# Racket Notes—Fall 2013

Original by Luke Hunsberger (this version edited by Jenny Walter)

## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>I Introduction</td>
<td>4</td>
</tr>
<tr>
<td>II Primitive (Atomic) Data Types</td>
<td>5</td>
</tr>
<tr>
<td>II-A Primitive (built-in) functions</td>
<td>5</td>
</tr>
<tr>
<td>II-B Numbers</td>
<td>5</td>
</tr>
<tr>
<td>II-C Quoted Symbols</td>
<td>6</td>
</tr>
<tr>
<td>II-D Booleans</td>
<td>6</td>
</tr>
<tr>
<td>II-E The Empty List</td>
<td>7</td>
</tr>
<tr>
<td>II-F Characters</td>
<td>7</td>
</tr>
<tr>
<td>II-G Void data type</td>
<td>7</td>
</tr>
<tr>
<td>III Evaluating Primitive Types</td>
<td>8</td>
</tr>
<tr>
<td>III-A Expression Evaluation</td>
<td>8</td>
</tr>
<tr>
<td>III-B Evaluating Symbols</td>
<td>9</td>
</tr>
<tr>
<td>III-C The Interactions and Definitions Windows</td>
<td>10</td>
</tr>
<tr>
<td>III-D The Global Environment</td>
<td>11</td>
</tr>
<tr>
<td>III-E Evaluation of Primitive Data</td>
<td>12</td>
</tr>
<tr>
<td>III-F Evaluation of Primitive Functions</td>
<td>12</td>
</tr>
<tr>
<td>III-G Evaluation of Mathematical Expressions</td>
<td>13</td>
</tr>
<tr>
<td>Prefix versus Infix Notation</td>
<td>13</td>
</tr>
<tr>
<td>Process to convert infix to prefix</td>
<td>14</td>
</tr>
<tr>
<td>Looking up primitive functions</td>
<td>15</td>
</tr>
<tr>
<td>IV Compound Data Types</td>
<td>16</td>
</tr>
<tr>
<td>IV-A Strings</td>
<td>16</td>
</tr>
<tr>
<td>IV-B Images</td>
<td>16</td>
</tr>
<tr>
<td>IV-C Non-empty Lists</td>
<td>16</td>
</tr>
<tr>
<td>IV-D Evaluation of Non-Empty Lists (the Default Rule)</td>
<td>17</td>
</tr>
<tr>
<td>Formal definition of the Default Rule</td>
<td>19</td>
</tr>
<tr>
<td>IV-E Structs</td>
<td>20</td>
</tr>
<tr>
<td>IV-F Vectors</td>
<td>20</td>
</tr>
<tr>
<td>V Special Forms: define and lambda (λ)</td>
<td>21</td>
</tr>
<tr>
<td>V-A The define Special Form: Naming values in the global environment</td>
<td>21</td>
</tr>
<tr>
<td>V-B The Lambda Special Form: Creating functions</td>
<td>23</td>
</tr>
<tr>
<td>Syntax of λ Expression</td>
<td>23</td>
</tr>
<tr>
<td>Semantics of λ Expression</td>
<td>24</td>
</tr>
<tr>
<td>Unnamed λ Expressions</td>
<td>24</td>
</tr>
<tr>
<td>Summary of Using Unnamed lambdas</td>
<td>25</td>
</tr>
<tr>
<td>V-C Using the define Special Form to create named functions</td>
<td>25</td>
</tr>
<tr>
<td>Function naming conventions</td>
<td>26</td>
</tr>
<tr>
<td>Distinguishing Arguments from Parameters</td>
<td>26</td>
</tr>
<tr>
<td>Local versus Global Environment</td>
<td>27</td>
</tr>
<tr>
<td>VI Introduction to Writing and Running a Program</td>
<td>28</td>
</tr>
<tr>
<td>VI-A Comments/Documentation</td>
<td>28</td>
</tr>
<tr>
<td>Introduction to documenting functions</td>
<td>28</td>
</tr>
<tr>
<td>VI-B Indentation, Code Coloring, and Line Length</td>
<td>29</td>
</tr>
</tbody>
</table>
### VII Printing Strings
- VIII-A String functions ........................................... 30
- VIII-B The print, display, and newline functions ................. 30
- Escape Characters .................................................. 30
- Multiple printfs in function body .................................. 31
- Debugging ........................................................... 32

### VIII Asking Questions with Predicate Functions
- VIII-A Type-checker predicates .................................... 33
- VIII-B Arithmetic Predicates: Relational Operators ............. 33

### IX Making Decisions—Special Forms: cond, if, and, or
- IX-A The if Special Form: The basic yes/no decision-maker .... 35
- Using an if expression in a function ............................... 36
- IX-B The Boolean (Logical) Operators: and, or, not ............. 36
- The not function ................................................... 36
- The and special form ............................................. 36
- The or special form ............................................... 37
- IX-C Making Multiple Choice Decisions: The cond Special Form 37
- Function with cascading if statement ............................. 37
- Function using cond instead of if ................................ 38
- IX-D Using the Stepper to Trace Execution of Functions ....... 39

### X The Design Recipe: How to write a function
- X-A Pre-function Comments ........................................ 40
- X-B Pre-Function Testing: The check-expect and check-within Special Forms 40
- Writing pre-function tests ......................................... 40
- X-C Function definition and parameter naming ..................... 41
- Naming parameters .................................................. 42
- Comments inside functions ....................................... 42
- X-D Post-function testing — the TESTER function ............ 42
- X-E Literal values in function definitions ......................... 43
- Defining Constant Values before Functions ..................... 43

### XI Adding chance to functions: the random Function ....... 45

### XII Recursion
- XII-A Recursive Functions ......................................... 46
- Function with Infinite Loop ....................................... 46
- General rules for recursive functions ............................. 46
- Steps in defining recursive functions ............................ 46
- XII-B Recursion over numbers ..................................... 47
- The factorial function ............................................ 47
- Recursion and local environments .................................. 49
- Summary of recursive function features ......................... 50
- XII-C Accumulator-style (Tail-Recursive) Functions over numbers 51
- XII-D Wrapper functions ........................................... 52
- Information Hiding .................................................. 53

### XIII A Self-Referential Data Type: Flat Lists
- XIII-A I thought we were already using lists... ................ 54
- XIII-B Using cons to Create Pairs (i.e., Cons Cells) ............. 55
- XIII-C Picturing Non-Empty Lists ................................ 55
- XIII-D The Self-Referential Structure of Lists & List Recursion 56
- XIII-E Accumulator versions of functions that consume lists .... 57
- XIII-F Functions that return lists ................................ 58
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XIV</td>
<td>Lists Within Lists—Deep Lists</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>XIV-A Arbitrarily Nested Lists</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>XIV-B Lists Containing Lists of Equal size</td>
<td>59</td>
</tr>
<tr>
<td>XV</td>
<td>Structs: Containers for fixed-size data</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>XV-A Defining new data types—defstruct</td>
<td>60</td>
</tr>
<tr>
<td>XVI</td>
<td>Local Variables and Local Functions—the let and local special forms</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>XVI-A The let special form</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>The syntax of let</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>The semantics of a let expression</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>XVI-B The local/define special form</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>XVI-C More recursion: Keeping track of indices to process strings</td>
<td>66</td>
</tr>
<tr>
<td>XVII</td>
<td>Higher-order functions</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>XVII-A The filter higher-order function</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>XVII-B The map higher-order function</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>XVII-C The apply and foldl higher-order functions</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>XVII-D The build-list higher-order functions</td>
<td>70</td>
</tr>
<tr>
<td>XVIII</td>
<td>Interactive programs: Using the read function</td>
<td>71</td>
</tr>
<tr>
<td>XIX</td>
<td>Vectors: Fast access data containers</td>
<td>73</td>
</tr>
</tbody>
</table>
I. Introduction

Most kinds of communication are based on some kind of language, whether written, spoken, drawn or signed. To be used successfully, the syntax and semantics of a language must be understood.

- The syntax rules of a language specify the legal words, expressions, statements or sentences of that language.
- The semantic rules of a language specify what the legal words, expressions, statements or sentences mean.

For example, the syntax rules of the English language tell us that person, tall, told, the, a, me and joke are legal words, and that The tall person told me a joke is a legal sentence, whereas pkrs, shrel and fdadfa are not legal words, and Person tall told a me the is not a legal sentence. The semantic rules tell us what each of the words mean (e.g., what objects the nouns denote and what processes the verbs convey), as well as what the entire sentence means (i.e., that a particular tall person told me a joke). In a similar way, people use a programming language to communicate with a computer. And each programming language has an associated set of syntax rules that specify the legal expressions (or statements or sentences or programs) that can be used in that language, and a set of semantic rules that specify what the legal expressions mean (i.e., what computations the computer will perform). For most computer programming languages, the constituents of the language, whether they are called expressions, statements or entire programs, are usually sequences of typewritten characters.

Although people can effectively communicate using the English language based on an informal, imprecise, intuitive understanding of its syntax and semantics, trying to program a computer based on an informal, imprecise, intuitive understanding of the syntax and semantics of a given programming language typically leads to trouble. Therefore, it is important to be explicit about the syntax and semantics of the programming language being used. Indeed, while programming, it is extremely helpful to have a mental model of the computations the computer is performing. To enable us to enter the world of programming as quickly and painlessly as possible, it is helpful to use a programming language for which the syntax and semantic rules are relatively simple. Racket is just such a language.

Most computer languages display two types of error messages when a program is translated from a high level language like Racket into something the machine can read (in all 1’s and 0’s):

1) syntax error messages are displayed when either illegal words or expressions are evaluated and
2) runtime error messages are displayed when, for example, functions receive the wrong type or number of arguments.

You will undoubtably have lots of experience with error messages during this course, but errors help you learn.

Note: Racket programs are made up of keywords, variables, structured forms, literal data values (numbers, characters, strings, vectors, quoted lists, quoted symbols, etc.), whitespace, and comments. The next section presents the legal key sequences to represent different data types in Racket.
II. PRIMITIVE (ATOMIC) DATA TYPES

Any program in Racket is a sequence of characters. The syntax rules of Racket tell us which character sequences are legal to use in programs. These character sequences are known as “valid” or “legal Racket expressions”.

→ Each Racket program consists of one or more valid Racket expressions. For example, as you’ll soon discover, 3, true, #t, and ’() are legal expressions in Racket.

→ A primitive datum is a valid Racket expression that is atomic, in the sense that it does not have any smaller parts that you can access. In other words, primitive data is already in simplest form.

In Racket, each legal expression denotes a datum (i.e., a piece of data). The semantic rules of Racket tell us which datum each legal expression denotes. For example, in Racket, the legal expressions 3, #t, and ’() respectively denote the number three, the truth value true, and the empty list.

We begin with primitive data expressions. Each primitive data expression denotes a Racket datum of a particular kind. The universe of Racket data is populated by numbers, truth values (called booleans), quoted symbols, symbols that represent names and primitive functions, among many others. Importantly, each datum has a unique data type. For example, a Racket datum might be a number or a quoted symbol, but cannot be both. Stated differently, the universe of Racket data is partitioned according to data type and an item of a particular data type can only evaluate to itself.

A. Primitive (built-in) functions
Racket includes a variety of primitive (or built-in) functions that are identified by unique character sequences. The English word primitive may make you think of the Stone Age, but in mathematics the word means that something “can’t be defined in terms of anything simpler”. It is important to realize that each primitive function is a Racket datum, just like numbers and booleans. The majority of these primitive functions are written for a particular data type, and most do not work on multiple data types\(^1\). Examples of functions that consume each of the primitive data types are given in the subsections that pertain to each primitive type (sections B–G) below.

Functions in Racket are very similar to functions in mathematics, in that they consume data as input and produce data as output. Functions may also be called procedures or subroutines.

There are different ways to look up the name and usage of a particular function in DrRacket. These will be discussed in section III-F. The functions you write will frequently use one or more primitive functions to produce more complex functions tailored to specific tasks.

**Functions that consume functions:** There are primitive functions that consume other functions. These are called “higher order” functions and we will cover them later in the semester.

B. Numbers
According to the syntax rules of Racket, character sequences such as 3, -44, 34.9 and 85/6 are legal Racket expressions. According to the semantics of Racket, these expressions denote the numbers three, negative forty-four, thirty-four point nine and eighty-five sixths, respectively.\(^2\) Each number is an example of a Racket datum and each is also an example of what computer scientists call a “literal value”, meaning the character sequence cannot be used as a placeholder for different values.

As a reminder of the importance of distinguishing the character sequences you type from the data they denote, we use the following sort of notation to denote evaluation: \( \text{character sequence} \rightarrow \text{datum} \)

For example, you can use this notation to describe numeric character sequences that evaluate to numbers:

---

\(^1\)With the exception of a type of primitive functions presented in Section VIII called “type-checkers”, which work on any valid data type.

\(^2\)All numbers are presented in base 10 notation.
3 → the number 3
-44 → the number -44
85/6 → the fraction eighty-five sixths

In some cases, multiple Racket expressions denote the same datum. For example, each of the following character sequences denotes the number zero in Racket: 0, 000 and 000000.

0 → the number zero
000 → the number zero
000000 → the number zero

As programmers, you type the character sequences; however, behind the scenes, the computer is performing computations on the numbers (i.e., Racket data) denoted by these character sequences. We don’t need to know how they are represented in the computer, we can only see the input and output representations.

Racket numbers include exact and inexact integers, rationals, reals, and complex numbers. Exact integers and rational numbers have arbitrary precision, i.e., they can be of arbitrary size. Inexact numbers are preceded by #i.

Functions that consume numbers: Racket’s syntax rules for numerical expressions are quite similar to those for numerical expressions you’ve seen in math classes. The primitive functions that consume numbers include all the ones you’re already used to, such as +, −, *, /, =, >= (≥), >, <= (≤), <, and many more. In programming languages, the +, −, *, / operators are functions that consume numbers and return numbers and the =, >=, >, <=, < operators are functions that consume numbers and return either true or false (booleans).

C. Quoted Symbols
According to the syntax rules of Racket, the character sequences, ’a and ’b, are legal Racket expressions. According to the semantics of Racket, these expressions respectively denote the characters a and b. A quoted symbol is a sequence of one or more keyboard characters (excluding white space) preceded by a single quote (‘):

’cat → the quoted symbol cat
’dog → the quoted symbol dog

Like numbers, quoted symbols are character sequences that evaluate to atomic data. Their purpose is to represent things like names, job titles, and so on. Any quoted symbol is also a literal value, meaning that any quoted symbol is in simplest form and evaluates only to itself. If the same symbol is not preceded by a single quote, then it is simply a symbol or variable, and it is evaluated as a place-holder for data. This means that, without the single quote, a symbol evaluates to something that is different and is therefore not a primitive type.

Functions that consume quoted symbols: The functions you’ll use on quoted symbols include the symbol? type checker, the symbol=? equality checker for two symbols and the symbol->string function that converts a quoted symbol to a string.

D. Booleans
According to the syntax rules of Racket, the character sequences, true (or #t) and false (or #f), are legal Racket expressions. According to the semantics of Racket, these expressions respectively denote the truth values true and false, as illustrated below:

#t → the true truth value
true → the true truth value
#f → the false truth value
false → the false truth value

Keep in mind the difference between the character sequences and the truth values they denote. The boolean data type consists solely of these two truth values, unlike the infinite variety of numbers, functions, and quoted symbols. As programmers, we type the character sequences; behind the scenes, the computer is working with the corresponding truth values. The keywords true and false, and their equivalent abbreviations #t and #f, respectively, are literal values.
Functions that consume booleans: To check if two boolean expressions are the same, there is a boolean= equality checker function that consumes two boolean expressions, returning true if they are equal and false otherwise. There are also logical operators designed to consume and produce booleans: and, or, and not.

Any function ending with a ? returns either true or false, and you should follow this convention when writing a function that returns true or false.

E. The Empty List
According to the syntax rules of Racket, the character sequence consisting of a single quote, and an open and close parentheses, '(), is a legal Racket expression. According to the semantics of Racket, it denotes the null datum, which is also called the empty list, also represented by keywords empty and null.

'() → the empty list
empty → the empty list
null → the empty list

We’ll encounter uses for non-empty lists later on. The empty data type includes only this one datum, but it is a very important and frequently used datum because it allows us to detect when we have processed all the items in a list.

Functions that consume empty lists: The primitive functions used most often with the empty data type are the type checker functions empty? or null?. These functions return true only if their argument is an empty list.

F. Characters
A character in Racket is preceded by hash-backslash (#\). For example:

#\a → the letter a
#\H → the letter H

You’ll use character data later in the semester, when you start writing functions that consume strings. Any character, such as the letter c, written as #\c, is a literal value and can only evaluate to itself.

Functions that consume characters: Primitive functions that consume characters include char-upper-case?, char-lower-case?, char-numeric?, and many more.

G. Void data type
A “no value” value. Used to indicate that a function returns nothing and only has a side-effect. Side-effects include, among other things:

- printing text to the screen and
- defining variable and function names.

Several of the output functions whose purpose is to print to the screen (e.g. printf, display, and newline) have a return type of void.

The void data type is not the same as the empty list. The keywords void and empty (null) do not refer to the same entity.

Functions that consume void: There is a type-checker, void? for the void data type.
III. Evaluating Primitive Types

In this class, we will use the software package called DrRacket. DrRacket is free and can be downloaded (see the link on the course web page) and installed on any computer. DrRacket is an example of an “Integrated Developer’s Environment” (IDE), so called because it provides facilities for entering programs, running them, and testing the results, all in the same program. We will write and run our programs in DrRacket and it will be available for use on any of the machines in the CS department.

![DrRacket startup screen](image)

Fig. 1. A DrRacket startup screen. The top window is called the Definitions Window and the bottom is the Interactions Window. The language is selected by pulling down the Language menu or using the drop-down menu in the lower left corner of the window.

Fig. 1 shows the DrRacket startup screen. The top window is called the Definitions Window and the bottom is the Interactions Window (these windows may be shifted to be displayed side-by-side by choosing Use horizontal layout from the View menu).

DrRacket simulates the operation of a computer that understands the Racket programming language, enabling you to interact with that simulated computer. In effect, you use DrRacket as a communication medium between yourself and the simulated computer.

When you first open DrRacket, there will be no language chosen. You should pull down the Language menu, select “Choose Language...”, click on “Full Swindle”, and click OK. You must press the “Run” button to make the new language active.

Before starting a detailed discussion of the DrRacket IDE, you need to know a little more about how expressions are evaluated in DrRacket.

A. Expression Evaluation

Evaluation is the one and only thing that DrRacket does—namely, it evaluates Racket data. Because of this, it is important to carefully describe the evaluation process. The good news is that the process of evaluation can be described
easily.

We’ve seen that a variety of character sequences (e.g., 34, ’xyz, ’() and #t) constitute legal expressions according to the syntax rules of Racket. In addition, we’ve seen that each legal expression denotes a piece of data of a particular kind. For example, 34 denotes the number thirty-four, ’xyz denotes the quoted symbol ’xyz, ’() denotes the empty list, and #t denotes the boolean value true. The character sequences we type are expressions; the data they denote belong to the universe of Racket data. As programmers, we type character sequences; the computer deals with the corresponding Racket data.

Evaluation (eval) is a function—in the mathematical sense. In particular, the eval function takes one Racket datum as its input, and generates another Racket datum as its output, as depicted in Fig. 2.

