# Racket Notes—Fall 2016

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

## Contents

<table>
<thead>
<tr>
<th>I</th>
<th>Introduction</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>II</td>
<td>Primitive Data Types</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>II-A Numeric Type</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>II-B (Quoted) Symbol Type</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>II-C Image Type</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>II-D Boolean Type</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>II-E String Type</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>II-F Character Type</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>II-G Void Type and Side Effects</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>II-H Built-in Function Names</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>II-I Empty List Type</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>II-J Non-empty List Type</td>
<td>10</td>
</tr>
<tr>
<td>III</td>
<td>Writing Programs in DrRacket</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>III-A Expression Evaluation</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>III-B The Global Environment</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>III-C Evaluation of Primitive Data</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>III-D Evaluation of Primitive Functions</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>III-E Evaluation of Mathematical Expressions</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Prefix versus Infix Notation</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Process to convert infix to prefix</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Looking up primitive functions</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>III-F Evaluation of Non-Empty Lists (the Default Rule)</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>III-G Comments/Documentation</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>III-H Indentation, Code Coloring, and Line Length</td>
<td>20</td>
</tr>
<tr>
<td>IV</td>
<td>Defining Variables and Functions</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>IV-A Creating named identifiers in the global environment using define</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>IV-B Using the define special form to create named functions</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Function naming conventions</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Distinguishing Arguments from Parameters</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>IV-C Creating Functions Using the Lambda Special Form</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Semantics of (\lambda) Expression</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Unnamed (\lambda) Expressions</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Summary of evaluating (\lambda) expressions</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Local versus Global Environment</td>
<td>26</td>
</tr>
<tr>
<td>V</td>
<td>Displaying Strings within Programs</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>V-A The printf, display, and newline functions</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>V-B Using printf with escape characters</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>V-C Multiple expressions in Racket function body: The begin special form</td>
<td>29</td>
</tr>
<tr>
<td>VI</td>
<td>Asking Questions with Predicate Functions</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>VI-A Type-checker predicates</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>VI-B Arithmetic Predicates: Relational Operators</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>VI-C The Boolean (Logical) Operators: and, or, not</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>The not function</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>The and special form</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>The or special form</td>
<td>32</td>
</tr>
</tbody>
</table>
VII Making Decisions—Special Forms: cond & if
VII-A The if Special Form: The basic yes/no decision-maker
Using an if expression in a function
VII-B Making Multiple Choice Decisions: The cond Special Form
Function with cascading if statement
Function using cond instead of if
VII-C Using the Stepper to Trace Execution of Functions

VIII The Design Recipe: A disciplined approach to writing functions
VIII-A Pre-function Comments
VIII-B Pre-Function Testing: The check-expect and check-within Special Forms
Writing pre-function tests
VIII-C Function definition and parameter naming
Naming parameters
Writing comments inside functions
VIII-D Post-function printing using the printf function
VIII-E Literal values in function definitions
Defining Constant Values before Functions

IX Recursion: Repetition over Data Sets of Unknown Size
IX-A Recursive Functions
Function with Infinite Loop
General rules for recursive functions
Steps in defining recursive functions
IX-B Recursion over numbers
The factorial function
Recursion and local environments
Summary of recursive function features
IX-C Accumulator-style (Tail-Recursive) Functions over numbers
Tail recursion
IX-D Wrapper functions
Information Hiding
IX-E Local Variables and Local Functions—the local special form
The syntax of local
The semantics of a local expression

X Flat Lists: A Self-Referential, Compound Data Type
X-A I thought we were already using lists...
X-B Using cons to Create Pairs (i.e., Cons Cells)
X-C Picturing Non-Empty Lists
X-D The Self-Referential Structure of Lists & List Recursion
X-E Accumulator versions of functions that consume lists
X-F Functions that return lists
X-G More recursion: Keeping track of indices to process strings
X-H Yet more recursion: Using recursion to draw a scene

XI Interactive programs: Using the read function
XI-A Begin, prompt, define, and read pattern
XI-B Adding chance to functions: the random Function
XI-C Using mutation of global variables in place of an accumulator

XII Higher-order functions
XII-A The filter higher-order function
XII-B The map higher-order function
XII-C The apply higher-order function
XII-D The build-list higher-order function
XII-E The andmap and ormap higher-order functions
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 undoubtedly have lots of experience with error messages during this course, but keep in mind that errors help you learn.

**Note:** Racket programs are made up of keywords, variables, structured forms, 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 in Racket.
II. Primitive 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”. The programming software package we will use for creating programs is called DrRacket.

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

A primitive datum is a valid Racket expression that is 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, lists, characters, strings, among many others. Importantly, each datum has a unique data type. For example, a Racket datum might be a number or a boolean, but it 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. The primitive data types are introduced in parts A–J, in no particular order, below.

In the following sections describing each primitive type, we give the characters that you would type to create primitive data literals, actual values of each data type. After each new primitive data type is introduced, some primitive function names that consume each type are given, along with a sample listing from the on-line help desk. One of the first things you need to know to be successful in this class is how to find and use built-in functions. You are encouraged to look these function names up in the on-line help system in DrRacket.

A. Numeric Type

Characters that represent 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. 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–each evaluates only to itself.

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. Leading 0’s on numbers are dropped.

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 +, −, *, and / operators are arithmetic functions that consume numbers and return numbers and the =, >, ≥, >, <, ≤, and < operators are logical operators that consume numbers and return either true or false (booleans).

A version of the Help Desk entry for the + function is shown below. In the first line of this Help Desk entry, the ellipses (...) tell us that the + function may take any number of inputs (0 or more).

\[
(+ \ z \ \ldots) \rightarrow \text{number}
\]

\[
z: \text{number}
\]

Returns the sum of the numbers, adding pairwise from left to right. If no arguments are provided, the result is 0.

\[\text{1} \text{All numbers in this handout are presented in base 10 notation.}\]
Examples typed in the DrRacket interactions window are shown below, where the characters typed to the right of the input prompt \( > \) are entries by the user and the following line is the result of computer evaluation of the input:

\[
\begin{align*}
> & (+ 1 2) \\
& 3 \\
> & (+ 4 5 6.7) \\
& 15.7 \\
> & (+) \\
& 0
\end{align*}
\]

B. (Quoted) Symbol Type

Characters that represent quoted symbols: According to the syntax rules of Racket, the character sequences, ‘a and ‘b, are legal Racket expressions called quoted symbols. A quoted symbol is a sequence of one or more keyboard characters (blank spaces are not allowed), preceded by an apostrophe (‘), known in programming circles as a single quote.

The purpose of quoted symbols 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 an identifier symbol that stands for a variable name, a constant name, or that is undefined. Non-quoted identifiers are evaluated as place-holders for data. Without the leading apostrophe, a symbol evaluates to something different, and non-quoted symbols are therefore not a primitive type. The mechanism for attaching non-quoted symbols to data values or to functions is covered in Section IV.

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.

A version of the Help Desk entry for the symbol=? function is shown below:

Note that a and b as used below are parameter names:

\[
(symbol=? a b) \rightarrow \text{boolean}
\]

\[
a : \text{quoted symbol}
\]

\[
b : \text{quoted symbol}
\]

Returns true if a and b are the same quoted symbol.

Examples:

Note that a and b as used below are literal arguments substituted for parameters:

\[
> (symbol=? 'a 'b)
\]

false

\[
> (symbol=? 'cat 'cat)
\]

ture

C. Image Type

Representation of images: DrRacket provides us the ability to insert various types of images (e.g., .jpg, .gif files) 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. An image is not represented by characters, it is a picture included in your program.

Images can only evaluate to themselves, so evaluating an image returns the same image. We can also manipulate and name images in Racket programs. Images have parts that you can access, such as width and height.

Functions that consume images: Examples of functions that consume images include image-height, image-width, empty-scene, beside, scale, above, and place-image.

A version of the Help Desk entry for the image-width function is shown below:

\[
(image-width i) \rightarrow \text{exact non-negative integer}
\]
i : image
Returns the width of i.

Examples:
> (image-width (ellipse 30 40 "solid" "orange"))
30
> (image-width (circle 30 "solid" "orange"))
60
> (image-width (beside (circle 20 "solid" "orange")
(circle 20 "solid" "purple")))
80
> (image-width (rectangle 0 10 "solid" "purple"))
0

Look up the Help Desk entries for ellipse, circle, beside, and rectangle and see if you can figure out how to call these functions with various arguments in the DrRacket Interactions Window.

D. Boolean Type

Characters that represent booleans: According to the syntax rules of Racket, the character sequences, true (or #t) and false (or #f), are legal Racket expressions.

The keywords true and false, and their equivalent abbreviations #t and #f, respectively, are literal boolean 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.

A version of the Help Desk entry for the not function is shown below:

\[
\text{(not v)} \rightarrow \text{boolean}
\]
\[v : \text{boolean}
\]
Returns #t if v is #f, #f otherwise.

Examples:
> (not #f)
#t
> (not #t)
#f

E. String Type

Characters that represent strings: Syntactically, strings in Racket are sequences of keyboard characters, sometimes including spaces, bounded by quotation marks (often called double-quotes). For example, “hi there” and “Howdy!” are character sequences that denote string data.

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:

<table>
<thead>
<tr>
<th>Index:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td>w</td>
<td>o</td>
<td>r</td>
<td>l</td>
<td>d</td>
<td>!</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Conceptual storage of string “Hello world!”. Note that the position with index 5 contains a space character.

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.
Functions that consume strings: 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.

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. The evaluation function is executed in the Interactions window whenever the user presses return.

> "hi"
"hi"
> "Howdy!"
"Howdy!"

There are many useful primitive functions that consume strings. 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, in order from left to right 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.

A version of the Help Desk entry for the **substring** function is shown below:

```
(substring str start [end]) -> string?
str : string
start : exact-nonnegative-integer
end : exact-nonnegative-integer
```

Returns a new string that is end minus start characters long, and that contains the same characters as str from start, inclusive, to end, exclusive. The [] around end mean that this number is optional, in which case the function returns the substring from start to the last character of str.

Examples:

> (substring "Apple" 1 3)
"pp"
> (substring "Apple" 1)
"pple"

The **substring** function takes at least 2 and at most 3 arguments. The number of arguments is specified in the function definition.

F. Character Type

Characters that represent single 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 write functions that process character sequences (also known as 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?**, **char-upcase**, and many more.

A version of the Help Desk entry for the **char-upcase** function is shown below:

```
(char-upcase ch) -> char?
ch : char
```
Produces a character that is the uppercase version of ch. If ch has no uppercase version, char-upcase produces ch.

Examples:
> (char-upcase #\a)
#\A
> (char-upcase #\%)
#\%
> (char-upcase #\B)
#\B

The notation (i.e., the hash-backslash) for characters is rather strange. It is important to remember that #\a is not the same as "a". The first expression is the character a and the second expression is the one-character string a. When characters are combined to form strings (covered in the last subsection), the hash-backslash before each character is omitted. You will see that there are functions to convert strings into sequences of characters, and vice versa.

G. Void Type and Side Effects
A “no value” value. Created by a function call that returns nothing and only has a side-effect.

