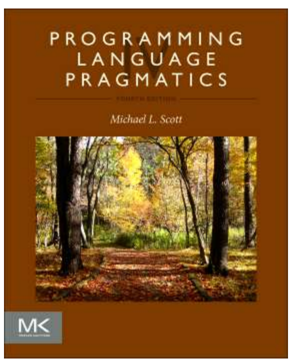
Chapter 11 :: Functional Languages

- CSE307/526: Principles of Programming Languages
- https://ppawar.github.io/CSE307-F18/index.html

Programming Language Pragmatics

Michael L. Scott



Historical Origins

- The imperative and functional models grew out of work undertaken 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 (algebric) manipulation, recursive function definitions, and combinatorics (area of mathematics primarily concerned with counting)
- 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
  - https://www.youtube.com/watch?v=gJQTFhkhwPA

Historical Origins

- Church's model of computing is called the *lambda calculus* (also written as **λ-calculus**)
  - is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution
  - based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter *λ*—hence the notation's name)
  - It is a universal model of computation that can be used to simulate any Turing machine

λ-calculus

- Terms are built using only the following rules producing expressions such as: producing expressions such as: $(\lambda x.\lambda y.(\lambda z.(\lambda x.z\ x)\ (\lambda y.z\ y))\ (x\ y))$

Syntax	Name	Description
x	Variable	A character or string representing a parameter or mathematical/logical value
(λx.M)	Abstraction	Function definition (M is a lambda term). The variable x becomes bound in the expression.
(M N)	Application	Applying a function to an argument. M and N are lambda terms.

- alpha equivalence: λa.a = λb.b
- beta substitution: (λa.aa) b = bb

- https://www.youtube.com/watch?v=eis11j_iGMs

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
- So how do you get anything done in a functional language?
  - Recursion takes the place of iteration
  - First-call functions take value inputs
  - Higher-order functions take a function as input

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

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - Extensive polymorphism – use function on as general a class of arguments
  - powerful list facilities
  - structured function returns – return structured types such as arrays from functions
  - Constructors (aggregates) for structured objects - newly created ones are initialized "all at once."
  - garbage collection

LISP (List Processing) family of languages

- 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 are statically scoped
  - –Common Lisp provides dynamic scope as an option for explicitly-declared special functions
  - –Common Lisp now THE standard Lisp
    - •Very big; complicated

ELSEVIER

LISP languages

- All of them use (symbolic) s-expression syntax: (+ 1 2).
- LISP is old - dates back to 1958 - only Fortran is older.
- Anything in parentheses is a function call (unless quoted)
  - (+ 1 2) evaluates to 3
  - (* 5 (+ 7 3)) evaluates to 50.
  - ((+ 1 2)) <- error, since 3 is not a function.
  - by default, s-expressions are evaluated. We can use the quote special form to stop that: (quote (+ 1 2))
  - Short form: ‘(+ 1 2) is a list containing +, 1, 2

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)
  - Names: Scheme is generally a lot more liberal with the names it allows:
    - foo? bar+ baz- <--- all valid names
    - x$_%L&=*! <--- valid name
      - names by default evaluate to their value

A Bit of Scheme

- Conditional expressions:
  - **(if a b c)** = if a then b else c
  - Example: (if (< 2 3) 4 5) ⇒ 4
  - Example 2: only one of the sub-expressions evaluates (based on if the condition is true):

(if (> a b) (- a 100) (- b 100))

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

A Bit of Scheme

- Lamba expressions:
  - (lambda (x) (* x x))
  - We can apply one or more parameters to it:

  ((lambda (x) (* x x)) 3)

  (* 3 3)

  9
- Bindings: (let ((a 1)   (b 2))   (+ a b))
  - in let, all names are bound at once. So if we did:

  (let ((a 1)   (b a))   (+ a b))
  - we'd get name from outer scope. It prevents recursive calls.
  - letrec puts bindings into effect while being computed (allows for recursive calls):

  (letrec ((fac (lambda (x) (if (= x 0) 1 (* x (fac (- x 1))))))) (fac 10))

A Bit of Scheme

- Control-flow:
  - (begin (display "foo") (display "bar"))
- Special functions:
  - eval = takes a list and evaluates it.