![Fig. 2. A conceptual depiction of the DrRacket eval function. Picture courtesy of Prof. Hunsberger.](image)

The result of applying the eval function depends on the type of data that it is applied to. Thus, in what follows, we describe what the eval function does for each type of data. Here are some examples of the eval function when applied to numbers, booleans, or the empty list:

- the number 0 → the number 0
- the boolean true → the boolean true
- the empty list → the empty list

In these notes, the → arrow is reserved solely for representing the application of the eval function on some Racket datum (called the input) to generate some, possibly quite different Racket datum (called the output).

![Fig. 3. Input of the number two and the boolean true to the eval function. Picture courtesy of Prof. Hunsberger.](image)

**B. Evaluating Symbols**

In Racket, unquoted symbols (character sequences that start with a letter and contain no spaces, generally referred to only as symbols) are used as variables. In Math, variables frequently have values associated with them so that the variable is actually a placeholder for a particular value. For example, the variable \( x \) may have the value 3. So it is with Racket. For this reason, the evaluation of symbols is different from the evaluation of primitive types. In particular, symbols typically do not evaluate to themselves; instead, they evaluate to the value associated with them. (Keep reading!)

The evaluation of a symbol is based on table lookup. In particular, the eval function may be thought of as having a private table called the global environment.\(^3\) The global environment is a table that contains many entries. Each

\(^3\)Since the global environment is a private appendage of the eval function, it is not an official Racket datum and is not available for direct inspection.
entry pairs a symbol with its corresponding value (which is a Racket datum). To evaluate a symbol, the evaluation function simply looks up the value associated with that symbol in the global environment. For example, if the global environment contains an entry that associates the number two with the symbol \textit{xyz}, then the result of applying the \texttt{eval} function to the symbol \textit{xyz} will be the number two:

\[
\text{the symbol } \textit{xyz} \implies \text{the number two}
\]

The Racket datum associated with a symbol in the global environment can be of any type. Thus, it might be that the boolean \texttt{true} is associated with the symbol \textit{pq}. Similarly, the empty list might be associated with the symbol \texttt{my-empty-list}.

\[
\begin{align*}
\text{the symbol } \texttt{pq} & \implies \text{the boolean } \texttt{true} \\
\text{the symbol } \texttt{my-empty-list} & \implies \text{the empty list}
\end{align*}
\]

In Section V, you will learn how to add symbols to the global environment.

\textbf{C. The Interactions and Definitions Windows}

There are two main windows available in DrRacket:

\begin{itemize}
\item The Definitions Window, which appears at the top (or left) side of the DrRacket display. This window holds expressions that can be run many times and saved in a file. To toggle the window to appear alone/not alone, use \texttt{control-E}.
\item The Interactions Window, which appears at the bottom (or right) side of the display. This window is intended to be used for fast evaluation and to test expressions. To toggle this window to appear alone/not alone, use \texttt{control-D}.
\end{itemize}

In the interactions window you interact directly with the simulated computer of DrRacket as follows:

1) Enter a typewritten character sequence at the \texttt{>} prompt in the interactions window\textsuperscript{4}.
2) The datum denoted by that character sequence is evaluated (i.e., fed into the Racket \texttt{eval} function as input), generating an output datum.
3) DrRacket displays some typewritten text in the interactions window describing the output datum to us, as shown in Fig. 4.

\textbf{DrRacket}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{interaction-diagram.png}
\caption{Interaction with the “read-evaluate-print” loop in DrRacket’s interactions window. Picture courtesy of Prof. Hunsberger.}
\end{figure}

More formally, when you type a sequence of characters, \texttt{C_{in}}, at the \texttt{>} prompt in the interactions window, and then press the Return (or Enter) key, DrRacket does the following:

1) It figures out which Racket datum, \texttt{R_{in}}, is denoted by the character sequence \texttt{C_{in}}.
2) It reads \texttt{R_{in}} as input to the evaluation function, which generates an output datum, \texttt{R_{out}} (i.e., \texttt{R_{in}} evaluates to \texttt{R_{out}}).
3) Finally, it displays some typewritten text, \texttt{C_{out}}, in the Interactions Window that describes the output datum, \texttt{R_{out}}. The operation of DrRacket’s Interaction Window is sometimes called an “read-evaluate-print” loop.

\textsuperscript{4}When the term “enter” is used in this document, it is referring to typing a character sequence and pressing the return key.
We can describe this process more succinctly as follows:

\[ C_{in} \rightarrow [R_{in} \implies R_{out}] \rightarrow C_{out} \]

where the \( \rightarrow \) arrows represent the translation from character sequences to the denoted Racket data and back, the \( \implies \) arrow represents the application of the \texttt{eval} function, and the square brackets indicate that you don’t get to see the Racket data, \( R_{in} \) or \( R_{out} \).

Anything typed in the interactions window is evaluated when you press return. Pressing the Run button makes all text typed in the interactions window disappear.

The definitions window is used to hold code that can be run any number of times by pressing the Run button. The code in the definitions window is usually saved to a file so it can be reloaded and run at a later time.

The \texttt{eval} function is available for us to use directly in some Racket dialects (including Swindle) and it is called every time data is entered at the interactions window prompt and every time a program is run in the definitions window. While you will not write programs that use the \texttt{eval} function, you’ll see that this function is useful when testing the output of functions. In particular, the \texttt{tester} function inside the “print-and-test.rkt” file uses the \texttt{eval} function to strip the quote off quoted expressions and evaluate the unquoted result.

\textbf{D. The Global Environment}

In Racket, symbols are used as variable names, which are place-holders for actual values. In math, constants like \( \pi \) and \( e \) have values associated with them, so this concept should not be too surprising. For example, the symbol \( X \) may be associated with the value 3.

The evaluation of symbols is different from the evaluation of primitive Racket types. In particular, symbols do not evaluate to themselves; instead, they evaluate to the value that is associated with them in the global environment.

The evaluation of every symbol is based on table lookup. Each entry in the global environment pairs a symbol with its corresponding value (which is a Racket datum), as shown in Fig. 5. The global environment is populated by over 200 entries, even before you make any entries of your own.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\text{SYMBOL} & \text{VALUE} \\
\hline
\hline
\end{tabular}
\caption{Global Environment:}
\end{table}

To evaluate a symbol, the evaluation function looks up the value associated with that symbol in the global environment. For example, if the global environment contains an entry that associates the number two with the symbol \( XYZ \), then the result of evaluating the unquoted symbol \( XYZ \) will be the number two:

\[ XYZ \implies \text{the number 2} \]

The Racket datum associated with a symbol in the global environment can be of any type. Thus, it might be that the boolean \texttt{true} is associated with the symbol \( PQ \). Similarly, the empty list might be associated with the symbol \texttt{MY-EMPTY-LIST}.

\[ PQ \implies \text{the boolean true} \]
\[ \text{MY-EMPTY-LIST} \implies \text{the empty list} \]

If a symbol does not have a corresponding entry in the global environment, it is not possible to evaluate that symbol. In other words, the result of applying the \texttt{eval} function to an unquoted symbol having no entry in the global environment is \texttt{undefined} and will cause an error. When we cover the \texttt{define} special form in Section V-A, you’ll see how to
insert new entries into the global environment, thereby enabling you to create and use variables and functions of your own.

A global environment is created every time you start DrRacket. This table is stored in the computer’s memory and is not available for you to read. Initially, the global environment contains only the built-in function names and a few named constants that are not functions.

E. Evaluation of Primitive Data

You can use DrRacket to confirm some of the things discussed in previous sections. In particular, you can enter character sequences (i.e., expressions) into the interactions window and then examine the results reported by DrRacket. In each case, you will only see the character sequences you type in next to the prompt (> ) and those reported back by DrRacket on the following line; you do not get to see the Racket data manipulated by the computer. For example, the following sequence of characters typed in the interactions window demonstrates that numbers, booleans, quoted symbols, and the empty list all evaluate to themselves, but undefined symbols (like x ) cause an error to occur:

```
> 3
3
> #f
#f
> 'cat
cat
> empty
()
> x
x: undefined;
cannot reference undefined identifier
```

One way to determine if a symbol name has already been entered into the global environment is to type it (unquoted) in the interactions window\(^5\). If there is a value associated with that name in the global environment, the value will be printed, and if there is no value associated with that name, an error like the one shown above for variable x in the example above will occur. In the interactions window, DrRacket uses the > character to prompt for input. Everything following the > character in the examples shown is typed by the programmer. The text on the next line is that generated by DrRacket in response to evaluation of the character input sequence. Thus, the excerpt from the interactions window shown above displays five separate interactions.

```
3 → [the number three ⇒ the number three] → 3
#f → [the boolean false ⇒ the boolean false] → #f
'cat → [the quoted symbol cat ⇒ the quoted symbol cat] → cat
empty → [the empty list ⇒ the empty list] → ()
x → [looks for x in GE ⇒ x is not found in GE] → ERROR
```

DrRacket need not use the same character sequence as the one you entered when reporting back the evaluated character sequence in response to an entry. For example:

```
0 → [the number zero ⇒ the number zero ] → 0
000 → [the number zero ⇒ the number zero ] → 0
000000 → [the number zero ⇒ the number zero ] → 0
```

F. Evaluation of Primitive Functions

As mentioned previously, DrRacket includes a variety of primitive (or built-in) functions. Examples include the addition function, the subtraction function, the multiplication function, and many others. It is important to realize that each primitive function is a Racket datum, just like numbers and booleans.

For each symbol defined as a built-in function, there is an entry in the global environment that associates that unique symbol with the function. Therefore, the evaluation of only that particular symbol can be used to gain access to the

\(^5\) Remember, if the name is preceded by a single quote (‘), then the name is considered to be a quoted symbol and it will not be looked up in the global environment.
corresponding function. The existence of primitive functions can be confirmed by typing characters like the following in the interactions window: (the > is the prompt in the interactions window)

```
> +
#<procedure:+>
> -
#<procedure:->
> *
#<procedure:*>
```

The term procedure is just another word for function. This excerpt shows that in DrRacket, functions evaluate to themselves. Once you discover how to create Racket functions of your own design in Section V-A, you’ll be able to give your new functions names, simply by placing appropriate entries into the global environment.

G. Evaluation of Mathematical Expressions
You can use the interactions window like a glorified calculator. There are many built-in functions that can be applied to various kinds of input in this window. Each built-in function has a name that follows naming conventions and for each built-in function there is an entry in the global environment that links a particular symbol to that function. This section gives an overview of how functions are written and applied in mathematics and relates this to how functions are applied in DrRacket.

**Example:** In a math class, you might see a function defined using an equation such as \( f(x) = x \times x \). In this case, the name of the function is \( f \), and we might casually describe it as the squaring function—because for each possible input value, \( x \), the corresponding output value is the square of \( x \) (i.e., \( x^2 \)).

Notice that the definition of the function, \( f \), gives a prescription for generating appropriate output values should \( f \) ever happen to be applied to any input values. In particular, the definition of \( f \) includes an input parameter, \( x \), which is used to refer to potential input values. In addition, the expression, \( (x \times x) \), on the right-hand side of the equal sign, indicates how to compute the corresponding output value for any given value of \( x \). The expression on the righthand side of the equal sign is sometimes referred to as the body of the function.

If you wanted to know the output value generated by \( f \) when given 3 as its input, you could get the answer by first substituting the argument 3 for each occurrence of the parameter \( x \) in the expression, \( (x \times x) \), yielding \( (3 \times 3) \). Evaluating the expression, \( (3 \times 3) \), would then yield the desired output value, 9. Similarly, if you wanted to know the output value generated by \( f \) when given the input argument 4, you would first substitute the argument 4 for each occurrence of the parameter \( x \) in the expression, \( (x \times x) \), yielding \( (4 \times 4) \), which evaluates to 16.

**Another Example.** In the preceding example, the function \( f \) took a single input value. However, you can similarly define functions that take multiple inputs. For example, the function, \( g \), defined below, takes two inputs, represented by the input parameters \( w \) and \( h \): \( g(w, h) = w \times h \).

Function \( g \) can be used to compute the area of a rectangle whose width is \( w \) and height is \( h \). To apply this function to the input arguments 3 and 7, you first substitute the argument 3 for the parameter \( w \) and the argument 7 for the parameter \( h \) in the expression, \( (w \times h) \), yielding \( (3 \times 7) \). Evaluating this expression results in the desired output value, 21.

You will see that Racket functions behave in a very similar manner to mathematical functions, with input values called arguments being substituted for parameters in the body of the function.

**Prefix vs. Infix Notation**
In the examples above, substituting values for parameters is known as invoking the function on input arguments. In Racket, all function invocations (also known as function calls) are written using prefix notation, where the operator precedes its operands. This notation is discussed below.

When a function is called:
- the name of the function is preceded by an open parenthesis,
• the name of the function is followed by the input arguments to the function, separated by spaces, and
• the name and the arguments are entirely contained within a set of parentheses.

This notation is different from what you are used to from math classes, where an operator usually occurs between its operands (known as infix notation). Below are some examples of infix and equivalent prefix expressions:

<table>
<thead>
<tr>
<th>Infix Expression</th>
<th>Equivalent Prefix Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 4 * 5</td>
<td>(+ 2 (* 4 5))</td>
</tr>
<tr>
<td>(2 + 3)/5</td>
<td>(/ (+ 2 3) 5)</td>
</tr>
<tr>
<td>2 + 3/5</td>
<td>(+ 2 (/ 3 5))</td>
</tr>
</tbody>
</table>

When writing an expression using prefix notation, it is critical to maintain whitespace between operators, operands, and left parentheses. The spacing is exaggerated in the table above to emphasize the importance of blank spaces in these expressions.

An advantage of using prefix notation is that in this type of expression many operations can take more than two parameters. For example, consider the infix expression 1 + 2 + 3 + 4 + 5. We could convert this to the prefix expression (+ 1 (+ 2 (+ 3 (+ 4 5)))) or (+ (+ (+ (+ 1 2) 3) 4) 5), but, since + can take more than two arguments in Racket, you can write the original expression more succinctly as (+ 1 2 3 4 5), which is shorter than its infix form because you don’t have to repeat the + sign.

There are many primitive Racket functions that can consume any number of input arguments.

You should familiarize yourself with the process of converting infix to prefix expressions. Here’s a step-by-step technique from the book *Picturing Programs: An introduction to computer programming* by Stephen Bloch, that may help in translating an expression in “standard” infix algebraic notation into Racket’s prefix notation:

1) Expand all the abbreviations and special mathematical symbols. For example, 3x really stands for 3 * x; x² really stands for x * x; and √3x uses a symbol that we don’t have on the computer keyboard, so we’ll write it as sqrt (3 * x).

2) Fully parenthesize the expression resulting from step 1, using the usual order-of-operations rules (PEMDAS: parentheses, exponents, multiplication/division, addition/subtraction, in decreasing precedence).

By the end of step 2, the number of operators, the number of left parentheses, and the number of right parentheses should all be equal. Furthermore, each pair of parentheses should be associated with exactly one operator and its operands, so that for any operator, you can point to its enclosing left- and right-parenthesis.

3) Move each operator to a position immediately to the right of its closest enclosing left-parenthesis, leaving everything else in the same order. In particular, make sure all the operands are in the same order in the resulting prefix expression as they were in the given infix expression.

**Example 1:** Write a Racket expression to represent the arithmetic expression 3x – 4 + 5.

**Solution:** In step 1, we insert a * between the 3 and the x to get 3 * x – 4 + 5.

Step 2 tells us to “fully parenthesize, using order of operations”. Since multiplication has higher priority than addition or subtraction, we rewrite the expression as (((3 * x) – 4) + 5). Note that there are three operators (*, +, and –), three left parentheses, and three right parentheses in this expression, as required.

In step 3, we move each operator to the right of its corresponding left parenthesis, to get (+ (– (* 3 x) 4) 5), a correct Racket expression in prefix notation.

**Example 2:** Write a Racket expression to represent the arithmetic expression \(7x - \frac{(3+x)}{(y+2)}\).
**Solution:** Step 1 expands the $7x$ to $7 \times x$.

Step 2 adds parentheses around the entire fraction, around $7 \times x$, and around the whole expression, to get $(7 \times x) - ((3 + x)/(y + 2))$. Note that there are now five operators ($\times$, $-$, $+$, $/$, and $+$), five left parentheses, and five right parentheses, as required.

Step 3 moves each of the five operators immediately to the right of its corresponding left parenthesis. We can read this from left to right to get $(- (\times 7 x)/(+ 3 x)(+ y 2))$, the expression in prefix notation.

**Different ways to look up functions that are available in DrRacket:**

1) You can search a list of available functions by pulling down the Help menu and choosing Help Desk. Then click on the How to Design Programs Languages link. Next, scroll down to section 5 and click on “Advanced Student”. Scroll down again until you see the predefined functions listed down the left side of the page, grouped according to the type of data they consume. Not all the functions listed for the HtDP languages are available in Swindle, but we will find the ones that are not as we learn the language.

2) If you think you know the name of a function, type the function name in the interactions window and press enter. If the function is defined, #<procedure...> will be displayed, meaning the function name is defined and has evaluated to itself. Then you can look up the usage of the function as described in step 3.

3) Once you know the name of the function, you can pull down the Help menu, choose Help Desk, and then type the function name at the top of the first Help Desk screen, following the links to the function description.

Note that looking up a function as described in step 1 will also find the particular usage of a function (i.e., what data type the function consumes and what type it produces). Finding a function name using step 2 will only let you know the function exists. Step 3 describes how to look up the input, output, and purpose of an existing primitive function. You should familiarize yourself with looking up functions in the Help Desk by finding the descriptions of functions introduced in this write-up and in class.
IV. COMPOUND DATA TYPES

Compound data types are, as the name suggests, divisible into parts. Many computer programs are written to disassemble compound data types and manipulate their parts for different useful applications.

A. Strings

Syntactically, strings in Racket are character sequences delimited by quotation marks (double quotes). For example, “hi” and “Howdy!” are character sequences that denote string data.

```
“hello”  →  “hello”
“Vassar college” → “Vassar college”
```

A large number of programs are written to process strings, so we will devote part of the semester to reading and writing functions that consume strings. Almost any sequence of keyboard characters enclosed in quotation marks are literal string values. Note in particular that any space inside quotation marks is treated as a blank part of the string.

Strings are stored as indexed sequences of characters. For example, the string “Hello world!” could be envisioned as being stored and indexed as shown below:

```
Index:  0  1  2  3  4  5  6  7  8  9 10  11
  Hello world!
```

![Fig. 6. Conceptual storage of string “Hello world!”](image)

The index numbers for characters in a string start at 0. This is a conventional practice in computer science. Starting the index at 0 is known as zero-based indexing. Later, you will discover how to access and manipulate characters in a string using their index number.

**Primitive functions that consume strings:** There are many primitive functions that consume strings, including `string-length`, `string-append`, `string-ref`, `substring`, and many more. We will be using these string functions in future labs and assignments.

B. Images

One way to make a programming assignment or user interface more interesting is to include images. DrRacket provides us the ability to insert various types of images (e.g., `.jpg`, `.gif`) directly into our programs by choosing “Insert Image...” from the Insert menu and then navigating to the directory where we have the image stored and selecting from that directory the image we want to insert.

Images can only evaluate to themselves, so evaluating an image returns the same image (or an indication of the image as `(struct:object:image% ...)` in Swindle). We can also manipulate and name images in Racket programs. Unlike primitive data types, images have smaller parts that you can access, such as width and height and so we group them with the composite data types.

**Primitive functions that consume images:** The primitive functions we will use that consume images include `image-height`, `image-width`, and `place-image`.