Side-effects include:
1) printing text to the screen,
2) drawing images to the screen, and
3) 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. Special forms to be covered in subsequent chapter of these notes, such as define and set! are also statements that return void.

It is important to remember that the void data type is not the same as the empty list. A void data type is generated by a function that has only side-effects.

Function that consumes void: There is a type-checker, void? that can be used to recognize the void data type. Look it up in the help desk.

H. Built-in Function Names
Characters that represent functions: All programming languages provide functions to perform basic operations (e.g., addition, subtraction, multiplication and many more). The existence of these functions allows you to use them in your own code to produce new, previously unavailable functions. Just try to think what it would be like to write the code to do addition of two numbers...you would need to know very intricate details of how the computer represents and stores numbers. Further complication arises from the fact that these details vary from one machine to another. Fortunately for us, most simple operations are already available for use when we start learning to program and we just need to know how to look them up and use them.

Racket includes a variety of built-in functions that are identified by unique character sequences and never preceded by an apostrophe. It is important to realize that a primitive function is a Racket datum, just like numbers and booleans. The difference is that functions may consume data of any valid type and may produce output of any valid type. The majority of the primitive functions are written to consume a particular data type, and most do not work on every data type.

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 are listed in section III-D. The functions you write will frequently use one or more primitive functions to produce more complex functions tailored to specific tasks.
One skill that you will need to develop quickly is interpreting the explanations of functions in the DrRacket Help Desk (under the Help menu). For example, if you look up the \texttt{sqrt} (for square root) function in the Help Desk, you will see something like the following:

\begin{verbatim}
(sqrt z) -> number
z : number
Returns the principal square root of z.
\end{verbatim}

Examples:
\begin{verbatim}
> (sqrt 4/9)
2/3
> (sqrt 2)
1.414213562370951
> (sqrt -1)
0+1i
\end{verbatim}

The first line of the \texttt{sqrt} entry tells us how the \texttt{sqrt} function is called, that it consumes one argument (\texttt{z}), and what type of value is returned. The second line says that the argument \texttt{z} must be a number. The third line explains the purpose of the function. Under Examples we are shown what occurs in the Interactions Window if the \texttt{sqrt} function is called with different arguments.

\textit{Functions that consume functions}: There are primitive functions that consume other functions. These are called “higher order” functions and we will cover them in Section XII.

I. \textbf{Empty List Type}

\textit{Characters that represent empty lists}: According to the syntax rules of Racket, the character sequence consisting of an apostrophe, followed immediately by a left and then a right parentheses, ’(), is a legal Racket expression.

According to the semantics of Racket, ’() denotes the \textit{null datum}, which is also called the \textit{empty list}. This datum is also represented by keywords \texttt{empty} and \texttt{null}.

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 non-empty list.

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

A version of the Help Desk entry for the \texttt{empty?} function is shown below:

\begin{verbatim}
(empty? v) -> boolean
v : list
The same as (null? v). Returns #t if v is the empty list.
\end{verbatim}

Examples:
\begin{verbatim}
> (empty? '(1 2))
#f
> (empty? '())
#t
\end{verbatim}

J. \textbf{Non-empty List Type}

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 \textit{array}. In Racket, the basic aggregate data structure is the \textit{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) A left parentheses that is NOT preceded by a single quote represents a function call if the first item after the open left parenthesis is a defined function name. Example character sequences that represent functions: `+, -, *, /, sqrt, expt, string-append, string-length, image-height`, and many more.

2) A left 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 (also know as a keyword). Example special form keywords include: `define, lambda, cond, else, if, local`, and more.

3) A left parentheses preceded by a single quote is a data container called a list. 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`. Primitive functions that produce lists include `cons` and `list`.

A left parentheses not preceded by a single quote but followed by the function `list` is the same as a left parenthesis preceded by a single quote. For example, `'(1 2 3)` is equivalent to `(list 1 2 3).

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.

A version of the Help Desk entry for the `first` function is shown below:

```
(first lst) -> any valid type
lst : list

Produces the first element in a non-empty list.
```

Example:

```
> (first '(1 2 3 4 5 6 7 8 9 10))
1
> (first (list "Hello" "world"))
"Hello"
```

We will look at each of the three types of non-empty lists 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!
In this class, you will use the software package called DrRacket. DrRacket is free and can be downloaded (see the link on the course web page; sect. 6.x is the most recent version) and installed on any computer. DrRacket is an example of an “Integrated Developer’s Environment” (IDE), so called because it provides facilities for typing programs, saving the typed programs, running the saved programs, and testing the results of running the programs, all in the same window. You will type and run your programs in DrRacket and it will be available for use on any of the machines in the CS department. The prologue of *How to Design Programs, 2nd edition* contains examples of interactions with DrRacket.

![Fig. 2. 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.](image)

Fig. 2 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). To toggle the view between only the Definitions Window and the both windows, type $\text{CTRL-E}$ or $\text{Command-E}$. To toggle the view between only the Interactions Window and the both windows, type $\text{CTRL-D}$ or $\text{Command-D}$.

The DrRacket Interactions Window simulates the operation of a computer that understands expressions typed line by line using the Racket programming language. This enables you to interact with the simulated computer by pressing return. In effect, you use the DrRacket Interactions Window as a communication medium between yourself and the simulated computer.

The DrRacket Definitions Window is used to store lines of expressions that may together form a program or that may just be definitions and invocations of unrelated functions. You must use the Definitions Window to type and save your program. The Interactions Window is only used for trying out expressions and function calls.

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

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 most important thing that DrRacket does—it converts Racket expressions to their simplest form. 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.

You’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, you’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 (often called eval in these notes) is a function that takes one Racket datum as its input, and generates another Racket datum as its output, as depicted in Fig. 3.

Fig. 3. A conceptual depiction of DrRacket evaluation. Picture courtesy of Prof. Hunsberger.

The result of evaluation depends on the type of data that is being evaluated.

Fig. 4. Input of the number two and the boolean true to the eval function. Picture courtesy of Prof. Hunsberger.

The eval function is not directly available for us to use in the teaching languages (like How to Design Programs Advanced Student), but you can consider it as a function applied by DrRacket during program execution (or whenever you type something and press return in the Interactions Window.)

B. The Global Environment

In Racket, non-quoted identifiers 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.

The evaluation of identifier symbols is different from the evaluation of primitive Racket types. In particular, non-quoted symbols do not evaluate to themselves; instead, they evaluate to the value that is associated with them in the
**global environment.** The **global environment** is a table that contains many entries. Each entry pairs a symbol with its corresponding value (a valid Racket type). To evaluate an identifier, the evaluation function looks up the value associated with that identifier in the global environment.

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

![Global Environment:](image)

Fig. 5. Conceptual depiction of the table lookup provided in DrRacket (the global environment), with a column for identifier names and a column for values associated with the names. Note that table shown is empty. This is slightly misleading because the global environment always contains a set of primitive identifiers and their values.

In Racket, unquoted symbols (character sequences that start with a letter and contain no spaces, generally referred to as **identifiers**) 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 stand for the value 3. 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 are looked up in the global environment.

The Racket datum associated with an identifier in the global environment can be of any valid type. Thus, it might be that the boolean **true** is associated with the identifier **pq**. Similarly, the empty list might be associated with the identifier **my-empty-list**.

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

If an identifier does not have a corresponding entry in the global environment, it is not possible to evaluate that identifier. Attempting to evaluate an identifier whose name is not written in the global (or local) environment will cause an error. When we cover the **define** special form in Section IV-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. And that’s called programming (so much fun)!

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 named variables.

**C. Evaluation of Primitive Data**

You can use DrRacket to enter character sequences (i.e., expressions) into the Interactions Window and then examine the results reported after evaluation. If you type in an expression that is already in simplest form, you will see the character sequences you type next to the prompt (> ) and those reported back by DrRacket on the following line. If the character sequence you type is undefined, you will see a red error message on the following line.

In the Interactions Window, DrRacket uses the > character to prompt for input. Everything following the > character in the examples shown was typed by a programmer. The text on the lines that do not start with > is that generated by DrRacket in response to evaluation of the character input sequence. Thus, the excerpt from the Interactions Window shown below displays seven separate interactions and demonstrates that numbers, booleans, quoted symbols, strings, the empty list, and the function + all evaluate to themselves, but undefined identifiers (like x) cause an error to occur:
The first five examples shown above are primitive types in simplest form. The fifth is looked up in the global environment and echoed back, showing the addition operator is defined. The last example, \( x \), is undefined.

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. 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, nor will it be evaluated. If there is a value associated with the unquoted 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.

D. Evaluation of Primitive Functions

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

For each identifier associated with a built-in function, there is an entry in the global environment that pairs that unique identifier with the function. Therefore, the evaluation of only that particular symbol can be used to gain access to the corresponding function. As mentioned previously, 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).

\[
\begin{align*}
&> + \\
&> - \\
&> *
\end{align*}
\]

This excerpt shows that, in DrRacket, functions evaluate to themselves. Once you learn how to create Racket functions of your own design in Section IV-A, you’ll be able to give your new functions names by placing appropriate entries into the global environment.

E. Evaluation of Mathematical Expressions

You can use the Interactions Window like a 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. The following examples give 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, \((x \cdot x)\), 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, \((x \cdot x)\), yielding \((3 \cdot 3)\). Evaluating the expression, \((3 \cdot 3)\), 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, \((x \cdot x)\), yielding \((4 \cdot 4)\), 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 \( h \):

\[
g(w, h) = w \cdot h.
\]

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, \((w \cdot h)\), yielding \((3 \cdot 7)\). Evaluating this expression results in the desired output value, 21.

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

**Prefix vs. Infix Notation**

Typing a function name after an open parenthesis and substituting arguments for parameters is known as *calling* or *invoking* the function on the input arguments. In Racket, all function calls are written using prefix notation, where an open parenthesis precedes every function identifier and the function identifier precedes the argument(s). This notation is discussed below.

When a function is called:
- the name of the function is preceded by a left parenthesis,
- the name of the function is followed by one or more spaces, then the input argument(s) to the function, each separated by one or more spaces,
- each argument is evaluated and substituted for the corresponding parameter in the function definition, and
- evaluation ends with a right parenthesis.

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 expressions and the equivalent prefix expressions:

<table>
<thead>
<tr>
<th>Infix Expression</th>
<th>Equivalent Prefix Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2 + 4 \cdot 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 (functions) and operands (arguments). The spacing is exaggerated in the table above to emphasize the importance of some blank spaces in these expressions.

An advantage of using prefix notation is that in this type of expression functions can take more than two arguments. 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).

Another important feature of expressions in prefix notation is that the expressions are evaluated from the inside out. That is, the leftmost expression that contains no sub-expressions, is evaluated first. Consider the prefix expression
(+ 2 (* 4 5))
In this expression, the (* 4 5) is evaluated and then the result of that evaluation is added to 2.

There are many primitive Racket functions that can consume any number of input arguments and you will learn to find such functions in the DrRacket Help Desk.

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 * x.

Step 2 adds parentheses around the entire fraction, around 7 * x, and around the whole expression, to get (((7 * x) − ((3 + x)/(y + 2))))). Note that there are now five operators (*, −, +, /, 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 (− (* 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 links to the predefined functions listed down the left side of the page, grouped according to the type of data they consume.

