1 Today’s topics

We’ll add four more rules of processing: cond, and, or, if.

We’ll write some more programs, following the Design Recipe each time.

We’ll “compose” two procedures, and briefly hint at how “decomposition” of problems leads to good programs.

2 Four new rules of processing

2.1 cond expressions

\[
\texttt{(cond}
\begin{array}{c|c}
\text{condition-1} & \text{result-1} \\
\text{condition-2} & \text{result-2} \\
\vdots & \\
\text{condition-n} & \text{result-n}
\end{array}
\text{\texttt{)}}
\]

2.2 and / or expressions

2.3 if expressions

3 An example for short-circuiting

4 A Recipe for Design

4.1 The Design Recipe

5 User-Defined Procedure Composition

6 Summary
is evaluated by first examining the first condition-result clause. We start by evaluating the condition -1, which must evaluate to a boolean or it’s an error (as it is in all subsequent cases as well). If it’s true, the value of the cond-expression is the value of the expression result-1. If it’s false, we move on to the next clause, and result1 never gets evaluated. We do the same thing for condition2: if it is true, the value of the cond-expression is the value of result-2, etc. Note that a cond-expression must contain at least one condition-result clause, and that the last condition may be not a boolean, but the keyword else; if we reach the last clause, and the condition else, then the value of the cond-expression is the value of result-n. If none of the conditions evaluate to true, and the last one is not an else, it’s an error and processing stops. Only the result for which the condition was true gets evaluated, none of the others do.

Here is an example of a procedure that uses cond:

```scheme
;; data definitions
;; example data:
;; string:"A", "G"
;; complementary-base: string -> string
;; 3. call structure
;; 4. specification
;; Inputs: base, a string, representing a DNA base, that can be "A", "C", "T", or "G"
;; Outputs: a string, representing a the complementary base, where "A", and "T" are complements and, "C" and "G" are complements.
;;
(define (complementary-base base)
  (cond
    [(string=? base "A") "T"]
    [(string=? base "T") "A"]
    [(string=? base "G") "C"]
    [(string=? base "C") "G"]))
;; test-cases
(check-expect (complementary-base "A") "T")
(check-expect (complementary-base "T") "A")
(check-expect (complementary-base "C") "G")
(check-expect (complementary-base "G") "C")
```

Should the conditions in cond cover everything in the domain? Firstly, it is important to keep in mind that the domain is not only defined in the signature but also in the input specification, where it may be further restricted. In the above example, the signature says the domain is of type string but the procedure can in fact take in only four specific stings- “A”, “C”, “T”, or “G”, as stated in the input specification. Now the answer to the original question is yes, you have to cover every possible input in the domain because otherwise, you’ve written a procedure different from what you’ve described in the design recipe. We will go even further and say that your procedure should handle inputs only in the domain and no more, as that would also mean your procedure is different from the one you’ve described. This means that you shouldn’t have an else case that for example returns an error message for any input not a part of the domain. The reason this is important is that a person using your program enters an invalid input, they will expect the program to halt and generate an error, which it won’t if there is an else case covering parts outside the domain. You are not responsible for how your program behaves on input outside the domain and should not address
errors if they are not a part of the specification.

2.2 and / or expressions

1. An and-expression \((\text{and } \exp_1 \ldots \exp_n)\) is evaluated by evaluating \(\exp_1\), whose value must be a boolean. If it is \textit{false}, the value of the and-expression is \textit{false} and the remaining expressions are never evaluated. If it is \textit{true}, however, we evaluate \(\exp_2\); again, the value must be a boolean or it’s an error. If the resulting value is \textit{false}, the value of the and-expression is \textit{false} and the remaining expressions are never evaluated. We proceed similarly through any remaining expressions. If all expressions evaluate to \textit{true}, then the value of the and-expression is \textit{true}.

