CSCI 275: Programming Abstractions

Lecture 13: Types

Spring 2025

Stephen Checkoway
Slides from Molly Q Feldman

Announcements

Take-home exam next Monday

- No lecture on Monday
- I’ll be in my office during the exam time to answer questions
- Review on Friday

HW 4 due Friday

Reminder: Structs

Reminder: Struct Data Types

`(struct name (field-a field-b) ...)`

Racket has a very general mechanism for creating data structures and their associated procedures 🎉

To create our point data type, we can instead use

```
(struct point (x y))
```

This will create a new type named point *and the following procedures*:
`(point x y)` produces a new point with the given coordinates
`(point? obj)` returns `#t` if `obj` is a `point`
`(point-x p)` returns the `x` field
`(point-y p)` returns the `y` field

`(provide (struct-out point))` will provide the definitions of the `point`

Example point

```
(struct point (x y))

(define p (point 3 4))

(point? p) ; returns #t

(point? 10) ; returns #f

(point-x p) ; returns 3

(point-y p) ; returns 4

p ; DrRacket prints this as #<point>

(point-x '(a b c)) ; raises an error
```

One more addition: Make the struct transparent

```
(struct point (x y) #:transparent)
```

`(point 3 4) => (point 3 4)` rather than #<point>

```
(equal? (point 3 4) (point 3 4)) => #t
```

`#:transparent` is a **keyword argument**

Why? Without it… Hard to Debug

```
(define (thing p)
  (cond [(negative? (point-x p))
         (error 'thing "Invalid point: ~s" p)]
        [else '...]))

(thing (point -3 2))
=> thing: Invalid point: #<point>
```

Why? Without it… Equality isn’t structural

```
; With lists, equal? performs structural
comparison
(equal? '(point 3 4) '(point 3 4)) => #t

; eq? asks if the arguments are the same object
(eq? '(point 3 4) '(point 3 4)) => #f

; With structs, equal? acts like eq? by
default!
(equal? (point 3 4) (point 3 4)) => #f
```

Let’s build a tree

complex recursive data type!

tree.rkt

Used heavily in
Part 2 of HW 4!

```
#lang racket

; Provide the procedures for working with trees.
(provide tree make-tree empty-tree
         tree? empty-tree? leaf?
         tree-value tree-children)

; Provide 8 example trees.
(provide empty-tree T1 T2 T3 T4 T5 T6 T7 T8)
```

Tree definition and a special value

```
; Definition of tree datatype
(struct tree (value children) #:transparent)

; An empty tree is represented by null
(define empty-tree null)

; (empty-tree? empty-tree) returns #t
(define empty-tree? null?)

; Convenience constructor
; (make-tree v c1 c2 ... cn) is equivalent to
; (tree v (list c1 c2 ... cn))
(define (make-tree value . children)
  (tree value children))
```

Reminder: variadic function!

Utility procedure

; Returns `#t` if the tree `t` is a leaf.

```
(define (leaf? t)
  (cond [(empty-tree? t) #f]
        [(not (tree? t))
          (error 'leaf? "~s is not a tree" t)]
        [else (empty? (tree-children t))]))
```

Example (number) trees

```
(define T1 (make-tree 50))
(define T2 (make-tree 22))
(define T3 (make-tree 10))
(define T4 (make-tree 5))
(define T5 (make-tree 17))
(define T6 (make-tree 73 T1 T2 T3))
(define T7 (make-tree 100 T4 T5))
(define T8 (make-tree 16 T6 T7))
```

A tree is represented as a struct: `(tree value children)`.

If you want to count how many children a particular (nonempty) tree `t` has, what's the best way to do it?

A. `(length (tree-children t))`

B. `(length (third t))`

C. `(length (rest t))`

D. `(length (rest (rest t)))`

E. `(length (caddr t))`

Talking about Types

Why do languages have types?

Why do you think some languages have static types?

Why do you think some languages have dynamic types?

Dynamically-checked types

Dynamically-typed languages assign types to values *at runtime*

In Racket, we can ask what the type of a value is:
`number?`, `list?`, `pair?`, `boolean?`, etc.

