Introduction

The plan:

- Racket review
- Textbook dialect: plait
- Some Racket Examples

Getting started

- Find a machine with DrRacket installed (e.g. the linux lab).
- Follow `https://www.cs.unb.ca/~bremner/teaching/cs3613/racket/setup` to customize DrRacket
- Documentation: the Racket documentation is your friend: `https://docs.racket-lang.org` is a good starting point
- There is very brief summary at `https://www.cs.unb.ca/~bremner/teaching/cs3613/racket/plait-demo.rkt`

Review from CS2613

- People missing CS2613 will have to do some extra work to catch up.
- Until the first midterm tutorial attendence is mandatory *only* for those without CS2613.
- The first tutorial of review material from CS2613 is available at `https://www.cs.unb.ca/~bremner/teaching/cs3613/tutorials/tutorial0`. Please complete this before Jan 14.

Starting files

- Racket files start like this:

1

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

- We will use a special dialect, simplifed and with static types:

2

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

Racket Expressions

We can program by interactively evaluating expressions.

3

```
;; Booleans
 #t #f
;; Numbers
1 0.5 1/2
;; Strings
"apple"  "banana cream pie"

;; Symbols
'apple 'banana-cream-pie
;; Characters
#\a  #\b  #\space
```

Prefix Expressions

Racket is a member of the lisp family and uses prefix notation.

4
```
(not #t)                ; => #f
(+ 1 2)                   ; => 3
(< 2 1)                   ; => #f
(string-append "a" "b")  ; => "ab"
```

5
```
(eq? 'apple 'apple)       ; Object identity
(equal? "apple" "apple")  ; => Content equality
(string=? "apple" "apple"); => ... for strings
(= 1 2)                   ; => for Numbers
```

Comments

```
; Comment until end of line.

;; multi-line comment

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

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

Conditionals

7

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

;; short circuit
(cond
  [#t 8]                ;
  [#f (/ 1 0)])      ; => 8
```

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

(cond                     ; and sometimes required
  [(< 3 1) 1]
  [(< 3 2) 2])
```

Racket Lists

9

```
;; Building lists
(list 1 2 3)              ; => '(1 2 3)

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

10

```
;; Functions on lists

(append (list 1 2)  (list 3 4))
(first (list 1 2 3))      ; => 1
(rest (list 1 2 3))       ; => '(2 3)
```

Defining Constants and Procedures/Functions

```
(define PI 3.14)

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

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

(define (length l)
 (cond
   [(empty? l) 0]
   [else (add1 (length (rest l)))]))
```

Racket and Types

- So far almost everything we saw is (un-typed) 'plai' Racket. 'plait' racket adds type annotations and a type checker.
- Most things we saw so far are also validly typed.
- Use `cond` or `list` to make an expression that is not 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 (static) types?

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

Why Racket with Types?

- Racket it good for experimenting with programming languages.
- Types are an important programming language feature
- Types enforce data-first design.

Definitions with type annotations

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

(define PI2 : Number (* PI PI))

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

(define (f [x : Number]) : Number
     (* x (+ x 1)))
```

Defining datatypes

```
(define-type Animal
  [Snake   (name : Symbol) (weight : Number)
           (food : Symbol)]
  [Tiger   (name : Symbol) (weight : Number)])

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

14

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

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

15

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

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

Datatype case dispatch

```
16  (type-case Animal (Snake 'Slimey 10 'rats)
      [(Snake n w f) n]
      [(Tiger n sc) n])


17  (define (animal-name a)
      (type-case Animal a
        [(Snake n w f) n]
        [(Tiger n sc) n]))

    (animal-name (Snake 'Slimey 10 'rats))
    (animal-name (Tiger 'Tony 12)) ; => 'Tony
```

```
(define (animal-weight a)
  (type-case Animal a
    [(Snake n w f) w]
    [else -1]))

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

Local binding

```
(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)))

(local [(define x 0)]
  (local [(define x 10) (define y (+ x 1))]
    (+ x y)))
```

First-class functions

```
(lambda [(x : Number)] (+ x 1))

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

((λ (x) (+ x 1)) 10) ; => 11

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

21

```
(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
```

22

```
(map (lambda (x) (* x x)) (list 1 2 3))
(map add1 (list 1 2 3))

(foldl (lambda (x y) (* x y)) 1 (list 1 2 3))
```

Examples

```scheme
(define (is-odd? x)
  (if (zero? x)
      #f
      (is-even? (- x 1))))

(define (is-even? x)
  (if (zero? x)
      #t
      (is-odd? (- x 1))))
(is-odd? 12)                ; => #f
```

24

```
(define (digit-num n)
  (cond [(<= n 9)     (some 1)]
        [(<= n 99)    (some 2)]
        [(<= n 999)   (some 3)]
        [else         (none)]))
```

25

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

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

(define (fact n)
  (helper n 1))
```

```
(define (every? pred lst)
  (or (empty? lst)
      (and (pred (first lst))
           (every? pred (rest lst)))))

(every? even? (list 2 3 4))
(every? even? (list 2 4 6))
```

A parser for arithmetic

```
(define (parse [s : S-Exp])
  (cond
     [(s-exp-number? s) (Num (s-exp->number s))]
     [(s-exp-list? s)
      (let* ([sl (s-exp->list s)]
             [op (s-exp->symbol (first sl))]
             [left (second sl)]
             [right (third sl)])
        (case op
          [(+) (Add (parse left) (parse right))]
          [(-) (Sub (parse left) (parse right))]))]
     [else (error 'parse-sexpr "bad syntax")]))

(parse `(+ 1 2))
```