1 Predicates

Predicates are a special type of function in Racket. They compare two arguments in some way, and return the result of that comparison as a boolean, or a true/false value. For instance:

\[
\begin{align*}
(> 3 5) &; \text{tests whether 3 is greater than 5} \\
(= 4 7) &; \text{tests equality of numbers} \\
(string=? "me" "you") &; \text{tests equality of strings}
\end{align*}
\]

There are many predicates built in to Racket that you can use, but you can also write your own! We’ll post a list of built-ins that you’re allowed to use on Piazza in the near future.

2 User-Defined Procedures

Last class, we discussed how you can define your own procedures, by writing something like this:

\[
(\text{define} \ (\text{next-num} \ n) \ (+ \ n \ 1))
\]

or more generally:

\[
(\text{define} \ (<\text{id}> <\text{id}> \ldots <\text{id}>) \ <\exp>)
\]

We’ll name the parts of such an expression like this:
(define (name arg1 ... argn) body)

where name and all the args are identifiers, and body is an expression.
Note the ( before name: this parenthesis is the marker for a user-defined procedure— it’s how you know it’s not a user-defined identifier.

2.1 Processing a User-Defined Procedure

The steps in processing a function-definition are:

1. Check that the procedure name is not already already bound in the TLE (top-level environment). If the name is already bound, an error occurs and the program halts.

2. Create a new closure, which is our name for an internal representation of a user-defined function. The closure consists of:
   - A finite sequence of “formal arguments” (namely, the identifiers arg1 ... argn)
   - A body (namely, the expression we’ve called body)

3. Bind the procedure name to this closure in the top-level environment. Note: The closure stores the body, not the result of evaluating the body.

2.2 Evaluating User-Defined Procedures

We’ve seen procedure-application-expressions, but now they come in two flavors: ones where the “proc” expression evaluates to a built-in procedure, and ones where it evaluates to a user-defined procedure (which we’re calling a closure).

The rules, in this second case, are pretty similar to the rules in the first case, with an important variation at the end:

User-Defined Procedure Application Expression:

(<proc> <arg1> ... <argn>)

1. Evaluate proc to get a closure, c
2. Evaluate arg1, ..., argn (called actual arguments) to get values v1 ... vn
3. Create a new environment that consists of the top-level environment with n new bindings added: each formal argument (from the closure’s formal argument list) is bound to the corresponding actual argument.
4. Evaluate the body of the procedure (the “body” part of the closure) in this new environment to get a value, val, which is the value of the user-defined-procedure expression.
5. Remove the bindings from step 1
6. The value of the procedure application expression is $val$

I like to think of an environment as a list of bindings written on a piece of paper, identifiers on the left, values on the right. I think of sticking on a few new identifiers as “taping a file card to the bottom of the piece of paper for a moment” and writing the bindings of formal arguments (“formals”) to actual arguments (“actuals”) and then doing the evaluations.

**Question:** Why can you write the procedure name with no arguments in Racket, without an error occurring?

**Answer:** The procedure name is a name, which is a type of expression in Racket. And, expressions are valid Racket programs! It may not make much sense right now to access the procedure and not apply it, but it’s possible.

**Question:** What if one of the formals is an identifier that’s already bound in the top-level environment?

**Answer:** A revised definition of “looking up” bindings: we’ll work bottom to top, so that any new binding, like the one on the note-card, “hides” any prior binding.

**Question:** Does this mean we can redefine things?

**Answer:** Nope. A definition results in the insertion of a binding in the top-level environment, not in this new augmented (or extended) environment. We still can define an identifier only once. On the other hand, it’s possible that during evaluation of a user-defined-procedure application, a single identifier might have multiple bindings in some augmented environment, in which case we use the binding that comes last.

All this stuff is just a way to make rules that mimic what you remember from math class. If we have

\[
\begin{align*}
f : \mathbb{N} &\to \mathbb{N} : n \mapsto n + 2 \\
g : \mathbb{N} &\to \mathbb{N} : n \mapsto n^2
\end{align*}
\]

then when we write $g(f(3))$, we first say “OK, $n$ is three inside $f$, so $f(3)$ is $3 + 2$ which is 5. And now $n$ is five inside $g$, so $g(5)$ is $5^2$ which is 25.”

### 3 A Recipe for Design

Today I am going to present a recipe that you will use in CS17 to guide you in the design of simple programs that operate on atomic data (ints, floats, booleans, and strings). Next time, I will add additional steps to the recipe; these will aid you in the design of more complex programs—programs that operate on *compound data*. Following this design recipe is not “Racket law,” but rather “house rules” in CS17.

While this recipe is valid regardless of programming language, it is applicable only after a programming language has been selected. After learning several languages, you will find that some are better suited to certain tasks than others, and that it is necessary when first presented with a problem to

\[\text{In CS17, we pretend strings are atomic.}\]

\[\text{The CS17 design recipes are adaptations of similar recipes that appear in a textbook by Professor Krishnamurthi (and others): How to Design Programs—www.htdp.org}\]
start by selecting an appropriate language\footnote{Often, this requires that you learn a new language!}. In CS17, however, we will always make the choice of language for you.

Many of the steps in the design recipe are mechanical. This is intended to eliminate the “blank page” phenomenon, whereby students do not know where to begin.

### 3.1 The Design Recipe for Atomic Data

1. **Data Definition**: say what kinds of data you’ll be working with. Choices are `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.

3. Specify the procedure’s **type signature**, which describes the (atomic) type of data the procedure consumes, and the (atomic) type it produces.

4. Following the type signature, describe the procedure’s **call structure**, i.e., define the procedure in Racket and give names to the procedure and its arguments, but don’t fill in anything else yet.

5. Write an **input-output specification** for the procedure. That is, *in words*, not code, state the relationship between the procedure’s input and output. As you describe each input argument, you may also possibly constrain the domain of those arguments (e.g., ‘positive integers’ instead of ‘all integers’).

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.

**Question**: Why write test cases before you write the code for the procedure?

**Answer**: It makes sure that you understand the procedure before you write the code for it. In general, when you’re writing a function, you should know what the output would be for a given input. Writing test cases makes you think about how you want your function to work, which solidifies your understanding of it.

### 3.2 Example

Prof. Hughes did a complete worked example with the “count-posts” procedure, starting at slide 15 in the powerpoint slides.

\footnote{In the next few lectures, this step will be broken down into two steps.}
4 Summary

You’ve learned about predicates, a type of built-in function that compares its arguments and outputs a boolean.

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.)

And then you’ve learned the design recipe (or at least one instance of it), a pattern that you’ll be following over and over throughout the whole course. Part of that recipe was a standard format in which to write a “type signature” which tells what sort of data comes into and out of a procedure.

And, if you look at the slides for this lecture, you’ve seen this recipe applied to write and test a post-counting procedure.

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: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)