Introduction to Typed Racket

The plan:

- Racket Crash Course
- Typed Racket and PL Racket
- Differences with the text
- Some PL Racket Examples

The Racket "Ecosystem"

- The Racket language is (mostly) in the Scheme family, or more generally in the Lisp family;
- Racket: the core language implementation;
- DrRacket: a Racket application, and IDE
- Our language(s)...
- Documentation: the Racket documentation is your friend (But beware that some things are provided in different forms from different places). `http://docs.racket-lang.org` is a good starting point

Getting started

- Find a machine with DrRacket installed (e.g. the linux lab).
- Follow the instructions at `http://www.cs.unb.ca/~bremner/teaching/cs3613/racket-setup` to customize DrRacket
- First part is based on `http://www.cs.unb.ca/~bremner/teaching/cs3613/racket/quick-ref.rkt`

Starting files

- Typed Racket files start like this:

1

```
#lang typed/racket
;; Program goes here.
```

but we will use a variant of the Typed Racket language, which has a few additional constructs:

2

```
#lang pl
;; Program goes here.
```

Racket Expressions

Racket is an expression based language. We can program by interactively evaluating expressions.

3

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Built-in atomic data

;; Booleans

true
false
#t ; another name for true, and the way
   ; it prints
#f ; ditto for false

; (:print-type #f)
```

4

```
;; Numbers

1
0.5

1/2 ; this is a literal fraction, not a
    ; division operation

1+2i ; complex number

; (:print-type 1/2)
```

5

```
;; Strings

"apple"
"banana cream pie"
```

6

```
;; Symbols

'apple
'banana-cream-pie
'a->b
'#%$^@*&?!
;(:print-type 'apple)
```

7

```
;; Characters

#\a
#\b
#\A
#\space  ; same as #\  (with a space after \)
(string #\a #\b)
```

Prefix Expressions

Racket is a member of the lisp (scheme) family and uses prefix notation (and many parentheses).

```
;; Procedure application
;;  (<expr> <expr>*)

(not true)                    ; => #f

(+ 1 2)                       ; => 3
(< 2 1)                       ; => #f
(= 1 1)                       ; => #t

(string-append "a" "b")       ; => "ab"
(string-ref "apple" 0)        ; => #\a
```

9

```
(eq? 'apple 'apple)           ; Object identity
(eq? 'apple 'orange)          ; => #f
(eq? "apple" "apple")         ; => depends

(equal? "apple" "apple")      ; => Content equality
(string=? "apple" "apple")    ; => ... for strings

(null? null)                ; => #t

(number? null)               ; => #f
(number? 12)                  ; => #t
```

Comments

```
;; This is a comment that continues to
;; the end of the line.
; One semi-colon is enough.

;; A common convention is to use two
;; semi-colons for multiple lines of
;; comments, and a single semi-colon when
;; adding a comment on the same line as
;; code.

#| This is a block comment, which starts
   with '#|' and ends with a '|#'.
|#

#;(comment out a single form)
```

Conditionals

```
;;  (cond
;;    [<expr> <expr>]*)
;;  (cond
;;    [<expr> <expr>]*
;;    [else <expr>])

(cond                           ;
  [(< 2 1) 17]                  ;
  [(> 2 1) 18])                 ; => 18

;; second expression not evaluated
(cond
  [true 8]                      ;
  [false (* 'a 'b)])            ; => 8
```

12

```
;; any number of cond-lines allowed
(cond
  [(< 3 1) 0]             ;
  [(< 3 2) 1]             ;
  [(< 3 3) 2]             ;
  [(< 3 4) 3]
  [(< 3 5) 4])            ; => 3

;; else allowed as last case
(cond
  [(eq? 'a 'b) 0]         ;
  [(eq? 'a 'c) 1]         ;
  [else 2])               ; => 2

(cond
  [(< 3 1) 1]
  [(< 3 2) 2])            ; => prints nothing

(void)                    ; => prints nothing
```

Racket Lists

13

```
;; Building lists
null                         ; => '()

(list 1 2 3)                  ; => '(1 2 3)
(cons 0 (list 1 2 3))         ; => '(0 1 2 3)

(cons 1 null)                ; => '(1)
(cons 1 '())                  ; => '(1)
(cons 'a (cons 2 null))      ; => '(a 2)

(list 1 2 3 null)            ; => '(1 2 3 ())
```

```
;; Functions on lists

(append (list 1 2) null) ; => '(1 2)
(append (list 1 2)
        (list 3 4))        ; => '(1 2 3 4)
(append (list 1 2)
        (list 'a 'b)
        (list true))       ; => '(1 2 a b #t)

(first (list 1 2 3))       ; => 1
(rest (list 1 2 3))        ; => '(2 3)
(first (rest (list 1 2))) ; => 2

(list-ref '(1 2 3) 2)      ; => 3
```

Defining Constants and Procedures/Functions

15

```
(define PI 3.14)

(define (double x)
        (list x x))

(define (Not a)
  (cond
    [a #f]
    [else #t]))

