CSE583:
Programming Languages

David Notkin
11 January 2000
notkin@cs.washington.edu
http://www.cs.washington.edu/education/courses/583

Functional programming: two weeks

- Scheme
  - Gives a strong, language-based foundation for functional programming
  - May be mostly review for some of you
- Some theory
  - Theoretical foundations and issues in functional programming
- ML
  - A modern basis for discussing key issues in functional programming

Scheme: a Lisp descendant

- A statically scoped and properly tail-recursive dialect of the Lisp programming language developed by Steele and Sussman in the mid 1970s
  - Embodies an executable version of the lamda calculus
- Intended to have an exceptionally clear and simple semantics with few different ways to form expressions

Statically scoped (review)

- A free variable in a function definition is bound to the variable of the same name in the closest enclosing block
- Dynamically scoped: free variable bound to nearest match on the call stack

```
proc fork(x,y:int){
  proc snork(a:int){
    proc cork() {
      a := x + y;
    }
    x : int;
    a := x + y;
  }
}
```


Tail recursive

- The top level call returns a value identical to that of the bottom level call
- Who cares?
  - Performance!
  - Scheme *requires* tail recursion to be implemented as efficiently as iteration

```
(define member(e l)
  (cond
    (null l) #f
    (equal e (car l)) #t
    (member e (cdr l))
  ))
```

- Once an element is a member, it's always a member
- Once you start returning true, you keep on returning true until you unwind the whole recursive call stack

Scheme

- Dynamically typed, strongly typed
- Expression-oriented, largely side-effect-free
  - Functional languages are expression, not statement-oriented, since the expression define the function computation
- List-oriented, garbage-collected heap-based
- Good compilers exist

read-eval-print loop

- Programming language equivalent of fetch-increment-execute structure of computer architectures
- Heart of most interpreted languages
- Helps in rapid development of small programs; simplifies debugging

Scheme syntax

```
Program     ::= ( Definition | Expr )
Definition  ::=
    (define id Expr)
  | (define (id_fn id_formal1 … id_formalN) Expr)
Expr        ::= id | Literal | SpecialForm
              | (Expr_fn Expr_arg1 … Expr_argN)
Literal     ::= int | string | symbol | …
SpecialForm ::= (if Expr_test Expr_then Expr_else)
  | ...
```


Examples

- **Identifiers**
  - `x`
  - `x1_2`
  - `is-string?`
  - `make-string!`
  - `<=`
- **Literals** (self-evaluating expressions)
  - `3`
  - `0.34`
  - `-5.6e-7`
  - `"hello world!"`
  - `""`
  - `#t`
  - `#f`

Ex: Definitions, expressions and evaluation results

- `(+ 3 4)`
  7
- `(define seven (+ 3 4))`
  seven
- `seven`
  7
- `(+ seven 8)`
  15
- `(define (square n) (* n n))`
  square
- `(square 7)`
  49

Another example

```
(define (fact n)
  (if (<= n 0)
      1
      (* n (fact (- n 1)))))
fact
(fact 20)
2432902008176640000
(fact 1)
1
(fact "xyz")
???
```

- Everything is an expression, including what we usually think of as control structures
- Prefix operators and functions calls are regular, although not traditional

Special forms

- Standard rule: evaluate all arguments before invocation
  - This is called *eager evaluation*
- Can cause computational (and performance) problems
- Can define your own

```
(define x 0)
(define y 5)
(if (= x 0)
    0 (/ y x))
0
(define (my-if c t e)
  (if c t e))
(my-if (= x 0)
       0 (/ y x))
error!
```

```
(fact 3)
```

Other special forms

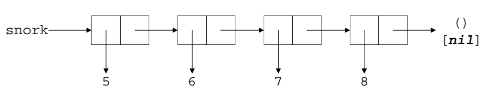
- `cond`: **like if-elseif-…else chain**
  ```
  – (cond ((> x 0) 1)
          (= x 0) 0)
          (else -1)))
  ```
- **short-circuit booleans**
  ```
  – (or (= x 0) (> (/ y x) 5) …)
  ```


Two more special forms

- **Local variable bindings**
  ```
  – (define x 1) (define y 2) (define z 3)
    (let ((x 5)
          (y (+ 3 4))
          (z (+ x y z)))
    (+ x y z)
    18
  ```
- **Sequentially evaluated local variable bindings**
  - **Replace** `let` **by** `let*`
  - **The same expression evaluates to** `27`

