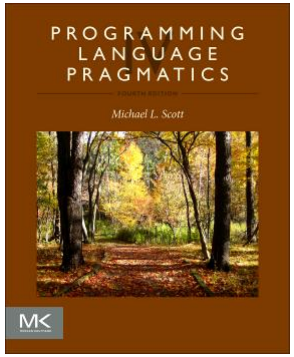
Chapter 11 :: Functional Languages

Programming Language Pragmatics

Michael L. Scott



ELSEVIER

Historical Origins

- The imperative and functional models grew out of work undertaken by Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
  - different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics
- These results led Church to conjecture that *any* intuitively appealing model of computing would be equally powerful as well
  - this conjecture is known as *Church's thesis*

Historical Origins

- Turing's model of computing was the *Turing machine* a sort of pushdown automaton using an unbounded storage "tape"
  - the Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables

Historical Origins

- Church's model of computing is called the *lambda calculus*
  - based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter λ—hence the notation's name.
  - Lambda calculus was the inspiration for functional programming
  - one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

Historical Origins

- Mathematicians established a distinction between
  - *constructive* proof (one that shows how to obtain a mathematical object with some desired property)
  - *nonconstructive* proof (one that merely shows that such an object must exist, e.g., by contradiction)
- Logic programming is tied to the notion of constructive proofs, but at a more abstract level
  - the logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs

Functional Programming Concepts

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language
- The key idea: do everything by composing functions
  - no mutable state
  - no side effects

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - serious polymorphism
  - powerful list facilities
  - structured function returns
  - fully general aggregates
  - garbage collection

Functional Programming Concepts

- So how do you get anything done in a functional language?
  - Recursion (especially tail recursion) takes the place of iteration
  - In general, you can get the effect of a series of assignments

```
x := 0          ...
x := expr1      ...
x := expr2      ...
```

from f3(f2(f1(0))), where each f expects the value of x as an argument, f1 returns expr1, and f2 returns expr2

Functional Programming Concepts

- Recursion even does a nifty job of replacing looping

```
x := 0; i := 1; j := 100;
while i < j do
      x := x + i*j;
      i := i + 1;
      j := j - 1
end while
return x
```

becomes `f(0,1,100)`, where

```
f(x,i,j) == if i < j then
      f (x+i*j, i+1, j-1) else x
```

Functional Programming Concepts

- Thinking about recursion as a direct, mechanical replacement for iteration, however, is the wrong way to look at things
  - One has to get used to thinking in a recursive style
- Even more important than recursion is the notion of ***higher-order functions***
  - Take a function as argument, or return a function as a result
  - Great for building things

Functional Programming Concepts

- Lisp also has the following (which are not necessarily present in other functional languages)
  - –homo-iconography
  - –self-definition
  - –read-evaluate-print
- Variants of LISP
  - –Pure (original) Lisp
  - –Interlisp, MacLisp, Emacs Lisp
  - –Common Lisp
  - –Scheme

Functional Programming Concepts

- Pure Lisp is purely functional; all other Lisps have imperative features
- All early Lisps dynamically scoped
  - Not clear whether this was deliberate or if it happened by accident
- Scheme and Common Lisp statically scoped
  - Common Lisp provides dynamic scope as an option for explicitly-declared *special* functions
  - Common Lisp now THE standard Lisp
    - Very big; complicated (The Ada of functional programming)

Functional Programming Concepts

- Scheme is a particularly elegant Lisp
- Other functional languages
  - ML
  - Miranda
  - Haskell
  - FP
- Haskell is the leading language for research in functional programming

A Bit of Scheme

- As mentioned earlier, Scheme is a particularly elegant Lisp
  - Interpreter runs a read-eval-print loop
  - Things typed into the interpreter are evaluated (recursively) once
  - Anything in parentheses is a function call (unless quoted)
  - Parentheses are NOT just grouping, as they are in Algol-family languages
    - Adding a level of parentheses changes meaning

```
(+ 3 4) ⇒ 7
((+ 3 4))) ⇒ error
(the ' ⇒' arrow means 'evaluates to‘)
```

