CS 42—Racket functions, programs, testing, and lists

Thursday, October 4, 2018

Summary

Today, we will see some features of Racket that let us write modular, robust code: functions, programs, and tests. We will also introduce Racket lists: their structure and the operations we can use to create them and use them. Racket lists are an example of an *inductive data structure*, which we can process using *recursive algorithms*.

Terminology

unit test: a way to test small piece of code, e.g., a single function, by describing a particular input and the corresponding expected behavior (i.e., the expected output or a particular kind of error).

test-driven development (TDD): a software-engineering practice where the programmer writes small pieces of a program at a time and where the programmer writes tests first, before writing code. The programmer repeats this process as the code grows and changes, always making sure that the tests pass before moving on to the next feature.

idiom: in programming, a commonly used way of expressing an idea. When a programming language gives us multiple ways to say the same thing, a programming community often adopts an idiom—an agreed-upon, single way to say that thing—to promote consistency in code.

In the second half of today's class, we'll be talking about ***recursion*** *and* ***induction****. Generally, we'll use recursion to mean "break things down into smaller pieces" and induction to mean "build things up from smaller pieces".*

recursive algorithm: a procedure for solving a larger instance of a problem by (1) solving one or more smaller, similar instances of the problem (called subproblems), then (2) using the solutions to the subproblems—plus some basic operations—to solve the original instance of the problem.

base cases: the parts of a recursive algorithm that compute the solutions to the smallest instances of a problem (for example, 0! = 1).

recursive cases: the parts of a recursive algorithm that create subproblems, then re-invoke the algorithm to solve the subproblem, then use the result to solve the original problem (for example, 4! = 4 * 3!).

inductive data structure: a data structure that defines (a) the smallest instance(s) of the structure and (b) a set of operations for building bigger instances of the structure from smaller instances.

linked list: an inductive data structure where the smallest list is the empty list. Given an existing list and a new element that we want to add to the list, we create a new, bigger list by prepending the element to the list (i.e., we add it to the beginning of the list).

head: (of a non-empty linked list) the first element of the list.

tail: (of a non-empty linked list) everything after the first element of the list.

pair *or* **cons-cell**: in Racket, a data structure used to implement one "link" in a list. A linked-list pair has two values: an element and a link to the tail of the list for which that element is the head.

Recap: functional programming in Racket

Global constants

We use `define` to "bind" a global name to a value in our program (we won't use this very much).

```
(define name expr)
```

"Variables"

They're called variables, but we won't vary them (i.e., their values are constant).

```
(let* ([var₁ expr₁]
       …
       [varₙ exprₙ])
  body-expr)
```

Conditionals

If you have exactly one condition, use `if`; otherwise, use `cond` (which is like if / else if / else).

```
(if conditional-expr
    true-expr
    false-expr)
```

```
(cond [condition₁ expr₁]
       …
      [conditionₙ exprₙ]
      [else else-expr])
```

Comments

Comments are any text that appears after a semi-colon (`;`), until the end of a line.

Functions

```
(define (function-name parameter₁ … parameterₙ)
   body-expr)
```

Functions should have a **header comment** that documents their inputs and output:

```
;; fact: computes n!
;;   inputs: n, a non-negative integer
;;   outputs: n!
(define (fact N)
  (if (= N 0)
      1
      (* N (fact (- N 1)))))
```

Programs

We use the definition pane of Dr. Racket to write programs. The first line of our program should be:

```
#lang racket
```

Imports and exports

If we want to use code that someone else has written (e.g., a library), we use the `require` function:

```
(require rackunit)   ; gives us access to the testing library
(require "fact.rkt") ; gives us access to the "fact" file's code
```

If we want to make a function available for someone else's code, we use the `provide` function:

```
(provide fact) ; this line "exports" the fact function
```

Writing and running unit tests in Racket

We'll use the `rackunit` library to write tests in Racket. Here's an example:

```
(check-equal? (fact 0) 1)
(check-equal? (fact 1) 1)
(check-equal? (fact 3) 6)
(check-equal? (fact 4) 24)
(check-equal? (fact 5) 120)
```

For more information about the kinds of tests you can write, see the rackunit documentation. The checks that we'll use most often are: `check-equal?`, `check-not-equal?`, `check-true`, and `check-false`.

To run the tests, just run the file that contains them. DrRacket will display error messages for any tests that fail. If all the tests pass, there will be no output.

It's good practice to separate our tests from our code.

In general, we should write our tests in a different file from our code. That way, we can run the code without seeing test output. To keep our code and tests separate, we `provide` all the functions we want to test, and we write a file with only tests that `requires` the testing library (e.g., `rackunit`) and the file or files that contain our code.

Note: *for this week's assignment, you don't need to keep the tests separate from the code; but you should still `provide` all the functions that need testing, so the autograder can access them. See the assignment for more details.*

Lists in Racket

Racket lists are linked lists, where each "link" in the list is a pair of element + list.

list-building operations	**Meaning**
`empty` *or* `'()`	constructs an empty list
`(cons <value> <list>)`	constructs a new list by prepending an element to an existing list (**Careful!** If the second element is not a list, then `cons` does not construct a new list.)
`(list <value1> ... <valueN>)`	constructs a list with the given arguments as elements
`'(<value1> ... <valueN>)`	
`(append <list1> ... <listN>)`	append multiple lists into a single list

list-accessing operations	**Meaning**
`(empty? <value>)`	returns true if the argument is an empty list
`(first <list>)`	returns the head of a non-empty list
`(rest <list>)`	returns the tail of a non-empty list

For more list operations, see https://docs.racket-lang.org/reference/pairs.html.

Next time: Functions as data!