C. Non-empty Lists

We have already seen one type of list—the empty list. A non-empty list is, as the name suggests, a set of parentheses enclosing other Racket data types or defined symbols. In many languages, the basic aggregate data structure is called an array. In Racket, the basic aggregate data structure is the *list*. 
Lists that are containers of data are written as sequences of objects separated by spaces, surrounded by parentheses and preceded by a single quote. For instance, ‘(1 2 3 4 5) is a list of five numbers, and ‘(“this” “is” “a” “list”) is a list of four strings. Lists need not contain only one type of object, so ‘(4.2 “hi” #f) is a valid list containing a number, a string, and a boolean. Lists may be nested (may contain other lists), so ‘((1 2) (3 4)) is a valid list with two elements, each of which is a list of two elements. Notice that the inner lists do not need to be preceded by single quotes; all symbols and lists inside a quoted list are treated as if they are also quoted.

There are three types of legal non-empty lists:

1) Open parentheses not preceded by a single quote represents function calls if the first item after the open left parenthesis is a defined function name. Example character sequences that represent function calls: +, -, *, /, sqrt, expt, string-append, string-length, image-height, and many more.

2) Open parentheses not preceded by a single quote can represent applications of special forms if the first item after the open left parenthesis is a special form keyword. Example special form keywords include: define, lambda, cond, else, if, quote, and half a dozen or so more.

3) Open parentheses preceded by a single quote are data containers called lists. Example character sequences that represent lists that are data containers were given at the beginning of this subsection and include: ‘(1 2 3), ‘(“Hello” “world”), ‘(abc def ghi), and infinitely many more. Primitive functions that consume quoted lists include first, rest, list?, empty?, length, reverse and list-ref. Primitives that produce lists include cons and list.

Any list that does not fit into one of the three types listed above is treated as an error. So be very careful where you use parentheses in a Racket program. If you include too many parentheses or put them in the wrong place, you will get an error and your code will not run.

We will look at each of the three types of non-empty list as we start using them. Non-empty lists are the major building block of every Racket program. In fact, Racket programs are composed entirely of lists!

D. Evaluation of Non-Empty Lists (the Default Rule)
Since a non-empty list is a Racket datum, a natural question arises: what kinds of character sequences can the programmer use to denote non-empty lists that are intended to be functions (i.e., what are the syntax rules for expressing non-empty lists when the lists are intended to be function calls)?

As already seen, the empty list evaluates to itself; however, the evaluation of a non-empty, non-quoted list is altogether different. This section presents the default rule for evaluating non-empty lists. Exceptions to the default rule involve keywords and are called special forms. Each keyword will be covered as it is needed.

We begin with some examples that confirm that something new is happening when DrRacket evaluates non-empty lists whose leftmost element is a primitive function in the interactions window:

\[
\begin{align*}
&> \ ( + \ 2 \ 3 ) \\
&5 \\
&> \ ( * \ 3 \ 4 \ 5 ) \\
&60 \\
&> \ ( + \ 2 \ ( * \ 3 \ 10 ) ) \\
&32 \\
&> \ ( + \ 2 \ ( * \ 3 \ ( + \ 4 \ 8 \ 6 ) ) ) \\
&56
\end{align*}
\]

In each of these examples from the interactions window, the expression entered by the programmer is a legal Racket expression in prefix notation that denotes a Racket list. In addition, the evaluation of each list appears to result in an arithmetic computation—in fact, the kind of arithmetic computations you’ve seen in math classes over the years. In each case, the list is being evaluated according to the default rule.

The resulting output datum is the result of evaluating the non-empty list given as input. Thus, the result of evaluating the list containing the + symbol, the number two and the number three, is (not surprisingly perhaps) the number five,
which DrRacket reports in the interactions window using the character sequence 5.

Here’s a summary of this example:

\[(+ \ 2 \ 3) \rightarrow \ [ \text{list containing + symbol, number two, and number three } \Rightarrow \text{number five} ] \rightarrow 5\]

The evaluation steps (\rightarrow) are explained by:

1) **First Step of Default Rule**—look up + symbol in global environment and evaluate arguments 2 and 3:
   - + symbol \Rightarrow addition function
   - number two \Rightarrow number two
   - number three \Rightarrow number three

2) **Second Step of Default Rule**:
   Addition function applied to two and three yields output of five.

Evaluation of all of the items in the list yields the addition function and two numbers. The second step in the default rule involves applying that function to the remaining items (i.e., feeding the remaining items as input into that function), as illustrated below:

![Fig. 7. Pictorial evaluation of addition function on inputs 2 and 3. Picture courtesy of Prof. Hunsberger.](image)

![Fig. 8. Pictorial version of evaluation of \((+ \ 2 \ 3)\). Picture courtesy of Prof. Hunsberger.](image)

There are several advantages to the default rule. First, it only has two steps, and they are always the same. Second, it can be used on arbitrarily complex lists without requiring any modifications.

For example, consider the following evaluation in the interactions window:

\[> (\ + \ 2 \ (* \ 3 \ 10))\]

\[32\]

If you follow the rules you already know, we will see that nothing new is needed to explain this interaction. First, the character sequence \((+ \ 2 \ (* \ 3 \ 10))\) is a legal Racket expression that denotes a non-empty list. The denoted list contains three items: the + symbol, the number two, and a subsidiary list. The subsidiary list contains three items: the * symbol, the number three and the number ten. To evaluate this list, we need to use the default rule.

The first step of the default rule requires us to evaluate each item in the list:

1) the + symbol \Rightarrow the addition function
   - the number two \Rightarrow the number two
   - the subsidiary list \Rightarrow WHAT?
Before we can complete the first step of the default rule, we must evaluate the subsidiary list (i.e., the list containing the * symbol, the number three and the number ten). Okay, so we pause for a moment and then proceed.

To evaluate the subsidiary list, we need to use the default rule. The first step of the default rule requires us to evaluate each item in the list:

1) the * symbol $\Rightarrow$ the multiplication function
   the number three $\Rightarrow$ the number three
   the number ten $\Rightarrow$ the number ten
2) The second step of the default rule requires us to apply the first item (i.e., the multiplication function) to the numbers three and ten. The result is the number thirty.

Now that we know the subsidiary list evaluates to thirty, we can pick up from where we left off when evaluating the original list. The first step of the default rule (for evaluating the original list) requires us to evaluate each item in the list:

1) the + symbol $\Rightarrow$ the addition function
   the number two $\Rightarrow$ the number two
   the subsidiary list $\Rightarrow$ the number thirty
2) The second step of the default rule then requires us to apply the first item (i.e., the addition function) to the rest of the items (i.e., the numbers two and thirty). The result is the number thirty-two and Racket returns the character sequence 32.

A Formal Description of the Default Rule:
Consider $(D_1 D_2 ... D_n)$, a list $L$ that contains $n$ data items, $D_1, D_2, ..., D_n$. The evaluation of the list $L$ is derived as follows:

1) First, evaluate each of the data items, $D_1, D_2, ..., D_n$. The result will be $n$ (possibly different) data items, $K_1, K_2, ..., K_n$ such that:
   $D_1 \Rightarrow K_1$
   $D_2 \Rightarrow K_2$
   ...
   $D_n \Rightarrow K_n$
2) Now, for the default rule to work, $K_1$ must be a function that is named in the global environment. (If $K_1$ is some other kind of datum, then DrRacket will report an error.)
3) The second step is to apply the function $K_1$ to the rest of the items, $K_2, ..., K_n$. In other words, the items $K_2, ..., K_n$ are each evaluated and fed as input arguments to the function $K_1$. (If the function $K_1$ cannot accept that number or type of inputs, then DrRacket will report an error.) The resulting output will be some datum, $P$.
4) The evaluation of the list $L$ is defined to be that datum $P$ (i.e., $L \Rightarrow P$).

It is possible for some things to go wrong in the process of evaluating a non-empty list. For example, the function $K_1$ might expect a different number of input arguments than are present in the rest of the original list. Or the attempt to evaluate the datum $D_i$ might be undefined. Or the application of the function $K_1$ to the arguments $K_2, ..., K_n$ might be undefined because, for example, the function expects numbers and it is asked to use some other type of data. In any of these cases, the result is undefined and DrRacket would report an error.

For example, none of the following lists can be evaluated under the Default Rule:
- $(1 2 3)$, a list containing the numbers one, two and three
- $(\text{empty} \ \text{empty})$, a list containing two instances of the empty list
- $(+ \ \text{true} \ \text{false})$, a list containing the + symbol, followed by the boolean true and the boolean false

You should try to explain, for each of the lists above, why they could not be evaluated by the default rule. It is important to understand that each of the above lists is a valid Racket datum: each one is a list. It’s just that these lists cannot be evaluated under the default rule (and therefore cannot be evaluated at all).

Example. Here’s an example of the default case of evaluating a non-empty list where things work out. Let $L$ be the list containing the following data:
$D_1$: the + symbol, $D_2$: the number one, $D_3$: the number two, $D_4$: the number three

These Racket data evaluate to the following:

$K_1$: the addition function, $K_2$: the number one, $K_3$: the number two, $K_4$: the number three

Since the first of these, $K_1$, is in fact a function, it can be applied to the evaluated arguments $K_2$, $K_3$ and $K_4$ (i.e., the numbers one, two and three). This results in the output six, which is itself a Racket datum. The number six is the result of evaluating the original list $L$, as illustrated below in an excerpt from the interactions window.

$$> (+ \ 1 \ 2 \ 3)$$

6

Notice that because addition is a primitive function, its operation is invisible to us. We observe the inputs going in and the output coming out, but we do not get to see how the output is generated.

Summary of Default Rule:

- The default rule for evaluating non-empty lists is how function application is made available to the Racket programmer. In particular, if you want to apply a given function to a bunch of input arguments, you create an expression that denotes the appropriate list and feed it to DrRacket. These applications may be nested, in which case the innermost values are computed first.
- The default rule has two steps. The first step involves evaluating each item in the original list, resulting in a bunch of new items. The second step involves applying the first new item to the rest of the new items (i.e., feeding the rest of the items as input, or arguments, to the function that is the first item). The result of this function application is taken to be the result of evaluating the original list.
- Racket is called a functional programming language because function application is the central part of the computational model of Racket. And the default rule is how the programmer gets function application to happen.

E. Structs

You can think of non-empty quoted lists as containers for an arbitrary amount of primitive or compound data. If you need to use multiple data containers that each hold a fixed number of primitive data types, you can create what are called structs. The functions to create a struct and access its parts are made at the same time the struct name is defined.

The keyword used to create new structs in Swindle is `defstruct`.

F. Vectors

A container for sequential, indexed data. Vectors are similar to the array data type used in other languages (e.g., Java & C++). The contents of a vector are numbered sequentially from left to right, starting at 0.

Vectors store data like lists do, but they provide faster access to all elements than is possible in a list. Vectors are “0-based” indexed, like strings, but vectors are more general than strings because they can hold anything, not just characters. Vectors are known as a “constant time, random access” data structure.

Primitive functions that consume vectors include `vector-ref` and `vector-length`. Primitives to produce vectors are `make-vector` and `vector`.

Structs and vectors will be covered later in the semester, after lists.
In DrRacket, there is a set of symbolic expressions called **special form keywords**. Examples of special form keywords include: **and**, **or**, **cond**, **define**, **else**, **if**, **lambda**, **let**, and **local**. Each of these keywords is a legal Racket expression when used in the proper way.

The interesting thing about special form keywords is this:  
\( \Rightarrow \) When the first element of a non-quoted, non-empty list is a keyword symbol, then that list is called a special form. And special forms have their own mode of evaluation that does not follow the default rule.

For example, each of the following character sequences denotes a representation of using a special form:

\(\text{(define } x \ 3)\)

\(\text{(if } \text{condition} \ \text{then-clause} \ \text{else-clause})\)

\(\text{(cond } \text{[condition1 return1]} \ \text{[condition2 return2]} \ \text{[else return3]}\)\)

A special form is evaluated according to a rule that is specific to the keyword used for that special form. There is one rule for evaluating **define** special forms, another rule for evaluating **if** special forms, and so on. Importantly, each **define** special form is evaluated in the same way. However, the rule for evaluating the **define** special forms is very different from the rule for evaluating the **if** special form. **In other words, each special form has its own rules for evaluation.**

In the default rule for evaluating non-empty lists, the first thing that happens is that each element of the list is evaluated, one after the other. In contrast, when evaluating a special form, which is also a non-empty list, some of the elements of that list may not be evaluated. Indeed, the first element of a special form (i.e., the keyword) is never evaluated, because no special form is written in the global environment.

In this section, we will introduce the **define** and **lambda** special forms. Other special forms will be covered as we need them.

**A. The define Special Form: Naming values in the global environment**

The purpose of the **define** special form is to create an entry in the global environment. The **define** special form is indicated by the **define** keyword.

As a character sequence, it has the form \(\text{(define } \text{C1 } \text{C2})\), where \text{C1} is some sequence of characters, and \text{C2} can be any expression denoting a Racket datum, call it \text{e}.

\[\text{Fig. 9. Evaluating the special form (define C1 C2). Picture courtesy of Prof. Hansberger.}\]

It is important to note that only \text{e}, the Racket version of the symbol \text{C2}, is evaluated (as shown in Fig. 9), the result being some datum \text{E} that is written into the global environment. The symbol \text{C1} is not evaluated, but an entry is created in the global environment in which the name \text{C1} is associated with the datum \text{E}.

Since the purpose of a **define** special form is to create an entry in the global environment, a **define** special form does not produce any output—it produces only **void** as an output type. Thus, DrRacket does not generate any output on the computer screen when a **define** special form is evaluated in the interactions window, as shown below:
After the `define` special form is executed as shown above, subsequent attempts to evaluate the symbol `X` in the interactions window will result in the display of the value 6, as illustrated in an excerpt from the interactions window below:

```
> X
6
> (* X 100)
600
> (* X 1000)
6000
> (* X X)
36
```

When the result of evaluating a special form generates no output (i.e. return value), we say that the result is a *side effect*. The side effect of the `define` special form is the inclusion of the symbol name and value in the global environment.

Creating a named value in the global environment is the way to name so-called *variables* in DrRacket. To be more accurate, in a language like Racket, a name associated with a value is never supposed to change value. So creating a named value in the global environment is really a way to name what in other languages is known as a *constant*.

Symbol names allow you to avoid using literal values in code. The usual practice is to start a program with all the constant definitions such that all constants used in the program are written in the same block of code. Any constant used in the code must be defined *before* it is used.

**Example:** Typing the character sequence `(define Y (+ 1 2 3))` into the interactions window and pressing return would result in the number 6 being associated with the symbol `Y` in the global environment, as illustrated below.

```
Evaluating the special form:  (define Y (+ 1 2 3))
Y
(+ 1 2 3)
Global Environment Entry:  The symbol Y  The number six
a list containing the + symbol and the numbers one, two, and three

Fig. 10. Evaluating the special form `(define Y (+ 1 2 3))`. Picture courtesy of Prof. Hunsberger.
```

However, DrRacket does not return any output value:

```
> (define Y (+ 2 4 3))
```

Subsequent attempts to evaluate the symbol `Y` would result in the value 9, as illustrated below:

```
> Y
9
> (* Y 8)
72
> (* Y 1000)
9000
> (* Y Y)
36
```

The following excerpt from the Interactions Window shows the definition of several constants:

```
> (define WIDTH 500)
> (define HEIGHT 400)
> (define DIAMETER (* 50 50 3.14159))
```
This sequence of interactions would result in setting the name WIDTH to be equal to the value 500, the name HEIGHT to be equal to the value 400, and the name DIAMETER to be equal to \( (50 \times 50 \times 3.14159) = 7853.975 \). The evaluation of mathematical expressions will be covered in an upcoming section. An important thing to note here is that the value of the mathematical expression in the third define statement is written in the global environment after the evaluation of the expression \( (50 \times 50 \times 3.14159) \).

Fig. 11. Adding the symbols WIDTH, HEIGHT, and DIAMETER to the Global Environment.

**Important:** Any usage of the define special form in the interactions window will have effect only until the Run button is pressed. Each time the Run button is pressed, all entries made to the global environment are rewritten. To make lasting changes to the global environment, you should type the define statements in the definitions window and press Run.

**B. The Lambda Special Form: Creating functions**

The Racket programming language provides the lambda special form to enable us to define functions. In simple terms, a function is a block of code that accomplishes some task. As a general rule of thumb, each function should perform one specific task. The lambda special form defines everything about a function except its name. Another way of specifying the lambda special form is by using the Greek letter \( \lambda \). To insert a \( \lambda \) character into a DrRacket program, pull down the Insert menu and choose “Insert \( \lambda \)”. The terms lambda and \( \lambda \) are used synonymously.

Like any special form in Racket, using the lambda special form is done by creating a list whose first element is the keyword lambda—in this case, the symbol lambda (or \( \lambda \)). The second element used in a lambda special form is the parameter list that specifies the names of the input parameter(s) for the function being defined. The rest of the elements when using the lambda special form constitute the body of the function being defined. The parameters and the body are always enclosed in a pair of (possibly nested) parentheses.

Here is an example of a lambda expression that calculates the hypotenuse of a right triangle, given side lengths \( a \) and \( b \):

\[
(\lambda \ (a\ b)\ (sqrt\ (+\ (*\ a\ a)\ (*\ b\ b))))
\]

In general, a lambda form consists of three parts:
1) The word lambda (or the symbol \( \lambda \)).
2) The names of the parameters \( (a \text{ and } b \text{ above}) \), in parentheses.
3) The body is \( (sqrt\ (+\ (*\ a\ a)\ (*\ b\ b))) \).

**The Syntax of a \( \lambda \) Expression:**

A \( \lambda \) expression has the following syntax: \( (\lambda \ (C_1\ C_2\ \ldots\ C_n)\ B) \) where:
- each \( C_i \) is a character sequence denoting some Racket symbol, \( r_i \);
- the symbols, \( r_1, r_2, \ldots, r_n \), are distinct (i.e., there are no duplicates); and
- \( B \) is a character sequence denoting a Racket datum, \( D_i \), of any kind.

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 examples of well-formed (i.e., valid) \( \lambda \) expressions:
1) \( (\lambda \ ()\ 44) \)
2) \( (\lambda \ (x)\ (*\ x\ x)) \)
3) \( (\lambda \ (w\ h)\ (*\ w\ h)) \)
4) \( (\lambda \ (r\ h)\ (*\ 1/3\ 3.14159\ r\ r\ h)) \)}
5) \((\lambda (x \ y \ z) (+ \ x (- y z)))\)

We often refer to functions based on the number of input parameters listed in the \(\lambda\) expression. For example, expression 1) is known as a zero-parameter function, 2) is a one-parameter function, 3) and 4) are two-parameter functions, and 5) is a three-parameter function. Expression 1) has only the number 44 for a body, so that expression will always evaluate to 44. Expressions 2), 3), and 4) have parameter lists \((x)\), \((w \ h)\), and \((r \ h)\) respectively, and bodies \((* x x)\), \((* w h)\) and \((* 1/3 3.14159 r r h)\), respectively. For expression 5), \((x \ y \ z)\) specifies the parameter list and \((+ x (- y z))\) specifies the body.

In contrast, the following are examples of malformed \(\lambda\) expressions:

1) \((\lambda (x \ y \ x) (* x y))\)
2) \((\lambda (x \ 10) (* x \ 10))\)
3) \((\lambda x)\)

Take a moment to describe what it is about each of the three expressions that makes them malformed.

The Semantics of a \(\lambda\) Expression:
The semantics of a \(\lambda\) expression stipulates what Racket datum the \(\lambda\) expression denotes, as well as how that Racket datum is evaluated. As suggested by the preceding examples, a \(\lambda\) expression invariably denotes a list—called a \(\lambda\) special form—and the evaluation of that list invariably results in a Racket function. The semantics of the \(\lambda\) expression also includes a description of the subsequent behavior of that function should it ever be applied to any input(s).