A Bit of Scheme

- Scheme:
  - Boolean values `#t` and `#f`
  - Numbers
  - Lambda expressions
  - Quoting

```
(+ 3 4) ⇒ 7
(quote (+ 3 4)) ⇒ (+ 3 4)
'(+ 3 4) ⇒ (+ 3 4)
```

  - Mechanisms for creating new scopes

```
(let ((square (lambda (x) (* x x))) (plus +))
(sqrt (plus (square a) (square b))))
```

A Bit of Scheme

- Scheme:
  - conditional expressions

```
(if (< x 0) (- 0 x))          ; if-then
(if  (< x y)  x  y)           ; if-then-else
(if (< 2 3) 4 5) ⇒ 4
(cond
  ((< 3 2) 1)
  ((< 4 3) 2)
  (else 3)) ⇒ 3
```

  - case selection

```
(case month
  ((sep apr jun nov)  30)
  (feb)  28)
  (else  31)
)
```

A Bit of Scheme

- Scheme:
  - Imperative stuff
    - assignments
    - sequencing (begin)
    - iteration
    - I/O (read, display)

A Bit of Scheme

- Scheme standard functions (this is not a complete list):
  - –arithmetic
  - –boolean operators
  - –equivalence
  - –list operators
  - –symbol?
  - –number?
  - –complex?
  - –real?
  - –rational?
  - –integer?

A Bit of Scheme

- expressions
  - –Cambridge prefix notation for all Scheme expressions:

```
(f x1 x2 … xn)

(+ 2 2)                    ; evaluates to 4
(+ (* 5 4) (- 6 2))   ; means 5*4 + (6-2)
(define (Square x) (* x x)) ; defines a fn
(define f 120)             ; defines a global
```

  - –Note: Scheme comments begin with ;

Source: Tucker & Noonan (2007)

A Bit of Scheme

- expression evaluation
  - three steps:
    1. Replace names of symbols by their current bindings.
    2. Evaluate lists as function calls in Cambridge prefix.
    3. Constants evaluate to themselves.

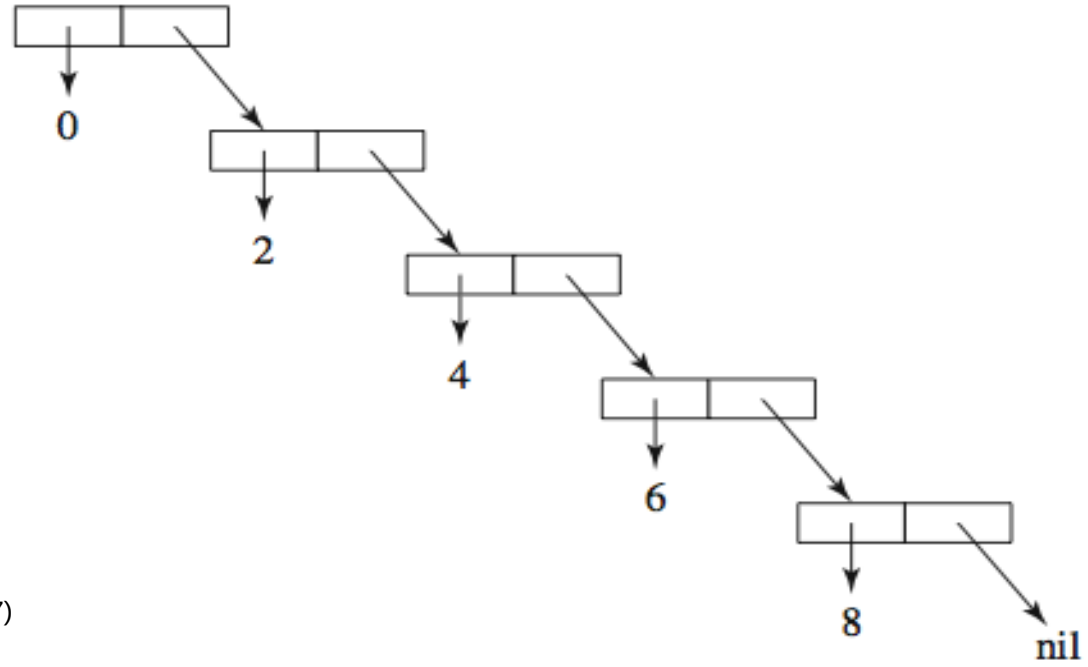
e.g.,

```
x                       ; evaluates to 5
(+ (* x 4) (- 6 2));evaluates to 24
5                       ; evaluates to 5
'red                    ; evaluates to 'red
```

