Lecture 11 – Part I

Introduction to Functional Programming

Introduction

x = x + 1 (Java) vs.
x = x + 1 (mathematical equation)

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*

  e.g. cube (x) = x * x * x

- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

  λ(x) x * x * x

  for the function cube(x) = x * x * x

Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., `(λ(x) x * x * x)(2)`

  which evaluates to `8`

e.g. Python

```
>>>(lambda x: x * x * x) (2)
>>>  8
```

Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both

$$HoF(f, g, x) \equiv f(g(x)+5)$$

Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both

$$\text{HoF}(f, g, x) \equiv f(g(x)+5)$$

function as parameter

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

  Form: $h \equiv f \circ g$

  which means $h(x) \equiv f(g(x))$

  For $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$,

  $h \equiv f \circ g$ yields $(3 * x) + 2$

Function composition is one (simple) type of functional form

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

  Form: α

  For $h(x) \equiv x * x$

  `α(h, (2, 3, 4))` yields `(4, 9, 16)`

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

  Form: α

  For $h(x) \equiv x * x$

  `α(h, (2, 3, 4))` yields `(4, 9, 16)`

Python map:

`map(h, [2,3,4]) => [4,9,16]`

Recursion

A factorial function

F(n):

```
if n <= 1 return 1;          //0! = 1
else return n*F(n-1);        //n! = n*(n-1)!
```

F (4) => 4 * F(3)   F(3) => 3 * F(2)   F(2) =>2*F(1)

F(1) = 1 (end of recursion)

Now, backward computation

F(2) = 2*1 = 2   => F(3) = 3*2 = 6   => F(4) = 4*6 = 24

Tail Recursion

- What is tail recursion?
  - A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function is more efficient
  - It can be automatically converted by a compiler to a non-recursive version, i.e. using iteration
- The factorial function defined previously is NOT in tail recursion.
  - The last operation is multiplication, not recursive call.

Convert to tail recursion w/ a helper()

Tail recursion

```
factorial (n) = helper (n, 1)

helper (n, s):
    if n <= 1 return s
    else return helper (n-1, n * s)
                //the last operation is a recursive call!
```

Convert to tail recursion w/ a helper()

Tail recursion

```
factorial (n) = helper (n, 1)

helper (n, s):
    if n <= 1 return s
    else return helper (n-1, n * s)
                //the last operation is a recursive call!
```

F(4) => h(4, 1)
h(4,1)
$\Rightarrow$ h(3,4*1)
$\Rightarrow$ h(2, 3*4)
$\Rightarrow$ h(1,2*12)
$\Rightarrow$ 24

Are the following tail recursions?

$F(n) = F(n-1) + F(n-2)$

$F(1) = 1$

$F(2) = 1$

$G(n) = G(n-2)$ $n > 10$

$G(n) = G(n-1)$ n in $[1, 10]$

$G(n) = 3$ $n <= 0$

Are the following tail recursions?

$F(n) = F(n-1) + F(n-2)$ //not tail recursion

$F(1) = 1$

$F(2) = 1$

$G(n) = G(n-2)\ n > 10$ //yes

$G(n) = G(n-1)\ n \text{ in } [1, 10]$

$G(n) = 3\ \ n <= 0$

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics
- *Referential Transparency* – In an FPL, the evaluation of a function always produces the same result given the same parameters

Reference Transparency – Side effect free

```
a = 3
y = f(a)
print(y)

b=3
z = f(b)
print(z)

//y equal z?
```

```
a = 3
y = f(a)
print(y)

b=3
z = f(b)
print(z)

//y=?
//z=?
```

```
extern int sum = 0;
          //a global variable

int f (int x) {
    sum += 1;
    return x + sum;
}
```

```
a = 3
y = f(a)
print(y)

b=3
z = f(b)
print(z)

//y=4
//z=5
```

```
extern int sum = 0;
          //a global variable

int f (int x) {
    sum += 1;
    return x + sum;
}
```

Characteristics of FP languages

- Functions
  - Built-in, user-defined, lambda expressions
- Recursions
- Higher-Order Functions
- Reference transparency
  - No Side-effects
- Lazy Evaluation
- Modularity