Assuming that

- each \(C_i\) denotes a Racket symbol, \(r_i\);
- the symbols, \(r_1, r_2, ..., r_n\), are distinct (i.e., there is no repetition); and
- \(B\) denotes some Racket datum \(D\),

then a \(\lambda\) expression of the form \((\lambda (C_1 C_2 ... C_n) B)\) denotes a Racket list whose elements are as follows:

- the \(\lambda\) symbol;
- a list containing \(n\) distinct symbols, \(r_1, r_2, ..., r_n\); and
- the Racket datum, \(D\)

This list is referred to as a \(\lambda\) special form.

Calling unnamed \(\lambda\) Expressions:
When you use the default rule to apply a unnamed \texttt{lambda} form to actual input values, it is often called an invocation or call of the function, as shown below:

\[
((\texttt{lambda} (a \ b) (\texttt{sqrt} (+ (* a a) (* b b)))) \ 3 \ 4)
\]

Racket evaluates an application involving a \texttt{lambda} form by matching the arguments (3 and 4 above) with the parameters \((a \ b)\). It then substitutes the arguments for the parameters in the body of the procedure \((\texttt{sqrt} (+ (* a a) (* b b)))\). The result is the body rewritten as \((\texttt{sqrt} (+ (* 3 3) (* 4 4)))\). The \texttt{lambda} keyword tells Racket that this substitution should occur.

The type of a \texttt{lambda} form is a function. The \texttt{lambda} form is the way the programmer expresses that some operations be performed on the arguments and return a value. \textit{Racket does not evaluate the elements of the body of the function until the function is invoked, or called, on the arguments.}

Another example showing how it is possible to define and apply a function without ever having given it a name is given below in an excerpt from the interactions window:

\[
> \ ((\lambda (x) (* x x)) \ 4)
\]

\(16\)

The default rule for evaluating non-empty lists is used to evaluate the expression typed at the \(>\) prompt above. In the process, each element of the list is evaluated. The first element of the first subsidiary list is the \(\lambda\) special form, whose body is \((* x x)\). The second element of the list typed at the \(>\) prompt evaluates to the number 4. The result
of replacing the parameter $x$ in the body of the function with the input argument 4 yields $(\ast 4 4)$, which in turn evaluates to the correct output, 16. Later on, we will encounter situations where it is more convenient to use functions without naming them, so it is important to understand how to use these unnamed $\lambda$ expressions.

**Example:** The Cube Function. The mathematical definition of the cube function is: $f(x) = x \ast x \ast x$.

This mathematical definition does three things for the function being defined:
1) it specifies a single input parameter, $x$;
2) it specifies a body, $(x \ast x \ast x)$; and
3) it specifies a name, $f$.

In Racket, the first two jobs are handled by the $\lambda$ special form. In particular, the following $\lambda$ expression can be used for a cubing function in Racket: $(\lambda (x) (\ast x x x))$

This $\lambda$ expression denotes a $\lambda$ special form (i.e., a Racket list whose first element happens to be the $\lambda$ keyword). Like any special form, a $\lambda$ special form has its own rule for being evaluated. For now, suffice it to say that the evaluation of a $\lambda$ special form always results in a function.

Thus, if the expression, $(\lambda (x) (\ast x x x))$, is typed into the interactions window, DrRacket will report that a function has been evaluated, as illustrated below:

```
> (\lambda (x) (\ast x x x))
#<procedure>
```

**Summary:** The $\lambda$ special form generates appropriate output values should it ever be applied to any input values. A $\lambda$ statement includes a list of input parameters and a body (the application). A function can be applied to appropriate input values as follows:
1) Replace the parameters as they occur in the first subsidiary list of the $\lambda$ with the arguments on a strict left-to-right basis (in terms of matching the parameter list and the arguments).
2) If there are nested function applications, work from the inside out by finding simple forms (ones which have no forms nested inside them) and evaluating them first.
3) Evaluate the body from the inside out until all parts are evaluated.

The desired input arguments are substituted for the appropriate input parameters in the body of the function. Next, the resulting expression is evaluated, thereby yielding the desired output value.

**Exercise V-C-1:** Evaluate each of the following Racket forms (try to figure out each of the outputs in your head and then try them on the computer):

(a) $((\lambda (x) (+ x 2)) 5)$
(b) $((\lambda (x y) (+ x (* y 3))) 7 4)$
(c) $(((\lambda (x)) ((\lambda (y) (* y 2)) (+ x 3))) 4)$

**Exercise V-C-2:** Write lambda forms that express the following algebraic formulas:

(a) $(x - 32)*\frac{5}{9}$
(b) $(x + 3 \ast y) \ast (x - y)$

The most important thing to know about the evaluation of a $\lambda$ special form is that the result is invariably a function; however, the evaluation of a $\lambda$ special form only creates the function; it does not apply it to any input(s).

**C. Using the define Special Form to create named functions**

You use the `define` special form to give a $\lambda$ statement a name, thereby naming a function. Naming functions is a way of reusing code because one name can result in the application of arbitrarily complex functions and the name is usable any number of times in a program.
For example, we can first give a function a name using a lambda special form inside the parentheses of a define special form and then applying the named function to a variety of input values. The following excerpt of an interactions window session demonstrates how to name a square procedure:

```
> (define square (λ (x) (* x x)))
> (square 5)
  25
> (square 7)
  49
> (square -8)
  64
```

Fig. 12. Adding the function square to the Global Environment.

The define special form is used to create an entry in the global environment that associates the symbol square with the function specified in the λ expression. When a define special form is evaluated, the given symbol—in this case, square—is not evaluated; however, the symbol and the given lambda expression—in this case, square and (λ (x) (* x x))—are written in the global environment. Thus, the value associated with the square symbol is the procedure that results from evaluating the given λ special form with a given argument.

Each of the expressions in Fig.12 is evaluated using the default rule for evaluating non-empty lists. In each case, the square symbol evaluates to the function that we defined earlier, which is then applied to the value of the input argument.

Additional Examples:
The following interactions window session demonstrates how to define, name, and apply a procedure analogous to the function, \( g(w, h) = w \times h \), seen earlier:

```
> (define rect-area (λ (w h) (* w h)))
> (rect-area 2 3)
  6
> (rect-area 3 8)
  24
```

Function naming conventions:

Racket adheres to certain conventions for function names.
- Functions that return a boolean (except the relational operators in numeric functions) end with a question mark (pronounced “huh”).
- Functions that compare non-numeric types for equality use the name of the type, followed by an equal sign, followed by a question mark. Examples include `string=?`, `boolean=?`, `symbol=?`.
- Functions that convert one type of data to another specify the name of the input type followed by `->`, followed by the name of the output type. Examples include `string->number`, `number->string`, `string->symbol`, and so on.

Distinguishing Arguments from Parameters

An argument is a specific value (e.g., a literal value or an expression that evaluates to a literal value) given in a function call, while a parameter is a “place-holder” introduced in the λ part of a procedure definition. An argument may evaluate to a number, string, image, constant, or other valid type, but a parameter is always a symbolic name, a sequence of keyboard characters. For example, the line `(rect-area 2 3)` has arguments 2 and 3 and these
numbers are substituted for the parameters \( w \) and \( h \) in the \( \lambda \) expression \( (\lambda (w \ h) (* \ w \ h)) \). The statement \( (\text{rect-area} \ 2 \ 3) \) is known as a \textit{function call}.

**Local vs. Global Environment:** When a \( \lambda \) expression is applied to an input value (i.e., when the expression is called with some argument value), a lookup table called the \textit{local environment} is created for the evaluation of that function. The names in the local environment are the parameter names listed in the \( \lambda \) expression and the values associated with the names are the arguments to which the \( \lambda \) expression is applied.

**Example—Applying the Squaring Function to Input Values:** Consider the \( \lambda \) expression, \( (\lambda (x) (* \ x \ x)) \). As noted above, it evaluates to a Racket function. When this \( \lambda \) function is applied to some input value, say 4, by typing \( ((\lambda (x) (* \ x \ x)) \ 4) \), as shown in Fig. 13, the following things happen:

![Fig. 13. Local environment of squaring function. Picture courtesy of Prof. Hunsberger.](image)

1) A local environment is set up containing a single entry which associates the value 4 with the symbol \( x \).
2) The expression \( (* \ x \ x) \) is evaluated with respect to the newly created local environment. \textit{This means that every occurrence of the symbol \( x \) is evaluated according to the local environment, not the global environment, even if \( x \) is also associated with a value in the global environment.} If \( x \) is associated with a value in the global environment, we say that the symbol \( x \) in the local environment \textit{shadows} the symbol \( x \) in the global environment.
3) That value, 16, is taken to be the output value that results from applying the \( \lambda \) function to the input value 4.
4) The local environment vanishes after the function returns a result.

Here’s an example of a named function that takes more than one input argument.

```racket
> (define discriminant
  (lambda (a b c)
    (- (* b b) (* 4 a c))))
```

```racket
> (discriminant 1 2 -4)
20
```

Notice that the syntax of Racket allows expressions to occupy multiple lines. This is quite useful when writing longer expressions. Sub-expressions are lists nested inside other lists. DrRacket automatically indents sub-expressions following a newline (return) to make longer expressions easier to read. Pressing the tab key will automatically cause the current line to snap to the appropriate amount of indentation \textit{unless there is some error in the syntax or semantics of the expression before that line.} In general, it is a good idea to create a newline before the opening parenthesis of each sub-expression.
VI. INTRODUCTION TO WRITING AND RUNNING A PROGRAM

Although we have portrayed the execution of expressions (and even short function definitions) in DrRacket’s Interaction Window, the real place to write code that is saved between times we exit and re-enter DrRacket is in the Definitions Window (at the top or left side of the window). All of the programs you write for this class will be written in the Definitions Window and saved in files. To command DrRacket to evaluate code written in the Definitions window, you press the Run button at the top right of the window. The result of running the program is displayed in the Interactions Window.

A. Comments/Documentation

Comments are used in every computer program as a way to document the purpose of the functions in that program. In Racket, the semi-colon character is used to signal a one-line comment. A comment may start at the beginning of a line (in column 0) or anywhere else in a line. Everything on the same line and to the left of a semi-colon should be a valid Racket expression, and everything to the right of the semi-colon is a comment and is ignored by the computer. Comments are easy to detect in Racket programs because DrRacket colors them brown.

Comments are as important to computer programs as captions and annotations in a paper or book. For that reason, a significant part of the grade you will receive for a program depends on how well the program is documented.

Another type of comment is made available through DrRacket: comment boxes. To start a comment box, pull down the Insert menu and choose Insert Comment Box. If you have already typed text and you want to comment it out with a comment box, highlight the text, pull down the Racket menu, and choose Comment out with a box.

Understanding pre-function comments in the documentation

Comments make code easier for people to understand. Programs written without comments are often useless to anyone who must later modify the code. For this class, comments are nearly as important as the functions you write. The comments shown below are written before a function that produces the area of a rectangle. First and foremost, every function should have a name, given in the comments, that fits the purpose of the function. For example, one style of pre-function comments are shown below:

```
;;; AREA
;;; -------------------------------------------
;;; Computes the area of a rectangle.
;;; INPUTS:
;;; WIDTH -- A number, integer or real
;;; HEIGHT -- A number, integer or real
;;; OUTPUT: A number representing the area of a rectangle with
;;; dimensions WIDTH and HEIGHT.
;;; SIDE EFFECTS: None.
```

Notice that the function and input names are capitalized in these pre-function comments, but this is just so they stand out. The actual function name and parameters could be in lowercase. There is nothing magical about preceding each line with 3 semicolons, doing so just adds visual interest. A single semicolon before each line would suffice.

The style of pre-function comments used in the “How to Design Programs” books and in the Help Desk of DrRacket would be shown as follows (for the area function):

```
; (area width height) -> number
; width: number
; height: number
; Calculates the area of a rectangle given
; the width and height.
; Examples:
; > (area 5 4)
; 20
```

The first line of a comment of this style is called a contract and it specifies:
1) the name to use when calling the function (e.g., area),
2) the parameter names (width and height) inside the function, and
3) the type of datum produced by the function (to the right of the ->) (e.g., a number).

The remaining lines tell us:
1) the type(s) of data arguments the function takes as input (e.g., two numbers) for placeholders width and height,
2) the purpose of the function, and
3) an example of using the function in the interactions window (at the > prompt).

In the Help Desk, the section of the documentation that covers the function string-length looks like this (dashes have been added to separate parts of the comment and are not shown in the HtDP books):

```
; (string-length s) -> nat num
;---------------------------------  
; s : string
;---------------------------------  
; Determines the length of a string.
;---------------------------------  
; Examples:  
; > (string-length "hello world")  
; 11
```

The contract tells us that the function string-length consumes a string argument for parameter s and produces a natural number. Next, there is a listing of the input parameter name s of type string. Then comes the purpose, followed by an example of using the function.

Now let's look up the function string-append. The documentation for this function looks like this:

```
; (string-append s ...) -> string
;---------------------------------  
; s : string
;---------------------------------  
; Juxtaposes the characters of several strings.
;---------------------------------  
; Examples:  
; > (string-append "hello" " " "world" " " "good bye")  
; "hello world good bye"
```

The ellipses (...) in the contract of the example above tells you that the function can consume any number of string inputs. There are many primitive functions that can take an arbitrary number of inputs, so you should know how to recognize them when you encounter them in the documentation.

You may choose to use one style or another for your pre-function comments, but you must include all necessary information in a clear and consistent manner.

B. Indentation, Code Coloring, and Line Length

DrRacket helps you write code by giving indications of when code is written correctly or incorrectly. For example, it colors keywords and function names in blue, literal values in green, comments in brown, and errors in red. DrRacket also indents nested code as it is typed and matches right parentheses with their corresponding left parentheses. You will learn to determine if the code you are writing is correct as you type it by paying attention to the color and indentation, as well as by matching parentheses.

In the bottom right corner of the DrRacket window is a pair of numbers separated by a colon (e.g., 67:34). The interpretation of these numbers is that the cursor is currently on line 67 and in column 34. For this class, you will be expected to keep track of the second number in this pair as you are typing programs in the Definitions Window. No line of code or single line comment should extend beyond column 80.

To show line numbers while you type code, pull down the View menu to Show Line Numbers.
VII. PRINTING STRINGS

A. String functions

Syntactically, strings in Racket are character sequences, sometimes including spaces, and bounded by double-quotes. For example, “hi” and “Howdy!” are character sequences that denote string data.

The following interactions window session demonstrates that the evaluation function behaves like the identity function when applied to string data, meaning that strings evaluate only to themselves.

```racket
> "hi"
"hi"
> "Howdy!"
"Howdy!"
```

Racket also includes a type-checker predicate for the string data type: string?, as demonstrated below.

```racket
> (string? "cat")
#t
> (string? 39)
#f
> (string? "a" "b" "c")
#f
```

The last example above shows that the function string? does not work for multiple input strings.

There are other primitive functions that consume strings that you will find useful. Most primitive functions for strings have self-explanatory names. For example,

- ```string-length``` consumes a string str and produces a number representing the length of str.
- ```string-append``` consumes any number of separate strings and joins them together to produce a single string.
- ```string-ref``` consumes a string str and a positive integer i, and returns the character that occurs in str at position i.
- ```substring``` consumes a string and one or two numbers specifying indices and produces the string from the first index onward or from the first index to one less than the second index.
- ```string``` consumes any number of characters and produces the characters in a string.

B. The printf, display, and newline functions

DrRacket includes built-in printf, display, and newline functions that produce void output but have the side-effect of printing to the interactions window.

The display function has a side effect of displaying its string argument to the interactions window. The display function interprets the character sequence \n in a special way, as a newline character.

The printf function also has the side effect of displaying its string argument to the interactions window, but it also allows the inclusion of symbols in the string at specified positions. The printf function interprets the character sequences, \%, and \A (or \a), in special ways when they are embedded in its first argument—a string. \% is a newline character and \A or \a are placeholders for values after the value has been converted to a string.

The character sequences \n, \%, \A and \a are commonly called escape characters because they cause something other than the exact character sequence they specify to be printed.

A call to the display, printf, or newline functions evaluates to void. That’s because the whole point of these functions is their side-effect. In reality, Racket provides a special datum that is interpreted as “no value”. This “no value” datum is the void data type. Like any other data type, there is a corresponding type-checker predicate for the void data type. It is called void?. Its use is demonstrated below.

```racket
> (void? (printf "hi\%n"))
hi
#t
```

In this example, the Default Rule for evaluating non-empty lists is used to evaluate the expression, (void? (printf "hi\%n")). In the process, the void? symbol evaluates to the built-in void? function and (printf
"hi˜%") evaluates to the special no value datum belonging to the void data type. The no value datum is fed as an argument to the void? function, resulting in the output value true, as reported by DrRacket. The character sequence, hi, was printed out as a side-effect of the evaluation of the expression, (printf "hi˜%").

The following example excerpt from the interactions window demonstrates the use of the printf, display, and newline functions:

```
> (printf "Hi there!")
Hi there!
> (display "Hi there!")
Hi there!
> (printf "Oh, ˜A I get it!" 'hey)
Oh, hey I get it!
> (display "Oh, hey\nI get it!")
Oh, hey
I get it!
> (printf "First thing: ˜A,˜%second thing: ˜A˜%" (+ 2 3) (* 6 7))
First thing: 5,
second thing: 42
> (newline)

The newline function causes a blank line to be printed in the interactions window.

The display function causes the string (i.e., its only argument) to be displayed in the interactions window such that:

- the quotation marks are omitted and
- each instance of the escape sequence \n is interpreted as a newline character, causing a newline to be printed in the interactions window.

The printf function causes the string (i.e., its first argument) to be displayed in the interactions window such that:

- the quotation marks are omitted;
- each instance of the escape sequence ˜ %, is interpreted as a newline character, and thus causes a newline in the Interactions Window; and
- each instance of the escape sequence ˜A or ˜a is replaced by a character sequence representing the value of one of the remainder of the arguments to printf, where each argument’s value is printed in the order they occur, from left to right, after the initial string parameter. The number of ˜As included in the first argument to printf (a string) must match the number of arguments that follow the first.

Putting Multiple printf Expressions in the Body of a λ Function.

The body of a λ function may contain multiple expressions in Swindle. When such a function is called, each of the expressions in the body is evaluated sequentially, from top to bottom. The value of the last expression is the output value for the function call. All expressions except the last should generate only side effects.

The following λ function, called verbose-func, contains multiple expressions in its body. When verbose-func is called, each expression in its body is evaluated, from top to bottom. The first four expressions cause the built-in printf function to be called, thereby generating several lines of side-effect printing in the Interactions Window. However, it is the evaluation of the last expression in the function’s body that generates an output value for the function call.

```
> (define verbose-func
  (lambda (a b)
    (printf "Hi. This is verbose-func!˜%")
    (printf "The value of the first input is: ˜A˜%" a)
    (printf "The value of the second input is: ˜A˜%" b)
    (printf "Their product is:˜%")
      (* a b)))
> (verbose-func 3 4)
Hi. This is verbose-func!
```
The value of the first input is: 3
The value of the second input is: 4
Their product is:
12
>

Even though printf statements generate only side-effects, we will use them in this class when we are printing results of function evaluation, learning to debug code, and when writing “interactive programs” that require input from the user.

**Debugging:**

Debugging is an important process (and really an art) a programmer goes through when a program has no syntax errors but still produces an incorrect result or otherwise fails as a result of logic errors. One common way professional programmers find logic errors in their programs is by writing printf statements to test the values of different parameters inside a function while the code is running.