2. An or-expression \((\text{or } \exp_1 \ldots \exp_n)\) is evaluated by evaluating \(\exp_1\), whose value must be a boolean. If it is \textit{true}, the value of the or-expression is \textit{true} and the remaining expressions are never evaluated. If it is \textit{false}, however, we evaluate \(\exp_2\); again, the value must be a boolean or it’s an error. If the resulting value is \textit{true}, the value of the or-expression is \textit{true} and the remaining expressions are never evaluated. We proceed similarly through any remaining expressions. If all expressions evaluate to \textit{false}, then the value of the or-expression is \textit{false}.

2.3 if expressions

An if-expression \((\text{if } \text{test-exp } \text{true-exp} \text{ false-exp})\) is evaluated by first evaluating \(\text{test-exp}\) to produce a value \(b\); if the value’s not a boolean, it’s an error. If the value \(b\) is \textit{true}, then the value of the if-expression is the result of evaluating \(\text{true-exp}\), and \textit{false-exp} is never evaluated; if the value of \(b\) is \textit{false} then the value of the if-expression is the result of evaluating \(\text{false-exp}\) and \(\text{true-exp}\) is never evaluated. Also note that while Racket will allow the values of the \(\text{true-exp}\) and \(\text{false-exp}\) to be different (that is to say \(\text{true-exp}\) could evaluate to a number while \(\text{false-exp}\) could evaluate to a string, for example), this is not how we will write code in CS17. The values returned by the evaluation of the \(\text{true-exp}\) and \(\text{false-exp}\) will be of the same type for this course.

The idea that in evaluating each of these four, and, or, if, cond, the rules involve \textit{not} necessarily evaluating all the expressions that appear after the keyword as mentioned before is called “short-circuiting”, and is amazingly helpful in some situations.

Of these four, \textit{cond} is certainly the one you’ll use most in CS17. It’s so important that in our next language, Ocaml, they developed special syntax so that you don’t even have to write a word like “cond” in many cases.

3 An example for short-circuiting

I’d like you to apply the rules of evaluation to a couple of expressions. I first need to mention that \((\text{zero? } x)\) tests whether \(x\) evaluates to 0.

Consider the following program:

\[
(\text{define } x 0) \\
(/ 1 x)
\]
This leads, as expected, to an error because division by zero is undefined.

Now: what happens when we process this program?

```
(define x 0)
(if (zero? x)
  0
  (/ 1 x))
```

Does this lead to an error? Although the `false-expr` part of the `if-expr` is clearly illegal, this procedure returns 0, producing no error. This is because the `test-expr` evaluates to true, as x is indeed equal to zero when checked by the predicate `zero?`. Therefore, the value of the `if-expr` becomes equal to the value of the `true-expr`, which is 0. The `false-expr` is ever evaluated (as indicated by a syntax highlighting in DrRacket) and any potential error in that part of the code is never found. If we had to follow the normal rules of evaluation for a procedure application expression (i.e., if `if` were a procedure rather than a keyword), we’d have to evaluate both the 0 and the `(/ 1 x)` expressions ... but evaluating the second would produce a divide-by-zero error. So the “if-rule” (and the other short-circuiting rules) can be really helpful.

4 A Recipe for Design

I presented a “design recipe” at the end of the last class; today I want to add a few steps that I (deliberately) omitted last time, and apply the recipe to make a couple of procedures. Last time was the design recipe for “atomic data”, which means things like numbers, strings, booleans, procedures. (We haven’t yet seen any non-atomic data!) Today, I’m going to enlarge it:

4.1 The Design Recipe

1. (Soon to change somewhat) Provide data definitions by naming the types of data used as inputs and outputs of your procedure
2. Provide examples of the data the procedure will process and produce.
3. Specify the procedure’s type signature, which describes the type of data the procedure consumes, and the type it produces.
4. Following the type signature, describe the procedure’s call structure, i.e., give names to the procedure and its arguments. (This involves writing the start of a Racket program rather than writing a comment.)
5. Write a specification for the procedure. That is, in words, not code, state the relationship between the procedure’s input and output (make sure to use the argument names you created in the call structure). This goes in a comment above the call structure.
6. Provide test cases that exemplify the procedure’s operation. These tests must follow its call structure and satisfy its specification.
7+8. Code the procedure This step may require you to be inventive, and should be fun, rather than rote.