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, its name 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. If the function is not defined, an error message will be displayed.

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 all functions introduced in this write-up and in class.

F. 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 special form keyword will be covered as it is needed.

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

```
> (+ 2 3)
5
> (* 3 4 5)
60
> (+ 2 (* 3 10))
32
> (+ 2 (* 3 (+ 4 8 6)))
56
```

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. In each case, the list is being evaluated according to the default rule.

The resulting output datum that appears after typing an expression and pressing return 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 the number five, which DrRacket reports in the Interactions Window using the character sequence 5.

Here’s a summary of this example:

```
(+ 2 3) → [ list containing + symbol, number two, and number three → number five ] → 5
```

The evaluation steps (→) are explained by:

1. First Step of Default Rule—look up + symbol in global environment and evaluate arguments 2 and 3:
   + symbol → addition function
   number two → 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 numeric arguments. The second step in the default rule involves applying that function to the arguments (i.e., feeding the remaining items as input into that function), as illustrated below:

![Diagram](image)

Fig. 6. Pictorial evaluation of addition function on inputs 2 and 3. Picture courtesy of Prof. Hunsberger.

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

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, you 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 you can complete the first step of the default rule, you must evaluate the subsidiary list (i.e., the list containing the $*$ symbol, the number three and the number ten).

To evaluate the subsidiary list, you need to use the default rule. The first step of the default rule requires you 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 you to apply the first item (i.e., the multiplication function) to the numbers three and ten. The result is the number thirty.

Now that you know the subsidiary list evaluates to thirty, you can pick up from where you left off when evaluating the original list. The first step of the default rule (for evaluating the original list) requires you 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 you 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.
It is possible for some things to go wrong in the process of evaluating a non-empty list. For example, the function might expect a different number of input arguments than are present in the rest of the original list. Or the attempt to evaluate a datum might be undefined. Or the application of the function to the arguments 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
- \((\empty\ \empty)\), a list containing two instances of the empty list
- \((+ \ \true\ \false)\), 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).

G. Comments/Documentation

Although we have portrayed the execution of expressions in DrRacket’s Interaction Window, the real place to write code that is saved between times you exit and re-enter DrRacket is in the Definitions Window (at the top or left side of the DrRacket 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.

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 start 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 see 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 using 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. Comment boxes have the disadvantage of making your files much larger, so you may want to stick to semi-colon comments.

H. 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.

After you have written a function, you can indent it correctly either by choosing “Reindent all” under the Racket menu, or by highlighting the whole function and pressing tab.

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.
IV. Defining Variables and Functions

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:

⇒ 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:

- `(define x 3)`
- `(if condition then-clause else-clause)`
- `(cond [condition1 return1] [condition2 return2] [else return3])`

A special form is evaluated according to a rule that is specific to the keyword used for that special form, not by using the default rule. There is one rule for executing **define** special forms, another rule for executing **if** special forms, and so on. Luckily for us, each **define** special form is executed in the same way, each **cond** is executed in the same way, and so on. However, the rule for executing the **define** special forms is very different from the rule for executing the **cond** special form. **In other words, each special form has its own rules for execution.**

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 executing 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, nor is it looked up, because no special form is written in the global environment.

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

A. Creating named identifiers in the global environment using **define**

The purpose of the **define** special form is to create an entry in the global environment, a side effect. The return type is always void.

As a character sequence, it has the form `(define C1 C2)`, where `C1` is an identifier, and `C2` can be any expression denoting a Racket datum, call it `e`.

```
Evaluating the special form:  (define C1 C2)
C1                                 C2
                                     e
Global Environment Entry: The symbol C1 E
```

Fig. 7. Evaluating the special form `(define C1 C2)`.

Only `e`, the Racket version of the symbol `C2`, is evaluated (as shown in Fig. 7), the result being some datum `E` that is written into the global environment. The symbol `C1` is not evaluated, but an entry is created in the global environment in which the name `C1` is associated with the datum `E`.

Because **define** statements generate void output, DrRacket does not display any output on the computer screen when a **define** special form is evaluated in the Interactions Window, as shown below:

```
> (define X 6)
>```

```
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, paired with $X$ in the global environment, 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 executing 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 global variables in DrRacket. Similar mechanisms exist to name local variables inside functions. Identifier names allow you to avoid using literal values in code. The usual practice is to start a program with all the variable definitions, such that all identifiers used in the program are written in the same block of code. Any global variable used in the code must be defined before it is used. You will see that it is more usual to define variables inside the function in which they are 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.

![Fig. 8. Executing the special form `(define Y (+ 1 2 3))`.](image)

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 value of the mathematical expression in the define statement for DIAMETER is written in the global environment after the evaluation of the expression \((50 \times 50 \times 3.14159)\).

```
Global Environment Entries:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WIDTH</td>
<td>500</td>
</tr>
<tr>
<td>HEIGHT</td>
<td>400</td>
</tr>
<tr>
<td>DIAMETER</td>
<td>7853.975</td>
</tr>
</tbody>
</table>
```

Fig. 9. 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 in the Interactions Window are rewritten. To make lasting changes to the global environment, you should type the define statements in the Definitions Window and press Run.

### B. Using the define special form to create named functions

You can also use the define special form to give a function a name. 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. Naming functions is a way of reusing code because one name can result in the application of a name that is usable any number of times in a program.

The following excerpt of an Interactions Window session demonstrates how to name a procedure that produces the cube of a number and then call the function several times:

```
> (define (cube x) (* x x x))
> (cube 5)
125
> (cube 7)
343
> (cube -8)
-512
```

Normally, functions are defined in the Definitions Window so they can be run repeatedly and written to files.

When define is used to add a new function name to the global environment, the identifier is followed by an open parenthesis, the function name, the parameter name, a close parenthesis, and an expression called the body of the function. The body of a function is evaluated when the function is called.

When naming a variable identifier, there is no left parenthesis between the keyword define and the identifier name.

**Additional Example:**

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

In this example, the function rect-area has two parameters, \(w\) and \(h\). The expression \((\ast w h)\) is the body of the function.

Function naming conventions:
Racket adheres to certain conventions for function names.

- Functions that return a boolean (except the relational operators for numbers) end with a question mark (traditionally 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) and is given when a function is called. The statement `(rect-area 2 3)` is known as a function call. A parameter is a “place-holder” introduced in the function definition. A parameter can be thought of as a variable identifier that exists only inside the parenthetical boundaries of a given function. An argument may evaluate to a number, string, image, constant, or any other valid type, but a parameter is always an identifier, 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 expression `(* w h)`.

C. Creating Functions Using the Lambda Special Form

In the last section, you were shown the “short-cut” way to define functions. It is important to know what is really happening when you define functions. Consider the `cube` function, defined as shown in section IV:

```
(define (cube x)
    (* x x x))
```

What the computer sees when this define statement above is executed is:

```
(define cube
    (lambda (x)
        (* x x x)))
```

The Racket family of programming languages provides the `lambda` special form to give us a mechanism to define functions. 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 Greek \( \lambda \) character into a DrRacket program, pull down the Insert menu and choose “Insert \( \lambda \)”. The terms `lambda` and \( \lambda \) are used synonymously.

The `define` special form is used to create an entry in the global environment that associates the identifier `cube` with the lambda expression `(lambda (x) (* x x x))`. The only difference between using define to create a variable identifier and using define to create a function is that, when naming a function, the `define` keyword is used to name a lambda expression. When a `define` special form is evaluated, the given symbol—in this case, `cube`—is not evaluated; however, the identifier `cube` and `(lambda (x) (* x x x))`—are written in the global environment under the name and value columns, respectively. Thus, the value associated with the `cube` identifier is the procedure that results from calling the function with a given argument in place of `x`.

Like any special form in Racket, the `lambda` special form is written in a list (a set of parentheses) whose first element is the keyword `lambda`—in this case, either `lambda` (or \( \lambda \)). The second element used in a `lambda` special form is the parenthesized parameter list containing the names of the input parameter(s) for the function being defined. The body of the function is an expression nested in the lambda expression after the parameter names, in a separate set of parentheses. The parameters and the body are always enclosed in a pair of (possibly nested) parentheses.

Here is an example of a `lambda` expression for a function that calculates the hypotenuse of a right triangle, given side lengths `a` and `b`:
(define hypotenuse
  (lambda (a b)
    (sqrt (+ (* a a) (* b b)))))

The definition above is a lambda form with three parts:
1) The word lambda (or the symbol λ, inserted by pulling down the “Insert” menu and choosing “Insert λ”).
2) The names of the parameters (a and b above), in parentheses.
3) The body is (sqrt (+ (* a a) (* b b))), enclosed in its own set of parentheses.

The following are examples of well-formed (i.e., valid), nameless λ expressions (in this class we will generally define a name for λ expressions). Try typing them in the Interactions Window and they should be identified as procedures.
1) (λ () 44)
2) (λ (x) (* x x))
3) (λ (w h) (* w h))
4) (λ (r h) (* 1/3 3.14159 r h))
5) (λ (x y z) (+ x (− y z)))

X-parameter functions:

We often refer to functions based on the number of input parameters listed in the λ expression. For example, expression 1) above 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. If you typed any of these lines in the Interactions window, the result would be a function because functions evaluate to themselves.

In contrast, the following are examples of malformed λ expressions:
1) (λ (x y x) (* x y))
2) (λ (x 10) (* x 10))
3) (λ x)

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

The Semantics of a λ Expression:

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

Assuming that
- each Ci denotes a Racket symbol, ri;
- the symbols, r1, r2, ..., rn, are distinct (i.e., there is no repetition); and
- B denotes some Racket datum D,
then a λ expression of the form (λ (C1 C2 ...Cn) B) denotes a Racket list whose elements are as follows:
- the λ symbol;
- a list containing n distinct identifiers, r1, r2, ..., rn; and
- the Racket datum, D
This list is referred to as a λ special form.