Functions are forced to check that the types of their input match the expected type

Racket and Python are examples of dynamically-typed languages

What does this code do?

```
(define (mul x y)
   (if (= x 0)
         0
         (* x y)))
(mul 0 'blah)
```

A. Syntax error
B. Contract violation
C. Runtime error
D. Warning about `'blah`
E. Returns 0

No explicit error checking!

```
(define (mul x y)
  (if (= x 0)
      0
      (* x y)))

(mul 10 'blah)
```

This gives a contract error:

```
*: contract violation
  expected: number?
  given: 'blah
```

Note that the contract error is on `*`, not `mul`

Implementing explicit error checking

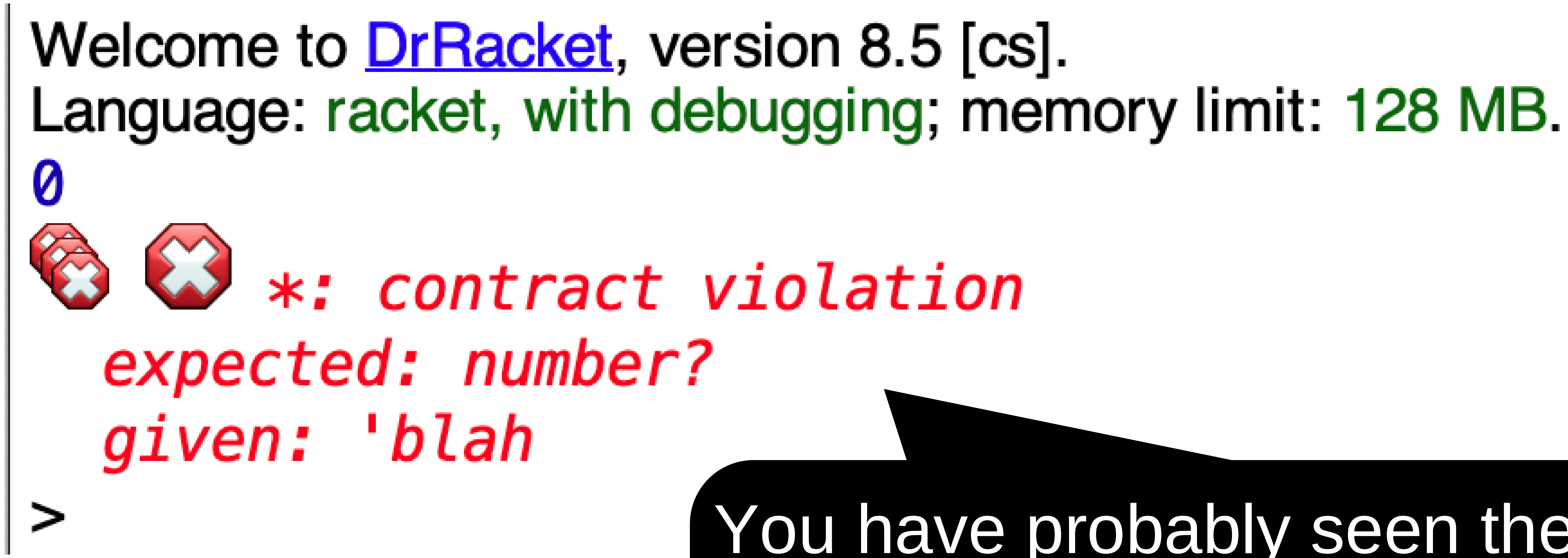
```
(define (mul x y)
  (cond [(not (number? x))
          (error 'mul "not a number: ~s" x)]
        [(not (number? y))
          (error 'mul "not a number: ~s" y)]
        [(= x 0) 0]
        [else (* x y)]))

(mul 0 'blah)
```

This gives the following error:

```
mul: not a number: blah
```

Aside: Contracts

Brief aside: Contracts



```
Welcome to DrRacket, version 8.5 [cs].
Language: racket, with debugging; memory limit: 128 MB.
0
*: contract violation
  expected: number?
  given: 'blah
>
```

You have probably seen these errors in all your Racket programming. But what exactly does “contract violation” mean here?