Here are a few more examples of using the printf function in the interactions window:

```lisp
> (printf "Line One!˜\% Line Two!!˜\% Line Three!!!˜\%")
Line One!
Line Two!!
Line Three!!!

> (printf "First ===> ˜a, Second ===> ˜A, Third ===> ˜a˜\%"
  (+ 4 2) (- 9 6.3) (* 4 100))
First ===> 6, Second ===> 2.7, Third ===> 400

> (printf "A symbol: ˜A, a string: ˜A, a boolean: ˜a˜\%"
  'I-am-a-symbol
  "I am a String!"
  (> 4 2))
A symbol: I-am-a-symbol, a string: I am a String!, a boolean: #t

Notice that the escape sequence ˜A is the same as ˜a, so they can be used interchangeably.
A function whose output is always a boolean (i.e., true or false) is called a *predicate*. (This is just convenient terminology; there is no predicate type in Racket.) This section describes some of the commonly used, built-in Racket predicates and illustrates their use.

### A. Type-checker predicates

Racket includes many primitive data types, discussed earlier. Racket also includes compound data types such as list and string. For each one of these data types, Racket includes a primitive function called a type-checker predicate. When a type-checker predicate is applied to some Racket datum, it outputs true if that datum belongs to the indicated data type; otherwise, it outputs false. Thus, the type-checker predicate associated with the number data type (*number?*) outputs true whenever the input belongs to the number data type. Similarly, the type-checker predicate associated with the list data type (*list?*) outputs true whenever the input datum belongs to the list data type or is an empty list, and so on.

Each type-checker predicate is a function that can be applied to a single input (i.e., each type-checker is a one-parameter function). That input can be any type of Racket datum. The type-checker predicates are unique in that they can be applied to any type of input without causing an error, a claim that is not true of the rest of the built-in functions. A type-checker predicate returns true if the given input datum is of the appropriate data type, and false otherwise, as illustrated in the following Interactions Window session:

```scheme
> (number? 3)
#t
> (number? true)
#f
> (boolean? #f)
#t
> (boolean? x)
#f
> (symbol? +)
#f
> (symbol? ’+)
#t
> (procedure? +)
#t
> (procedure? ’+)
#f
> (procedure? 34)
#f
> (list? ’(12 34))
#t
> (list? empty)
#t
```

Notice that the type checker predicate names mirror the names of the corresponding data types, except that the symbol associated with the type-checker predicate for functions is *procedure?*, not *function?* and the type-checker for quoted symbols is called *symbol?*.

Each of the expressions typed at the interactions window prompt in the excerpt above denotes a list that is evaluated according to the default rule for evaluating non-empty lists. In each case, the first element of the list is a symbol that evaluates to a function, which is then applied to whatever the second element evaluates to. Notice that the + symbol in (procedure? +) evaluates to the addition function, whereas the ’+ expression in (procedure? ’+) evaluates to the quoted + symbol. Notice too that the list? type-checker predicate returns true for any list, whether empty or non-empty.

### B. Arithmetic Predicates: Relational Operators

In addition to the primitive arithmetic functions for addition, subtraction, multiplication and division, Racket includes several arithmetic predicates, such as greater-than, less-than, and equal predicates and each is associated with a particular symbol in the global environment:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
</tbody>
</table>

Each of these predicates, when applied to two or more numeric inputs, generates the expected boolean output, as illustrated below:

```scheme
> (> 3 4)
#f
> (> 4 3 1)
#t
> (< 7 5 3 1)
#f
> (>= 4 3)
#t
> (= 3 4)
#f
> (= 3 3)
#f
```

Note that the creators of Racket decided to stick to mathematical convention and did not include a question mark at the end of each arithmetic predicate, even though that is the pattern for most predicate function names.

There are also versions of the some relational predicates for non-numeric data types. The use of these relational predicates are demonstrated below:

```scheme
> (string=? "hi" "hello")
#f
> (string>? "zoo" "yellow")
#t
> (symbol=? 'abc 'def)
#f
> (boolean=? (> 4 6) (< 7 2))
#f
```

Booleans and quoted symbols cannot be compared for anything but equality because they are atomic, unlike strings, which can be compared for size and alphabetical order. It is wise to look up the usage of such predicates in the Help Desk before assuming they exist.
A conditional expression is one whose evaluation depends on the value of a boolean expression—the boolean expression is called the condition. In Racket, a conditional expression can be composed using the `if` or `cond` special forms.

Conditional expressions influence what is known as the flow of control in a program. Usually, functions are evaluated line by line, from top to bottom. When a function includes decision statements like `if` or `cond`, some lines of code are not evaluated because the statements cause the flow of control to skip over certain lines.

The boolean condition in a conditional expression can be simple or complex. A complex boolean condition can be composed using the `and` and `or` special forms, as well as the built-in `not` function.

Frequently, it is useful to nest one `if` within another. In such cases, the resulting Racket expression can become quite complex. Thus, Racket provides another special form, the `cond` special form, to simplify nested conditional expressions.

The evaluation of the `if`, `cond`, and `or` special forms is called lazy or short-circuiting because only the computations needed to ascertain the final value are actually performed. What this means will be discussed along with each special form.

### A. The if Special Form: The basic yes/no decision-maker

Syntactically, the strictest version of an `if` special form looks like this:

```
(if boolExpr thenExpr elseExpr)
```

where:
- `boolExpr` evaluates to a boolean (i.e., either `#t` or `#f`); and
- `thenExpr` and `elseExpr` are any Racket expressions.

The semantics of Racket stipulates that an `if` special form is evaluated as follows:
- First, the boolean expression, `boolExpr`, is evaluated.
- If `boolExpr` evaluates to `#t`, `thenExpr` is evaluated—and the value of the if special form is whatever `thenExpr` evaluates to.
- On the other hand, if `boolExpr` evaluates to `#f`, then `elseExpr` is evaluated—and the value of the if special form is whatever `elseExpr` evaluates to.

The following expressions are examples of using the `if` special form:
- `(if (> 2 4) (* 8 2) (* 6 5))`
- `(if (> 4 2) 'then 'else)`
- `(if #f "then" "else")`

Notice that `boolExpr`, the first parenthesized expression after the `if`, is always evaluated; however, only one of the remaining expressions, `thenExpr` or `elseExpr`, is evaluated during a particular execution.

The following interactions window session demonstrates the evaluation of the if special forms seen above.

```
> (if (> 2 4) (* 8 2) (* 6 5))
30
> (if (> 4 2) 'then 'else)
then
> (if #f "then" "else")
"else"
```

In the first expression, the condition, `(> 2 4)`, evaluates to `#f`. Thus, the else expression, `(> 4 2)`, is evaluated. Its value, `30`, is the value of the entire if expression.
In the second expression, the condition, \((> 4 2)\), evaluates to \#t. Thus, the quoted symbol \'then is returned as the value of the entire if expression.

In the third expression, the condition, \(#f\), evaluates to \#f. Thus, the string, “else”, is the value of the entire if expression.

Using an if expression in the body of a function.

Below, a function, \(\text{howBig}\), is defined and run in the interactions window. If given a number less than 10 as an input, its output is the symbol \'small; otherwise, its output is the symbol \'big (assuming the input is a number\(^6\)).

\[
> \text{(define howBig} \\
> \quad (\text{lambda (num)} \\
> \quad \quad (\text{if (< num 10)} \\
> \quad \quad \quad \text{'}small\text{'} \\
> \quad \quad \quad \text{'}big))))
\]

\[
> \text{howBig 5} \\
\quad \text{small}
\]

\[
> \text{howBig 102} \\
\quad \text{big}
\]

B. The Boolean (Logical) Operators: and, or, not

This section introduces the boolean operators, and, or and not. The first two are implemented as special forms in Racket; in contrast, not is a built-in function.

The not Function

The global environment associates the \(\text{not}\) symbol with a built-in function. When given a boolean value as input, the \(\text{not}\) function returns the opposite boolean value, as demonstrated below in the interactions window:

\[
> \text{(not \#t)}
\quad \#f
\]

\[
> \text{(not \#f)}
\quad \#t
\]

The and Special Form

In the simplest case, the \(\text{and}\) special form takes two boolean inputs: \((\text{and boolOne boolTwo})\), where boolOne and boolTwo can be arbitrarily complex expressions that evaluate to true or false.

If both boolean expressions, boolOne and boolTwo, evaluate to true, then the \(\text{and}\) expression itself evaluates to true. If either boolOne or boolTwo evaluate to false, then the \(\text{and}\) expression evaluates to false. The following interactions window session demonstrates this behavior:

\[
> \text{(and \#t \#t)}
\quad \#t
\]

\[
> \text{(and (> 3 2) (< 5 9))}
\quad \#t
\]

\[
> \text{(and \#t \#f)}
\quad \#f
\]

\[
> \text{(and (> 3 2) (< 4 6) (= 5 9))}
\quad \#f
\]

\[
> \text{(and \#f \#t)}
\quad \#f
\]

\[
> \text{(and (> 2 5) \#t)}
\quad \#f
\]

\[
> \text{(and \#f \#f)}
\quad \#f
\]

\[
> \text{(and (< 2 5) (= 9 (/ 18 2)))}
\quad \#t
\]

\(^6\)If the input argument is not a number, what do you think the outcome of the function will be?
The **and** special form is called a “short-circuiting” function because it can be written to avoid run-time errors. For example, consider the definition of the following function **frac** that matches an input argument to its parameter \( x \) and returns the fraction \( \frac{1}{x} \) if \( x \) is a non-zero number and \( \frac{1}{x} > .4 \), returning \((x + 1)\) if \( x \) is equal to 0 (avoiding an arithmetic divide by zero error):

\[
> (define \textbf{frac}
  \ (\lambda (x)
        \ (if \ (\text{and} \ \ (\text{not} \ \ (= \ x \ 0)) \ (\text{>} \ \ (/ \ 1 \ x) \ .4))
          \ (/ \ 1 \ x)
          \ (+ \ x \ 1)))
> 
> (frac 0)
1
> (frac 1)
1
> (frac 2)
.5
> (frac 3)
4
\]

The function **frac** avoids a “divide-by-zero” run-time error by returning false if \( x = 0 \) and never executing the second boolean expression \((\text{>} \ ((/ \ 1 \ x) \ .4))\) when \( x = 0 \).

The **and** special form, like many of the built-in arithmetic functions, can take more than two input expressions. In such cases, the value of the **and** expression is true if and only if **all** of the input expressions evaluate to true, as demonstrated below:

\[
> (\text{and} \ \ #t \ \ #t \ \ #t \ \ #t)
#t
> (\text{and} \ \ #t \ \ #t \ \ #f \ \ #t \ \ #t)
#f
> (\text{and} \ \ (> \ 3 \ 2) \ \ (= \ 9 \ 9) \ \ (\leq \ 5 \ 20))
#t
\]

**The or Special Form**

The **or** special form evaluates to boolean true if and only if at least one of the input expressions evaluates to true. The behavior of the **or** special form is illustrated below:

\[
> (\text{or} \ \ #f \ \ #f \ \ #f \ \ #f)
#f
> (\text{or} \ \ #f \ \ #f \ \ #t \ \ #f)
#t
> (\text{or} \ \ #t \ \ #t \ \ #t \ \ #t)
#t
> (\text{or} \ \ (= \ 9 \ 8) \ \ (> \ 7 \ 9) \ \ (\leq \ 4 \ 2))
#f
\]

Like the **and** special form, the **or** is short-circuiting, meaning it only evaluates its input, from left to right, until an expression evaluates to **#t**, in which case it returns **#t**.

C. Making Multiple Choice Decisions: The **cond** Special Form

If there are more than two choices for a decision statement, then it is possible to either nest **if** statements, or to use a **cond** special form. The **cond** special form is designed to handle decisions with 2 or more outcomes. For example, consider the following function that consumes a number representing a score and returns a symbol representing the letter grade for that number using a sequence of nested **ifs** to make the decisions:

\[
> (define \textbf{grader-v1}
  \ (\lambda (score)
        \ (if \ (\geq \ score \ 90)
          \ 'A
          \ (if \ (\geq \ score \ 80)
            \ 'B
              \ 'C
            )))
> 
> (grader-v1 90)
'A
> (grader-v1 80)
'B
> (grader-v1 70)
'C
> (grader-v1 60)
'C
> (grader-v1 50)
'C
> (grader-v1 40)
'C
> (grader-v1 30)
'C
> (grader-v1 20)
'C
> (grader-v1 10)
'C
> (grader-v1 0)
'C
\]
(if (>= score 70) 'C
  (if (>= score 60) 'D 'F)))))

> (grader-v1 95)
A
> (grader-v1 75)
C

The grader-v1 function is long enough that it makes more sense to define it in the definitions window. This definition is shown below, where the function grader-v2 is written using a cond statement instead of nested if:

; (grader-v2 score) -> quoted symbol
;-------------------------------------
; score: number
;-------------------------------------
; Return a letter grade for the given score.
;-------------------------------------
; > (grader-v2 99)
; A
; > (grader-v2 10)
; F
;
; Function definition:
(define grader-v2
  (lambda (score)
    (cond
      ; A for score 90-100
      [(>= score 90) 'A]
      ; B for score 80-89
      [(>= score 80) 'B]
      ; C for score 70-79
      [(>= score 70) 'C]
      ; D for score 60-69
      [(>= score 60) 'D]
      ; F for score <= 59
      [else 'F]]))

You should pay attention to the comments before each part of the cond given above. As a general rule, each branch of a decision statement should be preceded by a comment.

The general syntax of a cond special form is as follows:

(cond
  [boolExpr1 expr1]
  [boolExpr2 expr2]
  ...
  [else expr_else])

where:

- each [boolExpr expr] pair is called a clause;
- each boolExpr is a condition that evaluates to a boolean;
- the condition in the last clause is either else or #t; and
- each expr is some Racket expression.

The value of such a cond expression is determined as follows:
1) The first condition, boolExpr1, is evaluated. If it is true, then the value returned by the whole cond expression is the value of the corresponding expression, expr1 and none of the lines following are evaluated; if boolExpr1 is false, expr1 is not evaluated and the flow of control goes to boolExpr2, and so on,
2) for every boolExpr that evaluates to false, the corresponding expr is not evaluated; and
3) if all the boolExprs have evaluated to false, the expr_else is the return value.
Note that each cond clause starts with a [ and ends with a ]. And since each boolExpr is (more often than not) a function call, this is one of the only expressions you will use that has a [ followed by a (, as shown in the second version of the grader-v2 function above. Braces can be used interchangeably with parentheses in Racket.

D. Using the Stepper to Trace Execution of Functions

It is often advantageous to use a feature in DrRacket called “the Stepper”, to see every evaluation involved in the application of a function. The Stepper is not available in the Advanced Student language or Swindle, but it is available in the language Intermediate Student with Lambda. To run the Stepper, follow these directions:

1) Copy and paste the function (and any constants or helper functions it uses) into a new definitions window: Highlight the function, pull down the Edit menu and choose Copy, then pull down the Edit menu to choose New Tab. When your cursor is in the new window, pull down the Edit menu and choose Paste.

2) Choose the language Intermediate Student with Lambda and press Run.

3) Type at least one function call into the new definitions window under the function definition and press Run to make sure the function call is legitimate.

4) Press the Step button. You should see the function with all the arguments in the call substituted for the matching parameters. Press Step to go through the evaluation.

Note that some functions in Swindle and Advanced Student language (ASL) are not defined in the Intermediate Student with Lambda (ISL) language. In particular, the printf, display, and newline functions are not defined and function bodies can contain only one expression in ISL.
X. THE DESIGN RECIPE: HOW TO WRITE A FUNCTION

A. Pre-function Comments

You really can't write a function to perform some computation if you don't already know the correct solution to the function for some subset of possible inputs. For this reason, we will follow the convention that every function starts with a group of comments that give the function name, parameter names, output type, argument data types, and function purpose as shown in section VI-A.

After a function is written, it must be called in order to produce a result. However, calling a function requires that the function already be written. The creators of DrRacket have provided a way for you to write the expressions to test your functions before you write them.

B. Pre-Function Testing: The check-expect and check-within Special Forms

As an example of testing a function before it is written, suppose you wanted to write the grader-v2 function from Section IX. Following the steps in the Design Recipe, we start writing the function with a pre-function comment:

`; (grader-v2 score) -> quoted symbol
;-----------------------------
; score: number
;-----------------------------
; Return a letter grade for the given score.

Now, before writing the actual function definition, we add another step to the Design Recipe—pre-function tests. In order to use pre-function tests in Swindle, you need to:

1) include the line
   (require test-engine/racket-tests)
   at the top of the program, and

2) include the line
   (test)
   at the bottom of the program.

The pre-function tests are intended to be written before you write the function, to show you know what the function should produce for given inputs. Writing these statements before you write the function causes you to think about how a problem should be solved. The lines to pre-test the grader-v2 function should look something like this:

`; Pre-function tests:
(check-expect (grader-v2 90) 'A)
(check-expect (grader-v2 80) 'B)
(check-expect (grader-v2 70) 'C)
(check-expect (grader-v2 69) 'D)
(check-expect (grader-v2 5) 'F)

Note that there is a separate test case for each branch of execution (each cond clause). Also, note that these lines will cause errors until the function definition is written.

The general form of a check-expect statement is:

(check-expect funcCall funcResult)

where funcCall is a call of the function to be written (a non-empty list) and funcResult is the value returned by the function.

You may ask how a function can be called before it is written. The answer is that DrRacket moves every check-expect function to the bottom of the program prior to running the code. This is why the line telling you how many tests passed always comes up last.
When testing a function, it is necessary to test every branch of execution. For example, each clause in a cond expression and each part of an if execution is a branch of execution and each should be tested separately.

The check-expect special form is useful only for computations that produce exact results (e.g., whole numbers, non-repeating decimal numbers, strings, symbols, booleans). There is another form of test statement called check-within that can be used to test the result of computations that may be inexact. The general form of a check-within special form is:

(check-within funcCall funcResult errorTolerated)

where funcCall is a call of the function to be written, funcResult is the value returned by the function, and errorTolerated is the amount the actual computed value can be expected to differ from funcResult. The check-within statements are also run after all function definitions have already been made, even though you write them before writing the function they test.

Neither check-expect nor check-within can be used with a function that has side-effects like printing.

C. Function definition and parameter naming

After the pre-function testing (if applicable) comes the actual function definition. The examples below put together all the information about the Design Recipe that we have covered thus far.

Example 1: Write a function arrange-name that consumes 2 string arguments, first and last representing first and last names, and returns a single string—last, first—the last name, followed by a comma and a space, and then the first name:

(require test-engine/racket-tests)

; (arrange-name first last) -> string
;-------------------------------
; first: string
; last: string
;-------------------------------
; Concatenate the input strings as follows:
; last, first

;Pre-function tests:
(check-expect (arrange-name "Donald" "Duck") "Duck, Donald")
(check-expect (arrange-name "Donald" "Trump") "Trump, Donald")
(check-expect (arrange-name "Matthew" "Vassar") "Vassar, Matthew")

;Function definition:
(define arrange-name
  (lambda (first last)
    (string-append last ", " first)))

(test)

Example 2: Write a function pythagorus to compute the Pythagorean rule:

(require test-engine/racket-tests)

; (pythagorus a b) -> number
;-------------------------------
; a: number
; b: number
;-------------------------------
; Compute the hypotenuse of a right triangle with
; sides of length a and b (sqrt (a^2 + b^2)).

;Pre-function tests:
(check-expect (pythagorus 3 4) 5)
(check-within (pythagorus 2 2) 2 1)
(check-within (pythagorus 1 1) 2 1)
(check-within (pythagorus 5 8) 9 1)
;
;Function definition:
(define pythagorus
  (lambda (a b)
    (sqrt (+ (* a a) (* b b)))))

(test)

Notice that check-within is used to pre-test the pythagorus function in the last 3 pre-function tests because the answer in these cases is not an exact number. Check-expect can be used in the first pre-function test because the answer is exact.

Because the lambda special form is where the names of the parameters are specified, we can use any names we like. Thus, for example, we could have written the lambda form for the pythagorus function as:

(lambda (bob fred) (sqrt (+ (* bob bob) (* fred fred))))

However, using random names such as bob, fred, xyyzy, or ghhtj makes the code unnecessarily difficult to read and code will be graded partially on readability.

Always be sure to include enough comments in the definition of a function so that a person with a reasonable amount of programming knowledge can easily determine the purpose of each line in the function. If the function has a decision statement, it is good practice to include a comment prior to each clause, or a comment to split up the clauses for easier reading.

Often, the comment before a function definition says all there is to say about what the function does. But if you have anything further to tell the reader about what a function is doing, be sure to include one or more comments within the code. Comments can be started after a line of code, but be sure no lines in a program are longer than 80 columns, not even comment lines.

D. Post-function testing — the tester function

After writing a program that contains only pre-function tests, all you get to see after running the code is the “All tests passed!” line generated by the check-expect statements. It is somewhat more satisfying to see the function call and also the result of that call and it is necessary when functions have only side-effects.

The file print-and-test.txt is provided for you on labs and assignments so that you can see the results of expression evaluation. In particular, the tester function is used to both print and evaluate an expression, making use of the eval function:

; (tester expr) -> valid Racket data type
; --------------------------------------------
; expr: a QUOTED valid Racket entity
; --------------------------------------------
; Produces the result of evaluating expr
; --------------------------------------------
; Displays the original Racket entity as a printf side-effect

Function definition:
(define tester ;line 1
  (lambda (expr) ;line 2
    (printf "A ===> " expr) ;line 3
    (eval expr))) ;line 4

In line 3 of the tester function, the quoted expression is printed. The action of the printf function ensures that no quote will be printed before the expression. In line 4, the eval function is used to strip the quote from expr and evaluate
the unquoted result.

The tester function was written by Prof. Luke Hunsberger and it provides a way to display and then evaluate functions in the interactions window. In order to use the functions in the print-and-test.txt file in your programs, you must include the line (load "print-and-test.txt") at the beginning of your program and be sure that the print-and-test.txt file is in the same directory as the program that contains the load statement. Using the tester function is shown in the grader-v3 function defined in the next section.

E. Literal values in function definitions

It is considered poor programming practice to include numbers, quoted symbols, strings, and images inside a function definition, although at times it is OK to do so. The next section shows how to define names for literal values.

An example function that uses too many literal values is the grader program from Sect. IX-C. After this program is written, it is conceivable that the cut-offs for each letter grade could change and it is even possible that the grade designations could change. For these reasons, it is better to precede the function with constant definitions as shown below in the grader-v3 procedure below. This example shows how the tester function is used after the function definition. Pre-function tests are not possible because of the side-effect printing inside the function.