Calling unnamed λ Expressions:

When you use the default rule to apply a unnamed lambda form to actual input values, it is an invocation or call of the function, as shown below:

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

The type of a **lambda** form is a **function**. The **lambda** form is the way the programmer expresses that some operations be performed on the arguments and return a value. *Racket does not evaluate the elements of the body of the function until the function is invoked (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:

```
> ((\(x\) \(\times\) \(\times\)) 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 \((\times \ \times)\). 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 \((\times \ 4 \ 4)\), which in turn evaluates to the correct output, 16. Later on, we will encounter situations where it is convenient to use functions without naming them, so it is important to understand how to use these **unnamed** \(\lambda\) expressions.

**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 expression that is evaluated for the return value). A function call progresses as follows:

1) Replace the parameters as they occur in the parameter 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) 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. When unnamed lambda expressions are preceded by a left parenthesis, the evaluation follows the Default Rule. In the following exercise, the value given after the right parenthesis that closes the lambda expression is the input argument (or arguments).

**Exercise IV-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) \(((\text{lambda} \ (x) \ (+ \ x \ 2)) \ 5)\)
(b) \(((\lambda \ (x \ y) \ (+ \ x \ (* \ y \ 3))) \ 7 \ 4)\)
(c) \(((\lambda \ (x)) \ ((\lambda \ (y) \ (* \ y \ 2)) \ (+ \ x \ 3))) \ 4)\)

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

(a) \((x - 32) \times (5/9)\)
(b) \((x + 3 \times y) \times (x - y)\)

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, you have to apply it to any input(s).

**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 **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, paired with the values of 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. 10, the following things happen:
1) A local environment is set up containing a single entry which associates the value 4 with the symbol x.
2) The expression (** x) is evaluated with respect to the newly created local environment. 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 shadows the symbol x in the global environment. The evaluation of (** x) therefore yields the result 16.
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.

\[
\text{> (define discriminant}
\text{  (lambda (a b c)}
\text{    (- (* b b) (* 4 a c)))}
\text{> (discriminant 1 2 -4)}
\text{20}
\text{> (discriminant 1 0 -3)}
\text{12}
\]

Notice that the syntax of Racket allows expressions to occupy multiple lines. This is essential when writing longer expressions. DrRacket automatically indents sub-expressions following a return to make longer expressions easier to read. Pressing the tab key while the cursor is at any position on a particular line will automatically cause the current line to snap to the appropriate amount of indentation unless there is some error in the syntax or semantics of the expression before that line. In general, it is a good idea to press enter before the opening parenthesis of each sub-expression.
V. Displaying Strings within Programs

A. 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 is a 1-parameter function that 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 is an n-parameter function that has the side effect of displaying its string argument to the Interactions Window, but it also includes the inclusion of escape characters in the string at specified positions. The printf function interprets the character sequences, \%, \n, and \A (or \a), in special ways when they are embedded in its first argument—a string. \% \n and \A are newline characters and \A or \a are placeholders for values after the value has been converted to a string.

B. Using printf with escape characters

The character sequences \n, \%, \A and \a are commonly called escape characters because they cause something other than the exact character sequence they are composed of 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 printing. 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.

> (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\n") 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\n").

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 is a 0-parameter function that 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 ^% (or \n), 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 order, 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.

C. Multiple expressions in Racket function body: The begin special form
In the HtDP languages, the body of a function may contain multiple expressions only if the keyword begin surrounds all the expressions. When a function containing a begin is called, each of the expressions in the body is evaluated sequentially, from top to bottom. The value of the last expression is the only output value for the function. All expressions except the last should therefore generate only side effects.

The following 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.

```racket
> (define (verbose-func a b)
  (begin
    (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 labeling problems, printing results of function evaluation, learning to debug code, and writing “interactive programs” that require input from the user.

Here are a few more examples of using the printf function in the Interactions Window:

```racket
> (printf "Line One!\n Line Two!!\n Line Three!!!\n")
Line One!
Line Two!!
Line Three!!!

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

> (printf "A symbol: ^A, a string: ^A, a boolean: ^a\n" '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. These escape sequences are used as placeholders in the string printed as a side effect. The values that are included in the string should match the number of ^a or ^A escape sequences in the preceding string.
VI. Asking Questions with Predicate Functions

A function whose output is a boolean (i.e., true or false) is called a predicate. 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. For each primitive or compound data type, Racket includes a primitive function called a type-checker predicate. When a type-checker predicate is applied to some Racket datum, the output is true if that datum belongs to the indicated data type; otherwise, the output is false. Thus, the type-checker predicate associated with the number data type number? outputs true whenever the input argument is a number. 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.

For convenience, each type-checker predicate has an easy-to-remember name that is the data type followed by a question mark. In other words, for each type-checker predicate there is an entry in the global environment that links a particular data type with that predicate. Thus, those data types can be used to refer to the type-checker predicates. For example, the symbol number? evaluates to the type-checker predicate for the number data type; the symbol boolean? evaluates to the type-checker predicate for the boolean data type; and so on. By convention, we pronounce the symbol number? as “number-huh”. Also by convention, every predicate function name ends with the ? symbol.

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:

```racket
> (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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</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.

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

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

```racket
> (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.

C. 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 not symbol with a built-in function. When given a boolean value as input, the not function returns the opposite boolean value, as demonstrated below in the Interactions Window:

```racket
> (not #t)
#f
> (not #f)
#t
```

The and Special Form

In the simplest case, the and special form takes two boolean inputs: (and boolOne boolTwo), where boolOne and boolTwo can be arbitrarily complex expressions that evaluate to true or false.
If both boolean expressions, \texttt{boolOne} and \texttt{boolTwo}, evaluate to true, then the \texttt{and} expression itself evaluates to true. If either \texttt{boolOne} or \texttt{boolTwo} evaluate to false, then the \texttt{and} expression evaluates to false. The following Interactions Window session demonstrates this behavior:

\begin{verbatim}
> (and #t #t)
#t
> (and (> 3 2) (< 5 9))
#t
> (and #t #f)
#f
> (and (> 3 2) (< 4 6) (= 5 9))
#f
> (and #f #t)
#f
> (and (> 2 5) #t)
#f
> (and #f #f)
#f
> (and (< 2 5) (= 9 (/ 18 2)))
#t
\end{verbatim}

The \texttt{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 \texttt{frac} that matches an input argument to its parameter \texttt{x} and returns the fraction $\frac{1}{x}$ if \texttt{x} is a non-zero number and $\frac{1}{x} > .4$, returning $(x+1)$ if \texttt{x} is equal to 0 (avoiding an arithmetic divide by zero error):

\begin{verbatim}
> (define frac
  (lambda (x)
    (if (and (not (= x 0))(> (/ 1 x) .4))
        (/ 1 x)
        (+ x 1))))
> (frac 0)
1
> (frac 1)
1
> (frac 2)
.5
> (frac 3)
4
\end{verbatim}

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

The \texttt{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 \texttt{and} expression is true if and only if all of the input expressions evaluate to true, as demonstrated below:

\begin{verbatim}
> (and #t #t #t #t)
#t
> (and #t #t #f #t)
#f
> (and (> 3 2) (= 9 9) (<= 5 20))
#t
\end{verbatim}

The \texttt{or} Special Form

The \texttt{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 \texttt{or} special form is illustrated below:

\begin{verbatim}
> (or #f #f #f #f)
#f
> (or #f #f #t #f)
#t
> (or #t #t #t #t)
#t
\end{verbatim}
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**.

```scheme
> (or (= 9 8) (> 7 9) (<= 4 2))
#f
```
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.

```racket
> (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, `(* 6 5)`, 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 \('\text{then}'\) is returned as the value of the entire if expression.

In the third expression, the condition, \(#f\), evaluates to \#f. Thus, the string, \("\text{else}"\), 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 \('\text{small}'\); otherwise, its output is the symbol \('\text{big}'\) (assuming the input is a number\(^2\)).

\[
\begin{align*}
> &\ (\text{define} \ (\text{howBig} \ \text{num}) \\
&\quad \ (\text{if} \ (< \ \text{num} \ 10) \\
&\quad \quad \ '\text{small} \\
&\quad \quad \ '\text{big}) \\
> &\ (\text{howBig} \ 5) \\
&\quad \ '\text{small} \\
> &\ (\text{howBig} \ 102) \\
&\quad \ '\text{big}
\end{align*}
\]

### B. 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 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 if\(s\) to make the decisions:

\[
\begin{align*}
> &\ (\text{define} \ (\text{grader-v1} \ \text{score}) \\
&\quad \ (\text{if} \ (> = \ \text{score} \ 90) \\
&\quad \quad \ 'A \\
&\quad \quad \ (\text{if} \ (> = \ \text{score} \ 80) \\
&\quad \quad \quad \ 'B \\
&\quad \quad \quad \ (\text{if} \ (> = \ \text{score} \ 70) \\
&\quad \quad \quad \quad \ 'C \\
&\quad \quad \quad \quad \ (\text{if} \ (> = \ \text{score} \ 60) \\
&\quad \quad \quad \quad \quad \ 'D \\
&\quad \quad \quad \quad \ 'F'))) \\
> &\ (\text{grader-v1} \ 95) \\
&\quad \ 'A \\
> &\ (\text{grader-v1} \ 75) \\
&\quad \ 'C
\end{align*}
\]

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 \(\text{grader-v2}\) is written using a cond statement instead of nested if\(s\):

\[
\begin{align*}
; &\ (\text{grader-v2} \ \text{score}) \to \text{quoted symbol} \\
; &\ \text{score is a number} \\
; &\ \text{Purpose: Return a letter grade for the given score.} \\
;
; &\ \text{Function definition:} \\
&\ (\text{define} \ \text{grader-v2} \\
&\quad (\lambda \ (\text{score}) \\
&\quad \ (\text{cond} \\
&\quad \quad ; \ 'A \ \text{for score} \ 90-100 \\
&\quad \quad [(> = \ \text{score} \ 90) \ 'A] \\
&\quad \quad ; \ 'B \ \text{for score} \ 80-89 \\
&\quad \quad [(> = \ \text{score} \ 80) \ 'B] \\
&\quad \quad ; \ 'C \ \text{for score} \ 70-79 \\
&\quad \quad [(> = \ \text{score} \ 70) \ 'C] \\
&\quad \quad ; \ 'D \ \text{for score} \ 60-69 \\
&\quad \quad [(> = \ \text{score} \ 60) \ 'D] \\
&\quad \quad ; \ 'F \ \text{for score} \ \leq \ 59
\end{align*}
\]

\(^2\)If the input argument is not a number, what do you think the outcome of the function will be?
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_i expr_i] pair is called a clause;
- each boolExpr_i is a condition that evaluates to a boolean;
- the condition in the last clause is either else or #t; and
- each expr_i 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_i that evaluates to false, the corresponding expr_i is not evaluated; and
3) if all the boolExpr_i s 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.

C. 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: To do this, 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 definitions 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.
VIII. The Design Recipe: A disciplined approach to writing functions

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.

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 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 describes the purpose of the function.

;;; (area width height) -> number
;;; width and height are numbers
;;; Calculates the area of a rectangle given the width and height.

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

The next two lines are:
1) the type(s) of data arguments the function takes as input (e.g., two numbers) for place holders width and height, and
2) the purpose of the function.

The style of pre-function comments used in the “How to Design Programs” books and in the DrRacket help desk look something like this (for the area function):

\[
\begin{align*}
\text{;; (area width height) -> number} \\
\text{;; width: number} \\
\text{;; height: number} \\
\text{;; Calculates the area of a rectangle given} \\
\text{;; the width and height.} \\
\text{;; Examples:} \\
\text{;; > (area 5 4)} \\
\text{;; 20}
\end{align*}
\]

In the Help Desk, the section of on-line documentation that covers the function string-length looks something like this (dashes have been added to separate parts of the comment and are not shown in the HtDP books; also, the headings Contract, Input, and Purpose have been added):

\[
\begin{align*}
\text{;; (string-length str) -> nonnegative-integer} \\
\text{;; str is a string} \\
\text{;; Returns the number of characters in str.}
\end{align*}
\]

The contract tells us that the function string-length consumes a string argument for parameter str and produces a non-negative whole number. Next, there is a listing of the input parameter name str and type. Lastly comes the purpose.