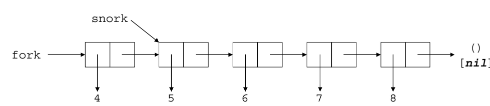
Lists

- **Standard data structure for aggregates**
- **A list is a singly-linked chain of** `cons`**-cells (i.e., pairs)**
- `(define snork (5 6 7 8))`



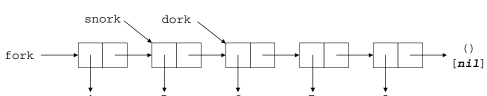
List creation

- `cons` **adds a cell at the beginning of a list, non-destructively**
  - **Remember, in the functional world, we don't destroy things (which would be a side-effect)**
- `(define fork (cons 4 snork))`
  - `fork` → **`(4 5 6 7 8)`**
  - `snork` → **`(5 6 7 8)`**



List accessors

- `car` **(head) returns the first element in a list**
  - `(+ (car snork) (car fork))` → **9**
- `cdr` **(tail) non-destructively returns the rest of the elements in the list after the first**
  - `(define dork (cdr fork))`
  - `fork` → **`(4 5 6 7 8)`** `snork` → **`(5 6 7 8)`**
    `dork` → **`(6 7 8)`**



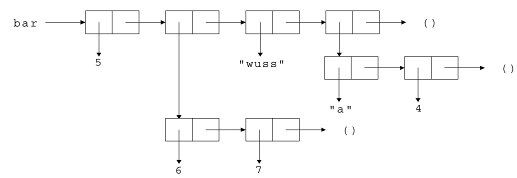
Why the names `car` and `cdr`?

The empty list

- `()` **is the empty list literal**
  - **In Lisp, this is the same as `nil`; in some Scheme implementations, it is the same as `#f` as well**
- `(cons 6 ())` → **`(6)`**
  - **The second argument to `cons` must be a list**
- `(cdr (cons 6 ()))` → **`()`**
- `(cdr 6)` → **???**

Lists: heterogeneous and nested

- `(define foo (5 6.7 "I am not a wuss"))`
- `(define bar (5 (6 7) "wuss" ("a" 4)))`



Quoting

- **How can we distinguish list literals from function invocations?**
  - `(cdr (cons 6 ()))` → **`()`**
  - `(cdr (cons (7 8) ()))` → **`error [function 7 not known]`**
  - `(cdr (cons (quote (7 8)) ()))` → **`()`**
  - **`quote (')` and `list` are special forms**
  - `(cdr (cons ' (7 8) ()))`
  - `(list 7 8)`

Programs and data

- **It should now be obvious that programs and data share the same syntax in Scheme (Lisp)**
- **To come full circle in this, there is a `eval` form that takes a list and evaluates it**
  - `(eval '(+ 2 3 4))` → **9**
- **Among other things, this makes it quite easy to write a Scheme interpreter in Scheme**

Equality

- **There are (more than) two different notions of equality in Scheme (and most other programming languages)**
  - **Are the two entities isomorphic? `(equal?)`**
  - **Are they the exact same entity? `(eqv?)`**

- `(eqv? 3 3)` → **`#t`**
- `(eqv? '(3 4) '(3 4))` → **`#f`**
- `(let ((x '(3 4))) (eqv? x x)` → **`#t`**
- `(equal? '(3 4) '(3 4))` → **`#t`**
- `(eqv? "hi" "hi")` → **`#f`**
- `(equal? "hi" "hi")` → **`#t`**
- `(eqv? 'hi 'hi)` → **`#t`**
- `(eqv? () ())` → **`#t`**

Other predicates

- **Numeric comparisons**
- **Type-testing predicates, available due to the use of dynamic typing**
  - `null? pair? symbol? boolean? number? integer? string?` …
- **The use of the `?` is a convention**

A note about binding and parameter passing

- All variables *refer* to data values, yielding a simple, regular model
- `let` binding and parameter passing do not copy
  - They introduce a new name that refers to and shares the right-hand-side value
  - "call by pointer value" or "call by sharing"
-
```
(define snork '(3 4 5))
(define (f fork)
  (let ((dork fork))
    (and (eqv? snork fork)
         (eqv? dork fork))))
(f snork) → #t
```


