

Racket Notes

Mark Engelberg

November 6, 2010

1 Which language?

I recommend using the full Racket language rather than any of the student languages for the contest. That way you have the full power of all the libraries available to you. To do this, select from the language menu “Determine language from source”. When you do this, any new file you create will automatically have the incantation

```
#lang racket
```

placed at the top of the file. This says that the file uses the full standard racket language. If you are editing a pre-existing file that was created in one of the student languages, you will need to add this language directive to the top of the file manually.

2 Getting more information

The Racket Help Desk is very good. The Help Desk is divided into two main sections, the Guide and the Reference. For the most part, you should stick to the Guide. It is a very readable overview of all the language features that you could even just read from beginning to end to get a full sense of what features are built-in to the full Racket language. The Reference provides an extra level of detail which is sometimes useful.

3 Prerequisites

To have any degree of success using Racket in a contest, students should already be comfortable working with numbers, strings, and lists (including lists of lists). A knowledge of higher-order functions (especially *map* and *filter*) can make programs significantly more concise.

Students will need to learn several built-in functions to handle input and output. These functions are not especially complicated, but they are fundamentally different from the kinds of pure functions your students are used to seeing in Racket. Input/output functions are useful for their side effects and

not necessarily for their return values, so using them effectively requires a few new techniques your students may not have seen.

First, they will need to understand how to create local definitions to bind the results of the input functions to local variables. Beginner students can do this using the *local...define* syntax. Intermediate students may want to learn that in certain contexts, you can drop the *local* keyword for brevity. Advanced students may want to learn the full distinction between **let**, **let***, and **letrec** and use these instead of *local...define* when appropriate.

Second, they will need to know how to sequence side-effecting functions using **begin**.

Processing lists for side effects is often more concise using Racket's iterative syntax (aka for-loops). For intermediate and advanced students, the guide section on iterations and comprehensions is recommended.

4 Reading standard input

For the purposes of the contest, the most useful function for reading from standard input is (*read-line*). When executed in DrRacket, this opens up a little prompt for the user to type some input, up until the point where the user hits the enter key, and it returns this inputted text as a string. You can explicitly convert this string to a number using *string->number*. You should also be aware that you can convert a string to a list of characters using *string->list*.

5 Printing to output

The simplest way to display output is to use the *display* function. This function intelligently handles both strings and numbers. (For advanced students, what is the difference between display and write?) As with many other languages, a `\n` character inside a string indicates a newline. Alternatively, you can use the no-argument function (*newline*) to print a newline character.

The most generally useful way to print output is *printf*. For contest purposes, the main placeholder you need to know about is `~a` which can handle both numbers and strings. So

```
(printf "~a's age is ~a\n" "Vonda" 15)
```

will print

```
Vonda's age is 15
```

For intermediate and advanced students, compare and contrast *printf* with *format*.

6 Java computation

Since the contest is written for Java programmers, and the answers are judged against the answers as produced by a Java program, it is important to under-

stand the ways in which Racket arithmetic differs from Java arithmetic. The most important difference is that Racket follows the IEEE standard for rounding floating point numbers (round to nearest integer; round ties to the nearest even integer). For some reason, Java does not follow the standard, and always rounds up in the event of a tie. [Discussion point — Java’s approach most closely matches the rounding method we usually learn in math class. Why does the IEEE standard for rounding floating point numbers specify the rounding of ties to even numbers?]

Unfortunately, this means rolling your own rounding algorithm to make sure you get the same output in Racket that Java would produce.

It is recommended that students memorize how to write the following functions, or an equivalent variation (for example, you could replace the internal definition of `10p` with `local`, `let`, or whatever local binding technique you plan to show your students):

```
(define (round n)
  (floor (+ n .5)))

; format-decimal produces a string from n, with precision p.
; Example: (format-decimal 2.3546 2) -> "2.35"
```

```
(define (format-decimal n p)
  (define 10p (expt 10 p))
  (real->decimal-string (/ (round (* n 10p)) 10p) p))
```

This isn’t guaranteed to be a perfect emulation of Java’s mechanism for rounding and printing floating point numbers, but it works well.

7 File Input

Batch processing input from a file is one of the trickier skills that needs to be learned for the contest.

7.1 Overall strategy

The input file is often presented as a series of lines, one “dataset” per line. Typically, the first line of file is the number of datasets contained in the file. In Racket, the dataset count is usually irrelevant – since lists can hold an arbitrary number of elements, there is little value to knowing how many datasets are in the file.

For the purposes of this section, let’s imagine you’re solving a problem in which each dataset is a name and an age, and the goal is to print a series of lines that say things like “Vonda’s age is 15.”

