Introduction to Problem Solving and Abstraction
Lecture on September 23, 2019

Topics to cover: Parts of chapters 8, 9, 11, and 13

- A lambda expression has the following syntax:

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

where each \(C_i\) is a character sequence denoting some symbol \(s_i\); the symbols \(s_1, s_2, \ldots, s_n\) are distinct (no repeats); and \(B\) is a character sequence denoting a Racket datum \(D\) of any valid type.

- Applying nameless functions to input values: This method works, but would be much too laborious for routine use.

E.g., \(((\text{lambda} (x \ y \ z) (* \ x \ y \ z)) \ 2 \ 3 \ 4) \Longrightarrow \ 2 * 3 * 4 \Longrightarrow \ 24.\)

This expression uses the DER\(^1\) because the first item in the non-empty list is a function, signified by the lambda special form.

- Evaluation of a function definition: Expressions in the body of a function can refer either to data stored in the local environment or in the GE. The local environment contains entries associated with the input parameters.

Consider this function definition:

\[
\text{(define threeDarea}
\text{(lambda (l w h)

\text{(* l w h)))})}
\]

When a function call such as \((\text{threeDarea} \ 5 \ 15 \ 6)\), is evaluated

1. a local environment is set up, containing an entry for each symbol, \(l \ w \ h\), in the parameter list, matching each of these symbols with the value of the corresponding argument in the function call \((l=5, w=15, h=6)\),

2. the body of the function is evaluated with respect to the local environment and the GE (for the value of \(*\)), and

3. a value \((450)\) is returned as the result of step 2.

Arguments are matched to parameters in sequence from left to right. In the example of the nameless function call, above, the parameter \(x\) is matched with the argument 2, the parameter \(y\) with argument 3, and the parameter \(z\) matched with the argument 4.

---

\(^1\)DER stands for “Default Evaluation Rule” for function evaluation.
It is critical to understand that each symbol in the parameter list of a function definition is defined only when the function is called and not before. It is not until a function call is made that the local environment is created and the parameter names are matched with argument values given in the function call. So if we have defined the function income-tax as follows:

```
(define income-tax
  (lambda (net bracket)
    (* net (/ bracket 100)))))
```

and subsequently clicked on Run, the symbols `net` and `bracket` are not defined, although the `income-tax` symbol is written into the GE and the lambda expression is saved as its value in the GE: (lambda (net bracket) (* net (/ bracket 100))).

It is not until we call the function using the DER like this:

```
(income-tax 300000 25)
```

that the parameters `net` and `bracket` are written into a local environment (created just for that particular call to `income-tax`), with values in the local environment shown below:

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>300000</td>
</tr>
<tr>
<td>bracket</td>
<td>25</td>
</tr>
</tbody>
</table>

What is wrong with the following lambda definition?:

```
(lambda (3) (* 3 3))
```

The answer is that the parameter list can contain only symbols that are placeholders for literal data. The number 3 is not a symbol and can evaluate only to 3.

- The `exact->inexact` function and Lab 3. Use the `exact->inexact` function when the answer may be an inexact number. This function converts the number from a large fraction (which is exact, but difficult to understand) to a decimal number.

- The `check-within` function is like the `check-expect` function except that it has one more argument, which is the error tolerance: `(check-within (+ 2.56 3.14) 8.03 .5)`, where the first argument is a call to the + function, the second argument is the expected result, and the .5 is the amount the 8.03 can vary (the error tolerance).
Chapter 8: Predicate functions

Chapter 8 covers predicate functions and we cover them here because predicates are commonly used when making decisions in code. Chapter 11 introduces the first decision statement: the IF special form.

• Type-Checker Predicates:

The Racket language provides a type-checker function for every Racket data type. These include (but are not limited to) the following functions:

- number?
- symbol?
- boolean?
- char?
- procedure?
- list?
- empty?
- string?

Notice that many of the predicate type-checkers mirror the type name, followed by a question mark. All these functions return #f except if the argument evaluates to the type specified in the predicate. So these functions take any data type as input and return a boolean.

Here is a contract for the symbol? function:

Name: symbol?
-------------------------------------
Usage: (symbol? x) --> boolean
Input: x can be any type of racket data
Output: #t if x is a quoted symbol and #f otherwise

Note that the symbol? function does not look in the GE to see if x is a defined variable or function name. It returns #t only if the argument for x is a quoted symbol.

• Section 7.2: The QUOTE special form

The quote special form was introduced in chapter 7, section 2. The quote special form can be used to shield symbols or lists from evaluation and it is normally written in the short-hand form “single quote” or ’ (apostrophe).

Quoted lists start with a ’ so that the list is not evaluated by the DER. Likewise, a quoted symbol is not looked up in either the Global or Local environment...it is just a sequence of non-blank characters preceded immediately by an apostrophe.

A quoted symbol is evaluated to a non-empty list that begins with the quote special form. For example:

'abc --> (quote abc) ==> abc
'(1 2 4) --> (quote (1 2 4)) ==> the non-empty list containing 1, 2, and 4
We will use quoted lists frequently when we want to treat lists like containers of data...not to be evaluated by the DER.