Global heap

- All data structures are implicitly allocated in the global heap
- Gives data structures unlimited lifetime (*indefinite extent*)
  - Simple and expressive
- Allows all data to be passed around and returned from functions and stored in other data structures

Garbage collection

- System automatically reclaims memory for unused data structures
  - Programmer needn't worry about freeing unused memory
  - Avoids freeing too early (dangling pointer bugs) and freeing too late (storage leak bugs)
  - System can sometimes be more efficient than programming

Recursion over lists

- Lists are a recursive data type
  - A list is either `()` [base case]
  - Or a pair of a value and another list [inductive case]
- Good for manipulation by recursive functions that share this structure

```
(define (f x …)
  (if (null? x)
     …base case on
       (car x)
     …inductive
       case on (cdr x)
  ))
```


Recursive examples

```
(define (length x)
  (if (null? x) 0
      (+ 1 (length (cdr x)))))

(define (sum x)
  (if (null? x) 0
      (+ (car x) (sum (cdr x)))))
```


Another example

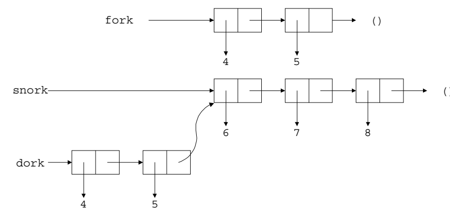
- Find a value associated with a given `key` in an association list (`alist`), a list of key-value pairs
-
```
(define (assoc key alist)
  (if (null? alist) #f
    (let* ((pair (car alist))
           (k (car pair))
           (v (cdr pair)))
       (if (equal? key k) v
           (assoc key (cdr alist))))))
(define Zips (list '(98195 Seattle)
                   '(15213 Pittsburgh)))
(assoc 98195 Zips) → #t
(assoc 98103 Zips) → #f
```

Yet another example

- Append two lists non-destructively
  - Requires a copy of the first list but not the second one
  - Sharing of list substructures is common and safe with no side-effects
- `(define (append x1 x2)`

```
(define (append x1 x2)
   (if (null? x1) x2
       (cons (car x1)
             (append (cdr x1)
                     x2))))
```


Append: in list form



- `(define snork '(6 7 8))`
- `(define fork '(4 5))`
- `(define dork (append fork snork))`

Side-effects

- There are several special forms that allow side-effects (note the `!` convention)
  - `set! set-car! set-cdr!`
- `set!` rebinds a variable to a new value
-

```
(define (test x)
  (set! x (cons 1 x))
  x)
(test '(2 3)) → (1 2 3)
```


Side effects on `cons` cells

- `set-car!` and `set-cdr!` do what you would expect
- You can use these, for instance, to define a destructive append, where the first list is reused instead of copied
- These tend to create more efficient functions that make the overall program more complicated due to more complex sharing
- Strictly, they are outside the basic notion of functional programming

Recursion vs. iteration

- These are comparable control structures
  - One or the other is needed for a language to be Turing-complete
  - Recursion is in some sense more general, since it has an implicit stack
- Recursion is often considered less efficient (time and space) than iteration
  - Procedures calls and stack allocation or deallocation on each "iteration" (recursive call)

Tail recursion (reprieve)

- In many cases, though, recursion can be compiled into code that's as efficient as iteration
- This is always possible in the face of tail recursion, where the recursive call is the last operation before returning

Tail recursion examples

```
• (define (last x)
    (if (null? (cdr x)) (car x)
        (last (cdr x))))
• (define (member e x)
    (cond ((null? x) #f)
          ((eqv? e (car x)) #t)
          (else (member e (cdr x)))))
```

- **The bold-italics invocations represent the final thing that the function does**
  - **After that, it's just a bunch of returns that don't "do" anything**

Converting to tail recursion

- **Programmers (and sometimes really smart compilers) can sometimes convert to tail recursion**

```
• (define (fact n)
    (if (= n 0) 1
        (+ n
        (fact (- n 1))
  )))
```

- **Not tail recursive (must do + after recursive call)**

```
(define (fact n)
  (f-iter n 1))
(define (f-iter n r)
  (if (= n 0) result
      (f-iter
        (- n 1)
        (* n result)
      )))
```