Introduction to Lisp

- Lisp (historically LISP)
  - LISt Processing
- Designed by John McCarthy, MIT
  - 1958; 62 years ago (one year younger than FORTRAN)
- Many dialects
  - Common Lisp: a lot of imperative features introduced
  - Scheme: 1970s
    - designed to be a cleaner, more modern, and simpler version
- Pure Lisp
  - Commonly refer to the pure functional programming features in Lisp

Lisp: Program and Data

- Lisp unifies the program and data
  - Data: atom (primitive values) and list
    - List: (3 (2 4) (5 2 45))
  - A Lisp program is represented by a list

    i.e. a program could serve as a data object
    - Able to write a Lisp interpreter using Lisp

  e.g. A sample Lisp program

```
(defun sqr (x ) (* x x))
(sqr 4)  ;;that will return 16

(setq prog (defun sqr(x) (* x x)))
(car prog)   ;;that will return the first item of prog, i.e. defun
```

Lisp: Expressions and Program Style

- Prefix notation

```
(+ (* 3 5) 7)
(> 3 (- a 2))
```

- Conditional form

```
(cond ((> a b) (+ a b))
      (T (- a b)))  ;;if (a > b) a + b else a – b
```

- Lots of parenthesis

```
(defun myFun (x  y)
      (cond ((> x y) (+ x y))
            (T (- x y))))
```

`LAMBDA` Expressions

- Form is based on λ notation

  e.g., `(LAMBDA (x) (* x x)`

  `x` is called a bound variable

- Lambda expressions can be applied to parameters

  e.g., `((LAMBDA (x) (* x x)) 7)` => 49

- `LAMBDA` expressions can have any number of parameters

  `(LAMBDA (a b x) (+ (* a x x) (* b x)))`

Output Functions

- Usually not needed, because the interpreter always displays the result of a function evaluated at the top level (not nested)
- Scheme has `PRINTF`, which is similar to the `printf` function of C
- Note: explicit input and output are not part of the pure functional programming model, because input operations change the state of the program and output operations are side effects

Control Flow

- A number of forms, COND, IF, UNLESS, …
  - Most popular COND

```
(COND ((> a b) 'large)
      ((= a b) 'same)
      (T     'small))
```

(if *a b c*) == (cond (*a b*) (T *c*))

- Loops

  Do loops are most popular, however loops are discouraged in functional programming

  Use recursion

Function Definition

```
(defun member (atm a_list)
  ;;check if an atm is in a_list or not
  (COND
     ((NULL a_list) 'nil)
     ((EQ atm (CAR a_list)) 'T)
     (T (member atm (CDR a_list)))
  ))
```

Using Lisp Interpreter

```
>(defun square (x) (* x x))
SQUARE
>(square 2)
4
>(square 1.4142158)
2.0000063289696399
```

Higher order functions

Passing functions as arguments

- Apply
  - Can be given a number of arguments, but the last one must be a list

```
>(apply #'+ '(1 2 3))
6
```

Funcall

  - Does the same thing as *apply* but doesn't need the arguments to be packed in a list

```
>(defun add2 (x) (+ x 2))
>(funcall #'add2 5)
>7
```

Lambda expression as function argument

- Examples

```
>(funcall #‘(lambda (N) (+ 1 N)) 3)
4

>(apply #‘(lambda (A B C) (* A (+ B C))) ‘(4 3 5))
32
;;make sure you use #’ in front of the function
parameter name
```

Mapping Functions

- Mapping function
  - Applied successively to elements of one or more lists
- Mapcar

```
>(mapcar #'numberp '(A 3  B 2 4))
 (nil T nil T T)

>(mapcar #'(lambda (n) (+ 1 n)) '(5 3 6 7 2))
(6 4 7 8 3)
```

Mapcar: More Examples

```
(defun double (x) (* 2 x))
(mapcar #'double '(1 2 3))
   => (2 4 6)
```

More mapping functions: maplist, mapc, …

```
(mapcar #'sqrt '(1 2 3 4))
        => (1 1.4132135 1.7320508 2)

(mapcar #'oddp '(1 2 3)) => (T nil T)

(mapcar #'(lambda (x) (1+ (* 2 x))) '(1 2 3))
        => (3 5 7)
```

Recursive Schemata

- Families of recursive functions which may help in solving Lisp problems.