  A list: '(+ 1 2) -> (+ 1 2)

  Evaluation of a list: (eval '(+ 1 2)) -> 3
  - apply = take a lambda and list: calls the function with the list as an argument.

A Bit of Scheme

- Evaluation order:
  - applicative order:
    - evaluates arguments before passing them to a function:

```
((lambda (x) (* x x)) (+ 1 2))
((lambda (x) (* x x) 3)
(* 3 3)
9
```

  - normal order:
    - passes in arguments before evaluating them:

```
((lambda (x) (* x x)) (+ 1 2))
(* (+ 1 2) (+ 1 2))
(* 3 3)
9
```

  - Note: we might want normal order in some code.

```
(if-tuesday (do-tuesday))  // do-tuesday might print something and we want it only if it's Tuesday
```

A Bit of Scheme

- ((lambda (x y) (if x (+ y y) 0) t (* 10 10))
- Applicative order:

  ((lambda (x y) (if x (+ y y) 0) t 100)
  (if t (+ 100 100) 0)
  (+ 100 100)
  200
  – (four steps !)

- Normal Order:

  (if t (+ (* 10 10) (* 10 10)) 0)
  (+ (* 10 10) (* 10 10))
  (+ 100 (* 10 10))
  (+ 100 100)
  200
  – (five steps !)

A Bit of Scheme

- What if we passed in nil instead?
- ((lambda (x y) (if x (+ y y) 0) nil (* 10 10))
- Applicative:

  ((lambda (x y) (if x (+ y y)) nil 100)

  (if nil (+ 100 100) 0)

  0

  – (three steps!)
- Normal

  (if nil (+ (* 10 10) (* 10 10)) 0)

  0

  – (two steps)
- Both applicative and normal order can do extra work!
- Applicative is usually faster, and doesn't require us to pass around closures all the time.

A Bit of Scheme

- Strict vs Non-Strict:
  - We can have code that has an undefined result.
    - (f) is undefined for

```
(define f (lambda () (f))) - infinite recursion
(define f (lambda () (/ 1 0)) - divide by 0.
```

  - A pure function is:
    - strict if it is undefined when any of its arguments is undefined,
    - non-strict if it is defined even when one of its arguments is undefined.
  - Applicative order == strict.
  - Normal order == can be non-strict.
  - ML, Scheme (except for macros) == strict.
  - Haskell == nonstrict.

Strict vs. Non-Strict Example

the following definition in Haskell:

```
noreturn :: Integer -> Integer
noreturn x = negate (noreturn x)
```

or the following Python function:

```
def noreturn(x):
    while True:
        x = -x

    return x # not reached
```

both fail to produce a value when executed.

- The following expression fails in Python (strict):
  2 in [2,4,noreturn(5)] -> innermost-first evaluation
- In Haskell the following expression returns True (non-strict):
  elem 2 [2, 4, noreturn 5] -> outermost-first evaluation (also called lazy evaluation)

Lazy Evaluation – Scheme Example

- Combines non-strictness of normal-order evaluation with the speed of applicative order.
- Idea: - Pass in closure. - Evaluate it once. - Store result in memo. - Next time, just return memo.
- Memoization refers to caching results of previous computations
- Example 1: ((lambda (a b) (if a (+ b b) nil)) t (expensivefunc))

(if t (+ (expensivefunc) (expensivefunc)) nil)
(+ (expensivefunc) (expensivefunc))
(+ 42 (expensivefunc)) <- takes a long time.
(+ 42 42) <- very fast.
84

- Example2: ((lambda (a b) (if a (+ b b) nil)) nil (expensivefunc))

(if nil (+ (expensivefunc) (expensivefunc)) nil)
nil ➔ never evaluated expensivefunc! win!

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

Currying

A common operation, named for logician Haskell Curry, is to replace a multiargument function with a function that takes a single argument and returns a function that expects the remaining arguments:

```
(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4)                          ⟹ 7
(define plus-3 (curried-plus 3))
(plus-3 4)                                    ⟹ 7
```

Among other things, currying gives us the ability to pass a "partially applied" function to a higher-order function:

```
(map (curried-plus 3) '(1 2 3))               ⟹ (4 5 6)
```

- Some languages use currying as their main function-calling semantics (ML): **fun add a b : int = a + b;** ML's calling conventions make this easier to work with: **add 1**
  add 1 2 (There's no need to delimit arguments.)