- **With a helper function, converted to tail recursion**

Partial conversions

- **Quicksort is a good example of a program for which there is no direct and simple conversion to tail recursion**
- **The second recursive call can be made tail recursive, but not the first**
- **Quicksort can be implemented using iteration, but only by implementing a stack internally**

First-class functions

- **Functions are themselves data values that can be passed around and called later**
  - **Which function to call can be computed as any other expression can be**
- **This enables a powerful form of abstraction where functions can take other functions as parameters**
  - **These parameterized functions are sometimes called *functionals***

Example: a find functional with a comparison parameter

```
• (define (find pred-fn x)
    (if (null? x) #f
      (if (pred-fn (car x)) (car x)
          (find pred-fn (cdr x)))))
```

- **`pred-fn` is a parameter that must be a function**
- `(define is-positive? n) (> n 0))` `(find is-positive? '(-3 -4 5 7))` → **5**
- `(find pair? '(3 (4 5) 6))` → **`(4 5)`**

`map`: apply a function to each element of a list, returning a list

- `(map square '(3 4 5))` → **`(9 16 25)`**
- `(map is-positive? '(3 -4 5))` → **`(#t #f #t)`**

```
• (defun map fn x)
    (if (null? x) ()
      (cons (fn (car x))
            (map fn (cdr x)))))
```

- The resulting list is always of the same length as the list to which `map` is applied

map2

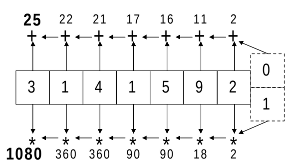
- (map2 + '(3 4 5) '(6 7 8)) → **(9 11 13)**
- (map2 list '(98195 15213)
            '("Seattle" "Pittsburgh"))
  → **((98195 "Seattle") (15213 "Pittsburgh"))**
- The resulting list is always of the same length as the first list argument
  - Why? (Note: I haven't provided enough information.)
  - What if the first and second list are of different lengths?
    - (map2 + '(3 4 5) '(6 7 8 9)) → **???**
    - (map2 + '(3 4 5 6) '(6 7 8)) → **???**

Anonymous functions

- **We can define functions that don't have names, to reduce cluttering of the global name space**
- (map (lambda (n) (* n n)) '(3 4 5))
  → **(9 16 25)**
- (map2 (lambda (x y)
           (if (= y 0) 0 (/ x y)))
           '(3 4 5) '(-1 0 2))
  → **(9 16 25)**
- **Define for functions (and indeed `cond`, `let`, `let*`, `and`, and `or`) is just syntactic sugar**
  - (define (square n) (* n n))
  - (define square (lambda (n) (* n n)))

Another functional

- reduce**: take a binary operator and apply to pairs of list elements to compute a result**
- (define (sum x) (reduce + 0 x))
- (define (prod x) (reduce * 1 x))



reduce

- **Lots of choices in defining it**
  - **left-to-right or right-to-left?**
  - **provide base value or require non-empty input list and symmetric operator?**
- **Right-to-left, with base value, used on previous slide**
- (define (reduce fn base x)
    (if (null? x) base
      (fn (car x)
          (reduce fn base (cdr x)))))

Functionals vs. recursion

- **There are many more common functionals like these**
- **How do you choose between using recursion and functionals?**
- **In general, functionals are clearer**
  - **Once you get over the learning curve**
  - **For instance, it's much easier to tell the structure of the resulting data**

Returning functions from functions

- **Functions are first-class (i.e., just like any other entity in Scheme)**
  - **So we can pass them to functions**
  - **Return them from functions**
  - **Store them in data structures**
- (define (compose f g)
    (lambda (x) (f (g x))))
  (define double-square
    (compose double square))

Currying

- **Every function of multiple arguments can reduce its number of arguments through *currying***
- **Take the original function, accept some of its arguments, and then return a function that takes the remaining arguments**
- **The returned function can be applied in many different contexts, without having to pass in the first arguments again**

Simple curry example

- We can think of any two-argument function in terms of two one-argument functions
- `(plus 3 5)`
- `(define (plus`$_3$` x) (+ 3 x))`
  `(plus`$_3$` 5)`
  `(plus`$_3$` 17)`
- `(define (plus f) (lambda (g) (+ g f)))`
  - `((plus 3) 5)` → `8`
- In essence we've taken a function with signature `int` × `int` → `int` and turned it into `int` → `(int` → `int)`