```scheme
;; (grader-v3 score) -> quoted symbol
;; -----------------------------
;; score: number
;; -----------------------------
;; Consumes a numeric score between 0 and 100
;; and returns a symbol representing the grade.
;; -----------------------------
;; Side-effect is printing the score and a comment.
;; > (grader-v3 90)
;; Oh my gosh! You did great!!!
;; A
;;
;; The function call to load allows use of tester function.
;; (load "print-and-test.txt")

;; Function constant definitions:
(define A 90) ; low score cut-off for A
(define B 80) ; low score cut-off for B
(define C 70) ; low score cut-off for C
(define ASYM 'A) ; representation of highest grade
(define BSYM 'B) ; representation of good grade
(define CSYM 'C) ; representation of average grade
(define DFSYM 'belowC) ; representation of low grade

;; Pre-function tests not possible due to side effect
;; printing.
;
;; Function definition:
(define grader-v3
  (lambda (num)
    (cond
      ;----------num=90->100------------------
      [(>= num A)
        (printf "Oh my gosh! You did great!!!~%")
        ASYM]
      ;----------num=80->89------------------
      [(>= num B)
        (printf "B is pretty good!! ~")
        BSYM]
      ;----------num=70->79------------------
      [(>= num C)
        (printf "C is considered average!~")
    ]
  )
)
```
The `grader-v3` procedure uses seven constants that are defined immediately before the function definition. These constants are used in the procedure in place of their literal values. In the `grader-v3` function, both numbers and quoted symbol literals are declared as constants before the function definition. Note that the constant names are defined in the global environment when the program is run and can be used anywhere inside the function definition or in subsequent definitions in the same program file.

Small constant numbers such as 0 and 1 are frequently used directly in code. But most literal values should be declared as constants outside the code, along with comments explaining the reason for creating these constants.

Once you have declared a name for a function or constant, that name may not be redefined inside that file. Also, if any files are loaded at the start of a program, you will not be allowed to define functions or constants with the same names as those defined in the loaded file.
XI. ADDING CHANCE TO FUNCTIONS: THE *random* FUNCTION

A game would be pretty boring if everything happened exactly the same way every time you played it. Likewise, we can write some interesting functions by using a primitive function called *random*.

The *random* function consumes a positive natural number \( n \) and generates a positive natural number between 0 and \( n - 1 \).

**Exercise**: Generate a random number that could be obtained from the roll of a single six-sided die.

**Exercise**: Generate a random number that could be obtained from the roll of two six-sided dice.
XII. Recursion

This section introduces recursive functions. Defining recursive functions in Racket requires no new computational constructs (i.e., no new special forms); instead, we simply combine existing constructs in a new way. In many cases, recursive functions can provide compact and elegant solutions to interesting computational problems.

We begin by recalling that the evaluation of a non-empty list according to the Default Rule typically involves the application of a function to zero or more inputs.

For convenience, we make the following definition:

Suppose expr is a Racket expression that denotes a non-empty list, $L$, whose evaluation is governed by the Default Rule. Then we say that expr is a function-call expression. Furthermore, suppose $f$ is the function that results from evaluating the first element of the list $L$. Then we say that expr calls $f$.

Thus, for example, the expression, $(+ 2 3)$, is a function-call expression that calls the built-in addition function. Similarly, $(\text{symbol? 'x})$ is a function-call expression that calls the built-in symbol? function. In contrast, the expressions, $(\text{define myVar 3})$ and $(\lambda (x) (* x x))$, denote special forms and, thus, are not function-call expressions.

A. Recursive Functions

A function, $f$, is said to be recursive if its body contains a function-call expression that calls $f$. To put this another way, a recursive procedure is a procedure that applies itself.

At first glance, this might seem like a crazy idea—after all, a function calling itself sounds like the kind of circularity that might lead to infinite loops. Such is the case with the following function, called goodbye:

```racket
; (goodbye) -> void
; No input; 0-parameter function
; -----------------------------
; Illustrates execution of infinite loop.
;
; Pre-function tests: Not possible due to void output and infinite loop.
;
; Function definition:
(define goodbye
  (\lambda ()
    (goodbye))
)
;
; Post-function tests: Not possible due to infinite loop.
```

If you call the goodbye function in the interactions window, the session will continue to run as long as the computer memory lasts. One way to detect such a non-stop execution in DrRacket is to observe the little green man in the lower right corner of the window...if the man is continually running, the program is not yet stopped. One way to stop such a runaway program is to press the square Stop icon on the upper right corner (you may have to press this button repeatedly).

The dreaded form of circularity exhibited by the goodbye procedure is generally easy to avoid, as explained below:

A recursive function typically includes a conditional statement that tests some stopping condition (or base case) first. If the stopping condition evaluates to true, then no recursive function call is made.

In cases where a recursive function call is made, it involves applying the function to different input arguments that are closer to the base case than the current argument.

Thus, as will be demonstrated, a typical sequence of recursive function calls is less like a circle that forever loops back on itself, and more like a spiral that converges on some stopping point.

Defining Recursive Functions in Racket.
In Racket, the typical characteristics of the definition of a recursive function, $f$, are (in order):
1) a \texttt{define} special form that names \( f \) as a lambda expression with one or more parameters;

2) a \texttt{cond} or \texttt{if} expression (in the function body) that:
   a) includes one or more clauses for the the base case(s) and
   b) includes one or more clauses with recursive function-call expressions that apply \( f \) to other input(s) such that these inputs are closer to the value of the base case.

No new Racket constructs are required to support recursion.

\textbf{B. Recursion over numbers}

\textbf{Example: The factorial function.} In mathematics, the factorial function, \( f(n) = n! \), is frequently defined as follows:

\[
f(n) = n! = n \times (n - 1) \times (n - 2) \ldots \times 3 \times 2 \times 1
\]

We can give a recursive definition of the factorial function, as follows:

- Base Case \((n = 1): 1! = 1\).
- Recursive Case \((n > 1): n! = n \times (n - 1)!\)

According to this definition, the following equalities hold:

- \(4! = 4 \times 3!\)
- \(3! = 3 \times 2!\)
- \(2! = 2 \times 1!\)
- \(1! = 1\)

Putting all of this information together yields:

\[
4! = 4 \times 3! = 4 \times (3 \times 2!) = 4 \times (3 \times (2 \times 1!)) = 4 \times (3 \times (2 \times 1)) = 24.
\]

The following Racket expression defines a recursive function, \texttt{facty-v1}, whose definition is based on the above insights. The main job of \texttt{facty-v1} is to use recursion to compute the factorial of its input, \( n \). We will use the design recipe to create this function in the Definitions Window:

\begin{verbatim}
; line to allow check-expect statements:
(require test-engine/racket-tests)

; line to load print-and-test.txt functions:
(load "print-and-test.txt")

; (facty-v1 n) -> positive integer
--------------------------
; n: positive integer
--------------------------
; Compute the factorial of n.
;
; Pre-function tests:
(check-expect (facty-v1 1) 1)
(check-expect (facty-v1 4) 24)
(check-expect (facty-v1 5) 120)
;
; Function definition:
(define facty-v1
  (lambda (n)
    (if (= n 1)
      1
      (* n (facty-v1 (- n 1))))))
;
; Post-function tests:
(tester '(facty-v1 1)
(tester '(facty-v1 3)
(tester '(facty-v1 4)

(test)
\end{verbatim}
In this function, an \texttt{if} decision statement is used instead of a \texttt{cond}. Either version works because the \texttt{cond} and \texttt{if} are interchangeable.

Each call of a recursive function to itself is known as an \textit{iteration}. Thus, we can refer to the number of iterations of a recursive call on any particular value.

Notice that the define special form gives the name, \texttt{facty-v1}, to the function defined by the \texttt{\lambda} special form. Notice, too, that the body of this function includes a decision that distinguishes the base case (i.e., when \( n = 1 \)) from the recursive case (i.e., when \( n > 1 \)). Finally, notice that the body includes a function-call expression that calls \texttt{facty-v1} with a value that is closer to 1, by using subtraction.

Okay, so what happens when the \texttt{facty-v1} expression is evaluated? Well, the expression is a \texttt{define} special form. So in the execution of the \texttt{define}, the name, \texttt{facty-v1}, is not evaluated but instead is written into the global environment. Then the third element of the \texttt{define} special form—namely the \texttt{\lambda} expression—is evaluated. Like any \texttt{\lambda} expression, the one above evaluates to a function. However, it is important to remember that evaluating the above \texttt{\lambda} expression only creates a function in the global environment. It does not call the function! Thus, the expressions in the body of the \texttt{\lambda} expression are not evaluated until the \texttt{tester} and \texttt{check-expect} functions calls are made.

After the lambda expression has been evaluated (to a function), the evaluation of the define special form can continue: in particular, by entering a value (i.e., the newly created function) for the parameter of \texttt{facty-v1}, in the global environment.

Before delving deeper into why \texttt{facty-v1} works, observe that we can define an equivalent function, \texttt{facty-v2}, using a \texttt{cond} expression, as follows:

\begin{verbatim}
(define facty-v2
  (lambda (n)
    (cond
      ;; Base Case: n=1
      [(= n 1) 1]
      ;; Recursive Case: n > 1
      [else (* n (facty-v2 (- n 1)))])))

(test)
\end{verbatim}

Another equivalent version of the factorial function is given below, this time called \texttt{facty}. This function differs only in that it contains some \texttt{printf} expressions that will help trace what happens when an expression such as \texttt{(facty 3)} is evaluated:
(load "print-and-test.txt")
(define facty
  (lambda (n)
    (cond
      ; Base Case: n = 1
      [(= n 1)
        (printf "Base Case (n = 1)\n")
        1]
      ; Recursive Case: n > 1
      [else
        (printf "Recursive Case (n = \"A\")\n"
          (* n (facty (- n 1))))])))

; Post-function tests:
(tester '(facty 1))
(tester '(facty 3))
(tester '(facty 6))

Evaluating (facty 3). Consider DrRacket’s evaluation of the expression, (facty 3). This is a function-call expression whose evaluation is governed by the Default Rule. Thus, the symbol facty and the number 3 must both be evaluated. The symbol facty evaluates to the function defined above; and the number 3 evaluates to itself. Next, the facty function is applied to the input 3.

The application of the facty function to the input 3 is depicted in Fig. 14. First, a local environment is created with an entry associating the input parameter n with the value 3. Next, the expression in the body of the facty function is evaluated with respect to that local environment.

---

Fig. 14. The local environments created in the evaluation of (facty 3). Picture courtesy of Prof. Hunsberger.
Since the value of n is 3 in the top local environment shown in Fig. 14, the condition, (= n 1), evaluates to #f. So flow of control skips to the second clause, else, which of course evaluates to #t. The expressions associated with the recursive case are evaluated from top to bottom. The first expression causes the line, “Recursive Case (n = 3)”, to be displayed in the interactions window. Then, the second expression, (* n (facty (- n 1))), must be evaluated—according to the Default Rule. The * symbol evaluates to the multiplication function, n evaluates to 3, and (facty (- n 1)) evaluates to ... Gosh, we need a new paragraph!

The expression, (facty (- n 1)), is evaluated according to the Default Rule. First, the facty symbol evaluates to the facty function; and (- n 1) evaluates to 2 (since n has the value 3). Next, the facty function must be applied to the input value 2, as depicted in the second box in Fig. 14.

Note that the evaluation of the expression, (* (facty (- n 1))), in the top function-call box cannot continue until the subsidiary expression, (facty (- n 1)), is evaluated. However, this value cannot be known until the output value for the second function-call box has been generated! In other words, the evaluation of the expression in the top box must be suspended, pending the outcome of the second box.

The application of the facty function to the value 2, depicted in the second function-call box in Fig. 14, is similar to the application of the facty function to 3 in the top box, except that the local environment in the second box associates the input parameter, n, with the value 2.

Crucially, the local environments in separate function-call boxes do not cause a conflict! They can’t see one another. Neither knows that the other even exists! Thus, although the two input parameters are both called n, they are quite distinct!

The evaluation of the body of the function in the second box proceeds in the environment where n has the value 2. The base case is skipped and the expression associated with the recursive case is evaluated. The evaluation of the expression, (* n (facty (- n 1))), leads to yet another recursive function call—this time the application of the facty function to the input value 1, as illustrated in the third box in Fig. 14.

Once again, the evaluation of the expression, (* n (facty (- n 1))), in the second box cannot continue until the output value for the third box has been generated. In other words, the evaluation of the expression in the second box must be suspended, pending the outcome of the third box.

The application of the facty function to the value 1 begins by creating a local environment entry that associates the input parameter n with the value 1. (Again, this is a new input parameter, distinct from the other n’s) Next, the cond expression in the body of the function is evaluated. This time, however, the condition (= n 1) evaluates to #t; thus, the base case expression is evaluated. The expression, 1, evaluates to itself, yielding the output value for the application of the facty function to the value 1 (i.e., the output value for the third box).

This output value, 1, is the value of the expression, (facty (- n 1)), that was being evaluated in the middle function-call box (where n has the value 2). Now that the value of (facty (- n 1)) is in hand, the evaluation of the expression, (* n (facty (- n 1))), in the middle box can continue. In that local environment, the multiplication function is applied to 2 and 1, yielding the output value 2 for the middle function-call box.

This output value, 2, is the value of the expression, (facty (- n 1)), that was being evaluated in the top function-call box (where n has the value 3). Now that the value of (facty (- n 1)) is in hand, the evaluation of the expression, (* n (facty (- n 1))), in the top box can continue. In that local environment, the multiplication function is applied to 3 and 2, yielding the output value 6 for the top function-call box. Phew!

Summary of features typical for recursive functions:
- The body of the function contains a conditional expression that enables a stopping condition—commonly called a base case—to be recognized. If that stopping condition evaluates to #t, then no more recursive function calls are made.
- The body of the function contains an expression that involves a recursive call to that same function—but with different input(s). It is crucial that the inputs to the recursive function call be different in some way; otherwise,
that recursive function call would lead to another identical recursive function call, and so on, infinitely.

- Because the inputs to the recursive function call use arguments that are closer to the base case, the recursive function call is not circular; instead, the sequence of recursive function calls is more like a spiral that eventually stops when the base case is arrived at.
- Although the expression in the body of the function is identical in each recursive function call, it is evaluated with respect to a different local environment. In other words, the evaluation of the body is affected by the value of the input parameter(s). This helps to avoid circularity and infinite loops.

**Exercise: Summing Squares.** Consider the function, \( g(n) = 1^2 + 2^2 + 3^2 + \ldots + n^2 \). The function \( g(n) \) sums the squares of the integers between 1 and \( n \), inclusive. We can define \( g \) recursively, as follows:

- Base Case (\( n=1 \)): \( g(1) = 1 \)
- Recursive Case (\( n>1 \)): \( g(n) = n^2 + g(n-1) \)

Notice that \( g(1) = 1 \), \( g(2) = 2^2 + 1^2 = 5 \), \( g(3) = 3^2 + 2^2 + 1^2 = 14 \), and so on. Define a recursive function, called `sum-squares`, that computes the sum of the squares from its input value \( n \) down to 1. Use the design recipe in your solution.

### C. Accumulator-style (Tail-Recursive) Functions over numbers

In the previous section, we saw functions that used recursion over numbers. In those functions, many local environments were created that could not be resolved until the evaluation of a subsidiary expression was completed. In this section, we consider a different style of recursion that creates just as many local environments as did the functions in the previous section, but in which only one local environment exists at the end of the computation. The difference in the recursive functions we’ll consider in this section is that they use an extra parameter that serves as an accumulator for the value being computed.

We begin by looking at an accumulator-style function to calculate \( n! \), called `facty-acc` (where the acc is to indicate an accumulator function):

```racket
(require test-engine/racket-tests)

;; (facty-acc n acc) -> integer
;;-------------------------------
;; n: positive integer
;; acc: positive integer, initial value = 1
;;-------------------------------
;; Compute the factorial of n, where the base case
;; value is stored in acc.
;;
;; Pre-function tests:
;; (check-expect (facty-acc 1 1) 1)
;; (check-expect (facty-acc 4 1) 24)
;; (check-expect (facty-acc 5 1) 120)
;;
;; Function definition:
(define facty-acc
  (lambda (n acc)
    (cond
      ;; Base Case: n=1 Return the accumulator
      [(= n 1) acc]
      ;; Recursive Case: n > 1
      [else
       ;; Recursive function call (tail-recursive)
       (facty-acc (- n 1) (* n acc))])))

;; Post-function tests:
(tester '(facty-acc 1 1))
(tester '(facty-acc 3 1))
```
An expression of the form, \((\text{facty-acc } n 1)\), will evaluate to the factorial of \(n\). In other words, the initial value of the accumulator must be 1 for this function to achieve its desired result.

Notice that the function, \(\text{facty-acc}\), is tail recursive because the recursive function-call expression, \((\text{facty-acc } (- n 1) (* n \text{ acc}))\), is not a subsidiary expression within some larger expression. Thus, the value of the last recursive function-call expression is the output value.

For \(\text{facty-acc}\), the accumulator, \(\text{acc}\), is multiplied by \(n\) to generate the value of the accumulator for the next recursive function call. Since \(\text{facty-acc}\) involves multiplying the current accumulator to generate the value of the next accumulator, the appropriate initial value for the accumulator is 1 (not 0). Thus, to use \(\text{facty-acc}\) to compute \(3!\), we would evaluate an expression such as \((\text{facty-acc } 3 1)\), as illustrated in Fig. 15.

The difference between this execution and the execution of \(\text{facty}\) shown in the last section is that only one local environment exists at a time and therefore there is no problem filling up memory unless the function is written incorrectly so that it continues infinitely.

**Exercise: Summing squares** The sum of squares from 1 to \(n\) is given as \(1^2 + 2^2 + \ldots + n^2\). Write a tail-recursive function that uses an accumulator for computing the sums of squares from 1 to \(n\). As you think about this problem, ask yourself: what should the base value of the accumulator be?

### D. Wrapper functions

One bothersome characteristic of accumulator-based functions is that the accumulators need to be given appropriate initial values to produce the correct results, meaning that the user must be aware of these values. Fortunately, this problem is easily overcome by providing a wrapper function. A wrapper function is a function that calls the
actual accumulator function and is designed to properly initialize any accumulators so that the user of an accumulator-based function need not remember the appropriate initial value. This section presents a wrapper function for the accumulator-based function seen earlier.

**Example: A wrapper for facty-acc.** The following code segment defines a wrapper function, facty-wr, for the accumulator-based function, facty-acc, defined in the last section. Notice that the wrapper function simply calls facty-acc with the accumulator appropriate initialized to 1.

