CMPU101-53 Spring 2019
Problem Solving and Abstraction

Calling Built-in Functions
Default Evaluation Rule

The default evaluation rule for non-empty lists is used every time a non-quoted, non-empty list expression is evaluated. By this rule, for an expression such as

\[(d_1 \ d_2 \ d_3 \ \ldots \ \ d_n),\]

*each* of the expressions \(d_1\ldots d_n\) are evaluated. \(d_1\) must evaluate to a function, and the results of evaluating \(d_2\ldots d_n\) are arguments to function \(d_1\).
Evaluation vs Look-up

Evaluation of an expression involves using the default rule to reduce the expression to simplest form.

Evaluation of a name that is defined in the GE involves looking up the name in the table. All values written in the GE are in simplest form.
A function can be defined to have any fixed number of inputs (≥ 0), but functions can return at most one output. A function can also create side-effects and the output can be void.
Side-Effect Printing

A function takes some number ($\geq 0$) input(s), and produces some number (0 or 1) output(s).

Functions that produce 0 outputs return $\text{void}$ and generally have side-effects. The side-effect we will concentrate on for now is $\text{side-effect printing}$. 
Side-Effect Printing

Side-effect printing is not considered an output because it is not a valid scheme type.
Commenting Code

All characters typed to the right of a semicolon (;) in DrRacket are called comments.

Comments are ignored by Racket—they are just for the programmers who have to read the code (like the future you or your present professor, me).
Comments inside functions

When writing a function, it is a good idea to include comments, either between lines of code or to the right of the code.

Comments are not evaluated. They are meant to make the purpose of the code clear to anyone reading it. Comments are an essential part of every program.
Calling functions

Functions are classified according to the number of parameters they have: 0-, 1-, 2-, 3-, ...k-parameter functions, where the number of arguments matches the number of parameters.

We can find out how many parameters a built-in function takes by looking up the function in the Help Desk. Look up the expt, remainder, max, and min functions.
Function Contracts

A function contract is written before a function definition, usually in comments, or in a call to built-in functions display or printf.

We will use the display function to make the function contract a part of side-effect printing which appears with the output of the function in the IW.
The display function

The display function has the following contract:

- **Contract:** \( \text{(display string-arg)} \rightarrow \text{void} \)
- **Input:** string-arg is a string.
- **Output:** void
- **Side-effect:** Prints string-arg to IW

The display function is called using the default-evaluation rule as shown below:

\( \text{(display "Hello, world!\n")} \)
The display function

If you try typing a string and evaluating it, the quotation marks will show in the output (because strings evaluate to strings). However, a string printed as a side-effect in a call to the display function will have no quotation marks.
The display function

You can embed "escape characters" inside the string argument. For example, typing \n (backslash-n) in the string will cause a linefeed in the printing and a " (backslash-quotation mark) will cause a " to appear inside ""s.

For a \ inside a string, use \\ (backslash-backslash)
The display function

A nice way to include a function contract in side-effect printing is shown below:

```
(display "
  Contract: (display string-arg) → void
  Input:   string-arg is a string.
  Output:  void
  Side-effect: Prints string-arg to IW \n")
```

The returns included in the string argument will cause the lines to print exactly as shown and the \n at the end will add a blank line after the string.
Calling Numeric Functions

In the lab today, you were asked to write calls to numeric operators. For example, suppose you are asked to write and evaluate the expression:

\[ 40 + 41 + 42 + 43 \]

If you look up the + function in the Help Desk, you will see a contract that looks like this:

- **Contract**: \( (+ \ z \ ... ) \rightarrow \text{number} \)
- **Input**: \( z \) and all other inputs are numbers
- **Output**: the result of adding all input numbers
Calling Numeric Functions

In Racket, you would write this expression as:

```
(+ 40 41 42 43)
```

Notice that this expression produces only a number in the IW output. To make the output more readable, it is good to display the function you are about to evaluate before you actually evaluate it:

```
(display "( + 40 41 42 43 ) --> ")
(+ 40 41 42 43)
```
Testing Functions

Check-expect is a function used to test the result of calling a function before you call it:

( check-expect ( + 40 41 42 43) 166 )

In lab, you used a check-expect to test each expression you wrote before you displayed the expression and showed its return value.
Length of Lines in Code

When you are writing a program, it is essential to write the code and the comments such that no typed characters extend beyond 80 columns.

Lines that extend beyond 80 columns tend to "wrap around" to the next line when they are printed.

DrRacket allows you to set a line in the code for the maximum line width you will allow.

(Preferences → Editing → General Editing)
Function Contracts

One type of comment is essential for all programs: Function Contracts. These comments explain the type of inputs, type of output, the purpose and any side effects.