- Each schema (plural schemata) represents a whole family of recursive functions which are similar in code.

OP-All (operate-all)

```
(defun OP-All (L)
      (cond ((null L) nil)
            (T (cons (f (car L)) (OP-All (cdr L))))))
```

- Examples
  - (SQ-ALL '(3 1 4 1)) -> (9 1 16 1)
  - (INC-ALL '(3 1 4 1)) -> (4 2 5 2)
  - (LISTIFY-ALL '(A 2 B)) -> ((A) (2) (B))
  - (Double-ALL '(3 1 4 1) -> (6 2 8 2)

OP-Some (operate-some)

```
(defun OP-SOME (L)
    (cond ((null L) nil)
          ((test (car L))
              (cons (f (car L)) (OP-SOME (cdr L))))
          (T  (cons (car L) (OP-SOME (cdr L))))))
```

- Examples

  (SQ-ODD ‘(3 5 4 7)) -> (9 25 4 49)

  (INC-ODD ‘(3 5 4 7)) -> (4 6 4 8)

  More: DOUBLE-EVEN, INC#, …

KEEP-SOME/DELETE-SOME

- KEEP-SOME/DELETE-SOME takes a list and returns another list, keeping/deleting some of the elements

  (KEEP-ODD '(3 1 4 1)) -> (3 1 1)

  (TOSS-ODD '(3 1 4 1)) -> (4)

More: KEEP-EVEN, KEEP#, KEEP-PS (perfect square), DEL-ODD, …

Other Functional Languages

- ML
- Haskell
- F#
- …

F#

- Based on Ocaml, which is a descendant of ML and Haskell
- Fundamentally a functional language, but with imperative features and supports OOP
- Has a full-featured IDE, an extensive library of utilities, and interoperates with other .NET languages
- Includes tuples, lists, discriminated unions, records, and both mutable and immutable arrays
- Supports generic sequences, whose values can be created with generators and through iteration

F# (continued)

- Sequences

```
let x = seq {1..4};;
```

  - Generation of sequence values is lazy

    ```
    let y = seq {0..10000000};;
    ```

    Sets `y` to `[0; 1; 2; 3;…]`

  - Default stepsize is `1`, but it can be any number

    ```
    let seq1 = seq {1..2..7}
    ```

    Sets `seq1` to `[1; 3; 5; 7]`

  - Iterators – not lazy for lists and arrays

```
let cubes = seq {for i in 1..4 -> (i, i * i * i)};;
```

Sets cubes to `[(1, 1); (2, 8); (3, 27); (4, 64)]`

F# (continued)

- Functions
  - If named, defined with `let`; if lambda expressions, defined with `fun`

```
(fun a b -> a / b)
let doubleIt (x : int) = 2 * x
```

  - No difference between a name defined with `let` and a function without parameters
  - The extent of a function is defined by indentation
  - If a function is recursive, its definition must include the `rec` reserved word

F# (continued)

- Why F# is Interesting:
  - It builds on previous functional languages
  - It supports virtually all programming methodologies in widespread use today
  - It is the first functional language that is designed for interoperability with other widely used languages
  - At its release, it had an elaborate and well-developed IDE and library of utility software

Multi-Paradigm Languages

- Support for functional programming is increasingly creeping into imperative languages
  - C#, Python, Ruby, …
  - Features: e.g. map, anonymous functions, …
  - e.g. Anonymous functions (lambda expressions)
    - JavaScript: leave the name out of a function definition
    - C#: `i => (i % 2) == 0` (returns true or false depending on whether the parameter is even or odd)
    - Python: `lambda a, b : 2 * a - b`

Functional vs. Imperative Languages

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
- Functional Languages:
  - Simple semantics
  - Simple syntax
  - Less efficient execution
  - Programs can automatically be made concurrent

Lecture 11 – Part II
Logic Programming and Prolog

A Brief Introduction

Introduction: Logic Programming Languages

- Characteristics of Logic programming languages
  - Programs are expressed in a form of symbolic logic
  - Use a logical inferencing process to produce results
  - *Declarative* rather that *procedural*:
    - Only specification of *results* are stated (not detailed *procedures* for producing them)
    - A built-in Engine for the language will conduct inferences and produce results
    - Programmer's focus: logical presentation of the problem, not detailed solution (i.e. algorithms are not needed.)

