

COMP 360

Programming Languages

Day 2



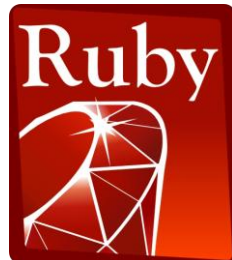
Scala



Swift



Racket



JavaScript



Dart

Review

- A Racket program is a sequence of **definitions** and **expressions**.
- A definition **binds** a variable to a value.
- An expression is something that can be evaluated.
- An expression always evaluates to a value (definitions never do).
- An **environment** holds variables and their values (bindings).

Review

- We have seen a number of different kinds of **expressions**:
 - Constants: `3`, `1/4`, `0.75`, `2+3i`, `#f`, `#t`
 - Evaluate to themselves.
 - Variables: `x`, `y`, `+`, `*`, `<`
 - Evaluate by looking up value in current environment.
 - `if` expressions: `(if test e1 e2)`
- What about `(+ e1 e2)` and `(< e1 e2)`?

New type of expression

- In Racket, all of the math operations are functions!
- Therefore, `(+ e1 e2)` and `(< e1 e2)` are governed by the same evaluation rules: those for ***function calls***.
- Syntax: `(f e1 e2 ... en)`
- Type-checking:
 - `f` must have type **`function`**
 - `e1`, `e2`, ..., `en` must be expressions.
- Evaluation: evaluate each argument expression to a value, then run the function code.

Examples of function calls

- `(+ 3 2) => 5`
- `(* 3 2) => 6`
- `(< 3 2) => #t`
- `(sqrt 4) => 2`
- `(expt 3 2) => 9`

Function definitions

Functions: the most important building block in the whole course

- Like Python/Java functions, have arguments and result
- But no classes, **this**, **return**, etc.

Example *function definition/binding*:

```
(define (add1 x)
  (+ 1 x))
```

Another example:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

A recursive example

; Note: correct only if $y \geq 0$

```
(define (pow x y)
  (if (= y 0)
      1
      (* x (pow x (- y 1))))))
```

Note: The *body* includes a (recursive) *function call*: `pow(x, y-1)`

Example, extended

```
(define (pow x y)
  (if (= y 0)
      1
      (* x (pow x (- y 1))))))
```

```
(define (cube x)
  (pow x 3))
```

```
(define sixtyfour (cube 4))
```

```
(define fortytwo (+ (pow 2 4) (pow 4 2) (cube 2) 2))
```


Recursion

- If you're not yet comfortable with recursion, you will be soon 😊
 - Will use for most functions taking or returning lists
- “Makes sense” because calls to same function solve “simpler” problems
- Recursion more powerful than loops
 - Will not normally use loops in Racket (they exist, but are usually poor style.)
 - Loops often (not always) obscure simple, elegant solutions

Function bindings

- Syntax: `(define (f x1 x2 . . . xn) b)`
 - (Will generalize in later lecture)
 - **f** is the name of the function.
 - **x1** through **xn** are the arguments (possibly none).
 - **b** is an expression that is the body of the function.
- Evaluation: ***The name of a function is a value!*** (it's a variable)
 - Different than in many other programming languages.
 - Adds **f** to environment so *later* expressions can *call* it.
 - (Function-call semantics will also allow recursion.)

Some gotchas

- Can't add extra parentheses like in Python/Java.
 - `(+ 1 2)` is fine... `(+ (1 2))` is not fine, and neither is `((+ 1 2))`.
 - Parentheses have a very particular meaning in Racket; they are not just used for changing precedence or grouping.
 - Using prefix notation for everything pretty much eliminates having to use parentheses for precedence.
- No “return” statement.
 - Functions only have a single expression as the body anyway.
 - Evaluating that statement becomes the return value.

Pairs and lists

So far: numbers, booleans (`#t` and `#f`), conditionals, variables, functions

- Now ways to build up data with multiple parts
- This is essential
- Java examples: classes, arrays

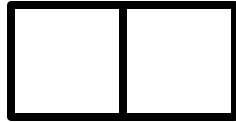
Rest of lecture:

- Pairs and lists
- These are our basic data structures that we use to create all other data structures.

Later: Other more general ways to create compound data

Cons cells

- Fundamental data structure for Racket (and pretty much every other "parentheses-based" programming language [Scheme, LISP])
- Two-piece structure:



- Left side is called the "car"
- Right side called the "cdr" (pronounced could-er)
- Each piece holds a pointer to something else (the something can be almost any data type)

Pairs

We need a way to *build* pairs and a way to *access* the pieces

Build:

- Syntax: `(cons e1 e2)`
- Evaluation: Evaluate `e1` to `v1` and `e2` to `v2`; result is `(v1 . v2)`
 - A pair of values is a value.
- Stored in a single cons cell.

