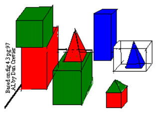
## Racket

- Based on *Scheme*, which is based on LISP, one of the earliest AI programming languages



John McCarthy, 1927-2011,
Inventor of Lisp

http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)


## Why Racket / Scheme / Lisp?

#1 Elegance

"If you ain't got elegance
You can never ever carry it off."
from "Hello, Dolly!"

#2 Minimal Syntax

#3 Mathematical

#4 Programs as Data

## Elegance

- *Functions* as primary building blocks — composability
- Lists are the main data structure — simplicity

Racket

List `'(1 2 3)`

Function Application `(fac 5)`

- In fact, lists are ***everything*** - data and program

## Ok, What is a Function, Anyway?

We are speaking of ***function*** in the *mathematical* sense.

## Possible Definitions

A **function** is a rule such that, for any given value, called the *argument*, another value, called the *result*, is uniquely determined.

The emphasis here is on the word *uniquely*: A given argument does not determine more than one value.

An argument must determine exactly *one* value.

x: the argument      f(x): the result

but in Racket, the result will be shown as (f x).

## A Function can be depicted as a Table

| Argument | Result |
|---|---|
| 1 | "Sunday" |
| 2 | "Monday" |
| 3 | "Tuesday" |
| . . . | . . . |
| 28 | "Sunday" |
| 29 | "Monday" |
| 30 | "Tuesday" |

Sometimes the table would need to be **infinite** in order to show all results.

## Functions as Sets of Pairs

A function can be regarded as a *set* of (argument, result) pairs.

Sometimes this set is called the ***graph*** of the function.

{(1, “Sunday”), (2, “Monday”), (3, “Tuesday”), … (30, “Tuesday”)}

The ***domain*** of the function is the set of first components of the pairs:
{1, 2, 3, …, 30}

The ***range*** of the function is the set of second components of the pairs:
{“Sunday”, “Monday”, “Tuesday”, …}

## Which of these sets is a function?

{(0, “red”), (1, “green”), (2, “blue”), (3, “red”)}

{(0, “red”), (1, “green”), (2, “blue”), (0, “red”)}

{(0, “red”), (1, “green”), (0, “blue”), (2, “green”)}

{(0, “red”), (1, “red”), (2, “red”)}

{ }

## Arity

Some functions have multiple argument value.
We can think of the arguments as being bundled as an n-tuple, and n is called the “arity” of the function.

Below we have a 2-ary function known as xor.

{((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 0)}

## Constant Functions

A constant function is one in which

the result value is always the same.

A 0-ary function is always a constant function.

{((), 99}

is the graph of a constant 0-ary function.

## A Function might be Computable

Example:

Given an argument,
the result is obtained by concatenating “-ary”:

This infinite function is exemplified by this table.

| Argument | Result |
|---|---|
| “1” | “1-ary” |
| “2” | “2-ary” |
| “3” | “3-ary” |
| . . . | . . . |
| “googol” | “googol-ary” |

## What Functions *Don’t* Do

Functions don’t necessarily “do” anything.

Although we often use functions to describe the results of some computation or process, it is the process, not the function itself, that is the doing.

So technically, functions don’t “take” arguments.

Also, functions don’t modify their arguments.

## Overloading

It is common in computer science, and most other fields, to find "overloaded" terminology.

This means that one word is used to mean more than one thing.

"function" is sometimes used to mean "procedure", for example, where the arguments could be modified, the result is not unique, etc.

It is best to avoid overloading, but sometimes this is impossible due to historical precedent.

## Side Effects

A procedure might not modify its arguments, and still might not qualify as a function.

For example, it could modify something else, such as a global value.

When a procedure does this, it is said to have a "side effect".

Functions are representable by procedures that don't have side effects.

## Advantages of Not Having Side-Effects

(Discuss)

## Functional Language

A programming language in which procedures have no side effects sometimes called a

"functional language".

This does not simply mean that the language "works",

just like "operating system" doesn't mean that the system "operates".

## Mostly-Functional Languages

Many languages have a functional subset that is able to compute any computable function.

Some have non-functional components (ones with side-effects), that integrate with the functional ones.

Racket/Scheme is an example.

