Lecture 06: Short-circuiting, More Design Recipe
10:00 AM, Sep 16, 2019

Contents

1 Today’s topics

2 Four new rules of processing

2.1 cond expressions

3 An example for short-circuiting

4 A Recipe for Design

4.1 The Design Recipe

5 Summary

1 Today’s topics

We’ll talk about user-defined procedures and add four more rules of processing: cond, and, or, if.

2 Four new rules of processing

2.1 cond expressions

\[(\text{cond}\n\text{[ condition-1 result-1]}\n\text{[ condition-2 result-2]}\n...\n\text{[ condition-n result-n]}\]

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 in 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 result-1 never gets evaluated. We do the same thing for condition-2: 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
;;(define (complementary-base base) ...)
;; 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", "C" and "G" 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")
```

Note: The example data need not necessarily be the ones used in procedure but for style purposes it is preferable to use examples used in the procedure.

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. 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. In CS17, 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.

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:

\[
\begin{align*}
\text{(define } x \text{ 0)} \\
\text{(/ 1 } x) \\
\end{align*}
\]

This leads, as expected, to an error because division by zero is undefined.

Now: what happens when we process this program?

\[
\begin{align*}
\text{(define } x \text{ 0)} \\
\text{(if (zero? } x) 0 0) \\
\text{(/ 1 } x)) \\
\end{align*}
\]

Does this lead to an error? Although evaluating the false-expr part of the if-expr would lead to an error, 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 \(\text{zero?}\). Therefore, the value of the if-expr becomes equal to the value of the true-expr, which is 0. The false-expr is never evaluated (as indicated by a syntax highlighting in DrRacket) and any potential evaluation 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. **Data Definition** say what kinds of data you’ll be working with. Choices are \(\text{int, bool, string, float}\) (the first being ”integers” and the last being ”any real number at all”). [This rule will soon change slightly.]

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.

9. Run your program on your test cases.

5 Summary

You’ve learned how to evaluate a procedure-application expression that involves a user-defined procedure by temporarily enlarging the environment and then evaluating the body in this enlarged environment. (We’ll be slightly adjusting this rule in a few more classes.)

Ideas

- We have 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, km-to-miles and miles-to-feet, and composed them to create one function km-to-feet.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci10170/feedback](http://cs.brown.edu/courses/csci10170/feedback)

1In the next few lectures, this step will be broken down into two steps.