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

\[
\begin{align*}
\text{;; (string-append str ...) -> string} \\
\text{;; str : string}
\end{align*}
\]
The ellipses (...) in the contract of the example above tells you that the `string-append` 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 by the ellipses in their contracts when you encounter them in the documentation.

For this class, you should always precede a function definition with a group of comments consisting of

- **Contract:** A comment that gives the name of the function and the name of each of its parameters inside a set of parentheses, followed by an arrow and then the data type of the value returned. If there are multiple possible types returned, represent the output type as the union of all the possible types, within a set of parentheses.
- **Input:** A listing of the function parameter names and each of their data types.
- **Purpose:** A comment that is a brief but complete statement of what the function computes. Reference may be made to any of the parameter names, since they are already introduced in the previous comments.

After a function is defined, it must be called in order to produce a result. Calling a function requires that the function is already defined. The creators of DrRacket have provided a way for you to write the expressions to test your functions before you write the functions. You can’t run these tests until after you defined the function...but making the effort of thinking of a few concrete cases of using the function almost always help you write the function.

### 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 VII. Following the steps in the Design Recipe, we start writing the function with a group of pre-function comments:

```scheme
;; (grader-v2 score) -> quoted symbol
;;
;; score is a number
;;
;; Return a letter grade corresponding to the given score.
```

Now, before writing the actual function definition, we add another step to the Design Recipe—pre-function tests. The general form of a `check-expect` statement is:

```scheme
(check-expect funcCall funcResult)
```

where `funcCall` is a call of the function to be written and `funcResult` is the value returned by the function.

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 and to give you insight into how to write the function.

Writing test 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 from section should look something like this:

```scheme
;; 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 running the check-expect statements will cause errors until the function definition is written.

You may ask how a function can be called before it is written. The answer is that DrRacket moves every check-expect function call 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:

(\text{check-within } funcCall funcResult errorTolerated)

where \textit{funcCall} is a call of the function to be written, \textit{funcResult} is the value returned by the function, and \textit{errorTolerated} is the amount the actual computed value can be expected to differ from \textit{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.

\textbf{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.

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

\begin{verbatim}
;; (arrange-name first last) -> string
;;
;; first and last are strings
;;
;; Concatenate the input strings as follows:
;; last, first (inserting the , into the output string)
;;
;;Pre-function tests:
(check-expect (arrange-name "Donald" "Duck") "Duck, Donald")
(check-expect (arrange-name "Xena" "Warrior-Princess") "Warrior-Princess, Xena")
(check-expect (arrange-name "Matthew" "Vassar") "Vassar, Matthew")

;;Function definition:
(define arrange-name
  (lambda (first last)
    (string-append last ", " first)))
\end{verbatim}

\textbf{Example 2:} Write a function \texttt{pythagorus} to compute the Pythagorean rule:

\begin{verbatim}
;; (pythagorus a b) -> number
;;
;; a and b are numbers
;;
;; Compute the hypotenuse of a right triangle with
;; sides of length \textit{a} and \textit{b} using the formula($\sqrt{a^2 + b^2}$).
;;
;;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:
\end{verbatim}
(define pythagorus
  (lambda (a b)
    (sqrt (+ (* a a) (* b b)))))

Notice that check-within is used to pre-test the \texttt{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.

We can use any names we like for parameters. Thus, for example, we could have written the pythagorus function as:

\begin{verbatim}
(define(pythagorus bob fred)
  (sqrt (+ (* bob bob) (* fred fred))))
\end{verbatim}

However, using random names such as bob, fred, xyzzy, or ghjit makes the code unnecessarily difficult to read and code that uses parameter names that have meaning to the problem being solved is better.

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.

\textbf{D. Post-function printing using the printf 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. The printf function was covered in Section V-A.

\textbf{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 \texttt{grader-v2} program from Sect. VII-B. 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 \texttt{grader-v3} procedure below. This example shows how the printf statements are used inside a function definition.

\begin{verbatim}
; Contract: (grader-v3 score) -> quoted symbol
;-----------------------------
; Input: score is a number
;-----------------------------
; Purpose: Consumes a numeric score between 0 and 100
; and returns a symbol representing the grade.
;
; Side-effect is printing a comment and returning
; the grade.
;
;Pre-function tests:
(check-expect (grader-v3 98) 'A)
(check-expect (grader-v3 88) 'B)
(check-expect (grader-v3 70) 'C)
(check-expect (grader-v3 69) 'belowC)
\end{verbatim}
; 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

; Function definition:
(define (grader-v3 score)
  (cond
   ;----------score=90->100--------------
   [(>= score A) ASYM]
   ;----------score=80->89---------------
   [(>= score B) BSYM]
   ;----------score=70->79--------------
   [(>= score C) CSYM]
   ;----------score<70------------------
   [else DFSYM]))

; Post-function tests:
(printf "(grader-v3 90) should be "a and is "a
\n  'A
  (grader-v3 90))
(printf "(grader-v3 89) should be "a and is "a\n\n  'B
  (grader-v3 89))
(printf "(grader-v3 79) should be "a and is "a\n\n  'C
  (grader-v3 79))
(printf "(grader-v3 60) should be "a and is "a\n\n  'belowC
  (grader-v3 60))

The grader-v3 procedure uses seven global variables that are defined before the function definition. These variables 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 constants such as 0 and 1 are frequently used directly in code. But most literal values should be declared as global variables outside the code, along with comments explaining the reason for creating these variables.
IX. Recursion: Repetition Over Data Sets of Unknown Size

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 $(\text{lambda } (x) (* x x))$, 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 unending program execution—finite loops. Such is the case with the following function, called goodbye:

; Contract: (goodbye) -> void
;-----------------------
; Input: none; this is a 0-parameter function
;-----------------------
; Purpose: Illustrates execution of infinite loop.
;
; Pre-function tests: Not possible due to void output and infinite loop.
;
; Function definition:
(define (goodbye)
  (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 define special form that names \( f \) as a lambda expression with one or more parameters;
2) a cond or if expression (in the function body) that:
   a) includes one or more stopping clauses for 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.

B. Recursion over numbers

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 program 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:

```racket
; Contract: (facty-v1 n) -> positive integer
;-------------------------
; Input: n is a positive integer
;-------------------------
; Purpose: Compute the factorial of n.
;
; Pre-function tests:
(check-expect (facty-v1 0) 1)
(check-expect (facty-v1 1) 1)
(check-expect (facty-v1 4) 24)
(check-expect (facty-v1 5) 120)
;
; Function definition:
(define (facty-v1 n)
  ; Base case
  (if (or (= n 1) (= n 0))
      1
      ; Recursive case
      (* n (facty-v1 (- n 1)))));

; Post-function tests:
(printf "(facty-v1 2) should be \"a\" and is \"a\n\n\"
        2
        (facty-v1 2))

(printf "(facty-v1 6) should be \"a\" and is \"a\n\n\"
        720
        (facty-v1 6))
```
In this function, an if decision statement is used instead of a cond. Either version works because the cond and if are interchangeable (so you can use your favorite in most cases :).

Each call of a recursive function to itself is known as an 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, facty-v1, to the function defined by the λ special form. Notice, too, that the body of this function includes a decision that distinguishes the base cases (i.e., when \( n = 0 \) or 1) from the recursive case (i.e., when \( n > 1 \)). Finally, notice that the body includes a function-call expression that calls facty-v1 with a value that is closer to 1, by using subtraction.

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

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

```scheme
; Contract: (facty-v2 n) -> positive integer
;-------------------------
; Input: n is a positive integer
;-------------------------
; Purpose: Compute the factorial of n.

; Pre-function tests:
(check-expect (facty-v2 0) 1)
(check-expect (facty-v2 1) 1)
(check-expect (facty-v2 4) 24)
(check-expect (facty-v2 5) 120)

; Function definition:
(define (facty-v2 n)
  (cond
    ; Base Case: n=1
    [(= n 1) 1]
    ; Recursive Case: n > 1
    [else (* n (facty-v2 (- n 1)))]))

; Post-function tests:
; Post-function tests:
(printf "(facty-v2 2) should be ~a and is ~a\n\n" 2
 (facty-v2 2))

(printf "(facty-v2 6) should be ~a and is ~a\n\n" 720
 (facty-v2 6))
```

Another equivalent version of the factorial function is given below, this time called facty. This function differs only in that it contains some printf expressions that will help trace what happens when an expression such as (facty 3) is evaluated:
Contract: (facty n) -> integer

Input: n is a positive integer

Purpose: Compute the factorial of n.

Print output to the Interactions Window

as a side-effect.

Pre-function tests:
(check-expect (facty 0) 1)
(check-expect (facty 1) 1)
(check-expect (facty 6) 720)

Function definition:
(define (facty n)
  (cond
    ; Base Case: n=0
    [(= n 0)
      (begin
        (printf " Base Case (n = 0)\n\n"
        1))]
    ; Base Case: n=1
    [(= n 1)
      (begin
        (printf " Base Case (n = 1)\n\n"
        1))]
    ; Recursive Case: n > 1
    [else
      (begin
        (printf "Recursive Case (n = ~A)\n" n)
        (\* n (facty (- n 1))))]])

Post-function tests:
(printf "(facty 2) --> ") (facty 2) (newline)
(printf "(facty 5) --> ") (facty 5) (newline)
(printf "(facty 12) --> ") (facty 12) (newline) (newline)

The \begin is used to allow more than one expression to be executed in the body of an expression (a requirement when using the HtDP Racket languages). When (begin ...) is used, only the last expression is returned. The printing that precedes the last expression is a side-effect that appears in the output when the function is called.

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. 11. 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.

Since the value of n is 3 in the top local environment shown in Fig. 11, 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. 11.
→ Note that the evaluation of the expression, \((\ast \ (\text{lcty} \ (- \ n \ 1)))\), in the top function-call box cannot continue until the subsidiary expression, \((\text{lcty} \ (- \ n \ 1))\), 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. 11, 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, \((\ast \ n \ (\text{lcty} \ (- \ n \ 1)))\), 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. 11.

→ Once again, the evaluation of the expression, \((\ast \ n \ (\text{lcty} \ (- \ n \ 1)))\), 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.

Here is what it looks like when (facty 3) is evaluated in the Interactions Window:

> (facty 3)
Recursive Case (n = 3)
Recursive Case (n = 2)
Base Case (n = 1)

> 6

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):
An expression of the form, (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. For example, to use facty-acc to compute 3!, we would evaluate an expression such as (facty-acc 3 1), as illustrated in Fig. 12.

Notice that the function, facty-acc, is tail recursive because the recursive function-call expression, (facty-acc (- n 1)(* n acc)), is not a subsidiary expression within some larger expression. Thus, the value of the last recursive function-call expression (the “tail” end of the function calls) is the output value.

The difference between this execution and the execution of 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.

A function call is in tail position with respect to a lambda expression if its value is returned directly from the lambda expression, i.e., if nothing is left to do after the call but to return from the lambda expression. When a procedure tail-calls itself as facty-acc does, the result is called tail recursion.

Incidentally, the following description of the output value for the function, facty-acc, is more general, in that it allows the accumulator to have values other than 1:

→ The output value for (facty-acc n acc) is equal to n! * acc.

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
; Contract: (facty-wr n) -> integer
; -----------------------
; Input: n is a positive integer
; -----------------------
; Purpose: 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 n)
  (facty-acc n 1))

; Post-function tests:
 printf "(facty-wr 2) should be ~a and is ~a " 2 (facty-wr 2))(newline)
 (printf "(facty-wr 5) should be ~a and is ~a " 120 (facty-wr 5))(newline)
 (printf "(facty-wr 12) should be ~a and is ~a " 479001600 (facty-wr 12))
```

The wrapper function shields the user from having to know the value of the accumulator. In fact, the user of facty-wr
need 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!!