## Racket/Scheme Notation

You are used to notation for functions results such as

f(x),
g(x, y),
…

Racket avoids commas and puts the function as the first element in an "S expression":

(f x)
(g x y)

"S" stands for symbolic.

## Racket/Scheme Terminology

Racket uses “procedure” to refer to both functions and procedures with side-effects.

## Racket/Scheme Notation

Furthermore, this notation is used for what might have been infix notation:

| | | |
|---|---|---|
| 2 + 3 | becomes | (+ 2 3) |
| 2 < 3 | becomes | (< 2 3) |

## Just the Right Number of Parentheses

In Racket, parentheses are neither optional nor redundant:

(+ 2 3)

Not + 2 3

Not ((+ 2 3))

Not (+ (2) (3))

## Some Functions have Arbitrary Arity

| | |
|---|---|
| (+ 2 3) | 2-ary |
| (+ 2 3 4) | ok, 3-ary |
| (+ 2 3 4 5 6) | ok, 5-ary |
| (+ 2) | ok, 1-ary |
| (+) | ok, 0-ary<br>(result is the additive identity, 0) |

This can be viewed as an example of overloading.

## Primitive Data Types in Racket

| Type | Meaning | Examples |
|---|---|---|
| Boolean | true or false value | #f, #t, (anything else behaves as if #t) |
| Integer | integer (arbitrary precision) | 42, 12345 |
| Rational | rational number | 2/3 |
| Float | floating-point numeral | 2.34e-5 |
| Complex | complex number | 2+3i |
| Symbol | a unique thing in memory, with a printable value | foo |
| String | a string of characters | “this is one” |
| Character | a single character | #\c |

## List Data vs. Function Application

`'(f 10)`

Is a **literal list** consisting of two elements… ~ a “quoted” list

`(f 10)`

Is the value of the expression resulting when the function f is run on the single input, 10.

## Uniform Syntax for Function Apps

A function application **always** appears as if a parenthesized list with:

The function first.

Zero or more arguments following

| | |
|---|---|
| `(f 5)` | *1 argument* |
| `(g 3 4 5)` | *3 arguments* |
| `(h)` | *No arguments* |
| `()` | *Disallowed* |

## Special Forms

Some expressions resemble function apps, but really aren't. These are called **special forms** and are distinguished by their first element, which is a keyword.

```
(define (add5 x) (+ x 5))
```

The word ***define*** identifies this as a special form. This is **not** a function called "define" being applied to two arguments. Rather, it signals the definition of a function called ***add5***, which adds 5 to its argument.

## Quoting is a Special Form

The ubiquitous **quote**, as in

```
'(to be or not to be)
```

is really an abbreviated special form:

```
(quote (to be or not to be))
```

## Some Built-in Functions

| | |
|---|---|
| `>` | arguments are numbers, value is boolean |
| `and or not` | boolean operators |
| `+ - * /` `modulo quotient` | arithmetic |
| `max min expt` `sqrt sin cos …` | mathematical functions |

## Conditional Special Forms

```
(if b
    true-branch
    false-branch)
```

A form that behaves like a function, almost.

```
(cond
   [ test     result ]
   [ test     result ]
     . . .
   [ else     result ]
  )
```

Brackets are an alternative, to make it more readable.

Regular parentheses can be used instead.

## Evaluation

"Evaluation" means the process of computing the result of an expression, such as, but not limited to, a function application.

This result is called the *value* of the expression.

We will show evaluations by a snapshot of an interaction with the Dr. Racket user interface.

| | |
|---|---|
| > (+ 2 3) | Expression |
| 5 | Expression's value |

## More Evaluation Examples

```
> (+ 2 3)
5

> (+ 2 3 4 5 6)
20

> (+ 2)
2

> (+)
0

> (- 3 2)
1

> (- 3)
-3

> (-)
 -: arity mismatch;
 the expected number of arguments does not match the given number
  expected: at least 1
  given: 0
```

## More$^2$ Evaluation Examples

```
> 5                                       numbers self-evaluate
5

> (+ 2 (* 3 4))
14

> (* 7 (+ 2 4))
42

> (> 7 (+ 2 4))
#t

> (if (> 7 (+ 2 4)) 5 6)
5

> (if (= 7 (+ 2 4)) 5 6)
6

> (exp (* -i pi))
-1.0-1.2246467991473532e-16i              numerical roundoff error
```

## Evaluation Process

If the expression is self-evaluating: return its value.