(: length : (Listof Any) -> Natural)
(define (length l)
 (cond
   [(null? l) 0]
   [else (add1 (length (rest l)))]))
```

Racket and Typed Racket

- So far almost everything we saw is (un-typed) Racket.
  Typed racket adds *type annotations* and a *type checker*.
- Type annotations and *type inference* reduce the amount declarations needed.
  Everything we saw so far is also validly typed.

Types of Typing

- Who has used a (statically) typed language?
- Who has used a typed language that's not Java?
- Who has used a dynamically typed language?

Why types?

- Types help structure programs.
- Types provide enforced and mandatory documentation.
- Types help catch errors.

Why Typed Racket?

- Racket it is an excellent language for experimenting with programming languages.
- Types are an important programming language feature; Typed Racket will help us understand them.
- Data-first design. The structure of your program is derived from the structure of your data.
  Types make this pervasive – we have to think about our data before our code.
- A language for describing data; Having such a language means that we get to be more precise and more expressive talking about code.

Definitions with type annotations

16

```
(define PI 3.14159)
(* PI 10)                    ; => 31.4159

;; (: <id> <type>)
(: PI2 Real)
(define PI2 (* PI PI))

(: circle-area : Number -> Number )
(define (circle-area r)
  (* PI r r))
(circle-area 10)             ; => 314.159
```

17

```
(: f : Number -> Number)
(define (f x)
   (* x (+ x 1)))

;; Less commonly in this course:
(define: (f2 [x : Number]) : Number
    (* x (+ x 1)))
```

Defining datatypes

18

```
;;  (define-type <id>
;;     [<id> <type>*]*)

(define-type Animal
  [Snake   Symbol Number Symbol]
  [Tiger   Symbol Number])

(Snake 'Slimey 10 'rats)
(Tiger 'Tony 12)
```

19

```
#;(Snake 10 'Slimey 5)
; => compile error: 10 is not a symbol

(Animal? (Snake 'Slimey 10 'rats)) ; => #t
(Animal? (Tiger 'Tony 12)) ; => #t
(Animal? 10)               ; => #f
```

20

```
;; A type can have any number of variants:
(define-type Shape
  [Square  Number] ; Side length
  [Circle  Number] ; Radius
  [Triangle Number Number]) ; height width

(Shape? (Triangle 10 12)) ; => #t
```

Datatype case dispatch

21

```
;;  (cases  <expr>
;;    [(<id> <id>*) <expr>]*)
;;  (cases <expr>
;;    [<id> (<id>*) <expr>]*
;;    [else <expr>])


(cases (Snake 'Slimey 10 'rats)
  [(Snake n w f) n]
  [(Tiger n sc) n])
```

22

```
(: animal-name : Animal -> Symbol)
(define (animal-name a)
  (cases a
    [(Snake n w f) n]
    [(Tiger n sc) n]))

(animal-name (Snake 'Slimey 10 'rats))
; => 'Slimey

(animal-name (Tiger 'Tony 12)) ; => 'Tony

#;(animal-name 10)        ; => error: Type error
```

23

```
(: animal-weight : Animal -> (U Number #f))
(define (animal-weight a)
  (cases a
    [(Snake n w f) w]
    [else #f]))

(animal-weight (Snake 'Slimey 10 'rats))
(animal-weight (Tiger 'Tony 12))
```

Short Circuit And/Or

24

```
;;  (and <expr>*)
;;  (or <expr>*)

(and true true)             ; => #t
(and true false)            ; => #f
(and (< 2 1) true)          ; => #f
(and (< 2 1) (+ 'a 'b))     ; => #f short circuit
```

25

```
(or false true)               ; #t
(or false false)              ; #f

(and true true true true) ; => #t
(or false false false)    ; => #f

(and true 1)              ; => 1

(or true (/ 1 0))
```

Local binding

26

```
;;  (let ([<id> <expr>]*) <expr>)

(let ([x 10]
      [y 11])
  (+ x y))

(let ([x 0])
  (let ([x 10]
        [y (+ x 1)])
    (+ x y)))

(let ([x 0])
  (let* ([x 10]
         [y (+ x 1)])
    (+ x y)))
```

First-class functions

27

```
;; Anonymous function:
;;  (lambda (<id>*) <expr>)
;;  (lambda: ( (<id> : <type>)* ) <expr>)

(lambda: [(x : Number)] (+ x 1))
;; => #<procedure>

;; Note the type annotation is not needed here

((lambda (x) (+ x 1)) 10) ; => 11
```

28

```
(define add-one
  (lambda: [(x : Number)]
    (+ x 1)))
(add-one 10)                  ; => 11
```

29

```
;; Similarly note here the inner lambda does
   not need annotation
(: make-adder : Number -> (Number -> Number))
(define (make-adder n)
  (lambda (m)
    (+ m n)))
(make-adder 8)                ; => #<procedure>
(define add-five (make-adder 5))
(add-five 12)                 ; => 17
((make-adder 5) 12)           ; => 17
```

```
(map (lambda: [(x : Number)]
       (* x x))
     '(1 2 3))              ; => (list 1 4 9)

(andmap (lambda: [(x : Number)] (< x 10))
        '(1 2 3))           ; => #t
(andmap (lambda: [(x : Number)]
                   (< x 10))
        '(1 20 3))          ; => #f
```

31