(display "
  Contract:  (add3 n1 n2 n3) --> number
  Input:      n1, n2, and n3 are numbers.
  Output:     n1 + n2 + n3, a number\n")
Lab 1

I will briefly cover the files used for lab 1, which takes place today at 3:10 pm.
Like all other programming languages, Racket has what are called KEYWORDS. These are character sequences that are a part of the language syntax. Keywords in Racket are called SPECIAL FORMS because the evaluation of each one is slightly different, and none of them are the same as the default evaluation rule.

Using a special form looks like using the default evaluation rule but special forms are each evaluated in their own way.
The define special form

Programs in Racket are combinations of function and value definitions (constants).

The **define** special form is the mechanism by which you can add entries to the global environment (GE). Define can be used to add names for functions and values (constants) to the GE.
The special form `define` is used as shown below to name constants in the GE:

```
(define INNINGS 9)
(define STRIKE-LIMIT 3)
(define STARTERS 10)
```
Evaluation of a DEFINE expression

When a left parenthesis occurs before the keyword `define` there are always exactly 2 arguments that follow the keyword. Neither the keyword `define` nor the first argument is evaluated. Only the second argument is evaluated to simplest form.

```
(define TURTLE-WT (* 50 BEAK-LENGTH))
```

<table>
<thead>
<tr>
<th>(define</th>
<th>TURTLE-WT</th>
<th>(* 50 BEAK-LENGTH) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp form</td>
<td>constant name</td>
<td>value</td>
</tr>
<tr>
<td>(not eval)</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; argument (not evaluated)</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; argument (evaluated)</td>
</tr>
</tbody>
</table>
The define expression is processed as follows:
1. **define** is read but not evaluated,
2. the 1\textsuperscript{st} argument is a *name* for a function or value and is not evaluated but is written in the *name* column of the GE, and
3. the 2\textsuperscript{nd} argument is evaluated and written in the *value* column of the GE.

\[(\text{define } \text{STARTERS } (+ (* 5 2) 3)))\]

would write the name STARTERS and the value 13 in the GE
When defining a name for a number or other value, only the second argument is evaluated to simplest form (in the 2\textsuperscript{nd} example, it is an expression).

\begin{verbatim}
(define NUM-RINGS 100)
(define TREE-AGE (* NUM-RINGS 10))
\end{verbatim}

The first statement writes the name NUM-RINGS in the GE. The second statement looks up the value of NUM-RINGS and then evaluates the expression (* NUM-RINGS 10), setting the value 1000 for TREE-AGE in the GE.
Defining Functions

There are 2 ways we will define functions. The first is a simplified version to avoid complexity for beginners.

The second form gives the actual value of the function as it is stored in the GE.
Defining Functions - Method 1

A function is written to the GE when the `define` keyword is followed immediately by an uncommented open-parenthesis containing $\geq 1$ name. The name to the right of define is the name of the new function, the other names are parameter names. This parenthesis is closed and followed by the body of the function in its own set of parentheses.

Ex: `(define (horse-info shoulder-ht ear-ht shoe-ht)
     (- (+ shoulder-ht ear-ht) shoe-ht))`
Defining Functions – Method 1

Ex: **red** indicates function name,
    **blue** indicates parameter names
    **green** indicates body of function and return value

(define (horse-info shoulder-ht ear-ht shoe-ht)
  (- (+ shoulder-ht ear-ht) shoe-ht))
Defining Functions – Method 2

The **lambda** special form is the way Racket programs actually store a function. For example, the definition of the function horse-info is really evaluated to the following before it is executed:

```
(define horse-info
  (lambda (shoulder-ht ear-ht shoe-ht)
    (- (+ shoulder-ht ear-ht) shoe-ht)))
```

In either of the cases for function definition, the define special form returns void.
Calling Functions

If you try *calling* either version of the function horse-info with 3 numeric *arguments*, where calling is writing the function name preceded by an unquoted open parenthesis, i.e.

\[(\text{horse-info } 63 \ 72 \ 2)\]

you will get a numeric return value, in this case 133.

Default evaluation is applied here.
Calling Functions

The define special form for defining function names returns void, but has the important side-effect of writing a name and the associated lambda (function) in the GE.

Calling the function produces an output, but define has only a side-effect and void output.
Side-Effects of Functions

Side effects include, e.g.:

• Writing a name in the Global Environment.
• Printing to the interactions window.
• Changing the value of a constant or parameter defined in the Global Environment (later...).
Naming Functions

In summary, the **define** special form is used to write a name associated with a **lambda** special form in the GE:

```
(define cube1
  (lambda (z) (* z z z)))
```

has exactly the same meaning as

```
(define (cube2 z) (* z z z))
```

The first expression writes the name `cube1` and the second writes the name `cube2` in the GE as **lambda expressions**.
Calling a Defined Function

To use the cube function after it has been written to the GE, you type a left parenthesis, followed by the name of the function, followed by values for the argument(s):

\[
(cube\ 50) \;; \rightarrow 125000
\]

Calling this function has the output of 125000 but no side-effects.