Using wrapper functions requires another function to be added to the code. What you will more often do in this type of situation is to make the wrapper function a “local” function, as shown in the next section.

E. Local Variables and Local Functions—the local special form

The purpose of a local 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 local expression is the value of the last expression in its body. Once a local expression has been evaluated, its local environment vanishes.

The syntax of local

\[
\text{(local \{[define } \text{var}_1 \text{ val}_1)\text{\})}
\begin{align*}
&\text{(define } \text{var}_2 \text{ val}_2) \\
&\ldots
\end{align*}
\begin{align*}
&\text{(define } \text{var}_n \text{ val}_n)\text{\}}
\begin{align*}
&\text{<body>}
\end{align*}
\]

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 that evaluate to a valid Racket type (including functions); and
- \(<\text{body}>\) is a Racket expression, the value returned from the local.
- the use of \([]\) brackets is optional...the expression can be written completely using only parentheses to divide the local expression into parts.

The semantics of a local expression

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

\[
\text{(local \{[define } \text{var}_1 \text{ val}_1)\text{\})}
\begin{align*}
&\text{(define } \text{var}_2 \text{ val}_2) \\
&\ldots
\end{align*}
\begin{align*}
&\text{(define } \text{var}_n \text{ val}_n)\text{\}}
\begin{align*}
&\text{<body>}
\end{align*}
\]

is evaluated as follows:
- First, the expressions, \(\text{val}_1, \ldots, \text{val}_n\), are evaluated.
- Second, a local environment is created containing \(n\) entries—one for each of the var/val pairs in the local expression. In particular, each symbol \(\text{var}_i\) is associated with the result of evaluating the corresponding \(\text{val}_i\) expression.
- Third, the expressions in the body of the local 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 \(\text{var}_i\) 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 local expression.

Each \(\text{var}_i\) is available to be used in the body. The sequence of the definitions is also important because any \(\text{var}_{i-1}\) is available to be used in the definition of any \(\text{var}_i\).

Using local expressions inside functions

The most frequent use of local expressions is to define variables and functions inside functions. We start by looking at a function that uses variables defined in the global environment and then look at a function that does the same thing only using local variables.
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.

;Contract: (monthly bal) -> number
;Input: bal is a number
;Purpose: Compute monthly interest on bal.
;
; Pre-function tests:
(check-expect (monthly 90) 0.0)
(check-expect (monthly 1000) 2.5)
(check-expect (monthly 50000) 125)
;
; Constant declarations in Global Environment:
(define MIN-BALANCE 100)
(define INT-RATE 0.03)
(define MONTHS 12.0)
;
; Function definition:
(define (monthly bal)
  (if (> bal MIN-BALANCE)
      ;; if bal is over some minimum value,
      ;; calculate monthly balance
      (/ (* INT-RATE bal) MONTHS)
      ;; otherwise, return 0
      0.0))
)
;
; Post-function tests:
(printf "(monthly 90) ==> should be 0 and is ")
(monthly 90)(newline)
(printf "(monthly 1000) ==> should be 2.5 and is ")
(monthly 1000)(newline)
(printf "(monthly 50000) ==> should be 125 and is ")
(monthly 50000)(newline)(newline)

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

;Contract: (monthly-v2 bal) -> number
;Input: bal is a number
;Purpose: Compute monthly interest on bal.
;
; Pre-function tests:
(check-expect (monthly-v2 90) 0.0)
(check-expect (monthly-v2 1000) 2.5)
(check-expect (monthly-v2 50000) 125)
;
(define (monthly-v2 bal)
  ; Give names in the local expression for
  ; the minimum balance, the interest rate,
  ; and number of months.
  (local [(define min-balance 100)
          (define int-rate 0.03)
          (define months 12.0))]
    ; Body of the local expression is the only place the
    ; symbols min-balance, int-rate, and months are visible
    (if (> bal min-balance)
      (/ (* int-rate bal) months)
      0.0))) ;end of local statement
Using the **local** expression to declare the literal values needed in a function has the advantage of keeping the identifiers close to the code in which they are used. When you create a **local** special form inside a function, you are creating a local environment inside the local environment of the function.

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 writing functions and their helper functions (like writing 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:

```scheme
;Contract: (factorial n) -> integer
;Input: n is a positive integer
;Purpose: Compute the factorial of n.
;
; Pre-function tests:
(check-expect (factorial 1) 1)
(check-expect (factorial 4) 24)
(check-expect (factorial 5) 120)
;
; Function definition:
(define (factorial n)
  (local
    ;Start of definitions part of local
    [(define ACC 1)
     (define (fact-helper 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.
     ; Below is call to the local function and it is also the
     ; end of the local expression
     (fact-helper n ACC))
  )
)
```

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**.

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. You may see the **let** special form in CMPU 145. Like the **local** special form, a **let** 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 **let** expression.
Any helper function can be defined within the parentheses of another function and often it makes more sense to do so, using local, because this practice keeps functions and their helper functions close together. However, a good rule of thumb dictates that one function should not be composed of 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.
X. Flat Lists: A Self-Referential, Compound Data Type

You have already seen that there are many different ways to use lists. In this chapter, we will discuss quoted lists: data types 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 explanation:

```racket
; Contract: (cons x l) -> list
; -------------------
; Inputs: x = any data type
; l = list
; -------------------
; Purpose: 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 formal data definition of a list is given below.

```racket
; A list is either:
; 1. empty, or
; 2. it’s a (cons val lst), where val is any valid Racket type and lst is a list.
```

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

```racket
> (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 certainly 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 list shown above is `'(1 2 3).

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

C. Picturing Non-Empty Lists

Fig. 13 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.

Notice that in the top of Fig. 13 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. 13 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. 13. 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)
> (rest (rest (rest exlist)))
empty
> (rest (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 having a known stopping point allows us to do recursion over lists. Remember that a recursive function requires a base case and a recursive case. 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 built-in and has the name length, but it is good to try writing it just for practice):

```scheme
; Contract (list-length LST) -> number
; ----------------------------
; Input: LST = list of anything
; ----------------------------
; Purpose: 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 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)))]))

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 the data definition you saw in the last section:

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

With this definition, we can start with an empty list and use the 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 a list of numbers (lon) as a data type in any subsequent function comments.**

Write a function called `sum-list` to sum all the elements in a list of numbers.

;; Contract: (sum-list listy) -> number
;; ------------------------
;; Input: listy = lon
;; ------------------------
;; Purpose: Compute the sum of numbers in listy
;; Pre-function tests:
(check-expect (sum-list empty) 0)
(check-expect (sum-list (list 3 4 6)) 13)
(check-expect (sum-list (list 8 4 3 42 16)) 73)

;; Function definition:
(define (sum-list listy)
  (cond
   ; Base case: return 0 because listy is empty
   [(empty? listy) 0]
   ; Recursive case: add first number on list to
   ; recursive call on rest
   [else (+ (first listy) (sum-list (rest listy)))]))

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:

;; Contract: (sum-list-acc listy) -> number
;; ------------------------------
;; Input: listy: list of numbers
;; ------------------------------
;; Purpose: Accumulator-style version of function to sum all the
;; numbers in listy.
;; Pre-function tests:
(check-expect (sum-list-v2 '(3 3 3)) 12)
(check-expect (sum-list-v2 '(54 13 4)) 71)
\( \text{(check-expect (sum-list-v2 '()) 0)} \)

; Function definition:
(define (sum-list-v2 listy)
  ; initial parameter name is listy
  (local
    [(define (help-sum lst acc)
      ; inside help-sum, listy is named lst
      (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,
    ; passes 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)))

\section*{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 \texttt{cons} function to put items on the list to be returned. For example, consider the function below that generates a list of random numbers between 1 and 100:

\texttt{Contract: (gen-ran-list num) \to list of numbers}
\texttt{Input: num is a nat-num}
\texttt{Purpose: Generates a list of num random numbers}
\texttt{between 1 and 100.}

; Pre-function tests: Not possible to test contents of list
; because they are random, but we can test the length of the
; list as shown:
(check-expect (length (gen-ran-list 8)) 8)
\texttt{length is primitive function that returns the length of}
\texttt{a given list}
;
; Function definition:
(define
  (gen-ran-list n)
  (cond
    ; base case, return smallest list
    [ (= n 0) empty]
    ; recursive case, cons next random number onto
    ; return list
    [else
     (cons (add1 (random 100)) (gen-ran-list (sub1 n))))]))

Any time you need to perform some operation a certain number of times, you can use recursion in Racket. In the function above, the base case is when \texttt{n = 0}, and the recursive case \texttt{cons}es the next random number (generated using \texttt{add1}—why?) onto a recursive call to the function on \texttt{n - 1}.

\section*{G. 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-letters and follow the design recipe.

As you’re thinking about how to solve this problem, consider the following:
1) Each character in the input string will need to be referenced. Find a primitive string function that will reference an indexed character in a string.
2) Each character in the input string will need to be appended twice onto the string that is returned. Find a primitive function that appends strings.
3) Find a function to convert a character into a string.

The answers to these questions can be looked up in the Help Desk.

The first version of the double-letters 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.

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

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

; Function definition:
(define (double-letters str)
  (local
    [(define len (string-length str))
     (define (doub-letters-helper index)
       (cond
        [(= index len) ""]
        [else
         (string-append
          (string (string-ref str index))
          (string (string-ref str index))
          (doub-letters-helper (add1 index)))])])
  (doub-letters-helper 0))

; Post-function tests:
(printf "(double-letters \"mississippi\") should be \"a and is \"a"
"mmississiiissiippipi"
(double-letters "mississippi"))(newline)
(printf "(double-letters \"kitty\") should be \"a and is \"a"
"kkiitttyy"
(double-letters "kitty"))(newline)(newline)

Look up the string functions used inside the double-letters function (string-length, string-ref, string, string-append) to be sure you understand their purpose. Remember, every function used inside
double-letters must be already defined, otherwise the function would produce an error.

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

```Scheme
; Contract: (double-letters-acc str) -> string
; --------------------------------
; Input str is a string
; --------------------------------
; Purpose: 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-letters-acc "Howdy!") "HHoowwddyy!!")
(check-expect (double-letters-acc "aeiou") "aaeeiioouu")
(check-expect (double-letters-acc "") "") ; "" is the empty string
; Function definition:
(define
 (double-letters-acc str)
 (local
 [((define len (string-length str))
   (define (doub-letters-acc-helper index acc)
     (cond
      ; If index is less than 0, return the acc
      [(< index 0) acc]
      [else
       (doub-letters-acc-helper
        (sub1 index)
        (string-append (string (string-ref str index))
                      (string (string-ref str index))
                      acc))]))]
  ; Initial call to doub-letters-helper
  (doub-letters-acc-helper (sub1 len) "")))
; Post-function tests:
(printf "(double-letters-acc \"Hi\\") should be \"a and is \"a"
  "HHii!!") (double-letters-acc "Hi")(newline)
(printf "(double-letters-acc \"Yippee\\") should be \"a and is \"a"
  "YYiiippppeeee") (double-letters-acc "Yippee")(newline)(newline)

In the post-function test of these functions, notice that a quotation mark inside a string must be preceded by a backslash.