Another curry example

- `(define (mapc fn)`
  `(lambda (x) (map fn x)))`
- `((mapc square) '(3 4 5))`
  → **`(9 16 25)`**
- `(define squarer (mapc square))`
- `(squarer '(3 4 5))`
  → **`(9 16 25)`**
- `(define sum-of-squares`
  `(compose sum squarer))`
- `(sum-of-squares '(3 4 5))`
  → **`50`**

Another curry example

- `(define (reducec fn base)`
  `(lambda (x) ...))`
- `(define sum (reducec + 0))`
- `(define prod (reducec * 0))`
- As an aside, one can write a function that curries other functions

Lexical binding (reprise)

- **Scheme (like many languages) has a hierarchy of name bindings**
  - **There's a global scope of all `define`d names**
  - `lambda`, `let`, `let*` **define nested scopes**
    - **Remember, `define` is sugar for `lambda`**
- **What happens to free variables?**
  - `(define (f x) (+ x y))`

Example

```
(define x 100)
(define bar (lambda (x) (foo 1 2))
(define foo (lambda (x y)
  (let ((w (+ y 1)))
    (let ((y w))
      (+ x y)))))
```

- **Using lexical scoping,** `(bar 5)` → **`103`**
- **Using dynamic scoping,** `(bar 5)` → **`4`**
  - **The call stack is followed to find the binding of the variables**

First-class functions and scoping

- What if you return a function that has free variables?
  - In lexical scoping, they should be bound in the context in which it is defined
  - In dynamic scoping, they should be bound in the context in which it is used
- This is a classic issue, called the (upwards) funarg problem

Funarg example

```
(define (compose f g)
  (lambda (x) (f (g x))))
(define double-square
    (compose double square))
(define square-double
    (compose square double))
(let ((square (lambda (y) (* y y))))
   (define d-square
     (compose double square))
   (double-square 5)           ; which square?
   (square-double 5)           ; which square?
   (d-square 5))               ; which square?
```

Remember: compose returns a function like
```
(lambda (x)
  (double (square x)))
```


Closures

- The solution for static scoping is to return functions as a *closure*
- A closure defines the code for the function and its *environment*
  - Environment records bindings of free variables
  - Closure is no longer dependent on the enclosing scope
  - The closure is heap-allocated
  - Languages with closures include Scheme, ML, Haskell, Smalltalk-80, Cecil

More on closures

- If you can only pass nested procedures downward (downward funarg) then there are cheaper, stack-based allocation schemes
  - Ex: Pascal, Modula-3
- If allow nested procedures but not first-class procedures, then even cheaper
  - Ex: Ada
- If allow first-class procedures but no nesting, then also cheap
  - Ex: C, C++

Functions in data structures

```
• (define FileMenu
    (list (list 'Open… open-fn)
          (list 'Save save-fn)
          (list 'SaveAs… save-as-fn)
          (list 'Quit quit-fn))
• (define (click key)
    (let ((fn (assoc FileMenu key))) (fn)))
```


Control constructs

- Basic methods
  - Function call and return
  - Conditional execution
  - Looping
- Advanced methods
  - break, continue
  - Exception handling
  - Coroutines, threads
  - …

Continuations

- Scheme supports all advanced control mechanisms (including looping) with one primitive called *continuations*
- A continuation is a procedure that can be called (with a result value) to do "the rest of the program", exiting the current task
  - Enables parameterization of a procedure by "what to do next"
  - Enables having multiple return places, not just one normal return, for different outcomes

Continuation example

```
(define (find pred x if-found if-not-found)
  (cond ((null? x) (if-not-found))
        ((pred (car x)) (if-found (car x)))
        (else (find pred (cdr x)
                        if-found if-not-found
              )))))
(find is-positive? '(2 5 -0)
      (lambda (y) 'Yes) (lambda () 'No))
```


Current continuation

- The normal return point is an implicit continuation
  - It takes the returned value and "does the rest of the program"
- Scheme makes this continuation available

Features of continuations

- Continuations can be used to program
  - Exception handling, stack unwinding code, coroutines, threads, backtracking, etc.
  - No other special features are needed
- Because they are first-class data values, they are very powerful in Scheme
  - But they can be confusing