---

1 In the next few lectures, this step will be broken down into two steps.
9. Run your program on your test cases.

5 User-Defined Procedure Composition

Just as we can compose mathematical functions, writing things like $g(f(3))$, we can compose Racket functions, too. Let’s write two racket functions using the design recipe, and then compose them.

The first will be km-to-miles. This is supposed to convert measurements in km to the corresponding measurement in miles, so let’s start with the recipe:

Step 1 is “data definitions”:

```racket
;;; data definition:
;;; float
```

Note about floats: Floats don’t need a “.” at the end in Racket. You can think of them as being essentially “Racket numbers” which means some internal and rather complicated representation of numbers that Racket has. So the “/” procedure will return this “Racket number” which is for all purposes, a rounded float.

Step 2 is “data examples”:

```racket
;;; data definitions
;;; example data:
;;; float: 16.2, 7
```

Step 3: type-signature

```racket
;;; data definitions
;;; example data:
;;; float: 16.2, 7
;;; km-to-miles: float -> float
```

Step 4: the call structure:

```racket
;;; data definitions
;;; example data:
;;; float: 16.2, 7
;;; km-to-miles: float -> float
;;; (define (km-to-miles dist-in-km)
;;;   ...
;
```

Step 5: Specification:

```racket
;;; data definitions
;;; example data:
;;; float: 16.2, 7
;;; km-to-miles: float -> float
;;;
;; input:
;; dist-in-km, a float indicating a distance measured in kilometers
;; output:
;; the same distance measured in miles
(define (km-to-miles dist-in-km)
  ...)

Step 6: Test cases

;; data definitions
;; example data:
;; float: 16.2, 7
;; km-to-miles: float -> float
;;
;; input:
;; dist-in-km, a float indicating a distance measured in kilometers
;; output:
;; the same distance measured in miles
(define (km-to-miles dist-in-km)
  ...
  (check-expect (km-to-miles 1.6) 1)
  (check-expect (km-to-miles 1) (/ 5 8))
  (check-expect (km-to-miles -1.6) -1)
  (check-expect (km-to-miles 0) 0)

Step 7+8: Code the procedure

;; data definitions
;; example data:
;; float: 16.2, 7
;; km-to-miles: float -> float
;;
;; input:
;; dist-in-km, a float indicating a distance measured in kilometers
;; output:
;; the same distance measured in miles
(define (km-to-miles dist-in-km)
  (* dist-in-km (/ 5 8)))
  (check-expect (km-to-miles 1.6) 1)
  (check-expect (km-to-miles 1) (/ 5 8))
  (check-expect (km-to-miles -1.6) -1)
  (check-expect (km-to-miles 0) 0)

Now at this point, I’m going to pause and say that numbers like 0.625 don’t belong in the middle of programs. They should be defined constants with mnemonic names. Only for this identifier binding, following all of the design recipe’s steps is not necessary; just put in a comment describing what this new identifier along with its constant represent. So I’m going to rewrite:

(define miles-per-km (/ 5 8)) ;; international standard
;; data definitions
;; example data:
;; float: 16.2, 7
Step 9: Run the program. It runs and all tests pass.

Let’s do this whole process again, writing miles-to-feet.

Step 1 is “data definitions”:

```scheme
;; data definitions
;; example data:
;; float: 5
```

Step 2 is “data examples”:

```scheme
;; data definitions
;; example data:
;; float: 5
```

Step 3: type-signature

```scheme
;; data definitions
;; example data:
;; float: 5
;; miles-to-feet: float -> float
```

Step 4: the call structure:

```scheme
;; data definitions
;; example data:
;; float: 5
;; miles-to-feet: float -> float
(define (miles-to-feet dist-in-mi)
  ...
)
```

Step 5: Specification:

