backslash n} \) escape character in a string to add a newline to the printing.

There is another side-effect-only printing function we will use, and that is thePRINTF function. The printf function allows for more escape characters to be included in its string argument: ~a (or ~A) to embed a value in the string and ~% , ~n, or \( \text{\backslash n} \) for newline.

There is also a 0-parameter NEWLINE function that inserts a blank line in the output. Use it like this:

\[
\text{NEWLINE}
\]

---
Testing expressions
The htdp languages allow us to write tests for our functions. Your program may contain many different functions, and each function should be thoroughly tested. The CHECK-EXPECT function should be used to test functions that do not have side-effects. For example,

(check-expect (expt 2 6) 64)
(check-expect (expt 2 0) 1)
All check-expect statements are run AFTER your program and the result are reported at the bottom of the file.

The check-expect function works with Swindle too, but you need to have the following statement at the top of the program:

(require test-engine/racket-tests)
and, at the bottom of the program:

(test)
Any tests that do not pass stop the program execution and show you which test failed. You need to fix failed tests from the top down.

Program Contracts
(display "CONTRACT: (decrement-then-square x) --> number"
"INPUT: x is a number"
"OUTPUT: a number, subtract 1 from x and then square the result"
"SIDE-EFFECTS: none \(\text{w}^\text{r}\)"
This known as a function contract.
Function contracts are often written entirely as comments, but doing it that way does not show the contract with the output.
Writing the contract in a display statement as shown will put the contract in the interactions window output.

Note about semicolons:
All characters typed to the right of a semicolon (;) are called comments.
Using comments in your code is essential. They are often the only way another person can understand the intent of your code.
There are other ways to comment code, but using ; is the standard method.

Comments are ignored by the computer...they are only information that can be read by anyone reading your program. They may seem redundant or just statements of the obvious, but they are necessary.

DrRacket colors comments a shade of brown ... different from any other statements in your code.

Comments inside functions
When writing a function, it is a good idea to include comments, either between lines of code or to the right of the code.
Comments are not evaluated. They are meant to make the purpose of the code clear to anyone reading it. They are essential.