Introduction

The plan for today:

- Boring admin stuff (Syllabus, mark breakdown)
- Racket review
- Textbook dialect: plait
- Some Racket Examples

Getting started

Install racket `https://download.racket-lang.org`

Customize `https://www.cs.unb.ca/~bremner/teaching/cs4613/racket/setup`

Documentation `https://docs.racket-lang.org`

Tour `https://www.cs.unb.ca/~bremner/teaching/cs4613/racket/plait-demo.rkt`

Review from CS2613

- People missing CS2613 will have to do some extra work to catch up.
- The first tutorial of review material from CS2613 is available at `https://www.cs.unb.ca/~bremner/teaching/cs4613/tutorials/tutorial0`. Please complete this before Jan 18.

Starting files

- Racket files start like this:

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

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

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

Racket Expressions

We can program by interactively evaluating expressions.

```
;; 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 uses prefix notation.

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

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

Conditionals

```
;; 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

;; sometimes required
(cond
  [(< 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

animals

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

(define slim (Snake 'Slimey 10 'rats))
(define anthony (Tiger 'Tony 12))
```

animals

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

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

Accessors

animals

```
(Snake-name slim)
#;(Snake-name anthony) ; run time error
```

```
;; 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
```

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)))
```

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-food a)
  (type-case Animal a
    [(Snake n w f) f]
    [else (error 'animal-food
                 "data unavailable")]))

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

Option

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