Source: Tucker & Noonan (2007)

A Bit of Scheme

- lists
  - –series of expressions enclosed in parentheses
  - –represent both functions and data
  - –empty list written as ()
  - –e.g., (0 2 4 6 8) is a list of even numbers
    - – stored as



Source: Tucker & Noonan (2007)

A Bit of Scheme

- list transforming functions
  - –using cons (construct):

```
(cons 8 ())                          ; gives (8)
(cons 6 (cons 8 ()))                 ; gives (6 8)
(cons 4 (cons 6 (cons 8 ())))        ; gives (4 6 8)
(cons 4 (cons 6 (cons 8 9))) ; gives (4 6 8 . 9)
```

  - –Note: the last element of a list should be a null list

Source: Tucker & Noonan (2007)

A Bit of Scheme

- list transforming functions
  - –suppose we define the list evens to be (0 2 4 6 8), i.e., we write `(define evens '(0 2 4 6 8))`. Then,

```
(car evens)                    ; gives 0
(cdr evens)                    ; gives (2 4 6 8)
(cons 1 (cdr evens))           ; gives (1 2 4 6 8)
(null? '())                    ; gives #t, or true
(equal? 5 '(5))                ; gives #f, or false
(append '(1 3 5) evens) ; gives (1 3 5 0 2 4 6 8)
(list '(1 3 5) evens)   ; gives ((1 3 5) (0 2 4 6 8))
```

Note: the last two lists are different!

A Bit of Scheme

- more on car/cdr

```
(car (cdr (evens))          ; gives 2
(cadr evens)                ; gives 2
(cdr (cdr (evens))          ; gives (4, 6, 8)
(cddr (evens)               ; gives (4, 6, 8)
(car '(6 8))                ; gives 6
(car (cons 6 8))            ; gives 6
(car '(8))                  ; gives 8
(cdr '(8))                  ; gives ()
```

Source: Tucker & Noonan (2007)

A Bit of Scheme

- defining functions

```
(define (name  arguments) function-body)

(define (min x y) (if (< x y) x y))
(define (abs x) (if (< x 0) (- 0 x) x))

define (factorial n)
   (if (< n 1) 1 (* n (factorial (- n 1))))
))
```

Note: be careful to match all parentheses

Source: Tucker & Noonan (2007)

A Bit of Scheme

- even simple tasks are accomplished recursively

```
((define (mystery1 alist)
 (if (null? alist) 0
        (+ (car alist) (mystery1 (cdr alist)))
))
```

```
(define (mystery2 alist)
 (if (null? alist) 0 (+ 1 (mystery2 (cdr
  alist)))
))
```

Source: Tucker & Noonan (2007)

A Bit of Scheme

- `subst` function

```
(define (subst y x alist)
 (if (null? alist) '())
    (if (equal? x (car alist))
       (cons y (subst y x (cdr alist)))
       (cons (car alist) (subst y x (cdr alist)))
)))
```

e.g., `(subst 'x 2 '(1 (2 3) 2))`

returns `(1 (2 3) x)`

Source: Tucker & Noonan (2007)

A Bit of Scheme

- `let` expressions allow simplification of function definitions by defining intermediate expressions

```
(define (subst y x alist)
   (if (null? alist) '()
  (let ((head (car alist)) (tail (cdr alist)))
     (if (equal? x head)
       (cons y (subst y x tail))
       (cons head (subst y x tail))
)))
```

Source: Tucker & Noonan (2007)

A Bit of Scheme

- functions as arguments
  - applies the function to each member of a list

```
(define (mapcar fun alist)
     (if (null? alist) '()
         (cons (fun (car alist))
               (mapcar fun (cdr alist)))
))
```

e.g., if `(define (square x) (* x x))` then

`(mapcar square '(2 3 5 7 9))` returns

`(4 9 25 49 81)`

Source: Tucker & Noonan (2007)

