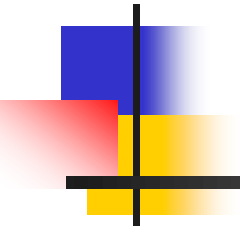
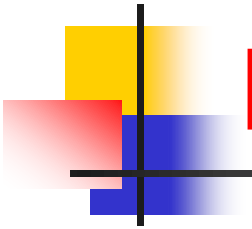


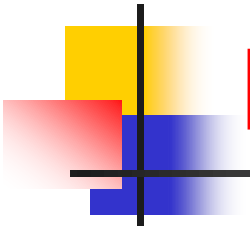
Lambda the Ultimate and Reduction Semantics

Corky Cartwright
Department of Computer Science
Rice University



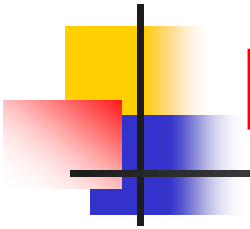
Functions Are Values

- In most functional languages, functions are data values. They can be
  - bound to variables;
  - returned as results from functions;
  - dynamically constructed during computation just like ordinary data values, but this process is clumsy without better notation that we have used so far.
- Terminology
  - A functional language where functions are not values is called a *first-order* functional language. SISAL is such a language. Java is often called a first-order language because methods are not data values.
  - A functional language where functions are regular data values (as described above) is called a *higher-order* functional language. Nearly all modern functional languages are higher-order.



Motivation for λ-notation

- Often, functions are used only once
  - Examples: arguments to functions like
    - `map`,
    - `filter`,
    - `fold`, and many more "higher-order" functions (functions that take functions as arguments)
- Sometimes we want to build new functions in the middle of a computation. The `local` construct suffices but it is notationally clumsy for this purpose.
- λ provides simpler, more concise notation
- Invented by Alonzo Church in the 1930s



Basic Idea

- λ-notation was invented by mathematicians. For example, given
  f (x) = $x^2 + 1$
  what is f? f is the function that maps x to $x^2 + 1$ which we might write as
  $x \rightarrow x^2 + 1$
  The latter avoids naming the function. The notation
  λ x . $x^2 + 1$ evolved instead of $x \rightarrow x^2 + 1$
- In Scheme, we write `(lambda (x) (+ (* x x) 1))` instead of λ x . $x^2 + 1$.
- `(define (f x) (+ (* x x) 1))` abbreviates
  `(define f (lambda (x) (+ (* x x) 1)))`

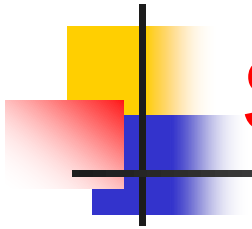
Why λ?

- The name was used by its inventor
  - Alonzo Church, logician, 1903-1995.
  - Princeton, NJ
  - Introduced lambda in 1930's to formalize mathematical proofs
- Church is my academic great-grandfather

  Alonzo Church -> Hartley Rogers ->

  David Luckham -> Corky Cartwright

Scope for a Lambda Abstraction



- Argument scope: `(lambda (`x_1 `...` x_n`)` *body*`)` introduces the variables x_1 `...` x_n which have *body* as their scope (except for holes)
- Example:

  ```
  (lambda (x) (+ (* x x) 1))
  ```

- Scope for variable introduced by `define`. At the top-level,

  ```
  (define f rhs)
  ```

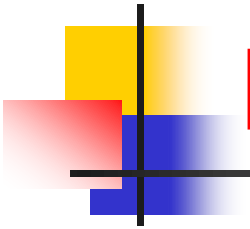
  introduces the variable `f` which is visible everywhere (except inside holes introduced by local definitions of `f`). Inside

  ```
  (local [(define f1 rhs1) ... (define fn rhsn)) body)
  ```

- the variables f_1 `...` f_n have the entire `local` as their scope.
- Recursion comes from `define` not `lambda`! It is possible to define recursive functions solely using `lambda` (and whatever primitive operations exist in the language) but it is surprisingly hard. The solution is called the Y-operator.

Some PL researchers are crazy about λ!

Prof.
Phil Wadler
University of
Edinburgh



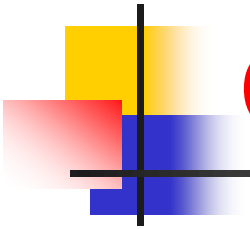
Example

Now we can write the following program concisely

```
(define l1 '(1 2 3 4 5))
(define l2
  (local ((define (square x) (* x x)))
    (map square l1)))
```

as

```
(define l1 '(1 2 3 4 5))
(define l2 (map (lambda (x) (* x x)) l))
```



Careful Definition of Syntax

- Official specification of what expressions that use `lambda` look like:
  - *exp* = ... | (**lambda** (*var**) *exp*)
- Interesting points
  - In Racket/Scheme, may have multiple arguments
  - May have have no arguments
- Application of a function with no arguments
  - `(define blowup (lambda () (/ 1 0)))`
    `(blowup)`

Currying: An Important Formulation of Lambda Notation

- In the original formulation of the Lambda Calculus, lambda abstraction was limited to a single argument because every function

  f: $D_1 \times \ldots \times D_n \rightarrow D$

  is isomorphic to a function

  f_c: $D_1 \rightarrow \ldots \rightarrow D_n \rightarrow D$

  eliminating the need for abstractions involving more than one variable. The function f_c is called the "curried" equivalent of f, in honor of Haskell Curry, who was prominent among the small group of logicians who formalized a system of mathematical notation starting in the 1920s, called combinatory logic, in which functions are ordinary values. The originator of the idea as recorded in the published literature was Moses Schönfinkel, a Russian logician and a colleague of Hilbert who warrants more fame. (Perhaps "schönfinkeling" is too much of a mouthful.)