```scheme
;; data definitions
;; example data:
;; float: 5
;; miles-to-feet: float -> float
;;
```
;; input:
;; dist-in-mi, a real number indicating a distance measured in miles
;; output:
;; the measure of that distance in feet
(define (miles-to-feet dist-in-mi)
  ...)

Step 6: Test cases

;; data definitions
;; example data:
;; float: 5
;; miles-to-feet: float -> float
;;
;; input:
;; dist-in-mi, a real number indicating a distance measured in miles
;; output:
;; the measure of that distance in feet
(define (miles-to-feet dist-in-mi)
  ...
)(check-expect (miles-to-feet 1) 5280)
(check-expect (miles-to-feet 0) 0)
(check-expect (miles-to-feet 0.25) 1320)
(check-expect (miles-to-feet -1) -5280)

Step 7+8: Code the procedure

;; data definitions
;; example data:
;; float: 5
;; miles-to-feet: float -> float
;;
;; input:
;; dist-in-mi, a real number indicating a distance measured in miles
;; output:
;; the measure of that distance in feet
(define (miles-to-feet dist-in-mi)
  (* dist-in-mi 5280))
(check-expect (miles-to-feet 1) 5280)
(check-expect (miles-to-feet 0) 0)
(check-expect (miles-to-feet 0.25) 1320)
(check-expect (miles-to-feet -1) -5280)

Again, we’ll replace 5280 with a defined constant with a mnemonic name. We’ll rewrite what we have to include this change and our end result is:

(define feet-per-mile 5280) ;; international standard
;; data definitions
;; example data:
;; float: 5
;; miles-to-feet: float -> float
;;
Step 9: Run the program. It runs and all tests pass.

Now let’s compose the two functions we’ve written to define a function \( \text{km-to-feet} \). This is very similar to how, in algebra, you might’ve done things like \( f(g(x)) \). We’ll only show the end result complete with design recipe comments and test cases here:

```
;; data definition
;; example data:
;; float: 5, 5.5
;; km-to-feet: float -> float
;; input:
;; dist-in-km, a real number indicating a distance measured in kilometers
;; output:
;; the measure of that distance in feet
(define (km-to-feet dist-in-km)
  (miles-to-feet (km-to-miles dist-in-km)))
(check-expect (km-to-feet 1.6) 5280)
(check-expect (km-to-feet 0) 0)
(check-expect (km-to-feet 1) 3300)
(check-expect (km-to-feet -1.6) -5280)
```

“Procedure composition” allows us to take a problem and decompose it into smaller, easier problems and then combine them, which is a really useful technique.

You might also be wondering why we allow for negative distances here. This was a design decision made so that we can account for something like traveling away from your destination. As an example, picture that you’re going from Providence to New York City. Going to Boston from Providence first can be thought of as traveling a negative distance, away from your destination of NYC.

## 6 Summary

### Ideas

- We’ve now been introduced to the idea of short-circuiting, and its role in the evaluation of both \( \text{and} \)-expressions and \( \text{or} \)-expressions. Essentially, as soon as \( \text{and} \) encounters a \textbf{false} value, the other arguments passed to it are not evaluated. Similarly, with \( \text{or} \), as soon as a \textbf{true} value is encountered, the other arguments passed to it are not evaluated.
• We have also further expanded our design recipe to include commenting and testing for procedures.

• Lastly, we showed that user defined procedures in Racket can be combined in much the same way functions in algebra are composed together.

Skills

• The Design Recipe includes several steps for writing procedures, and following all of these steps is crucial in understanding exactly what your procedure is doing, and how the procedure runs.

• We are beginning to see how through procedure composition, larger programs that do a variety of things based on a few inputs can be written. As an example, we wrote two procedures, \textit{km-to-miles} and \textit{miles-to-feet}, and composed them to create one function \textit{km-to-feet}.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: \url{http://cs.brown.edu/courses/cs017/feedback}.