If the expression is a function application: evaluate the function and each argument using this process. Then apply the function to the argument values.

Let **[** . . . **]** mean the value of expression . . . .

## Evaluation Process

**[** 5 **]** = 5 self-evaluation

**[** (+ 2 5) **]** = apply **[**+**]** to **[**2**]** **[**5**]**
= apply + to 2 5
= 7

**[** (* 6 (+ 2 5)) **]** = apply **[***]** to **[**6**]** **[**(+ 2 5)**]**
= apply * to 6 **[**(+ 2 5)**]**
**. . . what goes in here? ...**
= apply * to 6 7
= 42

## Evaluation Process for **if**

For if, we deviate from the basic process:

**[** (**if** B C D) **]**:
Compute **[** B **]**.
If the result is #f, return the value of **[** D **]**.
Otherwise return the value of **[** C **]**.

In other words, anything other than #f is interpreted as true.

## Example: Evaluation Process for **if**

**[** (**if** (> 3 4) 5 (+ 6 7)) **]** =

**if [** (> 3 4) **] then [** 5 **] else [** (+ 6 7) **]** =

**if** #f **then [** 5 **] else [** (+ 6 7) **]** =

**[** (+ 6 7) **]** =

apply **[**+**] to [** 6 **] [** 7 **]** =

apply + to 6 7 =

13

## What about the process for **cond**?

```
(cond
   [ test    result ]
   [ test    result ]
         . . .
   [ else    result ]
 )
```

## Names for the Evaluation Process

Racket uses **applicative order** evaluation.

The arguments are evaluated first, then the
the function is applied.

In some languages, the function is “applied” *before* the actual arguments are evaluated. This is called **normal order** evaluation. It is much rarer.

Special forms are evaluated in special ways, which may be similar to normal order.

## User-Defined Functions

The form of a definition is:

(define ( <functionName> <**formal** arguments> )
<body>)

For example:

```
(define (quadValue a b c x)
    (+ (* (+ (* a x) b) x) c))
```

## Using a Defined Function

```
> (define (quadValue a b c x)
      (+ (* (+ (* a x) b) x) c))

> (quadValue 3 4 5 6)
137
```

## Evaluation of Defined Function

A ***first approximation*** is this:

In an application (<function> … <actual args>)

the <actual args> are evaluated first.

Then the values of the ***actual*** arguments are substituted for the corresponding ***formal*** arguments in the body of the function.

Then the body of the function is evaluated to produce the result.

[Later we will see why this is only a first approximation.]

## Example

**[** (quadValue 3 4 5 6) **]** =
apply **[** quadValue **]** to **[**3**]** **[**4**]** **[**5**]** **[**6**]** =
apply **[** quadValue **]** to 3 4 5 6 =

**[** (+ (* (+ (* 3 6) 4) 6) 5)) **]** =

. . .
137

## Function Types

A function f that accepts an argument from set A and produces a result in set B is denoted:

$f: A \rightarrow B$

The sets can be thought of as the types for the argument and result.

The type of *f* is denoted $A \rightarrow B$

## Functions with Multiple Arguments

$f: A \times B \rightarrow C$

Denotes the type of a function that accepts two arguments, one from A and one from B, and returns a result in C.

Example:

The type of > is $N \times N \rightarrow B$
where N is the set of numbers and B is {#t, #f}.

## List types

If A is a type then A* denotes the type

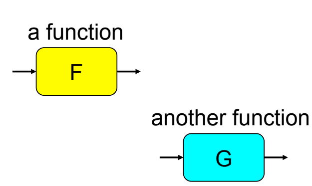
“list of things of type A”.

## High-Level Functional Programming

## Why Functional Programming is Important

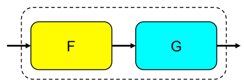
- No side effects:
  - Easier to debug
  - Easier to get parallel execution (e.g. on multi-core system)
- Composability: Easier to compose complex functions from simpler ones

## Composabilty



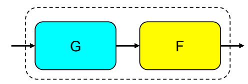
## Composabilty

a third function



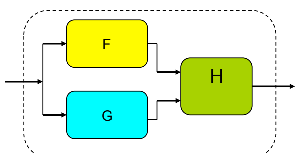

## Composabilty

a fourth function, maybe?