Pattern Matching

- It's common for FP languages to include pattern matching operations:
  - matching on value,
  - matching on type,
  - matching on structure (useful for lists).

–ML example:

```
fun sum_even l =
    case l of
            nil => 0
            | b :: nil => 0
            | a :: b :: t => h + sum_even t;
```

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
  - http://xahlee.info/ocaml/ocaml_list.html

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;;
```

https://v1.realworldocaml.org/v1/en/html/variants.html

```
type <variant> =
  | <Tag> [ of <type> [* <type>]... ]
  | <Tag> [ of <type> [* <type>]... ]
  | ...

type basic_color =
 | Black | Red | Green | Yellow | Blue | Magenta | Cyan | White ;;
```

  - Pattern matching

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

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
    - 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

More….

- Go through the following programs for your own interests

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

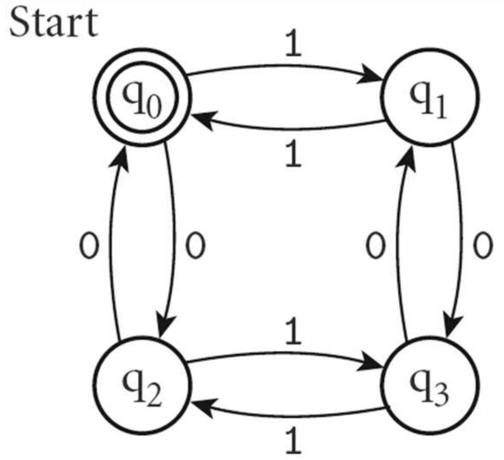
Example program - Simulation of DFA

```
(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

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

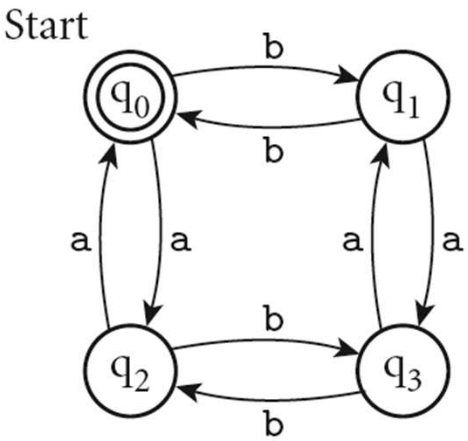
Example program - Simulation of DFA

```
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
        try helper new_moves (move d2 hd) tl
        with Not_found -> (None, new_moves) in
  match helper [] d input with
  | (None, moves) -> (rev moves, Reject)
  | (Some last_state, moves) ->
      ( rev (last_state :: moves),
        if mem last_state d.final_states then Accept else Reject);;
```

Figure 11.3 OCaml 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*. If the search fails, `find` raises exception `Not_found`, which propagates out of `move`; otherwise `move` returns a new machine with the same transition function and final states, but with *s* as its "start" state. Note that the code is polymorphic in the type of the input symbols. 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 encapsulating function then checks to see if the helper ended in a final state; it returns the (properly ordered) series of moves, together with an `Accept` or `Reject` indication. The built-in `option` constructor (Example 7.6) is used to distinguish between a real state (`Some s`) and an error state (`None`).

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 `as` and `bs` 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.

Concluding remarks (courtesy: Paul Fodor)

Conclusion for this course:

- That is all!
- Where languages will be going: languages that combine:
  - Multiparadigm
  - High-level data structures
  - With: speed, simplicity (dynamic weakly typed)
- JavaScript frameworks, node.js, Google Go, Swift, and what else?
  - More and more languages every day!!! What should we learn? All!
    - Youtube is implemented with Python,
    - IBM Watson uses Prolog,
    - Wikipedia is implemented with PHP,
    - Microsoft F# is a functional programming language, etc.
  - More Scripting Languages: Writing programs by coordinating pre-existing components, rather than writing components from scratch.

Concluding remarks

- I'm hoping that this course prepared you for the change the future will bring in programming languages!
- Thank you!! 감사합니다

- Fill in the course review form
- https://stonybrook.campuslabs.com/courseeval/