WELLESLEY

Programming Languages

CS 251

Fall 2021

Carolyn Anderson

Programs as Data

I/O in Racket

We've already seen how to read from files in Racket:

Open a file:

```
(define input (open-input-file "text.txt")
```

Read a single line from the file:

```
(read-line input)
```

Close file:

```
(close-input-port out)
```

I/O in Racket

Open a file:

(define input (open-input-file “text.txt")

Read first 100,000 characters of file as a string:

(read-string 100000 input)

If file contents are shorter than 100,000, all of the file will be read.

I/O in Racket

Here's how to write to files:

Open a file:

(define outfile (open-output-file "text.txt"))

Write a string to file:

(write "cat" outfile)

Throws an error if file already exists!

I/O in Racket

Open a file:

```
(define outfile
        (open-output-file #:exists 'truncate “text.txt”)
```

Write a string to file:

```
(write “cat” outfile)
```

'truncate overwrites existing contents of file

I/O in Racket

Open a file:

(define outfile

(open-output-file #:exists ‘append “text.txt”)

Write a string to file:

(write “cat” outfile)

'append appends to end of existing file contents

Quoting

Quote is a way to express data literals.
Given any Racket expression, **quote** returns the contents of the expression as data.

The quoted data remains unevaluated.

Quoting

(quote 3)	=> 3	a number
(quote “hi")	=> "hi"	a string
(quote a)	=> a	a symbol
(quote (+ 3 4))	=> (list '+ 3 4)	a list
(quote (a b c))	=> (list 'a 'b 'c)	a list
(quote (define x 25))	=> (list 'define 'x 25)	a list
(quote (lambda (x) (+ x 3)))	=> (list 'lambda (list 'x) (list '+ 'x 3))	a list

Symbols

Quoting a variable name does not produce a string, but another datatype: a symbol.

If we didn't have this datatype, we wouldn't be able to distinguish quoted names from strings.

```
'(define x 10)    => (list 'define 'x 10)        'define  is a symbol
'("define" x 10)  => (list "define" 'x 10)       "define" is a string
```

Writing a Racket program to file

Quoting gives us a way to write out Racket programs without evaluating them— which is exactly what we want to do when we write programs to file.

Shorthand for Quote

' is short-hand for (quote):

```
> (first ' 'road)
'quote
> (first '(quote road))
'quote
```

Print and Write Revisited

As we saw early on, Racket has two print operators: **print** and **write**. For quoted expressions, they act differently!

print prints a value in the same way that is it printed by the REPL.

write prints a value in such a way that read on the output produces the value back.

```
>(print #f)
 #f
>(print (quote
          (lambda (x)(x))))
 '(lambda (x)(x))
```

```
>(write #f)
 #f
>(write (quote
          (lambda (x)(x))))
 (lambda (x)(x))
```

Print versus Write

Print prints a value in the same way that is it printed by the REPL.

Write prints a value in such a way that read on the output produces the value back.

```
>(print #f)
  #f
>(print (quote
            (lambda (x)(x))))
  '(lambda (x)(x))
```

```
>(write #f)
  #f
>(write (quote
            (lambda (x)(x))))
  (lambda (x)(x))
```

Defining Data Structures

Struct

What do **data structures** look like in Racket?

Defining Data Structures

Struct

What do **data structures** look like in Racket?

Struct

What do **data structures** look like in Racket?

We can define them using **struct**.

(**struct** struct-id (field-id ...))

Struct

Say that we wanted to represent different orders at Truly's. We'll start with a scoop struct.

Struct

Next, we want to represent an ice cream cone. Let's say that we need to specify the cone type and up to two scoops.

Struct

struct automatically defines getter methods for its fields. We can use these to access the cone type and scoops of a cone.

Struct

What issues have we run into?

Problem 1: fixed number of fields

We said that we wanted to allow cones to have up to two scoops. How do we add optional fields?

Problem 1: fixed number of fields

We can do this using the #:auto-value option. This fills in a default value if a field is not filled.

Problem 2: validating fields

As we have seen, by default, **struct** imposes no requirements on the values of its fields. But we can add some checks using a **guard.**

Case analysis

So far we've relied on **cond** and **if** for control flow operators. Another useful construct is **match.**

In a **match**, the programmer defines a number of cases. Racket finds the first pattern (left) that matches the result of the given expression, and returns the result of evaluating its body (right).

```
(match exp
       (pat1   body1)
       (pat2   body2)
            ....
       (patn   bodyn))
```

Match

```
> (match 5
        (5    “five”)          ; Check if x is 5
        (10   “ten”)           ; Check if x is 10
        (20    “twenty”))      ; Check if x is 20
```

“five”

Special match syntax: (? exp pat)

(? *exp pattern*) is a special feature of match.

It checks whether *exp* applied to *pattern* is true.

This is useful for type-checking, since *pattern* refers to the **value** of the matched item, not its type.

Special match syntax: _

_ is the match equivalent of else in a conditional: it matches any expression.

You should only use _ in your last case, since otherwise, none of your other cases will be evaluated.

Special match syntax: ...

You can omit named sub-expressions in a case using …

Ellipsis acts like the Kleene star (*) in regular expressions.

```
(match lst
   ((list 1)  “length 1”)
   ((list x … 10) “length 10”))
```

Exercise: check for duplicates

Write a function that takes a list of strings and checks whether the first item in the string ever re-occurs:

```
> (dups? '("cat" "is" "cat"))
#t
> (dups? '("cat" "says" "meow"))
#f
```

Exercise: generic add

Write a generic addition function using **match**:

- If given a list of strings, your function should join them together into a single string.
- If given a list of numbers, your function should sum them together.
- If given any other kind of list, your function should return void.

Returning functions

The right-hand-side of match cases can return any kind of Racket expression, including functions.

```
(match x
  (0 +)
  (1 *))
```