```scheme
(require test-engine/racket-tests)

;; (facty-wr n) -> integer
;; -----------------------
;; n: positive integer
;; -----------------------
;; Compute the factorial of n.
;;
;; Pre-function tests:
;; (check-expect (facty-wr 2) 2)
;; (check-expect (facty-wr 4) 24)
;; (check-expect (facty-wr 6) 720)
;;
;; Function definition:
;; (define facty-wr
;;   (lambda (n)
;;     (facty-acc n 1)))
;;
;; Post-function tests:
;; (tester '(facty-wr 2))
;; (tester '(facty-wr 4))
;; (tester '(facty-wr 6))
;
;;(test)
```

The wrapper function shields the user from knowing the value of the accumulator. In fact, the user of facty-wr should not even be aware that an accumulator is being used at all. A basic tenet of software engineering called *information hiding* encourages the practice of allowing the user of software to be blissfully ignorant of how the software really works. So consider yourself on your way to becoming a software engineer!!
You have already seen that there are many different ways to use lists. In this chapter, we will discuss quoted lists that contain no internal lists...these are also known as “flat lists”.

We have already seen that an empty list is a primitive datum in Racket, denoted by ’(). This section introduces non-empty lists as chains of pairs. In this context, a pair is a data structure that contains two parts, commonly called first and rest. The first part of a pair is used to hold an element of a list; the second part of a pair is used to hold the rest of the list, itself a list. Thus, the first part of a pair can be any kind of Racket entity; in contrast, the rest of a pair must be a list (either empty or non-empty). If a pair has its rest equal to the empty list, then that pair is the last link in the chain.

There are a few different ways to denote a list that is composed of pairs:
- use a single quote before the open parenthesis that starts the list: ’(1 2 3 4),
- use the function list after the open parenthesis: (list 1 2 3 4), or
- use repeated application of the cons function, ending with the empty list: (cons 1 (cons 2 (cons 3 (cons 4 empty))))

Note that in a list that contains quoted symbols, the single quote before the left parenthesis causes the list to be evaluated as if every symbol was quoted. Therefore, the list ’(a b c) is equivalent to the list (list ’a ’b ’c), which in turn is equivalent to the list (cons ’a (cons ’b (cons ’c empty))). When the list or cons functions are used to specify a list of quoted symbols, each quoted symbol must be preceded by a quote.

Although it may seem strange to represent a non-empty list in terms of its first part (i.e., the first element of the list) and its rest part (i.e., the rest of the elements in the list), this kind of representation is extremely advantageous because it allows us to define recursive functions on lists. List-based recursion is quite similar to numerical recursion.
- There is a base case: the empty list (analogous to n = 0); and
- there is a recursive case: a non-empty list (analogous to n > 0).

Frequently, even quite simple recursive functions can process lists of any length. To support the use of lists and list-based recursion, Racket provides the following built-in functions:
- cons: a function that creates an instance of a pair (i.e., a link in the chain for a non-empty list);
- empty?: a type-checker predicate for the empty list;
- cons?: a type-checker predicate for pairs;
- first: a function that provides access to the first part of a pair; and
- rest: a function that provides access to the rest of a pair.

Surprisingly, these functions are all that is required to support list-based recursion.

A. I thought we were already using lists...

It is true that the Racket programming language uses lists all over the place. We define functions using the define and lambda special forms, which are examples of lists. All of the other special forms are also lists. We also use lists to apply functions to inputs, courtesy of the Default Rule for evaluating non-empty lists. So, why do we need anything else?

These uses of lists have so far enabled us to define functions and apply them to inputs, but they haven’t enabled us to process lists as containers of data. When viewing lists as containers of data, we typically don’t want their contents to be evaluated all at once. So far, the only way we have seen of shielding a list from evaluation has been by using a single quote before the list. However, the single quote notation is too limiting because it shields everything in the list from evaluation. For example, in the expression, ’(a b c), the quote shields the symbols a, b and c from evaluation, treating them as quoted symbols. However, oftentimes, when creating lists as containers of data, we want to create, for example, lists containing the placeholders, or constants, a, b, and c. In such cases, we want these symbols to be evaluated and replaced by their associated values in the global environment. In addition, we have not yet seen any way of accessing the elements of a given list, which is essential to doing any meaningful computation on lists-as-data.
B. Using cons to Create Pairs (i.e., Cons Cells)

Historically, the pairs that serve as links in the chains of non-empty lists have been called cons cells. The cons function has the following contract:

- (cons x l) -> list
- -------------------
  - x: any data type
  - l: list
  -------------------
  - Constructs a list.

⇒ The only requirement when using the cons function is that the second argument must be a list! So we need a list to create a list. Luckily, there is the empty list that can be used as the “innermost” list.

The following interactions window session demonstrates that the cons cells created using the cons function are treated as non-empty lists by DrRacket.

> (cons 8 '(a b c))
(list 8 'a 'b 'c)
> (cons 2 '(3 4 a b c))
(list 2 3 4 'a 'b 'c)
> (cons 64 '())
(list 64)

Notice that, in each example, the list represented by the newly created cons cell contains one more element than the second argument to cons, a list. For example, the cons cell (i.e., the non-empty list) created by (cons 8 '(a b c)) contains four elements, one more than '(a b c) contains. Similarly, the cons cell (i.e., the non-empty list) created by (cons 64 '()), also written (cons 64 empty), contains one element, which is one more element than the empty list contains.

You will probably use the list function to create lists much more often than you use cons. The list function is actually using the cons function behind the scenes for each item added to the left side of a list. For example, (cons 1 (cons 2 (cons 3 empty))) produces the same result as the expression (list 1 2 3). Even the presence of the empty list is hidden when using the list function. An even more concise way to create lists is by typing a single quote before the left parenthesis. An equivalent way to create the lists shown above is '1 2 3).

Don’t forget how to use the cons function, because it will be necessary when writing recursive functions that consume lists.

C. Picturing Non-Empty Lists

Fig. 16 shows two different ways of depicting the non-empty list, (list 3 4 6). The top of this figure shows the list as a set of boxes, each containing a smaller box that is the rest of the list. The bottom shows the list as a chain of cons cells.

![Fig. 16. The non-empty list (list 3 4 6) as a box containing smaller boxes (top) and as a chain of cons cells (bottom). Figure courtesy of Prof. Hunsberger](image)

Notice that in the top of Fig. 16 the list is indeed represented as a cons cell (the biggest one in the picture). The first element of this cons cell is 3, the rest of this cons cell is itself a cons cell (i.e., non-empty list)—namely, the cons cell whose first element is 4 and whose rest is (drumroll, please) another cons cell! This innermost cons cell has as its first
part, 6, and its rest, empty (i.e., the empty list). Since the rest of this cons cell is the empty list, it is the end of the list. Notice that the list represented by this box of cons cells has three elements: 3, 4 and 6. Notice, further, that it also has three cons cells!

⇒ A list containing n elements is represented by a chain of n cons cells--one cons cell per element in the list.

Although the top part of Fig. 16 is an accurate depiction of a chain of cons cells for a non-empty list, this kind of picture would get awfully difficult to draw for lists containing more than five or ten elements. For this reason, we prefer to depict chains of cons cells using arrows, as illustrated in the lower part of Fig. 16. It is important to realize that the non-empty list depicted in the upper figure is the same list as that depicted in the lower one (i.e., we have two kinds of picture-syntax for one semantic list!).

The following interactions window excerpt shows how to name a list using the define special form and then shows the action of the functions first, rest, cons? and empty? on the (list 3 4 6):

> (define exlist (list 3 4 6))
> (first exlist)
  3
> (rest exlist)
  (list 4 6)
> (first (rest exlist))
  4
> (rest (rest exlist))
  (list 6)
> (first (rest (rest exlist)))
  empty
> (rest (rest (rest exlist))))
  rest: expects a non-empty list; given: empty
> (cons? exlist)
  true
> (empty? exlist)
  false

D. The Self-Referential Structure of Lists & List Recursion

It is important to remember that the empty list is at the end (or core) of every quoted non-empty list. This fact is critical because it allows us to do recursion over lists. Remember that a recursive function requires a base case and a recursive case. For numbers, the base case is usually either 0 or 1 and the recursive case calls the function on a whole number > 0 or > 1. For a list, the base case is generally the empty list, and the recursive case calls the function on the rest of the list.

A data definition for a list of any data type could be phrased as follows:

A list is either:
1. empty, or
2. (cons x list), for any data item x and any list.

As a simple example of list recursion, write a function called list-length to find the length of a list (this function is already provided for you with the name length):

(recommend test-engine/racket-tests)

; (list-length LST) -> number
; ----------------------------
; LST: list
; ----------------------------
; Compute the number of elements in LST
; Pre-function tests:
(check-expect (list-length empty) 0)
(check-expect (list-length (list 3 4 6)) 3)
(check-expect (list-length (list 8 'a 3 42 "whoa")) 5)
; Function definition:
(define list-length
  (lambda (LST)
    (cond
      ; Base case: return 0 because the length of empty list is 0
      [(empty? LST) 0]
      ; Recursive case: add 1 for first and make recursive call on rest
      [else (+ 1 (list-length (rest LST)))]))))

; Post-function tests:
(tester '(list-length empty))
(tester '(list-length (list 1 2 3)))
(test)

Let's look at another example. This time we want to be more specific about the contents of the list, so we’ll begin with a data definition:

A list of numbers (LON) is either:
1. empty, or
2. (cons number LST), where LST is a LON.

Because of this definition, we can start with an empty list and use this data definition to construct any list of numbers...and that’s the purpose of making the data definition in the first place. Writing the definition above in comments allows us to refer to an LON as a data type in any subsequent function contract.

Write a function called \texttt{sum-list} to sum all the elements in a LON.

E. Accumulator versions of functions that consume lists

We need a base case when doing accumulator-style recursion over lists, just like we did when doing recursion over numbers. Write a tail-recursive accumulator function that consumes a list of numbers and that returns the sum of all the numbers in the list:

; (sum-list-acc listy) \rightarrow number
; ---------------------------------------------
; listy: LON
; ---------------------------------------------
; Accumulator-style version of function to sum all the
; numbers in listy.

; Pre-function tests:
(check-expect (sum-list-v2 '(3 3 3 3)) 12)
(check-expect (sum-list-v2 '(54 13 4)) 71)
(check-expect (sum-list-v2 '()) 0)

; Function definition:
(define sum-list-v2
  ; initial parameter name is listy
  (lambda (listy)
    (local
      [(define help-sum
        ; inside help-sum, listy is named lst
        (lambda (lst acc)
          (cond
            ; base case: lst empty so return acc
            [(empty? lst) acc]
            ; recursive case: pass in rest of lst as first argument
            ; and pass the result of adding the first number on lst
            ; to acc as the second argument
            [else
              (help-sum (rest lst) (+ (first lst) acc)))]))
      ; call to internal function help-sum, pass in whole list listy as
      ; the first argument and pass in 0 for the initial value of acc,
      ; the second argument
      (help-sum listy 0))))

F. Functions that return lists

We have seen functions that consume lists...but we should also look at functions that produce lists (and some
functions do both!)

Functions that return lists most often use the cons function. For example, consider the function below that generates
a list of random numbers between 1 and 100:

; Contract: (gen-ran-list natural-number) -> listof numbers (lon)
; Header: (define gen-ran-list (lambda (n) ...))
; Purpose: Generates a list of n random numbers between 1 and 100.
; Pre-function tests: Not possible to test contents of list because they
; as shown:
; (check-expect (length (gen-ran-list 8)) 8)
; length is primitive function that returns the length of a given list
;
; Function definition:
(define gen-ran-list
  (lambda (n)
    (cond
     [(= n 0) empty]
     [else
      (cons (add1 (random 100)) (gen-ran-list (sub1 n)))])))

Any time we need to perform some operation a certain number of times, we need to use recursion (in Racket). In
the function above, the base case is when n = 0, and the recursive case conses the next random number (generated
using add1—why?) onto a recursive call to the function on n - 1.
XIV. LISTS WITHIN LISTS—DEEP LISTS

Some quoted lists contain sublists, and processing these sublists requires slightly different recursion patterns.

A. Arbitrarily Nested Lists
   Go over template for processing lists that may have arbitrary nesting patterns.
   Main difference is that the first of the list may itself be a list, and this must be dealt with in functions that consume deep lists.

B. Lists Containing Lists of Equal size
   Go over template for processing deep lists in which every sublist is a pair.
XV. STRUCTS: CONTAINERS FOR FIXED-SIZE DATA

So far, we’ve seen primitive and compound data. Primitive data types are single entities that are in simplest form. The compound data we’ve seen (strings and quoted lists) are containers for an arbitrary number of characters (in the case of strings) or an arbitrary number of valid Racket entities (in the case of lists). There should be a mechanism that allows us to create a data type that has a fixed number of fields, such as a coordinate in the x-y plane. One solution would be to make a number of 2-element lists, so that we could access the first element (the x coordinate) and the first of the rest (the y coordinate). As the fixed number of elements became larger, the notation to access the parts would get very complex.

A. Defining new data types—defstruct

In Racket, you can package a finite number of elements in an entity called a struct, in which each field has a unique name. When you define a new struct, you are really defining a new data type.

To define a struct to represent a point in the x-y plane in the Swindle dialect of Racket, you would use the following syntax:

(defstruct <posn> () x y)

In this statement, the name of the struct defined is posn and the fields are x and y. The defstruct special form creates a number of functions that can be used on a posn and these are given below:

- make-posn: A constructor function whose purpose is to create new items of the posn type. Examples of using this function are given below:
  
  (make-posn 1 2) ; creates a posn with x field 1 and y field 2
  (make-posn 5 8) ; creates a posn with x field 5 and y field 8

Of course, making a posn without a name is not always a good strategy, particularly if you are going to refer to the same posn more than once in a program. To define names for struct objects, you need to use the define special form, as shown below:

(define point1 (make-posn 1 2)) ; creates and names a posn
(define point2 (make-posn 5 8)) ; creates and names a posn

- posn-x and posn-y: Functions known as “accessors” because they return the value of the x or y field in a particular posn. Examples of using these functions on the point1 and point2 posns in the interactions window are given below:

  > (posn-x point1)
  1
  > (posn-y point1)
  2
  > (posn-x point2)
  5
  > (posn-y point2)
  8

- posn?: A type-checker for posns, this function can consume any data type and returns a boolean—#t or #f.

  > (posn? point1)
  #t
  > (posn? "canary")
  #f

- set-posn-x! and set-posn-y!: Functions known as “mutators” because they change the value of a field in an existing posn. As shown in the example below, these functions return void but have the side effect of changing the field values in a posn struct.

  > (set-posn-x! point1 100)
  > (posn-x point1)
We will explore the actions of mutator functions in an upcoming Chapter.
A. The let special form

The purpose of a let special form is to set up a local environment, much like those that exist inside a function-call box, and then to evaluate one or more expressions with respect to that local environment. The value returned by a let expression is the value of the last expression in its body. Once a let expression has been evaluated, its local environment vanishes. The let expression is like a lambda expression without any parameter list...it can’t be called with arguments because they are already written inside the let.

When introducing any special form, it is important to specify both the syntax and the semantics.

The syntax of let

\[
\text{let } [(\text{var}_1 \\text{val}_1) \\
(\text{var}_2 \\text{val}_2) \\
... \\
(\text{var}_n \\text{val}_n)] <\text{body}>)
\]

where:
- \(\text{var}_1, \ldots, \text{var}_n\) are character sequences representing \(n\) distinct Racket symbols, for \(n \geq 0\);
- \(\text{val}_1, \ldots, \text{val}_n\) are \(n\) Racket expressions; and
- \(<\text{body}>\) is a Racket expression, the value returned from the let.
- the use of [] brackets is optional...the expression can be written completely using only parentheses to divide the let expression into parts.

Notice that a let can include zero or more var/val pairs; however, the body of a let can include more than one expression only if it contains a begin.

Example: Some legal let expressions

The following expressions are all legal let expressions:

- (let ()
true)
- (let [(x (+ 2 3))]
  (* x x))
- (let [(x (+ 2 3))
  (y 3)
  (z (* 2 2))]
  (begin
    (printf “x: ~A, y: ~A, z: ~A\%” x y z)
    (+ x y z)))

The first let expression includes no var/val pairs, as indicated by the empty subsidiary list. Its body consists of the single expression, true. The second let expression includes a single var/val pair: (x (+ 2 3)). Its body consists of the single expression, (+ x x). The third let expression includes three var/val pairs: (x (+ 2 3)), (y 3) and (z (* 2 2)). Its body consists of one begin expression holding two subsidiary expressions: a printf expression and (+ x y z).

The semantics of a let expression

As usual, we specify the semantics of a let special form by the special way in which it is evaluated. A let expression of the form

\[
\text{let } [(\text{var}_1 \\text{val}_1) \\
(\text{var}_2 \\text{val}_2) \\
... \\
(\text{var}_n \\text{val}_n)]
\]
is evaluated as follows:

- First, the expressions, $val_1, ..., val_n$, are evaluated.
- Second, a local environment is created containing $n$ entries—one for each of the var/val pairs in the let expression.
  - In particular, each symbol var$_i$ is associated with the result of evaluating the corresponding val$_i$ expression.
- Third, the expressions in the body of the let special form are evaluated, in turn, with respect to that newly created local environment. Thus, in the process of evaluating these expressions, if any of the symbols var$_i$ ever needs to be evaluated, its value is drawn from the newly created local environment. For other symbols, the parent environment—which is often the global environment—is used.
- The value of the expression in the body is the value of the entire let expression.

Example: Evaluating let expressions

The following interactions window session demonstrates the evaluation of the sample let expressions seen earlier.

