Mosaic+ Mixer
9/13 @4:30-6pm
Location: Front Lawn of Barus&Holley

Email us to join the listserv
mosaic.plus.brown@gmail.com
Hack@Brown 2020

Come to our info session:
https://tinyurl.com/info2020

And apply here:
https://tinyurl.com/hackatbrown2020
Due 9/13
Functions and procedures
Rules of Processing
Problem statement (short form)
;; Data Definition
;; Example data:
;; num: 0, -6, 41, 7.2
;;
;; count-posts: num * num -> num
;; inputs:
;; width, an integer, the number of posts along one
;; side of the property, at least 2.
;; depth, an integer, the number of posts along the perpendicular
;; side of the property, also at least 2.
;; output: the total number of posts needed to fence in the
;; property, an integer.
(define (count-posts width-count depth-count)
  (- (+ (* 2 width-count) (* 2 depth-count)) 4))
;; test cases for count-posts
(check-expect (count-posts 2 2) 4)
(check-expect (count-posts 2 5) 10)
(check-expect (count-posts 7 2) 14)
(check-expect (count-posts 5 8) 22)
;; Data Definition
;; Example data:
;; num: 0, -6, 41, 7.2
;;
;; count-posts: num * num -> num
;; inputs:
;; width, an integer, the number of posts along one
;; side of the property, at least 2.
;; depth, an integer, the number of posts along the perpendicular
;; side of the property, also at least 2.
;; output: the total number of posts needed to fence in the
;; property, an integer.
(define (count-posts width-count depth-count)
  (- (* 2 (+ width-count depth-count)) 4))
;; test cases for count-posts
(check-expect (count-posts 2 2) 4)
(check-expect (count-posts 2 5) 10)
(check-expect (count-posts 7 2) 14)
(check-expect (count-posts 5 8) 22)
Named sets of numbers
Miscellany

• Office hours
Functions that produce booleans

• These are called *predicates*
• Names often end with a question-mark
  (zero? 12) => false
  (string? "hello") => true
  (string=? "abc" "def") => false
• Lots more, introduced as needed
Comparing sets
What does all that stuff in the “header” mean?

;; count-posts: num * num -> num
• Tells the *name* of the procedure, and its “type”, using Racket’s limited type-descriptions
"Header" stuff (2)

;; inputs:
;;   width, an integer, the number of posts along one
;;   side of the property, at least 2.
;;   depth, an integer, the number of posts along the perpendicular
;;   side of the property, also at least 2.

• Restricts the domain from “num” to something much smaller.
• Makes a contract with the user of this program: if you give me integers that are 2 or more, then I’ll correctly compute the number of fenceposts needed.
  • THIS IS THE ONLY PROMISE MADE, and the only one a user of the program (including its author) should rely on.
  • If you give me other data, I may do anything I like, and it will not violate my contractual obligation.
  • The tests/examples we use must be within-contract.
  • No part of your program may rely on any out-of-contract performance of the program.
What does all that stuff in the “header” mean?

;; output: the total number of posts needed to fence in the
”; property, an integer.
• Again, a restriction; this time we restrict the codomain.
• A final bit of the contract: I promise the output will not be just
  any number, but an integer.
• (from popsugar.com)
Understanding a program

- Definitions mostly make sense, right?
- But what’s *really* going on there?
- I can’t tell you that, because I didn’t write DrRacket
- But I can give you an explanation that correctly predicts what’ll happen, and that’s just as good
- I’ll refer to this as an “abstract machine”
  - “Abstract” in the sense that it’s imaginary, but captures all of the important parts of how Racket works
  - “Machine” in the sense that it’s mechanistic - it’s a fixed set of rules that describes how programs are processed
Abstract machines (2)

• I’m going to phrase things in the first person, because it’s shorter to say “I read the name” than “Racket reads the name”
  • I’ll be describing physical actions I take. Easier to imagine me writing on a pad of paper than Racket doing so.

• Recall the first-day expectations
  • A computer can associate a name to something else (MyEssay.docx is associated to a Word document, for instance)
  • A computer can somehow do arithmetic computations
  • A computer can split apart text into logical bits (e.g. words, or now, tokens)
A working BNF for CS17 racket, for now

<prog> ::= <defn>* [<top-level-expr>]
<defn> ::= <name-def> | <proc-def>
'name-def' ::= (define <name> <expr>)
<proc-def> ::= (define (<proc-name> <arg>* ) <body>)
<proc-name> ::= <name>
<arg> ::= <name>
<body> ::= <expr>
<top-level-expr> ::= <expr>
<name> ::= <CS17name> | <othername>
<CS17name> ::= sequence of letters, digits, hyphens,
    starting with a letter, usually lowercase
<othername> ::= token consisting of non-special characters
    that can’t be interpreted as a number and
    that isn’t a keyword
<expr> ::= next page!
A working BNF for CS17 racket (2)

<expr> := <name> | <number> | <boolean> | <string>
   | <proc-app-expr>
<number> := *any thing that looks like a number*
<boolean> := true | false
<string> := "*any characters except double-quotes*
<proc-app-expr> := (<name> <expr>*)
Example

\[
\text{<proc-app-expr> := ( <name> <expr>* )}
\]

- Now we know what

\(( + 3 4 )\)

is

- It’s a “procedure application expression”
- The + is a name; 3 and 4 are expressions.
- What kind of expressions are they?

\[
\text{<expr> := <name> | <number> | <boolean> | <string> | <proc-app-expr>}
\]
Rules of processing, version 1
Details before we proceed (1)

- When an error arises, I print an error message and halt processing.
- I described the TLE as a sheet of paper, but in fact it does not start out as a *blank* sheet
  - Details to follow
- I’ve said how to process a name-definition
  - Still need to describe how to process a procedure-definition
  - Still need to describe how to process an expression
Activity

Assuming the TLE starts out empty, what’s it look like after processing of the following?