Pairs

We need a way to *build* pairs and a way to *access* the pieces

Build:

- Alternate syntax: `' (v1 . v2)`
- Evaluation: Evaluates to the pair `(v1 . v2)`.
 - This is how to make a “literal” pair, where v1 and v2 are not evaluated.
 - Similar to using double quotes to make a string literal in Python/Java.
 - E.g.: `(cons (+ 1 2) (+ 3 4))` makes the pair `(3 . 7)`.
 - E.g.: `' (3 . 7)` also makes the pair `(3 . 7)`.
 - E.g.: However, `' ((+ 1 2) . (+ 3 4))` makes the pair `((+ 1 2) . (+ 3 4))`

Pairs

We need a way to *build* pairs and a way to *access* the pieces

Access:

- Syntax: `(car e)` and `(cdr e)`
- Evaluation: Evaluate `e` to a pair of values and return the first or second piece.
 - `e` must evaluate to a cons cell.

Examples

Functions can take and return pairs

```
(define (swap pair)
  (cons (cdr pair) (car pair)))
```

```
(define (sum-two-pairs p1 p2)
  (+ (car p1) (cdr p1) (car p2) (cdr p2)))
```

```
(define (div-mod n1 n2)
  (cons (quotient n1 n2) (remainder n1 n2)))
; returning more than one value is a pain in C++
```

```
(define (sort-pair pair)
  (if (< (car pair) (cdr pair))
      pair
      (swap pair)))
```

Lists

- Lists are built in Racket using linked lists of cons cells.

Need ways to *build* lists and *access* the pieces...

Building Lists

- The empty list is a value:

`' ()`

- In general, a list of values is a value; elements are separated by spaces:

`' (v1 v2 ...vn)`

- If **e1** evaluates to **v1** and **e2** evaluates to a list **(v2 v3 ... vn)**, then **(cons e1 e2)** evaluates to **(v1 v2 v3 ... vn)**

Accessing Lists

- `(null? e)` evaluates to `#t` if and only if `e` evaluates to `'()`.
- If `e` evaluates to `'(v1 v2 ... vn)` then `(car e)` evaluates to `v1`
 - throw exception if `e` evaluates to `'()`
 - Think of `car` as "get the first element of the list."
- If `e` evaluates to `(v1 v2 ... vn)` then `(cdr e)` evaluates to `(v2 ... vn)`
 - throw exception if `e` evaluates to `'()`
 - Think of `cdr` as "get everything but the first element of the list."
 - Notice result is a list

Example list functions

```
(define (sum-list lst)
  (if (null? lst)
      0
      (+ (car lst) (sum-list (cdr lst)))))
```

```
(define (countdown num)
  (if (< num 0)
      '()
      (cons num (countdown (- num 1)))))
```

Recursion again

Functions that process lists are usually recursive.

- Only way to “get to all the elements”
- What should the answer be for the empty list?
 - Usually, this is your base case.
- What should the answer be for a non-empty list?
 - Typically a combination of doing something with the **car** of the list and a recursive call on the **cdr** of the list.

Similarly, functions that produce lists of potentially any size will be recursive.

- You create a list out of smaller lists (with **cons**, **list**, or **append**).

Two other ways to build lists

- **list** function
 - Makes a list out of all arguments.
 - Arguments can be of any data type.
 - **(list e1 e2 ... en)** evaluates **e1** through **en** to values **v1** through **vn**; returns the list ' (**v1 v2 ... vn**) '.
- **append** function
 - Concatenates values inside lists given as arguments.
 - Arguments *must* be lists.
 - **(append e1 e2 ... en)** evaluates **e1** through **en** to values **v1** through **vn**;
 - If **v1 = (v11 v12 ...)** and **v2 = (v21 v22 ...)** etc, then return value is **(v11 v12 ... v21 v22 ...)**.

Lists of pairs

Processing lists of pairs requires no new features. Examples:

```
(define (sum-pair-list lst)
  (if (null? lst)
      0
      (+ (car (car lst)) (cdr (car lst)) (sum-pair-list (cdr
lst))))))
```

```
(define (firsts lst)
  (if (null? lst)
      '()
      (cons (car (car lst)) (firsts (cdr lst))))))
```

```
(define (seconds lst)
  (if (null? lst)
      '()
      (cons (cdr (car lst)) (seconds (cdr lst))))))
```

```
(define (sum-pair-list2 lst)
  (+ (sum-list (firsts lst)) (sum-list (seconds lst)))))
```