A Bit of Scheme
Example program - Symbolic Differentiation

- Symbolic Differentiation Rules

$$\frac{d}{dx}(c) = 0 \qquad c \text{ is a constant}$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx} \qquad u \text{ and } v \text{ are functions of } x$$

$$\frac{d}{dx}(u-v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d}{dx}(u/v) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right)/v^2$$

Source: Tucker & Noonan (2007)

A Bit of Scheme
Example program - Symbolic Differentiation

- Scheme encoding
  1. Uses Cambridge Prefix notation

     e.g., `2x + 1` is written as `(+ (* 2 x) 1)`
  2. Function `diff` incorporates these rules.

     e.g., `(diff 'x '(+ (* 2 x) 1))` should give an answer.
  3. However, no simplification is performed.

     e.g. the answer for (diff 'x '(+ (* 2 x) 1)) is

     `(+ (+ (* 2 1) (* x 0)) 0)`

     which is equivalent to the simplified answer, 2

Source: Tucker & Noonan (2007)

A Bit of Scheme
Example program - Symbolic Differentiation

- Scheme program

```
(define (diff x expr)
   (if (not (list? expr))
      (if (equal? x expr) 1 0)
         (let ((u (cadr expr)) (v (caddr expr)))
            (case (car expr)
               ((+) (list '+ (diff x u) (diff x v)))
               ((-) (list '- (diff x u) (diff x v)))
               ((*) (list '+ (list '* u (diff x v))
                          (list '* v (diff x u))))
               ((/) (list 'div (list '- (list '* v (diff x u))
                          (list '* u (diff x v)))
                          (list '* u v)))
))))
```

A Bit of Scheme
Example program - Symbolic Differentiation

- trace of the program

```
(diff 'x '(+ '(* 2 x) 1))
   = (list '+ (diff 'x '(*2 x)) (diff 'x 1))
   = (list '+ (list '+ (list '* 2 (diff 'x 'x))
                       (list '* x (diff 'x 2)))
                       (diff 'x 1))
    = (list '+ (list '+ (list '* 2 1) (list '* x (diff 'x 2)))
                       (diff 'x 1))
    = (list '+ (list '+ '(* 2 1) (list '* x (diff 'x 2)))
                       (diff 'x 1))
    = (list '+ (list '+ '(* 2 1) (list '* x 0))(diff 'x 1))
    = (list '+ (list '+ '(* 2 1) '(* x 0)(diff 'x 1))
    = (list '+ '('+ '(* 2 1) '(* x 0))(diff 'x 1))
    = (list '+ '('+ '(* 2 1) '(* x 0)) 0)
    = '(+ (+ (* 2 1) '(* x 0)) 0)
```

Source: Tucker & Noonan (2007)

A Bit of Scheme
Example program - Simulation of DFA

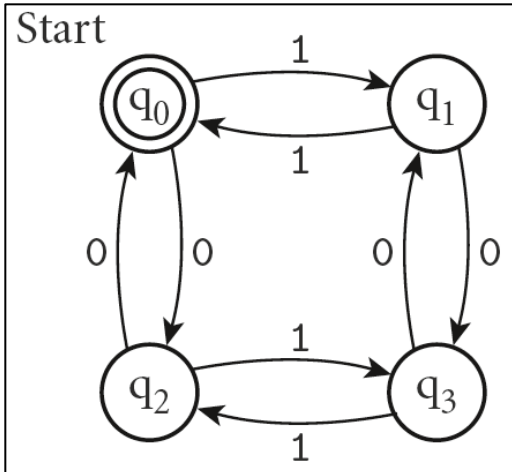
- We'll invoke the program by calling a function called 'simulate', passing it a DFA description and an input string
  - The automaton description is a list of three items:
    - start state
    - the transition function
    - the set of final states
  - The transition function is a list of pairs
    - the first element of each pair is a pair, whose first element is a state and whose second element in an input symbol
    - if the current state and next input symbol match the first element of a pair, then the finite automaton enters the state given by the second element of the pair