(define a 3)
(define b true)

[For the moment, assume that 3 evaluates to a number-value 3, and true evaluates to a Boolean-value “true”]
The word “value” means “something that I use to represent meaningful stuff.”

- I might choose to represent the number we call “three” by putting three dots on my piece of paper.
- I might choose to represent it as a binary number “11”
- I might represent it as a piece of text: “three”
- ...

All that matters is that it’s known to be a *number*, and the representation is unique (no other number has the same representation)

- The same thing goes for booleans and strings

To distinguish it from a piece of program text that looks like a number, I’ll always refer to this as a “number-value”
Details before we proceed (3)

• We’ll need a representation for procedures as well – details soon!

• The remaining undefined word is “evaluate”
  • “To draw a value from”
  • Evaluation turns expressions into values
  • Near-truth: Evaluation is a function: it consumes an expression, and produces a value
    • ....almost
  • Truth: Evaluation consumes an expression and an environment, and produces a value.
    • Recall an environment is a list of name-value pairs written on paper (for now)
Read-Eval-Print Loop (REPL)

This picture misses out on the processing of definitions.
Processing a procedure definition

• When I encounter a procedure definition of the form
  \((\text{define} \ (<\text{proc-name}> \ <\text{arg}*>) \ <\text{body}>)\)
  I enter the “proc-name” in the left-hand column of the TLE, and on the corresponding point in the right-hand side, I create a “closure”, my representation of a proc
  • The closure is a rectangular box containing
    • the list of argument names
    • the body
  • Example: processing \((\text{define} \ (f \ x \ (+ \ x \ 1)))\) adds a new line to the environment:

\[
\begin{array}{ccc}
  f & x \\
  (+ x 1)
\end{array}
\]
Rules, continued
Printed representations of values

• Printed representations are sequence of characters
• They tend to be fairly clear representations of the thing
• Example: if I have a number-value, representing the number we call ‘three’, the printed representation is 3
• If I have a boolean-value representing true, the printed rep is true
• If I have a procedure ...
  • the printed representation might be something like <user-defined-proc>, or <proc:0x823> or <function> or <built-in: +> or ...
  • ...because we humans haven’t got an agreed-on notation for procedures
Rules of evaluation (the essence of Racket)

• Tell us how to go **from** pieces of program text that are expressions **to** values.
• There are about 7 rules. We’ll see a few today.
• Recall BNF for expressions:
  \[
  \begin{align*}
  <\text{expr}> & : = <\text{name}> | <\text{number}> | <\text{boolean}> | <\text{string}> \\
  & \quad | <\text{proc-app-expr}>
  \\
  <\text{number}> & : = \text{anything that looks like a number}
  \\
  <\text{boolean}> & : = \text{true} | \text{false}
  \\
  <\text{string}> & : = "\text{any characters except double-quotes}"
  \\
  <\text{proc-app-expr}> & : = (<\text{name}> <\text{expr}>*)
  \end{align*}
  \]
Rules of Evaluation
Informal version of these first four rules

- Evaluating a number produces that number
- Evaluating a bool produces that bool
- ...

What’s misleading is that “number”, on the left, is a textual description of the number, a sequence of characters in a file “that number”, on the right, is a computer-based representation of the number, something that the computer can use in addition, subtraction, etc.

... but it sure is nice and short.
The last rule of evaluation (for now)

(define (f x) x)
(f 4)
The goal of the proc-app-expr rule
A new bit of syntax: cond
Cond-expression structure
Cond example

(define x 5)

(cond
  ((negative? x) "negative")
  ((positive? x) "positive")
  ((zero? x) "zero"))

The first test produces “false”; we move on.
The second test produces “true”; we evaluate the result to get “positive”

If we’d defined x to be 0, we’d have gotten “zero”.
Base-pairs

• In biology, there are things called “bases” and named A, T, C, G.
• They come in “complementary pairs”: A and T are complements, C and G are complements.
• We’ll represent these with strings, “A”, ”T”, “C”, “G”.
• Write a procedure “complement” that consumes a base and produces the complementary base.
;; Data definition
;; string: "abs", "T"
;;
;; complement: string -> string
;;
;; input:
;; base a string, which must be one of "A", "T", "C", "G"
;;
;; output:
;; the complementary base, one of "A", "T", "C", "G"
(define (complement base)
  (cond
    ((string=? base "A") "T")
    ((string=? base "T") "A")
    ((string=? base "C") "G")
    ((string=? base "G") "C"))
)

(check-expect (complement "A") "T")
(check-expect (complement "T") "A")
(check-expect (complement "C") "G")
(check-expect (complement "G") "C")