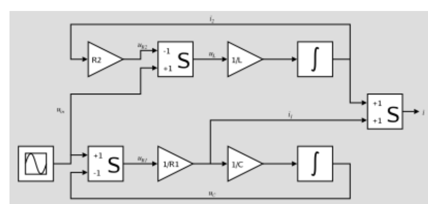



## More Ways to Compose

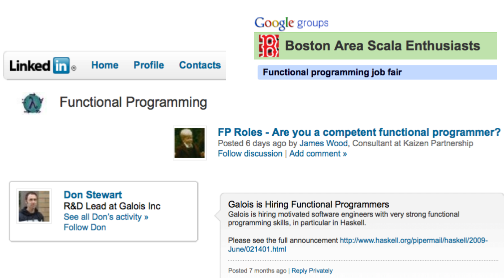
to get still more functions




## Engineering Applications

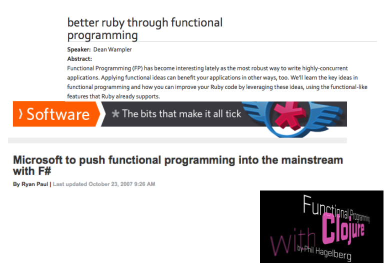



## But can I get a job?




## (Increasingly) Popular Functional Languages

- Haskell
- Erlang
- OCaml
- Scala
- F#
- Clojure



## So I should become a functional programmer?

- Yes, but don't stop there.
- Have functional skills in your toolkit, but be able to work outside that domain as well.
- Be "amphibious" and exploit the best of what the community has to offer.

## mapping over a list

- **map** applies a 1-ary function to each element of a list, returning a list of the same length, with the results of the applications in order

```
> (define (cube x) (* x x x))

> (map cube '(1 2 3 4 5))
'(1 8 27 64 125)
```

## Not all Equalities are Equal

- = Numeric equality
- equal? Content equality
- eq? Reference equality
- See Racket reference card I posted

| Equality | |
|---|---|
| (**equal?** $a_1$ $a_2$) | Return #t if data values $a_1$ and $a_2$ are equal, otherwise #f. |
| (**eq?** $a_1$ $a_2$) | Return #t if data values $a_1$ and $a_2$ are in the same memory locations, otherwise #f. |
| (= $n_1$ $n_2$ . . .) | Return #t if ***numbers*** $n_1$ . . . are equal, otherwise #f. |
| (**string=?** $s_1$ $s_2$) | Return #t if ***strings*** $s_1$ and $s_2$ are equal, otherwise #f. |
| (**char=?** $c_1$ $c_2$) | Return #t if ***characters*** $c_1$ and $c_2$ are equal, otherwise #f. |
| (**symbol=?** $s_1$ $s_2$) | Return #t if ***symbols*** $s_1$ and $s_2$ are equal, otherwise #f. |

## Possible Surprises

```
> (equal? "abc" (string-append "a"  "bc"))
#t                    (same content)

> (eq? "abc" (string-append "a"  "bc"))
#f                    (not same memory location)

> (= "abc" (string-append "a"  "bc"))
=: contract violation     (applies only to numbers)
```

When in doubt, probably should use **equal?**

## Symbols vs. Strings

- Strings and symbols are separate types.
- Explain on Board, and see Reference Card

```
> (equal?  'abc   "abc")
#f

> (symbol->string 'abc)
"abc"

>  (string->symbol "abc")
'abc
```

## More mapping examples

```
> (map symbol->string '(I should care))
'("I" "should" "care")

> (map string-length (map symbol->string '(I should care)))
'(1 6 4)

> (map reverse '( (Washington George) (Lincoln Abraham)
                  (Jefferson Thomas) (Obama Barack)))

'((George Washington) (Abraham Lincoln)
 (Thomas Jefferson) (Barack Obama))
```

# The type of map

- (map Function List) List
  A→B x A* → B*
- So map:(A→B) x A* → B*
- A and B could be the same type
- map **preserves the length** of its argument list

# n-ary map

- map will apply an n-ary function to n equal-length lists "pointwise"

```
> (map + '(1 2 3 4) '(9 8 7 6))
(10 10 10 10)

> (map list '(1 2 3 4) '(9 8 7 6))
((1 9) (2 8) (3 7) (4 6))
```

# The type of n-ary map

- map:$(A^n \rightarrow B) \times (A^*)^n \rightarrow B^*$
- All argument lists must be the same length in Racket

# foldr and foldl

- These functions "fold" a list into something like an element of the list.
- The first argument is a 2-ary function.
- The second argument is the result for an empty list.
- The third argument is the list being folded.

