CS4613 Lecture 1

David Bremner

March 21, 2023

Online resources

- https://www.cs.unb.ca/~bremner/teaching/cs4613/

- D2L will not be used for this course
- Homework and tests will be handed via a custom handin server https://www.cs.unb.ca/~bremner/teaching/cs4613/handin-server/
- Marked work will be returned via the same server.

Syllabus

- https://www.cs.unb.ca/~bremner/teaching/cs4613/printable/
- https://www.cs.unb.ca/~bremner/teaching/cs6905/printable/

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`

Background from CS2613

- Undergrads are expected to be familiar with Racket (and functional programming) from CS2613.
- Graduate students are expected to pick this up.
- The racket material from CS2613 will be available in condensed form at `https://www.cs.unb.ca/~bremner/teaching/cs4613/tutorials`

Starting files

- In CS2613 we used

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

or

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

- In CS4613 we will use a special dialect, simplified and with static types:

2

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

Racket and Types

- Most things we saw in CS2613 are also validly (statically) 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))
```

```
#;(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
```

```
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-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)]))
```

Tests

revtest

```
(define (reverse lst)
  (local
      [(define (rev lst acc)
         (cond
           [(empty? lst) acc]
           [else
            (rev
             (rest lst)
             (cons (first lst) acc))]))]
    (rev lst empty)))

(module+ test
  (test (reverse empty) empty)
  (test (reverse (list 1 2 3))
        (list 3 2 1)))
```

What defines a language?

- syntax
- semantics
- libraries
- idioms

1. Side-note:
   E.W. DIJKSTRA
   "Goto Statement Considered Harmful."
   This paper tries to convince us that the well-known goto statement should be eliminated from our programming languages or, at least (since I don't think that it will ever be eliminated), that programmers should not use it. It is not clear what should replace it. The paper doesn't explain to us what would be the use of the "if" statement without a "goto" to redirect the flow of execution: Should all our postconditions consist of a single statement, or should we only use the arithmetic "if," which doesn't contain the offensive "goto"? And how will one deal with the case in which, having reached the end of an alternative, the program needs to continue the execution somewhere else?
   The author is a proponent of the so-called "structured programming" style, in which, if I get it right, gotos are replaced by indentation. Structured programming is a nice academic exercise, which works well for small examples, but I doubt that any real-world program will ever be written in such a style. More than 10 years of industrial experience with Fortran have proved conclusively to everybody concerned that, in the real world, the goto is useful and necessary: its presence might cause some inconveniences in debugging, but it is a de facto standard and we must live with it. It will take more than the academic elucubrations of a purist to remove it from our languages.
   Publishing this would waste valuable paper: Should it be published, I am as sure it will go uncited and unnoticed as I am confident that, 30 years from now, the goto will still be alive and well and used as widely as it is today.
   Confidential comments to the editor: The author should withdraw the paper and submit it someplace where it will not be peer reviewed. A letter to the editor would be a perfect choice: Nobody will notice it there!

How important is each of these?

- libraries give you the run-time support, arguably not a crucial part of the language itself.
- idioms originate from both language design and **culture**. Which of the two JavaScript samples is idiomatic?

```
if (isExplorer) {
    document.onmousemove =
        function () { ... };
} else {
    document.onmousemove =
        function () { ... };
}
```

```
function foo(n) {
  return function(m) { return m+n; };
}
```

How important is each of these?
- libraries give you the run-time support, arguably not a crucial part of the language itself.
- idioms originate from both language design and **culture**. Which of the two JavaScript samples is idiomatic?

```
if (isExplorer) {
    document.onmousemove =
        function () { ... };
} else {
    document.onmousemove =
        function () { ... };
}
function foo(n) {
  return function(m) { return m+n; };
}
```

1. the line between "a library" and "part of the language" is less obvious than it seems.

Syntax versus semantics

- Syntax is mostly in the cosmetics department.
- Semantics is the real thing.
- Suppose **a** is an array with 3 elements.

```
a[25]+5                          (Java: exception)
(+ (vector-ref a 25) 5)          (Racket: exception)
a[25]+5                          (JavaScript:
                                   NaN or undefined)
a[25]+5                          (Python:exception)
$a[25]+5                         (Perl: 5)
a[25]+5                          (C: Undefined
                                       Behaviour)
```

How should we talk about semantics?

- A few well-known formalisms for semantics.
- We will use programs to explain semantics: the best explanation **is** a program.
- Ignore possible philosophical issues with circularity (but be aware of them).

How should we talk about semantics?

- A few well-known formalisms for semantics.
- We will use programs to explain semantics: the best explanation **is** a program.
- Ignore possible philosophical issues with circularity (but be aware of them).

1. Actually, they are solved: Scheme has a formal explanation that can be taken as a translation from Scheme to logic, which means that things that we write can be translated to logic.

Syntax and semantics in this course

3
```
ae: NUMBER
  | ae "+" ae
  | ae "-" ae
```

4
```
(define (eval expr)
  (type-case AE expr
    [(Num n)   n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]))

(eval (Add (Num 1) (Num 2)))
```