```
;; The apply function may be useful eventually:

(: f :  Number Number Number -> Number)
(define (f a b c) (+ a (- b c)))
(define l '(1 2 3))

#;(f l)                    ; => error: f expects
   3 arguments
(apply f l)                ; => 0

;; apply is most useful with functions that
   accept any
;;  number of arguments:

(apply + '(1 2 3 4 5))     ; => 15
```

Examples

```
(: is-odd? : Number -> Boolean)
(define (is-odd? x)
  (if (zero? x)
      false
      (is-even? (- x 1))))

(: is-even? : Number -> Boolean)
(define (is-even? x)
  (if (zero? x)
      true
      (is-odd? (- x 1))))
(is-odd? 12)                  ; => #f
```

33

```
(: digit-num : Number -> (U Number String))
(define (digit-num n)
  (cond [(<= n 9)     1]
        [(<= n 99)    2]
        [(<= n 999)   3]
        [(<= n 9999)  4]
        [else         "a lot"]))
```

34

```
(: fact : Number -> Number)
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))
```

```
(: helper : Number Number -> Number)
(define (helper n acc)
  (if (zero? n)
    acc
    (helper (- n 1) (* acc n))))

(: fact : Number -> Number)
(define (fact n)
  (helper n 1))
```

```
(: fact : Number -> Number)
(define (fact n)
  (: helper : Number Number -> Number)
  (define (helper n acc)
    (if (zero? n)
      acc
      (helper (- n 1) (* acc n))))
  (helper n 1))
```

```
(: every? : (All (A) (A -> Boolean)
                     (Listof A) -> Boolean))
;; Returns true if all pass pred.
(define (every? pred lst)
  (or (null? lst)
      (and (pred (first lst))
           (every? pred (rest lst)))))
```

A parser for arithmetic

38

```
(: parse-sexpr : Sexpr -> AE)
;; to convert s-expressions into AEs
(define (parse-sexpr sexpr)
   (match sexpr
    [(number: n) (Num n)]
    [(list '+ left right)
     (Add (parse-sexpr left) (parse-sexpr
        right))]
    [(list '- left right)
     (Sub (parse-sexpr left) (parse-sexpr
        right))]
    [else (error 'parse-sexpr
                 "bad syntax in ~s" sexpr)]))
```

Differences with the text

- PL Racket uses types, not predicates, in define-type.

39

```
(define-type AE
  [Num Number]
  [Add AE AE])

  ;;  versus

  ; (define-type AE
  ;   [Num (n number?)]
  ;   [Add (l AE?) (r AE?)])
```

Fancier type examples

Typed Racket has unions, and gathers type information via predicates.


```
(: foo : (U String Number) -> Number)
(define (foo x)
  (if (string? x)
    (string-length x)
    ;; at this point it knows that `x' is not a
       string, therefore it
    ;; must be a number
    (+ 1 x)))
```

Statically typed languages are usually limited to "disjoint unions". For example, in Haskell you'd write:


```
data StrNum = Str String | Num Int

foo::StrNum -> Int
foo (Str string) = length string
foo (Num num) = num

-- And use it with an explicit constructor:

a = foo (Str "bar")
b = foo (Num 3)
```

Unions and Subtypes

- Typed Racket has a concept of subtypes In fact, the fact that it has (arbitrary) unions means that it must have subtypes too, since a type is always a subtype of a union that contains this type.
- Another result of this feature is that there is an ‘Any’ type that is the union of all other types.
- Consider the type of ‘error’: it’s a function that returns a type of ‘Nothing’ – a type that is the same as an **empty** union: (U). This means that an ‘error’ expression can be used anywhere you want because it is a subtype of anything at all.

or Else what?

- An ‘else’ clause in a ‘cond’ expression is almost always needed, for example:

42

```
(: digit-num : Number -> (U Number String))
(define (digit-num n)
  (cond [(<= n 9)     1]
        [(<= n 99)    2]
        [(<= n 999)   3]
        [(<= n 9999)  4]
        [(>  n 9999) "a lot"]))
```

- if you think that the type checker should know what this is doing, then how about replacing the last test with

43

```
(> (* n 10) (/ (* (- 10000 1) 20) 2))

;;  or

(>= n 10000)
```

Polymorphic trouble

It is difficult to do the right inference when polymorphic functions are passed around to higher-order functions. For example:

44
```
(: call : (All (A B) (A -> B) A -> B))
(define (call f x)
  (f x))
(call rest (list 4))
```

In such cases, we can use ‘inst’ to ”instantiate” a function with a polymorphic type to a given type – in this case, we can use it to make it treat ‘rest’ as a function that is specific for numeric lists:

45
```
(call (inst rest Number) (list 4))
```

In other rare cases, Typed Racket will infer a type that is not suitable for us – there is another ‘ann’ form that allows us to specify a certain type. Using this in the ‘call’ example is more verbose:

46

```
(call (ann rest : ((Listof Number)
                   -> (Listof Number))) (list
                      4))
```

However, these are going to be rare and will be mentioned explicitly whenever they’re needed.