H. Yet more recursion: Using recursion to draw a scene

To make Racket draw pictures, you should include the following line at the top of your program:

(require 2htdp/image)

This require statement imports the image library into the program. The image library lets you draw circles, squares, and other basic shapes. It also allows you to place many images in a single scene.

The simplest scene is an empty white box outlined in black. One way to make this scene is to use the empty-scene function from the image library. The function most often used to place one image on top of another is called place-image. This function has the following contract:

(place-image image x y scene) -> image
where x and y are numbers indicating the x and y coordinates of the center of the image on the x-y coordinate system defined by the scene (which is really just another image).

Figure 14 shows how x and y coordinates are laid out by the Racket image library. The upper left corner is at position x=0, y=0. The x-coordinate increases from left to right, not too surprising. But the y-coordinate increases from top to bottom, which is totally surprising to most people. Since the empty-scene is just another image, you can use the `image-width` and `image-height` functions to access the width and height of a given empty-scene image.

Suppose the empty-scene is defined as a constant using the following statement:

```
(define MT (empty-scene 400 350))
```

To place a solid purple circle at the center of the empty-scene shown in Fig. 14, you could use the following expression:

```
(place-image
  (circle 50 "solid" "purple")
  (/ (image-width MT) 2)
  (/ (image-height MT) 2)
  MT)
```

This expression places a solid, purple circle with diameter 100 on the exact center of the scene called MT.

So, how can we use recursion to draw a picture? All we need is a minimum or maximum value to end the recursion, and this value is used to place the image.

For example, we must use recursion to draw figures like those shown in Fig. 15. The `square-chain-diag` function places a chain of outlined squares from upper-left to lower-right on an empty-scene with side-length 400. The result of running this function is shown on the left side of Fig. 15. A helper function inside this function uses the x,y coordinate of the current square to drive and to end the recursion. Lines of code within the function are numbered.
Fig. 16. Output of square-chain-diag (left), square-diag (middle), and square-d (right).

(require 2htdp/image)

(define MT (empty-scene 380 380))
(define SIDE 20)
(define MODE "outline")
(define CLR "black")

; Contract: (square-chain-diag) -> image
; Header: (define square-chain-diag (lambda () ...))
; Purpose: To draw a chain of squares with side-length 20 from upper-left to
; lower-right on a square scene.
;
; Pre-function tests: Not useful unless we draw the same image here.
;
; Function definition:
(define (square-chain-diag)
  (local
    ;need local function to change x and y coordinates
    ;repeatedly.
    (define (help-draw x y)
      (cond
        ;base case: x and y are off lower right edge,
        ;return the image--the empty scene
        [(and (> x (image-width MT)) (> y (image-height MT))) MT]
        ;recursive case: place another square and call
        ;help-draw with larger x and y
        [else
         (place-image (square SIDE MODE CLR) x y
                      (help-draw (+ x SIDE) (+ y SIDE)))])
    )
  (help-draw (/ SIDE 2) (/ SIDE 2)))

Exercise: Suppose you wanted every other square in the chain shown in Fig. 15 to be solid black. What changes
would you need to make to the square-chain-diag function?

Answer: One way to do this is to add another parameter to the internal help-draw function – an index that starts at 0.
In the function definition square-diag, shown below, all the constants from the square-chain-diag function are used,
plus one extra: MODE2, which contains the string “solid”. This function uses even? or odd? function to determine if
index is even or odd.

(define MODE2 "solid")

; Contract: (square-diag) -> image
; Input: None
; Purpose: To draw a chain of alternating outline and solid squares with
; side-length 20 from upper-left to lower-right on a square scene.
;
; Pre-function tests: Not useful unless we draw the same image here.
;
; Function definition:
(define (square-diag)
  (local
    (define (help-draw x y index)
      (cond
        ;base case: x and y are off lower right edge,
        ;return the image--the empty scene
        [(and (> x (image-width MT)) (> y (image-height MT))) MT]
        ;if index is even, place an outline square
        [(even? index) (place-image (square SIDE MODE CLR) x y
                                    (help-draw (+ x SIDE) (+ y SIDE)))]
        ;if index is odd, place a solid black square
        [else (place-image (square SIDE MODE2 CLR) x y
                            (help-draw (+ x SIDE) (+ y SIDE)))])
    )
  (help-draw (/ SIDE 2) (/ SIDE 2) 0)))
Note that the \texttt{square-diag} function could be written using less code with the if statement written inside the place-image function as shown below:

\begin{verbatim}
(place-image
 (square SIDE MODE2 CLR)
 (- (image-width MT) x) y
 (help-draw (+ x SIDE) (+ y SIDE) (add1 index)))
)
\end{verbatim}

\textbf{Example:} Suppose you wanted another diagonal line of squares to go from the upper-right to the lower-left of the scene, as shown in the right-most subfigure in Fig. 16. Modify the \texttt{square-diag} function to accomplish this. The placement of the squares from top right to bottom left will require the x coordinate to decrease while the y coordinate increases, resulting in an image with crossing lines of squares, as shown in the \texttt{square-d}, below.

\begin{verbatim}
(place-image
 (square SIDE
 (if (= (remainder index 2) 0)
 MODE
 MODE2)
 CLR)
 (help-draw (+ x SIDE) (+ y SIDE) (add1 index)))
)
\end{verbatim}
XI. INTERACTIVE PROGRAMS: USING THE 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). If character sequences are read, they are interpreted as quoted symbols.

The read function is used to get input from the user of the program and should always be used after a 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.

A. Begin, prompt, define, and read pattern

General pattern for reading input:

```
begin->printf->define->read
```

; Contract: (get-name-and-age) -> void; side-effect printing
; Input: none
; 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)
  (begin
   ;; prompt the user for their full name
   (printf "Please enter your full name in quotation marks:\n")
   ;; save the value they type in the local variable name
   (local
    [(define name (read))]
   (begin
    ;; prompt the user for their age
    (printf "\nPlease enter your age:\n")
    ;; save the number in the local variable age
    (local
     [(define age (read))]
    ;; print the result of the two read statements
    (printf "\nName: ˜a, Age: ˜a\n" name age)))))

This pattern of using 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 expression reads the data. The local and define special forms are used to combine the printing and reading in a single 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:

```
printf->let->read
```

; Contract: (get-name-and-age-let) -> void; side-effect printing
; Input: none
; 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 ()
  (printf "Please enter your full name in quotation marks:˜%")
  (let [(name (read))])))
B. 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. You can write some interesting functions that include an element of chance 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 \( n - 1 \). Here are some calls as they would appear in the interactions window to the random function to generate numbers between 0 and 9 (note that the return may be different each time the random function is called):

\[
\begin{align*}
> & (\text{random } 10) \\
& 9 \\
> & (\text{random } 10) \\
& 0 \\
> & (\text{random } 10) \\
& 5 \\
\end{align*}
\]

Here are some calls as they would appear in the interactions window to the random function to generate numbers between 1 and 10. The call to the random function in each of the 3 examples generates a number between 0 and 9 and the add1 function is wrapped around each call to (random 10) to make the result 1 through 10, as desired.

\[
\begin{align*}
> & (\text{add1 (random 10)}) \\
& 6 \\
> & (\text{add1 (random 10)}) \\
& 3 \\
> & (\text{add1 (random 10)}) \\
& 10 \\
\end{align*}
\]

**Exercise:** Write an expression that generates a random number that could be obtained from the roll of a single six-sided die (i.e., 1 through 6).

**Exercise:** Write an expression that generates a random number that could be obtained from the roll of two six-sided dice (careful, this is trickier than it seems).

Simple interactive programs can be written by using combinations of prompting the user, reading the input, and using the random function. A simple interactive game is called “Guess the number”, where person A (the computer) generates a random 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. You will be asked to write this program as a lab or homework assignment.

C. Using mutation of global variables in place of an accumulator

Using assignment and global variables is an alternative to using accumulators. In Racket, putting a new value into a previously defined variable is done by using the set! special form. This will be demonstrated in the programs below, where an accumulator function sum-list is rewritten as a function that maintains “state” over time in global variables. Care must be taken when using global variables because, unless the variables are reset to their initial values
before running the program, they may contain a value set in a previous run of the code.

; Contract: (sum-list listy) -> number
; Input: listy is a list of numbers
; Purpose: Uses an accumulator to return the result of summing
; the numbers in list

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

; Function definition:
(define (sum-list listy)
  (local
    [(define (sum-list-acc lst acc)
      (cond
        [(empty? lst) acc]
        [else
          (sum-list-acc (rest lst) (+ (first lst) acc))])])
  (sum-list-acc listy 0)))

The sum-list function is rewritten below using a global variable to hold the sum over time.

; First, define a global variable to hold the sum:
(define sum 0)

; Contract: (sum-list-starter listy) -> number
; Input: listy is a list of numbers
; Purpose: Entry point for the sum-list program.
; Ensures sum is initialized to 0 before
; calling recursive sum-list

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

; Starting point function definition:
(define (sum-list-starter listy)
  (begin
    ;; Be sure global variable is initialized to 0
    (set! sum 0)
    ;; Call recursive function to calculate sum
    (sum-list-v2 listy)))

; Contract: (sum-list listy) -> number
; Input: listy: list of numbers
; Purpose: State maintaining version of function to sum all the
; numbers in listy.

; Function definition:
(define (sum-list-v2 listy)
  (cond
    ; base case: listy empty so return current value of sum
    [(empty? listy) sum]
    ; recursive case: save the result
    ; of adding the first number on list
    ; to sum and pass in rest of listy
    ; as argument to recursive call.
    [else
      (begin
        (set! sum (+ (first listy) sum))
        (sum-list-v2 (rest listy)))]))
Another option using the `set!` function is to make the mutated variable local to the function as shown in the example below:

```scheme
; Contract: (sum-list-v3 listy) -> number
; Input: listy: list of numbers
; Purpose: Local state maintaining version of function to sum all the
; numbers in listy.

; Function definition:
(define (sum-list-v3 listy)
  (local
    [(define sum 0)
     (define (sum-list-setter lst)
       (cond
         ; base case: lst empty so return current value of sum
         [(empty? lst) sum]
         ; recursive case: save the result
         ; of adding the first number on lst
         ; to sum and pass in rest of lst
         ; as argument to recursive call.
         [else
          (begin
            (set! sum (+ (first lst) sum))
            (sum-list-setter (rest lst)))]))
    (sum-list-setter listy))
```

Whenever you are planning to write a program that uses mutation, you should remember to:

- Declare any variables to be mutated with a base value either as global or local variables outside any recursive function.

- If using global mutation, create an entry function that sets the global variables to their initial state and that calls the function to process the data.

