1 Quick review

We’ve seen rules of processing and rules of evaluation. At the end of last class, we looked at something like \((+ 3 4)\) and evaluated it using the rule for procedure-application expressions, which is, in brief: evaluate the first expression in the parentheses; if that value’s not a procedure, it’s an error. If it is a (builtin) procedure, then evaluate all the other expressions in order to get a collection of values (the arguments), and apply the procedure to the arguments to get a resulting value, which is the value of the procedure-application expression.

The value turned out to be 7. Notice that instead of writing 3 + 4 as we might in math class, in Racket we write \((+ 3 4)\). In the former, the operation “+” is in between the two operands, and the notation is called “infix”. In the latter, the operator precedes the two operands, and it’s called “prefix notation”. On the HW for this week, you’ll be converting a bunch of things from infix to prefix.

Why use prefix? It’s unambiguous. With infix notation, you look at \(3 + 2 * 7\) and you have to remember that “multiplication comes before addition” and turn this into \(3 + 14\) and then into 17. In prefix notation, the order is unambiguous.
Let’s look at \((+ 3 \, (* \, 2 \, 7))\).

First: let’s identify everything that’s an expression in there. There are quite a lot of them. The entire thing is an expression, the \(+\) and \(*\) are expressions, the numbers are all expressions, and \((* \, 2 \, 7)\) is an expression too. In the course of evaluating the outer expression, we’ll have to evaluate all the inner ones, too.

**Question:** Are the inner parentheses necessary in the above expression?

**Answer:** Yes! Say we tried to evaluate \((+ 4 \, (* \, 3 \, 2))\). We would evaluate the \(+\) and see it is a procedure. Then, we would evaluate all the other expressions. The \(*\) would evaluate to the multiplication built-in. After all expressions have been evaluated, we would then apply the addition procedure to the values. We would get an error as we could not apply the addition procedure to the multiplication procedure.

To evaluate this, we run through the rules of evaluation, which I’m going to do in shorthand:

It’s a proc-app expression; the first expression evaluates to the built-in addition proc, so we can proceed.

The second expression evaluates to 3, by the usual rules. So our current state is this:

\[
\begin{align*}
& (+ \quad 3 \quad (* \quad 2 \quad 7) \quad ) \\
- & \quad \text{addn-proc} \quad 3
\end{align*}
\]

Now we have to evaluate the third expression, and get a value for it, so that we can apply the addition procedure to these two values.

That third expression is a proc-app expression, so we evaluate its first expression, \(*\). The value, from lookup in the top-level environment, is the builtin multiplication procedure. Since that’s a proc, we can proceed. So our current state looks like this:

\[
\begin{align*}
& (+ \quad 3 \quad (* \quad 2 \quad 7) \quad ) \\
- & \quad \text{addn-proc} \quad 3
\end{align*}
\]

\[
\begin{align*}
- & \quad \text{mult-proc}
\end{align*}
\]

We can now finish up evaluating that inner expression by applying the mult-proc to the arguments 2 and 7 to get 14:
And now that we have values for all three expressions in the outer proc-app expression, we can apply the addition procedure to the values 3 and 14 to get 17:

\[
(+ \quad 3 \quad (* \quad 2 \quad 7) \quad )
\]

\[
\text{addn-proc} \ 3
\]

\[
\text{mult-proc} \ 2 \ 7
\]

\[
14
\]

\[
17
\]

Whew! It’ll seem easier after a while.

2 Foreshadowing

Today we’re going to learn a new data type (boolean), some boolean expressions, which have their own evaluation rules, how you can create your own procedures, and then how we will always do so in CS17. This last item, the “design recipe,” is incredibly useful. It gives you something to do when you don’t know how to approach a problem, helping you make the first step. And for most of the problems in CS17, that first step is a big one — the problems are crafted, in part, to make the design recipe work well for them, especially early on. But later problems are also amenable to it, not because we made them that way, but because the recipe is a really valuable way to organize your thinking about any computational problem. When you get good at it, it’ll start to become second nature to see how it might apply to a completely different problem (including “writing an essay for your history class”).