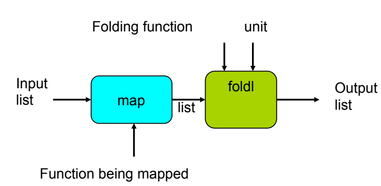
```
> (define (demo x y) (list '+ x y))

> (foldl demo 0 '(1 2 3 4 5))
(+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))

> (foldr demo 0 '(1 2 3 4 5))
(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))
```

```
> (foldl + 0 '(1 2 3 4 5))
15

> (foldl * 1 '(1 2 3 4 5))
120
```

# The type of foldl/foldr

- (foldl Function Unit List) Element
  AxA→A A A* → A
- So foldl:(AxA→A) x A x A* → A

# Composing map and foldl




The famous Google **mapReduce**!

# fold vs. reduce

- ***reduce*** is the functional programming identifier for folding when the direction is non-specific.
- reduce is not in Racket with that name.
- **map/reduce** in combination achieved fame from Google’s many uses of it.
- Google recently patented its use of map/reduce! 

# Google’s Patent

Google's patent on MapReduce could potentially pose a problem for those using third-party open source implementations. Patent #7,650,331, which was granted to Google on Tuesday, defines a **system and method for efficient large-scale data processing**:

> **A large-scale data processing system and method includes one or more application-independent map modules configured to read input data and to apply at least one application-specific map operation to the input data to produce intermediate data values, wherein the map operation is automatically parallelized across multiple processors in the parallel processing environment. A plurality of intermediate data structures are used to store the intermediate data values. One or more application-independent reduce modules are configured to retrieve the intermediate data values and to apply at least one application-specific reduce operation to the intermediate data values to provide output data.**

# Reactions

› Open Ended | # Open source and programming

**Google's MapReduce patent: what does it mean for Hadoop?**

**By Ryan Paul** | Last updated January 20, 2010 10:10 AM

Han Soete

The USPTO awarded search giant Google a software method patent that covers the principle of distributed MapReduce, a strategy for parallel processing that is used by the search giant. If Google chooses to aggressively enforce the patent, it could have significant implications for some open source software projects that use the technique, including the Apache Foundation's popular Hadoop software framework.

"Map" and "reduce" are functional programming primitives that have been used in software development for decades. A "map" operation allows you to apply a function to every item in a sequence, returning a sequence of equal size with the processed values. A "reduce" operation, also called "fold," accumulates the contents of a sequence into a single return value by performing a function that combines each item in the sequence with the return value of the previous iteration.

# Hadoop

## Hadoop

From Wikipedia, the free encyclopedia

**Apache Hadoop** is a Java software framework that supports data-intensive distributed applications under a free license.[1] It enables applications to work with thousands of nodes and petabytes of data. Hadoop was inspired by Google's MapReduce and Google File System (GFS) papers.

Hadoop is a top-level Apache project, being built and used by a community of contributors from all over the world.[2] Yahoo! has been the largest contributor[3] to the project and uses Hadoop extensively in its web search and advertising businesses.[4] IBM and Google have announced a major initiative to use Hadoop to support university courses in distributed computer programming.[5]

Hadoop was created by Doug Cutting (now a Cloudera employee),[6] who named it after his son's stuffed elephant.[7] It was originally developed to support distribution for the Nutch search engine project.[8]

# Our own map-reduce

```
> (define (map-reduce binary unit unary L)
    (foldr binary unit (map unary L)))

> (map-reduce + 0 length '((1 2 3) (4 5) (6) ()))
6
```

# Averaging a Non-Empty List

```
> (define (average L) (/ (foldl + 0 L) (length L)))

> (average '(1 2 3 4 5 6 7 8 9))
5

> (map average '((1 2 3) (2 3 4) (3 4 5) (5 6 7)))
(2 3 4 6)

> (average (map length '((1 2 3) (2 3 4) (3 4 5) (5 6 7))))
3

> (average (map average '((1 2 3) (2 3 4) (3 4 5) (5 6 7))))
3 3/4
```