Brief aside: Contracts

Contracts are a predicate that declares some fact about a value that must be true

`number?` – The value is a number

`list?` – The value is a list

`positive?` – The value is positive

`pair?` – The value is a cons cell

`any/c` – Every value satisfies this contract

Contracts can help us do runtime error checking!

```
(define/contract (mul x y)
  ; x, y, and return value are numbers
  (-> number? number? number?)
  (if (= x 0)
      0
      (* x y)))
(mul 0 'blah)
```

This gives a contract error:

```
mul: contract violation
  expected: number?
  given: 'blah
  in: the 2nd argument of
      (-> number? number? number?)
```

Challenges of Dynamic Typing

Errors like passing and returning the wrong types of values are not caught until run time, even with contracts

```
(define/contract (faclist n)
  (-> positive? (listof integer?))
  (cond [(equal? n 1) 1]
        [ else (cons n (faclist (sub1 n)))]))
```

This has a type error, but it won't be caught until runtime

```
faclist: broke its own contract
  promised: list?
  produced: '(6 5 4 3 2 . 1)
```

Statically-checked types

Statically-typed languages compute a static approximation of the runtime types

The type of an expression is computed from the types of its sub expressions

This can be used to rule out a whole class of type errors at compile time

C, Java, Rust, and Haskell are examples of statically-typed languages

A Decision!

For the rest of today, we’re going to talk about **static types**

We *could* have done a small vignette of a type functional programming language (Haskell, Ocaml, etc.)

A Decision!

For the rest of today, we’re going to talk about **static types**

Really helpful because can give you a **direct** comparison between dynamic and statically typed languages

Would recommend Racket over Typed Racket though in *most cases*

Instead: we will discuss types using ***Typed Racket***

Also used in a Summary Problems

The presentation here is adapted, with thanks, from the Typed Racket Guide: https://docs.racket-lang.org/ts-guide/index.html

Adding Types to Racket

To start off with, what are the types we have available?

`Boolean`

`String`

`Number` – but also a complex hierarchy here including `Integer`, `Float-Complex`, etc.

Adding Types to Functions

We provide type signatures as follows:

```
(: function-name (-> input-type output-type))
(define (function-name input)
  …)
```

Below is a sum method in Racket. What should its type signature be?

```
(define (asum x y)
  (+ x y))
```

A. `(: asum (-> Number Number))`

B. `(: asum (-> Number Number Number))`

C. `(: asum (-> (Listof Number) Number))`

D. Something else

Below is a sum method in Racket. What should its type signature be?

```
(define (bsum lst)
   (cond [(empty? lst) 0]
         [else (+ (first lst) (bsum (rest lst)))]))
```

A. `(: bsum (-> Number Number))`

B. `(: bsum (-> Number Number Number))`

C. `(: bsum (-> (Listof Number) Number))`

D. Something else

What is `Listof`?

We decided `(: bsum (-> (Listof Number) Number)`
is the type for summing the elements of a list.

`Listof` is **not** actually a type, but rather a **type constructor**

Supporting type constructors (for instance, lists, arrays, references) is non-trivial

`(Listof Integer)` is meaningful,
`(Listof Listof)` is not

Similarly, `(String String)` does not work

How can we support procedures that output multiple types?

Motivation: Racket's `member` procedure has the following behavior

`(member 4 (list 1 2 3))` gives `#f`

`(member 2 (list 1 2 3))` gives `‘(2 3)`

So… how to state the return type if we want to write

```
(: member (-> Number (Listof Number) ???)
```

Answer is Union Types!

`Union` here is inspired by mathematical set union

```
;number specific member implementation
(: nmem (-> Number (Listof Number)
         (U False (Listof Number))))
(define (nmem x lst)
  ...))
```

Next Up

Homework 4 is due **Friday** at 11:59pm

- First Commit due **tonight**