3 Booleans

George Boole invented what has come to be known as Boolean Algebra (named by others, not him), he called the system “The Laws of Thought.” Modest guy, huh?

Booleans are a new kind of value—they’re like numbers and strings, except they’re really simple: there are two of them and their names are true and false. Booleans are used in Racket to express
the idea of selecting from various options. Note, they are keywords and, as such, cannot be used as identifiers.

3.1 Useful Operations on Booleans

[Covered in next class]

3.2 A Few More Built-ins

- $(>\text{ num1 num2})$: Produces $\text{true}$ if $\text{num1}$ is larger than $\text{num2}$, $\text{false}$ otherwise. Produces an error if either argument is not a number. Similarly, you can use $(<\text{ num1 num2})$.
- $(\geq\text{ num1 num2})$: Produces $\text{true}$ if $\text{num1}$ is larger than or equal to $\text{num2}$, $\text{false}$ otherwise. Produces an error if either argument is not a number. Similarly, you can use $(\leq\text{ num1 num2})$.
- $(=\text{ num1 num2})$: Produces $\text{true}$ if $\text{num1}$ is equal to $\text{num2}$, $\text{false}$ otherwise. Produces an error if either argument is not a number.
- $(\text{string=}\text{? str1 str1})$: Produces $\text{true}$ if $\text{str1}$ and $\text{str2}$ are identical strings, $\text{false}$ otherwise. Produces an error if either argument is not a string.

While I’m mentioning further built-ins, there are things like

- $(\log\text{ num})$: Produces the logarithm of the positive real number $\text{num}$. Produces an error if $\text{num} \leq 0$.
- $(\text{expt a b})$: Produces $a^b$ for numbers $a$ and $b$. For negative values of $a$, may produce complex number results (which will never matter in CS17).

and just about any other mathematical function you can imagine, although the names will not always be obvious. We’ll use hardly any of these in CS17, but they’re there.

**Question:** Is $(\text{and true false})$ a procedure-application expression?

**Answer:** No, it is an and-expression. The first item within the parentheses of a procedure-application expression must be an identifier that evaluates to a procedure. But $\text{and}$ is not an identifier, as it is a keyword, and keywords cannot be identifiers.

4 User-Defined Procedures

We’ve seen built-in procedures like $+$ and $\geq$ and $\text{not}$.

You can also define your own procedures, by writing something like this:

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

or more generally:
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.

The steps in processing a function-definition are:

1. 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)

2. 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.

### 4.1 Evaluating user-defined functions

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, which is the value of the user-defined-procedure expression.
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:** 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.

**Question:** What happens to the new environments after a user-defined procedure has been evaluated?

**Answer:** New environments disappear after we’re done using them (think of taking off the file card and throwing it away) and we’re left only with the top-level environment (we keep the main piece of paper).

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

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

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.”

5 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 (numbers, booleans, and strings\(^1\)). 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\(^2\).

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 start by selecting an appropriate language\(^3\). In CS17, however, we will always make the choice of language for you.

\(^1\)In CS17, we pretend strings are atomic.

\(^2\)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.

\(^3\)Often, this requires that you learn a *new* language!
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.

5.1 The Design Recipe for Atomic Data

1. Data Definition: say what kinds of data you’ll be working with. Choices are int, bool, string, real (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., give names to the procedure and its arguments.

5. Write a 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.

6+7. 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.

5.2 Example

Prof. Hughes did a complete worked example with the “next-num” procedure, shown in the powerpoint slides.

6 Summary

You’ve learned a new datatype — boolean — and some associated expressions, namely and-expressions and or-expressions.

You’ve learned how to define a new procedure, and how to evaluate a procedure-application that involves one of these user-defined procedures 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.

---

4In the next few lectures, this step will be broken down into two steps.
And you’ve seen this recipe applied to write a future-value procedure, and to test it.

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)