Shift: From Scheme to theory

- What are the underpinnings of the functional programming paradigm?
- We've seen a lot of the basics
- Now for a bit more depth on the theory behind them

Functions and their combination

- Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages. – Backus
- Functions are first-class data objects: they may be passed as arguments, returned as results, and stored in variables. The principal control mechanism in ML is recursive function application. –Harper

The lambda calculus

- This prevalence of functions demands a basic understanding of the lambda calculus
  - Other models of computation (such as Turing machines) don't give us much insight into functional computation
- Why care about the model of computation underlying a programming language?
  - It helps us deeply understand a language, and thus be more effective in using it

Function definition

- A function in the lambda calculus has a single basic form
  - λx,y,z • expr
  - where the x,y,z are identifiers representing the function's arguments
- The value of the lambda expression is a mathematical function (not a number, or a set of numbers, or a float, or a structure, or a C procedure, or anything else)

Function application

- Application consists of applying this function given values for the arguments
- We show this by listing the values after the lambda expression
  - (λx,y •x+y) 5 6
  - (λx,y •if x > y then x else y) 5 6
  - (λx •if x > 0 then 1 else
    if x < 0 then -1 else 0) -9

That it!

- **That's just about the full definition of the λ-calculus**

Currying

- **Many definitions of the lambda calculus restrict functions to having a single argument**
- **But that's OK, since there exists a function that can curry other functions, yielding only functions of one argument**
- **So we can use a multiple argument lambda calculus without loss of generality**

Beta reduction rule

- Application is defined using the beta-reduction rule
  - This in essence replaces the formals with the values of the applied arguments
- (λx,y •if x > y then x else y) 5 6
  if 5 > 6 then 5 else 6
  6

Representation

- **For the λ-calculus to be Turing-complete, we need to address some representational issues**
- **Indeed, it only really has identifiers that don't really represent anything**
- **Even writing + is kind of a cheat**

(Non-negative) integers

- Here's a scheme for representing non-negative integers
  - **0** ≡ λf • λx • x
  - **1** ≡ λf • λx • f x
  - **2** ≡ λf • λx • f (f x)
  - **3** ≡ λf • λx • f (f (f x))
- That is, every time we see λf • λx • f (f x), we think "Oh, that's 2"
- You can think of f as "increment by 1"

Defining addition

- add ≡ λa • λb • λf • λx • a f (b f x)
- To add 1 and 2 we write
  - (λa • λb • λf • λx • a f (b f x))
    (λf • λx • f x) (λf • λx • f (f x))

Reduction

- Applying β-reduction to both arguments
  - λf • λx • (λf • λx • f x) f
    ((λf • λx • f (f x)) f x)
  - λf • λx • (λf • λx • f x) f (f (f x))
  - λx • (λf • λx • f x) (f (f x))
  - λf • λx • f (f (f x))
- Whew, 1 + 2 is 3
  - This is much like adding in unary on a Turing machine

Representing booleans

- true ≡ λt • λf • t
- false ≡ λt • λf • f

Defining cond

- cond ≡ λb • λc • λa • b c a
  - cond true 2 1
    (λb • λc • λa • b c a) true 2 1
  - (λc • λa • true c a) 2 1
  - (λa • true 2 a) 1
  - true 2 1
  - (λt • λf • t) 2 1
  - (λf • 2) 1
  - 2
- Of course, we could represent 1 and 2 explicitly as functions, too

Normal form

- A lambda expression has reached normal form if no reduction other than renaming variables can be applied
  - Not all expressions have such a normal form
- The normal form is in some sense the value of the computation defined by the function
  - One Church-Rosser theorem in essence states that for the lambda calculus the normal form (if any) is unique for an expression

Reduction order

- A *normal-order reduction* sequentially applies the leftmost available reductions first
- An *applicative-order reduction* sequentially applies the leftmost innermost reduction first
- This is a little like top-down vs. bottom-up parsing and choosing what to reduce when

Example

- (λx • y) ((λx • x x) (λx • x x))
  - never reduces in applicative-order
- (λx • y) ((λx • x x) (λx • x x))
  - reduces to y directly in normal-order

High-level view

- Normal-order defines a kind of lazy (non-strict) semantics, where values are only computed as needed
  - This is not unlike shortcircuit boolean computations
- Applicative-order defines a kind of eager (strict) semantics, where values for functions are computed regardless of whether they are needed