A Bit of Scheme
Example program - Simulation of DFA


```
(define simulate
  (lambda (dfa input)
    (letrec ((helper  ; note that helper is tail recursive,
               ; but builds the list of moves in reverse order
               (lambda (moves d2 i)
                 (let ((c (current-state d2)))
                   (if (null? i) (cons c moves)
                       (helper (cons c moves) (move d2 (car i)) (cdr i)))))))
      (let ((moves (helper '() dfa input)))
        (reverse (cons (if (is-final? (car moves) dfa)
                           'accept 'reject) moves))))))

;; access functions for machine description:
(define current-state car)
(define transition-function cadr)
(define final-states caddr)
(define is-final? (lambda (s dfa) (memq s (final-states dfa))))

(define move
  (lambda (dfa symbol)
    (let ((cs (current-state dfa)) (trans (transition-function dfa)))
      (list
       (if (eq? cs 'error)
           'error
           (let ((pair (assoc (list cs symbol) trans)))
             (if pair (cadr pair) 'error))) ; new start state
       trans                                ; same transition function
       (final-states dfa)))))               ; same final states
```

Figure 11.1 Scheme program to simulate the actions of a DFA. Given a machine description and an input symbol *i*, function `move` searches for a transition labeled *i* from the start state to some new state *s*. It then returns a new machine with the same transition function and final states, but with *s* as its "start" state. The main function, `simulate`, encapsulates a tail-recursive helper function that accumulates an inverted list of moves, returning when it has consumed all input symbols. The wrapper then checks to see if the helper ended in a final state; it returns the (properly ordered) series of moves, with `accept` or `reject` at the end. The functions `cadr` and `caddr` are defined as `(lambda (x) (car (cdr x)))` and `(lambda (x) (car (cdr (cdr x))))`, respectively. Scheme provides a large collection of such abbreviations.

A Bit of Scheme
Example program - Simulation of DFA




```
(define zero-one-even-dfa
 '(q0                                                    ; start state
   (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)      ; transition fn
    ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
   (q0)))                                                ; final states
```

Figure 11.2 DFA to accept all strings of zeros and ones containing an even number of each. At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 11.1.

A Bit of OCaml

- OCaml is a descendent of ML, and cousin to Haskell, F#
  - "O" stands for objective, referencing the object orientation introduced in the 1990s
  - Interpreter runs a read-eval-print loop like in Scheme
  - Things typed into the interpreter are evaluated (recursively) once
  - Parentheses are NOT function calls, but indicate tuples

A Bit of OCaml

- Ocaml:
  - Boolean values
  - Numbers
  - Chars
  - Strings
  - More complex types created by lists, arrays, records, objects, etc.
  - (+ - * /) for ints, (+. -. *. /.) for floats
  - `let` keyword for creating new names

```
let average = fun x y -> (x +. y) /. 2.;;
```

A Bit of OCaml

- Ocaml:
  - –Variant Types

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree;;
```

  - –Pattern matching

```
let atomic_number (s, n, w) = n;;
let mercury = ("Hg", 80, 200.592);;
atomic_number mercury;;  ⇒ 80
```

A Bit of OCaml

- OCaml:
  - Different assignments for references `':='` and array elements `'<-'`

```
let insertion_sort a =
        for i = 1 to Array.length a - 1 do
                let t = a.(i) in
                let j = ref i in
                while !j > 0 && t < a.(!j - 1) do
                        a.(!j) <- a.(!j - 1);
                        j := !j - 1
                done;
                a.(!j) <- t
        done;;
```

Example program - Simulation of DFA

- We'll invoke the program by calling a function called 'simulate', passing it a DFA description and an input string
  - The automaton description is a record with three fields:
    - start state
    - the transition function
    - the list of final states
  - The transition function is a list of triples
    - the first two elements are a state and an input symbol
    - if these match the current state and next input, then the automaton enters a state given by the third element

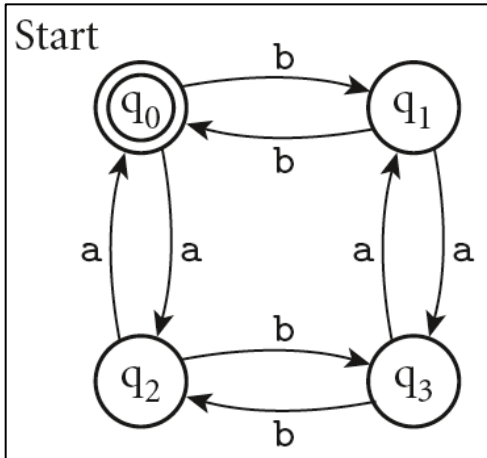
A Bit of OCaml
Example program - Simulation of DFA