Example: Sorting a List

- Describe the *characteristics* of a sorted list, not the *process* of rearranging a list

sort(old_list, new_list):

permute (old_list, new_list) $\cap$ sorted (new_list)

sorted (list):

$\forall_j$ such that $1 \leq j < n$, list(j) $\leq$ list (j+1)

Programmer: write problem specification.
System: solve the problem for you.

Main Components of Logic Programs

- Proposition
  - Facts
    - Objects, terms, compound terms, …
  - Query
    - Resolution
- Symbolic logic: Behind the engine!
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions

Example

```
factorial(0,1).   /*a fact: 0! is 1.*/

factorial(N,F) :- N>0, N1 is N-1,
                  factorial(N1,F1), F is N * F1.
      /*a rule to calculate F = N!*/
```

Now, how the ”Engine behind” calculates factorial(N,F), say factorial(4,F), i.e. F = 4!?

Resolution

- Resolution: a key feature that needs to be supported by logic language interpreter. It includes
  - *Unification*: finding values for variables in propositions that allows matching process to succeed
  - *Instantiation*: assigning temporary values to variables to allow unification to succeed
  - After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

e.g. Q(X) :- x + 1 /* what is Q(3)? Here, x will be unified with 3 */
like(X, Y) :- friends (X, Z), like (Z, Y)
like (mary, fb)? /*does Z exist? Or can you instantiate Z to some value?

Backtracking

8-Queen Problem (illustrated by 4-Queen)

	Q		
			Q
Q			
		Q	

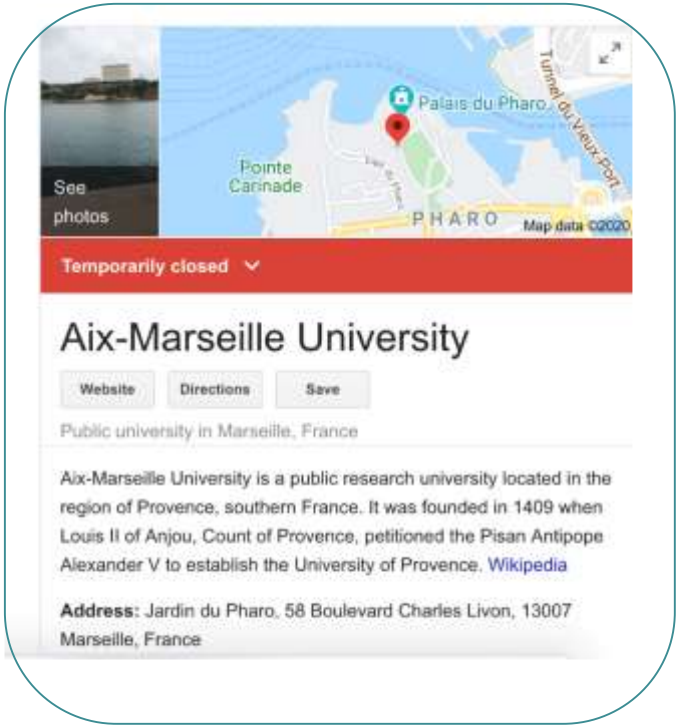
		Q	
Q			
			Q
	Q		

Logic Programming: Summary

- Declarative semantics
  - There is a simple way to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state how a result is to be computed, but rather the form of the result

The Origins of Prolog

- University of Aix-Marseille (Calmerauer & Roussel)
  - Natural language processing
- University of Edinburgh (Kowalski)
  - Automated theorem proving



Prolog Basic Components

- Facts: used for the hypotheses

  student(shelley).

  student (bill).

  likes(bill, fb).

  likes (shelley, yt).