## Filtering a List

- Function **filter** keeps elements that satisfy a predicate argument.

```
>  (define (even x) (= 0 (modulo x 2))) ;; = is a numeric test

> (filter even '(1 2 3 4 5 7 9 10))
(2 4 10)
```

## A-Lists and assoc

- Explain on board

```
> (assoc 'c '((a 1)(b 2) (c 3)))
'(c 3)

> (assoc 'd '((a 1)(b 2) (c 3)))
#f

(assoc 'c '((a 1 2 3)(b 4 5) (c 6 7 8) (d 9)))
'(c 6 7 8)
```

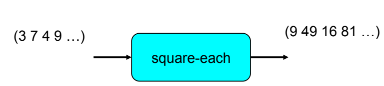
## Example: Generalized Anagram

- Suppose spaces “don’t count”.
- How would you define function anagram?
- Filter out the spaces.

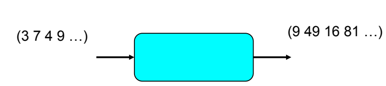
## Anonymous Functions

- We functions don’t need **names** to do our job.
- You can define us *anonymously* to:
  - Avoid having to think up names
  - Avoid cluttering up the namespace with temporary functions

## A Named Function




## An Anonymous Function Doing the Same Job



## Functions as “First-Class Citizens”

Disposition of data types in a typical language

| Type | Need name? | Use as argument | Return as result | Put in structure |
|---|---|---|---|---|
| Number | No | Yes | Yes | Yes |
| String | No | Yes | Yes | Yes |
| Function | ? | ? | ? | ? |

In Racket, the answer to these is Yes

## Lambda Expressions

- One way to specify an anonymous function uses the idea of “lambda expression”

    (lambda (x) …)

  means

  “**the function that**, **with argument x**, returns the value computed by …”

## Lambda Expression Examples

- (lambda (x) (+ 5 x))
  “the function that adds 5 to its argument”
- (lambda (x y) (expt y x))
  “the function that raises its second argument y to the power of the first argument x”
- (lambda (x) (list x x))
  “the function that makes a 2-element list of its argument twice in succession”

## Equivalent “bar-arrow” notation

- (Not seen often enough)
- Instead of (lambda (x) (+ 5 x))

  $x \longmapsto 5+x$ is more suggestive

  (Unicode arrow 21A6)

http://en.wikipedia.org/wiki/Function_(mathematics)
http://en.wikipedia.org/wiki/List_of_mathematical_symbols

## Lambda Expressions are *applied* just like any other function

```
> ( (lambda (x) (+ 5 x))  99)
104

> ( (lambda (x y) (expt y x)) 10 2)
1024

> ( (lambda (x) (list x x))  "foobar")
("foobar" "foobar")
```

## Lambda Expressions are *applied* just like any other function

> ( (lambda (x y) (expt y x)) 10 2)

In evaluation,
  y is bound to 10
  x is bound to 2
then (expt y x) is evaluated with those bindings.

## mapping and filtering with lambda expressions

- This kind of usage is very common:

```
> (map (lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)

> (filter (lambda (x) (> x 3)) '(1 2 3 4 5 6))
(4 5 6)
```

## Imported Variables in Lambda Expressions

- By "imported" I mean variables that are not arguments. These are sometimes called "**free variables**" in the lambda expression.
- These variables retain the meaning they had at the time of the function's definition. This is called **static scope**.
- They do not change their meaning based on context (which would be **dynamic scope**).

## Example of Imported Variable

- Below, b is ***imported*** into the lambda expression.

***imported***
(a free variable)

```
> (let (
       (b 99)
       )
   (map (lambda (x) (+ x b)) '(1 2 3 4 5))
   )

'(100 101 102 103 104)
```

***not imported***
(an *argument*)

## How to Spot Imported Variables

- They are not arguments.
- They are not defined locally *inside* the lambda expression (e.g. in a **let** form).