```racket
> (let ()
  true)
true
> (let [(x (+ 2 3)]
  (* x x))
25
> (let [(x (+ 2 3))
  (y 3)
  (z (= 2 2))]
  (begin
    (printf “x: A, y: A, z: A”
      x y z)
    (+ x y z)))
(printf “x: A, y: A, z: A”
      x y z)
12
```

In the first expression, the local environment contains no entries. Thus, when the body of the let is evaluated, the result is the same as if it were evaluated outside the let. In particular, the expression, true, evaluates to true, which is reported as the value of the entire let expression. Since the purpose of a let expression is to set up a local environment, it is rare to see a let expression that contains no var/val pairs.

In the second expression, the local environment contains a single entry that associates the value 5 with the symbol x. The body of the let consists of the single expression, (* x x), which evaluates to 25 in this context. Notice that 25 is reported as the value of the entire let expression.

In the third expression, the local environment contains three entries: one associating the value 5 with x, one associating the value 3 with y, and one associating the value 4 with z. The body contains one begin expression with two subsidiary expressions. The printf expression causes information to be displayed in the Interactions Window; the expression (+ x y z) is then evaluated, resulting in the value 12, which is reported as the value for the entire let expression.

Using let expressions inside functions

The most frequent use of let expressions is to define variables inside functions, as shown in the simple function below:

**Example: Computing monthly interest on a bank balance.** Write a procedure that computes the monthly interest on a bank balance. If the balance is less than a minimum, the interest is 0.0 and if the balance is greater, the interest rate is 0.03.

```racket
(require test-engine/racket-tests)

; (minimum-balance bal) -> number
; --
```
; bal: number
;--------------------------------
; Compute monthly interest on given bank balance, bal.
;
; Pre-function tests:
(check-expect (monthly 90) 0.0)
(check-expect (monthly 1000) 2.5)
(check-expect (monthly 50000) 125)
;
; Constant declarations:
(define MIN-BALANCE 100)
(define INT-RATE 0.03)
(define MONTHS 12.0)
;
; Function definition:
(define monthly
  (lambda (bal)
    (if (> bal MIN-BALANCE)
        (/ (* INT-RATE bal) MONTHS)
        0.0)))

; Post-function tests:
(tester '(monthly 90))
(tester '(monthly 1000))
(tester '(monthly 50000))

(test)

Now, we’ll re-write the function monthly as monthly-v2, defining the constants in a let special form inside the function:

(define monthly-v2
  (lambda (bal)
    ; Give names in the local environment of the let for
    ; the minimum balance, the interest rate, and number of
    ; months.
    (let [(min-balance 100)     
          (int-rate 0.03)      
          (months 12.0)]      
      ; Body of the let expression is the only place the
      ; symbols min-balance, int-rate, and months have values
      (if (> bal min-balance)
          (/ (* int-rate bal) months)
          0.0))))

Using the let expression to declare the literal values needed in a function has the advantage of keeping the constant values close to the code in which they are used. When you create a let special form inside a function, you are creating a local environment (for the let) inside the local environment of the function.

While the let special form has many uses, there is a more general form that allows both local constants and local functions to be defined. This is the local special form, covered in the next section.

B. The local/define special form

A local special form with one or more define forms can be used to define constants and helper functions inside a function. This form is well-suited to write functions and their helper functions like the facty-acc and facty-wr in a single function. Consider the factorial function given below. It uses a local special form to define both a constant (the initial value of the acc) and the definition of a local function called fact-helper to do the recursion:

(require test-engine/racket-tests)
(factorial n) -> integer

; Pre-function tests:
(= (factorial 1) 1)
(= (factorial 4) 24)
(= (factorial 5) 120)

; Function definition:
(define factorial
(lambda (n)
  (local
    ; start of definitions part of local
    [(define ACC 1)
     (define fact-helper
      (lambda (n acc)
        (cond
          ; Base Case: n=1 -- return the accumulator
          [(= n 1) acc]
          ; Recursive Case: n > 1
          [else
           ; Internal recursive function call (tail-recursive)
           (fact-helper (- n 1) (* n acc))]
        ))]
    ; Note: The next line is still inside the local expression,
    ; but outside the definitions section of that expression.
    (fact-helper n ACC)) ; call to local function and end of local expression
))

; Post-function tests:
(tester '(factorial 3))
(tester '(factorial 6))
(test)

A local special form has a definitions section enclosed inside square braces starting right after the keyword local, followed by an application section, in which there can be any number of expressions, but only the value of the last expression is output.

The structure of a local special form is somewhat similar to the cond special form, in that there can be any number of define clauses inside a single local expression, just like there can be any number of clauses inside a single cond.

A local special form groups together arbitrarily long sequences of definitions. The syntax of local is shown below:
(local
  [(define var_1 val_1)]
  (define var_2 val_2)
  ...
  (define var_n val_n)]
  <body>)
where:
- var_1, ..., var_n are character sequences representing n distinct Racket symbols, for n ≥ 0;
- val_1, ..., val_n are n Racket expressions, including λ expressions; and
- <body> is a Racket expression, the return value.

The semantics of a local special form encompasses the semantics of the let, define, and lambda special forms. Each var_i is available to be used in the body. The sequence of the definitions is also important because any var_{i-1} is available to be used in the definition of any var_i.
Since the `local` and `define` special forms do the same job as the `let` special form, we will use the `local` form most often to define local environments. Like the `let` special form, a `local` expression sets up its own local environment inside the local environment of its enclosing function and all definitions have value only inside the parentheses of the `local` expression.

Any helper function can be defined within the parentheses of another function and often it makes more sense to do so because this practice keeps functions and their helper functions close together. However, a good rule of thumb dictates that one function should not be over a screenful of lines (excluding comments). Also, if more than one function has need of the same helper function, it is a better idea to separate the helper from the functions that use it because another rule of good programming is that code should be reused, not rewritten.

C. More recursion: Keeping track of indices to process strings

Recall the compound primitive type string. Every string is a sequence of characters inside quotation marks and each character in a string has a unique index number (starting at 0). Suppose you were asked to write a function that consumes a string and produces a longer string in which each character of the input string is repeated twice. Just to make sure you understand the output of this function, we’ll call the function `double-each` and follow the design recipe.

The first version of this function uses an index with value varying as the string is processed. The function calls itself recursively and adds 1 to the index on each recursive call. In effect, we use recursion to maintain a counter whose value is a position in the string.

```scheme
(require test-engine/racket-tests)

;;; (double-each str) -> string
;;; ---------------------------
;;; str: string
;;; ---------------------------
;;; Produce a string in which every character of str is repeated twice, from left to right, in the output string.

;;; Pre-function tests:
(check-expect (double-each "Hi!") "HHii!!")
(check-expect (double-each "4") "44")
(check-expect (double-each "") ";" ;" is the empty string

;;; Function definition:
(define double-each
  (lambda (str)
    (local
      ; first, save the length of str
      [(define len (string-length str))]
      ; next, define an internal function that uses len
      (define doub-each-helper
        (lambda (index)
          (cond
            ;; If index is at end of str, return the empty string
            [(= index len) "]
            [else
              (string-append (string (string-ref str index))
                            (string (string-ref str index))
                            (doub-each-helper (add1 index)))])))]
      ;; Initial call to doub-each-helper, starting index at 0
      (doub-each-helper 0) ; end of local )
    ))

;;; Post-function tests:
(tester '(double-each "mississippi"))
(tester '(double-each "kitty"))
```
Look up the string functions used inside the double-each function to be sure you understand their purpose. Remember, every function used inside the double-each must be already defined, otherwise the function would produce an error.

An accumulator version of double-each, double-each-acc, is shown below:

```racket
(define double-each-acc
  (lambda (str)
    (local
      [(define len (string-length str))
       (define doub-each-acc-helper
         (lambda (index acc)
           (cond
             ;; If index is less than 0, return the acc
             [(< index 0) acc]
             [else
              (doub-each-acc-helper
                (sub1 index)
                (string-append (string (string-ref str index))
                              (string (string-ref str index))
                              acc))])))
      ;; Initial call to doub-each-helper
      (doub-each-acc-helper (sub1 len) "")))))
```

Post-function tests:
(test)
(tester '(double-each-acc "Hi!"))
(tester '(double-each-acc "Yippee"))
(test)
XVII. Higher-order functions

One of the advantages of a language like Racket is that it can pass named or unnamed functions as arguments to functions and a function can return a function as a result. Functions that consume or return functions are known as “higher-order functions”.

In this section, we present several higher-order functions that can greatly simplify functions that consume lists. They do this by consuming an extra parameter...a function.

In a contract for a higher-order function, the notation for a one-parameter function that consumes an argument of type X and returns a value of type Y is \((X \rightarrow Y)\). When passing a function as an argument, the function name is not preceded by a left parenthesis; when calling a function, the function name is preceded by a left parenthesis.

A. The \texttt{filter} higher-order function

One built-in higher-order function that can make our lives easier is the \texttt{filter} function. \texttt{Filter} applies its first argument, a one-parameter predicate procedure to each element in its second argument, a list and always produces a list. The list returned is less than or equal to the length of the input list and contains only those items in the input list for which the predicate procedure returns true.

The definition of \texttt{filter\%}, below, shows how we could go about writing a procedure equivalent to the built-in \texttt{filter} procedure:

```racket
; Contract: (filter\% (X -> boolean) lox) -> lox
; Header: (define filter\% (lambda (pred list-of-x) ...))
; Purpose: Produces a list containing only the elements in
; list-of-x for which pred is true
(check-expect (filter\% odd? '(1 2 3 4 5)) '(1 3 5))
(check-expect (filter\% odd? '()) '())
(check-expect (filter\% odd? '(1 1 3 3 5)) '(1 1 3 3 5))
(check-expect (filter\% odd? '(2 2 4 4 6)) '())

; Function definition:
(define filter\%
  (lambda (pred list-of-x)
    (cond
     [(empty? list-of-x) empty]
     [(pred (first list-of-x))
      ; pred is called in the line above
      ; pred is passed as an argument to filter\% in the 2 lines below
      (cons (first list-of-x) (filter\% pred (rest list-of-x)))]
     [else (filter\% pred (rest list-of-x))])))
```

The name for filter probably makes sense to you; the metaphor of the air filter that allows air through but doesn’t allow dirt, and so on, evokes something that passes some data and blocks other data.

B. The \texttt{map} higher-order function

A common pattern in processing lists is called mapping, consuming a single parameter procedure and a single list and returning the elements in the list after having applied the procedure to each of them.

The definition of \texttt{map\%}, below, shows how we would go about writing a procedure equivalent to the built-in \texttt{map} procedure:

```racket
; Contract: (map\% (X -> Y) lox) -> loy
; Header: (define map\% (lambda (oper list-of-x) ...))
; Purpose: produce list that results from applying oper to
; every element in list-of-x
(define map\%
  (lambda (oper list-of-x)
    (cond
     [(empty? list-of-x) empty]
     [else (cons (oper (first list-of-x))
       ; oper is called in the line above
      ))})
```
Like the map% procedure shown above, map is a built-in procedure that applies its first argument, a one-parameter procedure, to each of the elements in its second argument, a list. The following lines show the effect of mapping the add1 function to a list of numbers.

> (map add1 '(45 17 22 93))  
(list 46 18 23 94)

This result is the same as (list (add1 45) (add1 17) (add1 22) (add1 93)).

Map is a higher-order procedure that always produces a list that is the same length as the input list. Note the difference between passing a function as an argument and calling a function.

The term “map” comes from the mathematical study of functions, in which they talk about a mapping of the domain into the range.

It is possible for the map function to consume a function and any number of equal-length lists. If there are 2 list arguments, the function argument must consume at least 2 inputs; if there are 3 list arguments, the function argument must consume at least 3 inputs. For example, here is an invocation of map on two input lists and the + function and an invocation with three input lists and the * function.

> (map + '(1 2 3) '(4 5 6))  
(list 5 7 9)
> (map * '(1 1 1 1) '(2 2 2 2) '(3 3 3 3))  
(list 6 6 6 6)

C. The apply and fold higher-order functions

Some functions that consume lists are very similar. For example, consider the following function that consumes a list of numbers and returns the result of summing the numbers in the list.

; Contract: (sum-list lon) -> number  
; Header: (define sum-list (lambda (list-of-num) ...))  
; Purpose: Produces the sum of all numbers in list-of-num.  
; Pre-function tests:  
(check-expect (sum-list '(1 2 3 4 5)) 15)  
(check-expect (sum-list '(1 1 1 1 1)) 5)  
(check-expect (sum-list '()) 0) ; empty list => identity for +  
; ; Function definition:  
(define sum-list  
  (lambda (list-of-num)  
    (cond  
      [(empty? list-of-num) 0]  
      [else  
        (+ (first list-of-num) (sum-list (rest list-of-num))))]))

Compare the sum-list function, given above, to the mult-list function given below. These functions are so similar that the creators of Racket have provided the built-in function apply to be used whenever there is a combining operation for every element in a list.

; Contract: (mult-list lon) -> number  
; Header: (define mult-list (lambda (list-of-num) ...))  
; Purpose: Produces the product of all numbers in list-of-num.  
; Pre-function tests:  
(check-expect (mult-list '(1 2 3 4 5)) 120)  
(check-expect (mult-list '(1 1 1 1 1)) 1)  
(check-expect (mult-list '()) 1) ; empty list => identity for *  
; ; Function definition:  
(define mult-list  
  (lambda (list-of-num)
The **foldl** function provides a way to reduce the elements of a list into a single value by using items from a list as the arguments to a function that takes a variable number of arguments. The added parameters are for the combining operation and for the base case. For example, the following invocations of **foldl** to add all the elements in a list or to multiply all the elements in a list are given below:

```scheme
> (foldl + 0 (list 1 2 3 4))
10
> (foldl * 1 (list 1 2 3))
6
```

The contract, header, and purpose for **foldl** are given below:

```scheme
; Contract: (foldl (X Y -> Y) Y (listof X)) -> Y
; Header: (define foldl (lambda (func base lst) ..))
; Purpose: To apply func to every element of lst and at end of lst return base.
```

A very similar function to **foldl** is the **apply** function. This function has the following contract, header and purpose:

```scheme
; Contract: (apply (X Y -> Y) (listof X)) -> Y
; Header: (define foldl (lambda (func lst) ..))
; Purpose: To apply func to every element of lst.
```

In effect, **apply** works exactly like the **foldl** function except that no parameter needs to be supplied for the base case. In effect,

```scheme
(apply f (list x-1 ... x-n)) = (f x-1 ... x-n).
```

Examples of using this function are given below:

```scheme
> (apply + (list 1 2 3 4))
10
> (apply * (list 1 2 3))
6
```

The **apply** function is also called **reduce** or **foldl** in some dialects of Racket. The name **reduce** is intuitive because the procedural argument is used to produce a single value from the list argument.

**D. The build-list higher-order functions**

The **build-list** function consumes a non-negative natural number \( n \) and a one-parameter function \( f \). It produces a list that is the result of applying \( f \) to every element \( 0\ldots(n-1) \). In effect,

```scheme
(build-list n f) = (list (f 0) ... (f (~ n 1))).
```

Using **build-list** is a fast way to create lists for testing.

Also, remember that the function \( f \) can be an unnamed function instead of a built-in function.
XVIII. INTERACTIVE PROGRAMS: USING THE \texttt{read} FUNCTION

Read is a zero-parameter function that displays a box for entry in the interactions window and returns whatever is typed (up to, but not including the first space) as a quoted symbol, a number, or a string (the string must be typed in quotations).

Read is used to get input from the user of the program and should always be used after a \texttt{printf} that prompts the user for the data to be read. For example, the following function prompts the user for their full name and age and then prints the information to the interactions window.

**General pattern for reading input with local:**

\begin{verbatim}
begin->printf->local->read
\end{verbatim}

; Contract: (get-name-and-age) -> void; side-effect printing
; Header: (define get-name-and-age (lambda () ...))
; Purpose: Prompt for and read a person's name and age and echo back the
; information in a printf statement.
; Pre-function tests: Not possible due to void return type

; Function definition:
(define get-name-and-age
(lambda ()
  need a begin here because body of function is a printf and a local
  statement
  (begin
    (printf "Please enter your full name in quotation marks:" "%")
    (local
      [(define name (read))
        need a begin here because a printf will precede the next local
        statement
        (begin
          (printf " Please enter your age:" "%")
          (local
            [(define age (read))
              print the result of the two read statements
              (printf " Name: a, Age: a" name age))))])))

This pattern of combining a printf expression before a read statement is known as "prompt and read". The printf tells the user what entry is expected and the read statement reads the data. The local and define special forms are used to combine the printing and reading in a single function. Advanced Student Language does not allow a define statement to be used inside a function without being preceded by a local special form. This means that a new local statement must be used to store the result of every read function.

There is a slightly shorter form of the local expression that allows the definition of value holders within a function. This form (covered in a previous section) is called LET and is used in a similar way to local.

**General pattern for reading input with let:**

\begin{verbatim}
begin->printf->let->read
\end{verbatim}

; Contract: (get-name-and-age-let) -> void; side-effect printing
; Header: (define get-name-and-age-let (lambda () ...))
; Purpose: Prompt for and read a person's name and age and echo back the
; information in a printf statement.
; Pre-function tests: Not possible due to void return type

; Function definition:
(define get-name-and-age-let
(lambda ()
  need a begin here because body of function is a printf and a let
  statement
  (begin
    (printf "Please enter your full name in quotation marks:" "%")
    (let [(name (read))
      need a begin here because a printf will precede the next local
      statement
      (begin
        (printf " Please enter your age:" "%")
        (let [(age (read))
          print the result of the two read statements
          (printf " Name: a, Age: a" name age))))))))
Simple interactive programs can be written by using combinations of different programming constructs we’ve used in previous chapters. A simple interactive game is called “Guess the number”, where person A (the computer) thinks of a number between, say 1 and 100, and the other person, person B (the user), guesses the number. If the guess is correct, the game ends. If the guess is too high, person A says “lower”, and person B is allowed to guess again. If the guess is too low, person A says “higher”, and person B is allowed to guess again.

The following is an example of a function that plays this game with a user:

```scheme
; Contract: (guess-the-number) --> void; only side-effect printing
; Header: (define guess-the-number (lambda () ...))
; Purpose: The computer thinks of a number between 1 and 100 and the
; user guesses until they guess the right number.
; Pre-function tests: Not possible due to side-effect printing
; Function definition:
(define guess-the-number
 (lambda ()
   (local
     ; 1. Generate and save the random number
     (define CORRECT (add1 (random 100)))
     ; 2. Give user instructions
     (begin
       (printf "Type a number between 1 and 100 and I’ll give you hints\n")
       (printf "to let you know if the correct answer is higher or lower.\n")
     ; 3. Define a local function to loop until the correct answer is guessed
     (local
       [(define keep-guessing
           (lambda ()
             (begin
               ; 4. Prompt for the user’s guess
               (printf "Make your guess now: \n")
               (local
                 ; 5. Read the guess.
                 (define GUESS (read)))
               ; 6. if guess equals the correct answer, give them feedback.
               (if (= GUESS CORRECT)
                 (printf "YOU GOT IT!! The number is \n")
                 ; 7. if guess is less than correct answer, tell user to
                 ; guess higher
                 (if (> CORRECT GUESS)
                   (begin
                     (printf "Higher\n")
                     (keep-guessing))
                 ; 8. if guess is greater than correct answer, tell user
                 ; to guess lower
                 (begin
                   (printf "Lower\n")
                   (keep-guessing))));]))
     ; 9. Call inner guessing function
     (keep-guessing)))))))
```
XIX. VECTORS: FAST ACCESS DATA CONTAINERS