- Rules
  - Right side: *antecedent* (*if* part)
    - May be single term or conjunction
  - Left side: *consequent* (***then*** part)
    - Must be single term

A :- B
if B then A

Example Rules

```
friend(bill,shelley):-
      likes(bill,fb), likes(shelley,fb).
```

- Can use variables (*universal objects*) to generalize meaning:

```
parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).
grandparent(X,Z):- parent(X,Y), parent(Y,Z).
```

Goal Statements

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement*
- Example

  ?- friend(bill,shelley). %are bill and shelley friends?

- Conjunctive propositions and propositions with variables also legal goals

  ?- factorial(3,W). %W is 3!, what is W?

  W=6 %answer by prolog engine

Facts (database) and Rules

```
father(jack, susan).                /* Fact  1 jack is father of susan*/
father(jack, ray).                  /* Fact  2 */
father(david, mary).                /* Fact  3 */
father(david, john).                /* Fact  4 */
mother(karen, susan).               /* Fact  5 */
mother(karen, ray).                 /* Fact  6 */
mother(susan, mary).                /* Fact  7 */
mother(susan, john).                /* Fact 8 */

parent(X, Y) :- father(X, Y).       /* Rule  1 */
parent(X, Y) :- mother(X, Y).       /* Rule  2 */
grandparent(X, Y) :- parent(X, Z), parent(Z, Y). /* Rule  3 */

?- parent(susan, mary).     /*is susan a parent of mary? */
yes
?- parent(ray, john).
no
?- parent (X, susan).
jack
karen
no
```

Inferencing Process of Prolog

- Queries are called goals
- If a goal is a compound proposition, each of the facts is a subgoal
- To prove a goal is true, must find a chain of inference rules and/or facts. For goal Q:

```
P2 :- P1
P3 :- P2
...
Q :- Pn
```

- Process of proving a subgoal called matching, satisfying, or resolution

Approaches

- *Matching* is the process of proving a proposition
- Proving a subgoal is called *satisfying* the subgoal
- *Bottom-up resolution, forward chaining*
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
  - Works well with a large set of possibly correct answers
- *Top-down resolution, backward chaining*
  - Begin with goal and attempt to find sequence that leads to set of facts in database
  - Works well with a small set of possibly correct answers
- Prolog implementations use backward chaining

Subgoal Strategies

- Breadth-first search: work on all subgoals in parallel
- Depth-first search: find a complete proof for the first subgoal before working on others
  - Prolog uses depth-first search
    - Can be done with fewer computer resources
- Backtracking
  - With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution.
  - Begin search where previous search left off
  - Can take lots of time and space because may find all possible proofs to every subgoal

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-   speed(X,Speed),
                     time(X,Time),
                     Y is Speed * Time.
```

Simple unification and instantiation, no backtracking.

A query: `?- distance(chevy, Chevy_Distance).`

Example: With backtracking

```
father(jack, susan).          /* Fact  1 jack is father of susan*/
father(jack, ray).            /* Fact  2 */
father(david, mary).          /* Fact  3 */
father(david, john).          /* Fact  4 */
mother(karen, susan).         /* Fact  5 */
mother(karen, ray).           /* Fact  6 */
mother(susan, mary).          /* Fact  7 */
mother(susan, john).          /* Fact 8 */

parent(X, Y) :- father(X, Y).          /* Rule  1 */
parent(X, Y) :- mother(X, Y).          /* Rule  2 */
grandparent(X, Y) :- parent(X, Z), parent(Z, Y). /* Rule  3 */

?- grandparent (X, mary).
```

mary -> david -> no (now, backtracking)

mary -> susan -> jack (backtracking again)

mary -> susan -> karen

Deficiencies of Prolog

- Resolution order control
  - In a pure logic programming environment, the order of attempted matches is nondeterministic and all matches would be attempted concurrently
- The closed-world assumption
  - The only knowledge is what is in the database
- The negation problem
  - Anything not stated in the database is assumed to be false
- Intrinsic limitations
  - It is easy to state a sort process in logic, but difficult to actually do—it doesn’t know how to sort

Applications of Logic Programming

- Relational database management systems
- Expert systems
- Natural language processing

- New paradigm: declarative programming