## Imported Values Bind Statically in Racket

**Functions should not be chameleons.**

```
> (let* (
         (b 99)
         (f (lambda (x) (+ b x)))
         )
   ( (lambda (b) (f 1)) 1000)
   )

100                  NOT 1001
```

Early implementations of Lisp tended to get this wrong.
It was called the "Funarg problem". [Google it]

## How Static Binding is Implemented

- The compiler turns a function into a "closure".
- Racket shows closures **cryptically**, as <procedure>
- A **closure** contains:
  - Reference to **imported** values
  - **Code** for evaluating the function, given the arguments
- Closures live "on the heap". They do not disappear when the stack shrinks.

```
> (lambda (x) (+ 5 x))
#<procedure>
```

## Racket Dialog

```
>  (lambda (x y) (expt y x))
#<procedure>

> ((lambda (x y) (expt y x)) 10 2)
1024
```

## Closure Example

```
(let* (
        (b 99)
        (f (lambda (x) (+ b x)))
      )
```

Here the value of f is a **closure** containing:

The binding for b.

The code for evaluating (+ b x), given x and b as bound.

## Using previous symbology **[ ]**

- **[** ( <u>(lambda (x y) (expt y x))</u> 10 2) **]** = apply **[** (lambda (x y) (expt y x) **]** to **[**10**]** **[**2**]**
- **[** (lambda (x y) (expt y x) **]** is a closure.
- Racket knows how to apply a closure to a commensurate number of arguments.

## Racket tries to do "the right thing" yet still provide for the user's convenience

y has no value yet

```
> (define f (lambda (x) ( + x y)))

> (define y 5)
> (f 10)
15

> (let ( (y 100) )
    (f 10)
    )
15                    NOT 110
```

Does **not re-bind** y dynamically.

## Functions Returning Functions

```
> (define (add n)  (lambda (x) (+ x n)))

> (map (add 5) '(1 2 3 4 5))
'(6 7 8 9 10)

> (map (add 10) '(1 2 3 4 5))
'(11 12 13 14 15)
```

(add n) returns a function, for any argument n

## The Same Definition *Not* Using Lambda

```
> (define ((add n) x) (+ x n))

> (map (add 5) '(1 2 3 4 5))
'(6 7 8 9 10)
```

This is called "Currying" the add function,

in honor of logician Haskell B. Curry.

The idea is due to Moses Schönfinkel .
Linguists called it "Schönfinkelisation".

http://en.wikipedia.org/wiki/Currying
http://en.wikipedia.org/wiki/Moses_Sch%C3%B6nfinkel

## The Type of (add 5) in Racket

```
> (add 5)

#<procedure>
```

Opaque

## Currying Represented by Lambda Expressions

- (lambda (x) (lambda (y) (+ x y))
- **The function that**, with argument x

  returns **the function that**, with argument y

  returns the value (+ x y).

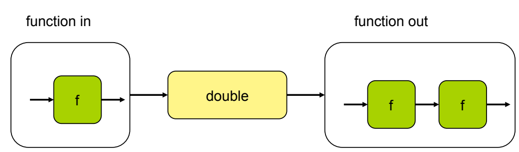
## Functions that both take and give functions as arguments

double returns a function that applies
its argument twice in succession.
(Not related to numeric doubling.)

```
> (define (double f) (lambda (x) (f (f x))))

> (define (square x) (* x x))
> ((double square) 5)

625
```

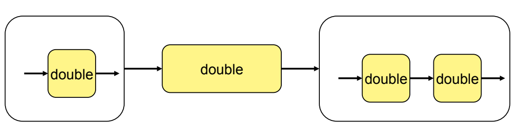
```
Alternate
> (define ((double f) x) (f (f x)))
> ((double square) 5)

625
```

## In Pictures




## (double double)

Is this meaningful?




What would it do?

## (double double)

```
> (double double)
#<procedure>

> (((double double) square) 5)
152587890625

> (square (square (square (square 5))))
152587890625
```

## How Big?

(((double (double double)) square) 5)

## How Big?

shot from my screen at 7-point font:

```
> (/ (log (((double (double double)) square) 5)) (log 10))
45,807 ; decimal digits
```

## Composing Functions Using Functions

```
> (define (compose f g) (lambda (x) (f (g x))))

> (define (cube x) (* x x x))
> ((compose cube square) 5)
15625

> ((compose square cube) 5)
15625

> (define (add2 n) (+ 2 n))
> ((compose add2 square) 5)
27

> ((compose square add2) 5)
49
```

Draw a picture