A different high-level view

- **To the first order, you can think of normal-order as substituting the actual parameter for the formal parameter rather than evaluating the actual first**
  - **It's closely related to call-by-name in Algol.**
- **To the first order, you can think of applicative-order (also called eager-order) as evaluating each actual parameter once and passing its value to the formal parameter**

Comparing them

- **For many functions, the reduction order is immaterial to the computation**
- `fun sqr n = n * n;` **// from D. Watt**
- `sqr (p+q) [say, p = 2, q = 5]`
- **For applicative-order, we compute p+q=7, bind n to 7, then compute 49**
- **For normal-order, we pass in "p+q" and evaluate 2+5 each time sqr uses n**
- **But we get the same answer regardless**

Strict functions

- **A strict function requires all its parameters in order to be evaluated**
- **`sqr` is strict, since it requires its (only) parameter**
  - `(define one (x) 1)` **is not-strict**
  - `(one 92)`

More strictness

- `fun cand (b1, b2) = if b1 then b2 else false`
- `cand(n>0,t/n>0.5) with n=2 and t=0.8`
  - **Eagerly, n>0 evaluates to true, t/n>0.5 evaluates to false, and therefore the function evaluates to false**
  - **Normal-order also evaluates to false.**
- **But what if n=0**
  - **Eagerly, n>0 evaluates to false but t/n>0.5 fails due to division-by-zero; so the function call fails.**
  - **But with normal-order, the division isn't needed nor done, so it's fine.**
- **This function is considered to be strict in its first argument but non-strict in its second argument**

Fixed-points (or fixpoints)

- The idea of defining the semantics of lambda calculus by reducing first every expression to normal form (for which a simple mathematical denotation exists) by a sequence of contractions is attractive but, unfortunately, does not work as simply as suggested… The problem is that, since every contraction step … removes a λ, we have deduced a bit hastily that it decreases the overall number of λs. We have neglected the possibility for a contraction step actually to *add* one λ, or even more, while it removes another. **– Meyer**

Example

- SELF ≡ λx • (x (x))
- SELF (λx • (x (x)))
- λx • (x (x)) (λx • (x (x)))
- What does this application of SELF to itself produce?
  - λx • (x (x)) (λx • (x (x)))
  - Itself, with no reduction in lambda's.

The good news

- However, we're still not in trouble
- Church proved a theorem that shows that any recursive function can be written non-recursively in the lambda calculus
  - So we can use recursion without (this) danger in defining programs in functional languages

But its complicated

- **Theorem: If there is a normal form for a lambda expression, then it is unique**
  - **There isn't always a normal form, however**
- **Theorem: If there is a normal form, then normal-order reduction will get to it**
  - **Applicative-order reduction might not**
- **So, it seems pretty clear that you want to define a functional language in terms of normal-order reductions, right?**

In theory, there is no difference between theory and practice

- Nope, since efficiency shows it's ugly head
  - Even for sqr above, we had to recompute values for expressions more than once
  - And there are lots of examples that arise in practice where "unnecessary" computations arise regularly
- So, applicative-order evaluation looks better again

But…

- But there are two problems with this, too
  - The "magic" approach to representing recursion without recursion falls apart for applicative-order evaluation; a special reduction rule for recursion must be introduced
  - It isn't always faster to evaluate

Example

- (λx•1)(* 5 4) in normal-order and in applicative-order
- (λx•1)((λx•x x) (λx•x x)) in normal-order and in applicative-order, as we know still stands as a problem
- Even with this, most early functional languages used applicative-order evaluation: pure Lisp, FP, ML, Hope, etc.

What do to?

- The basic approach to doing better lies in representing reduction as a graph reduction process, not a string reduction process; this allows sharing of computations not allowed in string reductions (Wadsworth)
- A graph-based approach to normal-order evaluation in which recomputation is avoided (by sharing) is called lazy evaluation, or call-by-need
  - One can prove it has all the desirable properties of normal-order reduction and it more efficient than applicative order evaluation.
  - Still, performance of the underlying mechanisms isn't that great, although it's improved a ton

Theory

- OK, that's all the theory we'll cover for functional languages
  - There's tons more (typed lambda-calculus, as one example)
- It's not intended to make you theoreticians, but rather to give you some sense of the underlying mathematical basis for functional programming