Skip Chapter 10.

Chapter 11: Conditional Expressions I–Making Decisions with the IF and WHEN special forms

Many problem solutions involve making decisions from a set of possible alternatives. Many data types have their own type-checkers and comparison predicates, plus there is a general comparison predicate called eq?. The eq? operator works on booleans, numbers, symbols, or the empty list.

• The IF special form

The syntax of the if special form is as follows:

\[(\text{if } \text{predicateExpr} \quad \text{thenExpr} \quad \text{elseExpr})\]

where \(\text{predicateExpr}\) is an expression that evaluates to \(\#t\) or \(\#f\); and \(\text{thenExpr}\) and \(\text{elseExpr}\) are any Racket expressions (i.e., they can evaluate to any Racket type).

The semantics of the if special form tell us that it is evaluated as follows:

1. First, the condition, \(\text{predicateExpr}\), is evaluated. After this, one of 2 things will happen:

   (a) Case 1: \(\text{predicateExpr}\) evaluates to \(\#t\). In this case, \(\text{thenExpr}\) is evaluated and \(\text{elseExpr}\) is not evaluated. The value of the if is whatever \(\text{thenExpr}\) evaluates to.

   (b) Case 2: \(\text{predicateExpr}\) evaluates to \(\#f\). In this case, \(\text{elseExpr}\) is evaluated and \(\text{thenExpr}\) is not evaluated. The value of the if is whatever \(\text{elseExpr}\) evaluates to.

In the if special form, the \(\text{predicateExpr}\) is always evaluated. However, the if special form does not adhere to the DER because only one of \(\text{thenExpr}\) and \(\text{elseExpr}\) is ever evaluated.

Examples of using if special form:

\[
> (\text{if} \ (> 7 9) \ (* 8 2) \ (* 6 5))
30
> (\text{if} \ (< 2 3) \ \text{'then} \ \text{'else} \ \text{then})
> (\text{if} \ #f \ "\text{then}" \ "\text{else}"
"else"

4
When interpreting `predicateExpr`, anything that does not evaluate explicitly to `#f` evaluates to `#t`.

The `eq?` function is used when comparing booleans. Remember, this function is just `eq?`, not `eq=?`. There is no `=?` operator.

---

2. The WHEN special form

The `when` special form is like an `if` except that you don’t need an else. In other words `when` is like an `if` expression that has no `elseExpr`. However, you can write multiple statements inside a `when` expression. Each one but the last should cause a side-effect because only the last expression evaluated is returned.

### Chapter 13: Conditional Expressions II

1. Nested IF statements:

   If there are more than two answers to choose from, you would like to have an `n-ary` special form that makes `multi-way` decisions. This can be done by using `nested` `if` expressions. For example, consider the function `letter-grade` with the following contract and definition:

```
;; Name: letter-grade
;;---------------------------------------
;; Usage: (letter-grade score) --> quoted symbol
;; Input: score is a number between 0 and 100
;; Output: One of the quoted symbols A, B, C, D

(define letter-grade
  (lambda (score)
    (if (>= score 90) ;; true part of 1st if
        'A ;; true part of 1st if
        (if (>= score 80) ;; false part of 1st if
            'B ;; true part of 2nd if
            (if (>= score 70) ;; false part of 2nd if
                'C ;; true part of 3rd if
                'D ) ;; false part of 3rd if
            ) ;; end of 2nd if
        ) ;; end of 1st if
    ) ;; end of lambda
  ) ;; end of define
```

You will find another special form in Chapter 13, called `cond`, that is most commonly used for making multi-way or `n-ary` decisions.

---

2 Multiple expressions are not allowed in the if special form without extra syntax
2. The COND special form

```scheme
;;; Name: letter-grade-v2
;;;---------------------------------------
;;; Usage: (letter-grade-v2 score) --> quoted symbol
;;; Input: score is a number between 0 and 100
;;; Output: One of the quoted symbols A, B, C, D

(define letter-grade
  (lambda (score)
    (cond
      ((>= score 90) 'A) ;; case 1: got an A
      ((>= score 80) 'B) ;; case 2: got a B
      ((>= score 70) 'C) ;; case 3: got a C
      (else 'D) ;; case 4: got a D
    ) ;; end of cond
  ) ;; end of lambda
) ;; end of define
```

Two more special forms are the logical operators AND and OR, and the function NOT.

3. The AND and OR special forms

The syntax of the and special form is as follows:

```scheme
(and boolean-expression1 boolean-expression2 ... boolean-expression-n)
```

The and special form takes in any number of boolean expressions and returns #t only if all the boolean-expression expressions evaluate to #t. Evaluation of the and special form stops when the first boolean-expression expression evaluates to #f. This is known as short-circuit or lazy evaluation and it is this feature that makes the and special form different from the DER.

The or special form is similar in that it takes in any number of boolean expressions. However, the or special form returns #f only if all its boolean expression inputs evaluate to #f.