- Make changes to the value being calculated inside the recursive function using the assignment special form, `set!`. 
XII. 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.

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 filter higher-order function

One built-in higher-order function that can make our lives easier is the filter function. Filter applies its first argument, a one-parameter predicate function, 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 function returns true.

The definition of my-filter, below, shows how we could go about writing a procedure equivalent to the built-in filter procedure:

```racket
; Contract: (my-filter pred? lox) -> list of valid Racket type
; Input:  pred? is a one-parameter function that produces a boolean;
;        lox is a list of elements for which pred? is valid
; Purpose: Produces a list containing only the elements in
;          lox for which pred? is true

; Pre-function tests:
(check-expect (my-filter odd? '(1 2 3 4 5)) '(1 3 5))
(check-expect (my-filter odd? '()) '())
(check-expect (my-filter odd? '(1 1 3 3 5)) '(1 1 3 3 5))
(check-expect (my-filter odd? '(2 2 4 4 6)) '())

; Function definition:
(define (my-filter pred? lox)
  (cond
   [(empty? lox) empty]
   [else (cons (first lox) (my-filter pred? (rest lox))))])
```

The names for filter-style functions probably make sense to you; the metaphor of the air filter that allows air through but doesn’t allow dirt, and so on, brings to mind something that passes some data and blocks other data.

B. The 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 my-map, below, shows how we would go about writing a procedure equivalent to the built-in map procedure:

```racket
;; Contract: (my-map oper lox) -> list of type y
;; Input: oper is one parameter function that consumes
;;        type x input, lox = list of type x
;; Purpose: produce list that results from applying oper to
;;          every element in lox

;; Pre-function tests:
(check-expect (my-map add1 '(1 2 3 4)) '(2 3 4 5))
(check-expect (my-map sub1 '(2 3 4 5)) '(1 2 3 4))
```
(check-expect (my-map abs ’(-1 -2 -3 -4)) ’(1 2 3 4))

(define (my-map oper lox)
  (cond
   [(empty? lox) empty]
   [else (cons (oper (first lox))
             (my-map oper (rest lox)))]))

;; Post-function tests:
(check-expect (my-map add1 ’(1 2 3 4)) ’(1 3 4 5))
(check-expect (my-map sub1 ’(2 3 4 5)) ’(1 2 3 4))

Like the my-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 using the built-in function map.

> (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(s). 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) ’(3 3 3 3))
(list 6 6 6 6)

C. The apply higher-order function

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 list-of-num) -> number
; Input: list-of-num = list of numbers
; 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 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.

\[
\text{Contract: } (\text{mult-list list-of-num}) \to \text{number}
\]
\[
\text{Input: } \text{list-of-num} = \text{list of numbers}
\]
\[
\text{Purpose: } \text{Produces the product of all numbers in list-of-num.}
\]

\[
\text{Pre-function tests:}
\]
\[
(\text{check-expect (mult-list '(1 2 3 4 5)) 120})
\]
\[
(\text{check-expect (mult-list '(1 1 1 1)) 1})
\]
\[
(\text{check-expect (mult-list '()) 1}) \text{ empty list => identity for *}
\]

\[
\text{Function definition:}
\]
\[
(\text{define (mult-list list-of-num)}
\[
\begin{cases}
\text{[(empty? list-of-num) 1]} \\
\text{[else}}
\[
(* \text{(first list-of-num)}
\[
\text{(mult-list (rest list-of-num))})])
\]
\]

The APPLY function:

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.

\[
\text{Contract: } (\text{apply func list-of-x}) \to \text{Y}
\]
\[
\text{Input: } \text{func is a one-parameter function, list-of-x is list of any type}
\]
\[
\text{Purpose: } \text{To apply func to every element of lst.}
\]

In effect,

\[
(\text{apply } f \text{ (list x-1 ... x-n)} = (f x-1 ... x-n)).
\]

Examples of using this function are given below:

\[
> (\text{apply } + \text{ (list 1 2 3 4)})
\]
\[
10
\]
\[
> (\text{apply } * \text{ (list 1 2 3)})
\]
\[
6
\]

D. The build-list higher-order function

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...(n-1). In effect,

\[
(\text{build-list n f}) = (\text{list (f 0)} \ldots (\text{f (}- \text{n 1}))).
\]

Using build-list is an easy way to create large lists for testing.

Also, remember that the function f can be an unnamed function or a function you have defined yourself, instead of a built-in function.

E. The andmap and ormap higher-order functions

The ANDMAP function is similar to map in the sense that a predicate function, pred?, is applied to each element of one or more equal-length lists, making a list equal in length to let, but consisting of only #t and #f values. After making this temporary list (which you never get to see) andmap applies the and special form to the boolean values in the temporary list, resulting in a true value only if all the items in the temporary list are true. The result is #f if any application of pred? produces #f, in which case pred? is not applied to later elements of the lsts.

The ORMAP function is also similar to map in the sense that a predicate function, pred?, is applied to each element of one or more equal-length lists, making a list equal in length to let, but consisting of only #t and #f values. After making this temporary list (which you never get to see) ormap applies the or special form to the boolean values in
the temporary list, resulting in a false value only if all the items in the temporary list are false. The result is true if any application of pred? produces true, in which case pred? is not applied to later elements of the lsts.

Remember that, for all the higher order functions mentioned in this section, the function passed in as an argument can be an unnamed function or a function you have defined yourself, instead of a built-in function.
XIII. Other 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. Structs: Containers for fixed-size data

You can think of non-empty quoted lists as containers for an arbitrary amount of primitive or compound data. In a list, only the first element is accessible within a function. The list must be traversed using recursion to access any of the other elements. Containers called structs allow you to create new data types in which every field is directly accessible, without using recursion. If you need to use data containers that 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 special form used to create new structs is define-struct.

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). define-struct is 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.

B. Defining new data types—define-struct

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 in which all the field values are accessible without using recursion.

To define a struct to represent a point in the x-y plane in Racket, the creators of DrRacket made the posn struct by using the following syntax:

(define-struct posn (x y))

In this statement, the name of the struct defined is posn and the fields are x and y. The define-struct 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

- **set-posn-x!** and **set-posn-y!**: Functions known as “mutators” because they change the value of the x or y field in a particular posn. Examples of using these functions on the point1 posn in the Interactions Window are given below:

```scheme
> (set-posn-x! point1 3) ;; void return
> (set-posn-y! point1 6) ;; void return
> point1
(make-posn 3 6)
```

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

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

C. **Using structs in animations**

The animation engine in DrRacket is a special form called **big-bang**. The way big-bang was designed makes it a necessity to use structs. The big-bang function can contain a number of “clauses”, commonly known as **event handlers**. There is an event handler to draw and redraw a scene (on-draw), another type that responds to clock ticks (on-tick), a type for responding to mouse events (on-mouse), and a type to respond to key presses (on-key). If the animation (also known as simulation) you are writing needs to stop when a given event occurs, there is a type of clause to do so (stop-when).

Each of these event-handlers are “higher-order functions” because they each consume a function that is called by the system when particular events occur.

Structs are used in big-bang animations because everything that changes from scene to scene must be put in a container so that only one argument that describes the current state is passed to any of the functions specified in the event-handling clauses.

For example, suppose you are writing an animation of a continually moving image. For any continual motion animations the on-tick clause is used. This clause consumes a one-parameter function that takes the state of the world as input and that produces the state of the world, either new or unchanged, as output.

To draw the current state as an image, the on-draw (or to-draw) clause consumes a one-parameter function that produces an image, but does not change the state of the world. The on-draw clause is the way we can determine how the current scene looks.

If the image(s) in an animation change position, the struct to describe that change is usually a posn. However, if the x and y coordinates do not change in the same way on every clock tick, more information may be needed in the struct that describes the current scene. For example, in a simulation of a perpetually bouncing ball that moves at a non-right angle to the sides of the scene, a bounce of that ball may change one of the coordinates. This change factor should be a part of the struct used to describe a scene.

When writing an animation in DrRacket, you need to be very detail-oriented, keeping track of all the fields contained in every struct involved with describing the state of the scene.
In a simulation where many ufo’s are dropping out of the sky and being shot at by a gun that can only move from right to left, the struct to describe the world would need a struct detailing the current coordinates of all the ufo’s plus the change in position of each one, probably in a list. That list would be included in a struct that describes the world, along with fields that describe the x coordinate of the gun. If the gun has “bullets”, the position of the bullets would be another list in the struct that describes the world scene. Hopefully, you are beginning to get a feel for how complex a struct used for even a simple “shoot the aliens” animation would be.

D. Vectors: Fast access data containers

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.

This section will be expanded in future versions of these notes.
XIV. PROGRAMMING USING FULL SWINDE

To learn about iteration and loops, you need to change the language to Full Swinde. This language is selected using Choose Language under DrRacket’s Language menu. In the next window, click the radio button next to “Other Languages”. Click the arrow next to the word “Swinle” and choose “Full Swinle”. Then click OK to exit the menu and click run to make the change stick.

A. Coding differences between HtDP and Swinle

There are some changes you need to make to a Racket program in order for it to run with no errors in Swinle:

1) A file must use a require statement for every teachpack used in the program. For example you can add the universe and image teachpack by typing (require 2htdp/universe) and (require 2htdp/image) near the top of the program.

2) If the file contains check-expect statements, you must include the line (require test-engine/racket-tests) near the top of the program and the line (test) at the bottom.

3) The syntax to create structs is different in Swinle. Instead of typing, for example, (define-struct posn (x y)), you would type (def-struct <posn> () x y). The constructor, accessor, mutator, and type-checker functions are all added to the Global Environment when this statement is run, and are given the same names as they had when you used a define-struct statement.

4) In Swinle, there is no need to use a begin special form around statements to be executed in sequence. Swinle will execute any number of sequential statements, returning the value of only the last such statement in the body of a function.

5) In Swinle, you can use iteration instead of just recursion. One of the iterative special forms you can use is the while statement. This special form is described below.

B. Iteration using while special form

It is important to know about iteration to prepare yourself to use object-oriented languages like Java and C++. Unlike recursive procedures that save the state of a computation in an accumulator, a while statement can be used to mutate state variables to accomplish the same goal. Since iteration generally involves mutation, it is imperative that you initialize the state of variables at the beginning of your program (so the variable does not contain values obtained from previous runs of the program).

The while keyword is used in the example below. In this example, we are calculating the sum of values from 1 to some positive integer n.

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

; Contract: (summation n) -> positive integer
; Input: n is a positive integer
; Purpose: To produce the sum of integers from 1...n

; Pre-function tests:
(check-expect (summation 5) 15)
(check-expect (summation 7) 28)
(check-expect (summation 0) 0)

(define summation
  (lambda (n)
    (local
      ;; initialization of local variables
      [(define x 1) ;; initialization of local variables
        (define sum 0)]
    )
  )
)```
(while (<= x n) ;; stopping condition
    (set! sum (+ x sum)) ;; mutators
    (set! x (add1 x)))
  sum))) ;; need to return value of sum
(test)

You are encouraged to look back over the programs you wrote earlier in the semester using recursion. It should be possible to write any recursive function using iteration, although the iterative version is usually longer.