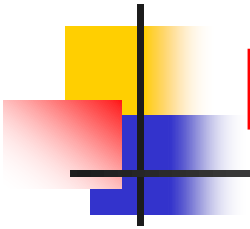
  Ironically, purely combinatory languages do not include lambda abstraction because they dispense with variables. This is a convenient and common "mathematical hack" in formal logic. Combinatory languages are not very intuitive. John Backus (the inventor of Fortran) developed a purely combinatory language called FP but it included a syntactic hack that I generalized in my paper *Lambda as a Combinator*.

Currying: An Important Formulation of Lambda Notation cont.

In contrast Racket supports multiple arguments including the option of no arguments. In statically typed functional languages (like Ocaml and Haskell) lambda abstraction is typically limited to a single argument implying that all functions are "curried". In practice, this convention has proven very convenient provided the parentheses around the argument in a function application are dropped. Infix notation can be supported as alternate notation for the full application of a curried binary function. Check out the Haskell language at haskell.org. (Perhaps it should called "Moses" instead of "Haskell".)

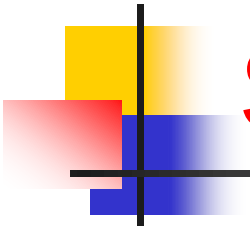
A simple example of the curried formulation of a familiar function is curried add. In Racket, it can be defined by the lambda abstraction

```
(lambda (x) (lambda (y) (+ x y)))
```



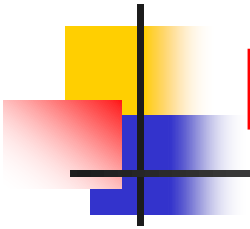
Reduction Semantics

- Simple Reduction Semantics: Essence of Functional Programming
- Idea: Evaluation of expressions is a familiar idea from grammar school.
- Grammar school:
  evaluate parenthesized arithmetic expressions
- Functional programming:
  evaluate arbitrary (functional program) text



Synopsis of Reduction

- Value are values are values …
- A value evaluates to itself so we stop evaluation when we reduce our original expression to a value.
- In most functional languages, always perform leftmost reductions because order matters



Evaluation of λ-expressions

- How do we evaluate a λ-abstraction

  `(lambda (`x_1 `...` x_n`)` *body*`)`

  It's a value!
- What about λ-applications?

  `((lambda (`x_1 `...` x_n`)` *body*`)` v_1 `...` v_n`)`

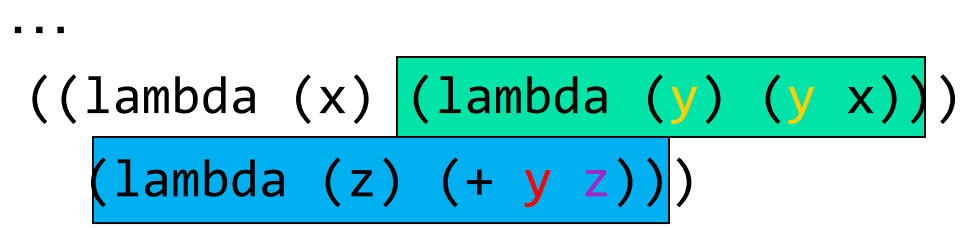
  `=>` *body*`[`$x_1 \leftarrow v_1$ `...` $x_n \leftarrow v_n$`]` (called β-reduction)

Examples:

```
((lambda (x) (* x 5)) 4) => (* 4 5) => 20
```

```
((lambda (x) (x x)) (lambda (x) (x x)))
=> ((lambda (x) (x x)) (lambda (x) (x x)))
=> ((lambda (x) (x x)) (lambda (x) (x x)))
```

Problem with Raw Substitution




```
=> (lambda (y) (y (lambda (z) (+ y z)))))
```

WRONG!

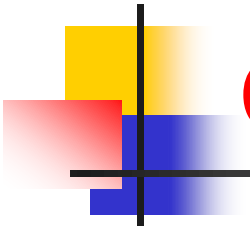
The meaning of `y` has changed! But it can *never* happen in the evaluation of Racket program text *provided* `lambda` *is the only binding construct*. Racket never reduces expressons inside a `lambda`.

Prevent Capture: Safe Substitution

- Must rename local variables in the code body that is being modified by the substitution to avoid capturing free variables in the argument expression that is being substituted.

```
   ((lambda (x) (lambda (y) (y x))) (lambda (z) (+ y z)))
=> ((lambda (x) (lambda (f) (f x))) (lambda (z) (+ y z)))
=> (lambda (f) (f (lambda (z) (+ y z))))
```

- Only necessary when:
  - Performing program transformations.
  - Performing a beta-reduction *body*[$x_1 \leftarrow v_1$ ... $x_n \leftarrow v_n$] where a variable u bound by `define` appears in some v_i and x_i appears free in *body* within a bound occurrence of a local variable named `u`.



Comprehensive Reduction Rules

- The document Laws Of Evaluation entitled ***Evaluating Core Racket Programs*** is a comprehensive description of the reduction semantics of functional Racket. In the literature, this form of semantics is often called a "rewrite-rule" semantics. A similar system, with minor technical differences, called Structural Operational Semantics is widely cited by computer science researchers (see Wikipedia!) but regrettably is almost completely ignored in the documentation of mainstream languages.
- As you saw in HW3, you need to understand it in detail.