So, for example, the file might be:

```
4
Vonda 15
```

```
Agatha 14
Molly 11
Sam 16
```

and the output would be:

```
Vonda's age is 15
Agatha's age is 14
Molly's age is 11
Sam's age is 16
```

The goal is to read in the file in such a way so that you end up with a list of datasets. For example, I think most students would be able to easily solve the problem if they were given the input as a list of dataset structs, where:

```
(define-struct dataset (name age))
```

and the input were presented as:

```
(list
  (make-dataset "Vonda" 15)
  (make-dataset "Agatha" 14)
  (make-dataset "Molly" 11)
  (make-dataset "Sam" 16))
```

This could be solved as follows:

```
(define (process-dataset ds)
  (printf "~a's age is ~a\n" (dataset-name ds) (dataset-age ds)))

; process-list-of-datasets: list-of-datasets -> void
; process-list-of-datasets takes a list of datasets and, as a side effect,
; prints a message to the screen for each dataset. Returns void.

(define (process-list-of-datasets datasets)
  (for-each process-dataset datasets))
```

Note that for-each is a higher-order function that essentially corresponds to this pattern:

```
(define (process-list-of-datasets datasets)
  (cond
    [(empty? datasets) (void)]
    [else (begin
              (process-dataset (first datasets))
              (process-list-of-datasets (rest datasets)))]))
```

So clearly the core logic of this problem is trivial. The key is to figure out how to take the file input and produce a list of datasets.

Discussion point: In the above code, process-list-of-datasets produces no return value (in Racket, we call the absence of a return value *void*), and prints

to the screen as a side effect. Racket programmers recognize that sometimes there is a genuine need to print something to the screen (which is why these functions are a part of Racket), but this style of programming is the exception not the norm. Racket programmers would be far more likely to write process-list-of-datasets as a function that returns a list of strings. For this example, the output would be

```
(list
  "Vonda's age is 15"
  "Agatha's age is 14"
  "Molly's age is 11"
  "Sam's age is 16")
```

Why do Racket programmers prefer this style? Which version is easier to test? Is there any way to name, reuse or manipulate the output of the version that prints to the screen?

As an experiment, try reformulating the above sample program into one that produces a list of strings. Then create a completely separate helper function that prints these strings to the screen. This separates the act of printing from the logic, rather than printing directly from the process-dataset function. Which version would you prefer for long-term maintenance of your code? Which version is easier to write under the time pressure of a contest?

7.2 Producing a list of datasets

At the time of last year's contest, the batch-io teachpack did not exist. I taught Alex how to use the built-in functions *file->string* and *file->lines* to read in the file as either one big string, or a list of line strings, respectively. Then, he used functions from Racket's regular expression library to chop up the string(s) into lists of words, used *string->number* to convert numeric strings to numbers, and various higher-order functions to manipulate this into a list of datasets.

7.3 Batch io

The new batch-io teachpack can be included with the line:

```
(require 2htdp/batch-io)
```

The most useful batch-io function for this example is *read-words/line*, which does almost what we want. With our example input file, (*read-words/line* "f:/data.txt") would produce

```
(list
  (list "4")
  (list "Vonda" "15")
  (list "Agatha" "14")
  (list "Molly" "11")
  (list "Sam" "16"))
```

It's not quite as easy to work with as a list of structs, but this is definitely something we can work with. Instead of a dataset being a struct, we now think of a dataset as a list of strings. We invoke *rest* on this to get rid of the dataset count. Then, we can use (*first dataset*) to extract the name, and (*string->number (second dataset)*) to extract the age.

I am currently lobbying to have two functions added to the batch-io library, namely *read-words-and-numbers* and *read-words-and-numbers/line* which would behave like their *read-words* counterparts but would have the added feature of automatically converting numeric strings to numbers. I doubt these functions will get added in time for the next contest, but I'll let you know if that changes.

8 Note to Contest Administrators

In dynamic languages like Racket and Python, it is easiest to ignore the dataset count and rely on the end-of-file marker to determine the number of datasets. However, this makes it all the more essential that the data file be well-formed. Specifically, *additional blank lines at the end of the file can really screw things up* for a programmer using this technique. These blank lines will get read in as additional empty datasets and will trigger a runtime error (and it seems unreasonable to expect students to check for these sorts of malformed data files). Contest administrators who wish to make their contest language-neutral should take special care to ensure that there are no additional blank lines at the end of the file. Similarly, administrators should verify that each line contains no additional whitespace characters hidden invisibly at the start or end of the line. Most tokenizing strategies will strip away these superfluous whitespace characters, but some will not. By taking care to avoid hidden whitespace characters in the data file, contest administrators can guarantee that the files can be easily processed by the greatest number of libraries in the greatest number of languages.

Although administrators should check that their own data files adhere to the most rigorous standards, contest judges should be lenient if a student's output contains extra blank lines at the beginning or end of the output. It is especially common for student programs to produce an extra blank line at the end depending on how the newline characters are woven into the for loop.