```
open List;;     (* includes rev, find, and mem functions *)

type state = int;;
type 'a dfa = {
  current_state : state;
  transition_function : (state * 'a * state) list;
  final_states : state list;
};;
type decision = Accept | Reject;;

let move (d:'a dfa) (x:'a) : 'a dfa =
  { current_state = (
      let (_, _, q) =
        find (fun (s, c, _) -> s = d.current_state && c = x)
             d.transition_function in
      q);
    transition_function = d.transition_function;
    final_states = d.final_states;
  };;

let simulate (d:'a dfa) (input:'a list) : (state list * decision) =
  let rec helper moves d2 remaining_input : (state option * state list) =
    match remaining_input with
    | [] -> (Some d2.current_state, moves)
    | hd :: tl ->
        let new_moves = d2.current_state :: moves in
```

lulate the actions of a DFA. Given a machine description
s transition for a transition labeled c from the start state to
tion raises exception Not_found, which propagates out of
machine with the same transition function and final states,
d the code is polymorphic in the type of the input symbols.
ulates a tail-recursive helper function that accumulates an
n it has consumed all input symbols. The computation
ts ended in a final state, it returns the (properly ordered)
pt, or Reject. Initially, the built-in option constructor
ts a real state (Some s) and an empty list (None).

Figure 11.3 OCaml program to si
and an input symbol x, function move
some new state s. If the search fails, i
move; otherwise, move returns a new
but with s as its "start" state. Note th
The main function, simulate, accepts
inverted list of moves, returning wh
function then checks to see if the hal
series of moves, together with an ex
(Example 7.6) is used to distinguish b

A Bit of OCaml
Example program - Simulation of DFA




```
let a_b_even_dfa : char dfa =
  { current_state = 0;
    transition_function =
      [ (0, 'a', 2); (0, 'b', 1); (1, 'a', 3); (1, 'b', 0);
        (2, 'a', 0); (2, 'b', 3); (3, 'a', 1); (3, 'b', 2) ];
    final_states = [0];
  };;
```

Figure 11.4 DFA to accept all strings of `a`s and `b`s containing an even number of each. At the bottom of the figure is a representation of the machine as an OCaml data structure, using the conventions of Figure 11.3.

Evaluation Order Revisited

- Applicative order
  - what you're used to in imperative languages
  - usually faster
- Normal order
  - like call-by-name: don't evaluate arg until you need it
  - sometimes faster
  - terminates if anything will (Church-Rosser theorem)

Evaluation Order Revisited

- In Scheme
  - functions use applicative order defined with lambda
  - special forms (aka macros) use normal order defined with syntax-rules
- A *strict* language requires all arguments to be well-defined, so applicative order can be used
- A *non-strict* language does not require all arguments to be well-defined; it requires normal-order evaluation

Evaluation Order Revisited

- Lazy evaluation gives the best of both worlds
- But not good in the presence of side effects.
  - delay and force in Scheme
  - delay creates a "promise"

High-Order Functions

- Higher-order functions
  - Take a function as argument, or return a function as a result
  - Great for building things
  - Currying (after Haskell Curry, the same guy Haskell is named after)
    - For details see Lambda calculus on CD
    - ML, Miranda, OCaml, and Haskell have especially nice syntax for curried functions

Functional Programming in Perspective

- Advantages of functional languages
  - lack of side effects makes programs easier to understand
  - lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
  - lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
  - programs are often surprisingly short
  - language can be extremely small and yet powerful

Functional Programming in Perspective

- Problems
  - –difficult (but not impossible!) to implement efficiently on von Neumann machines
    - lots of copying of data through parameters
    - (apparent) need to create a whole new array in order to change one element
    - heavy use of pointers (space/time and locality problem)
    - frequent procedure calls
    - heavy space use for recursion
    - requires garbage collection
    - requires a different mode of thinking by the programmer
    - difficult to integrate I/O into purely functional model