# Programming Languages:

# Lecture 9

# Chapter 15: Functional Programming Languages

**Jinwoo Kim**
**jwkim@jjay.cuny.edu**

# *Chapter 15 Topics*

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
- The First Functional Programming Language: LISP
- Introduction to Scheme
- COMMON LISP
- ML
- Haskell
- Applications of Functional Languages
- Comparison of Functional and Imperative Languages

# *Introduction*

- 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*

- A *lambda calculus (λ−calculus)*
  - Introduced in 1930's by Church and Kleene as part of investigation into the foundations of mathematics
  - Emerged as a useful tool in the investigation of problems in computability and recursion theory
  - Forms the basis of a paradigm of "Functional Programming"
    - **Primary features of Functional Programs**
      - **Stateless**
      - **Deals exclusively with functions which accepts and return data (including other functions)**
      - **Produce no side effects in "state" and do not alter "incomiing data"**
  - Most modern functional languages built on λ−*calculus*
    - **Lisp, Scheme, ML and Haskell**

# *Lambda Expressions*

- Every expression is a "unary function"
  - it accepts a single input ("argument") and returns a single value ("result")
  - Since every expression is a "unary function", every argument and result are functions too
    - **This makes λ–*calculus quite interesting and unique within both computation and mathematics***

- Example

$$(x, y) \mapsto x \times x + y \times y$$

$$((x, y) \mapsto x \times x + y \times y)(5, 2)$$
$$= 5 \times 5 + 2 \times 2 = 29$$

$$x \mapsto (y \mapsto x \times x + y \times y)$$

$$((x \mapsto (y \mapsto x \times x + y \times y))(5))(2)$$
$$= (y \mapsto 5 \times 5 + y \times y)(2)$$
$$= 5 \times 5 + 2 \times 2 = 29$$

# Lambda Expressions (Continued)

- A function is anonymously defined in Lambda expressions
  - Nameless functions
  - Example:

$$sqsum(x, y) = x \times x + y \times y$$
$$(x, y) \mapsto x \times x + y \times y$$

- 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`

# *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
  - Composition
  - Apply-to-all

- Higher-order functions are closely related to *first-class functions*
  - mathematical concept of functions that operate on other functions, while "*first-class*" is a computer science term that describes programming language entities that have no restriction on their use
  - *first-class functions* can appear anywhere in the program that other first-class entities like numbers can, including as arguments to other functions and as their return values

# *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 $(3 * x) + 2$

# *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: $\alpha$

For `h(x) ≡ x * x`

`α(h, (2, 3, 4))` yields `(4, 9, 16)`

# Fundamentals: 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*

- An expression is said to be ***referentially transparent*** if it can be replaced with its value without changing the program
  - in other words, yielding a program that has the same effects and output on the same input (no side effect)
  - E.g., compare *++x* and *int plusone(int x) { return x+1;}*

- In an FPL, the evaluation of a function always produces the same result given the same parameters

# *Referential Transparency*

- The chief advantage of writing a code in a referentially transparent style
  - Better static code analysis and code-improving transformations
  - E.g., expensive function call inside a loop

- Primary disadvantage from enforcing referential transparency
  - it makes the expression of operations that naturally fit a sequence-of-steps imperative programming style more awkward and less concise

# Coding Styles Comparison

- Imperative programs tend to emphasize the series of steps taken by a program in carrying out an action

```
#include <iostream>

// Fibonacci numbers, imperative style
int fibonacci(int iterations)
{
    int first = 0, second = 1; // seed values

    for (int i = 0; i < iterations; ++i) {
        int sum = first + second;
        first = second;
        second = sum;
    }

    return first;
}

int main()
{
    std::cout << fibonacci(10) << "\n";
    return 0;
}
```

# *Coding Styles Comparison (Continued)*

- Functional programs tend to emphasize the composition and arrangement of functions, often without specifying explicit *steps*

```
unsigned int recFib (unsigned int n) {
    if (n < 2)
        return n;
    else
        return recFib(n-1) + recFib(n-2);
```

# Coding Styles Comparison (Continued)

- Functional programs tend to emphasize the composition and arrangement of functions, often without specifying explicit *steps*

```
(defun fact (x) (if (= x 1) 1 (* x (fact (- x 1)))))
```

```
(fact 10)
```

```
unsigned int factorial(unsigned int N) {
    int fact = 1, i;

    // Loop from 1 to N to get the factorial
    for (i = 1; i <= N; i++) {
        fact *= i;
    }

    return fact;
}
```

# Applications of Functional Languages

- APL is used for throw-away programs
  - APL has long had a small and fervent user base
  - It was and still is popular in financial and insurance applications, in simulations, and in mathematical applications
    - **often where a solution changes frequently or where in a standard language yields excessive complexity**
  - E.g.

$$\sum_{i=1}^{100} i$$

$$(\sim \mathrm{R} \in \mathrm{R} \circ . \times \mathrm{R}) / \mathrm{R} \leftarrow 1 \downarrow \iota \mathrm{R}$$

$$+/\iota 100$$

# APL Example

- Fast Fourier Transformation (FFT)
  - “Programming in APL” by Wolfang K. Giloi, 1977, p.212
  - People thinking of Perl as an unreadable language have obviously never seen any APL code yet

```
    ∇ Z←FFT X;C;D;E;J;K;LL;M;N;O
[1]   LL←⌊2*-O-⍳M←⌊2⍟N,0⍴E←1-2×~O←⍳1.J←⍳L
      ←0,0⍴K←⍳N←¯1↑⍴X
[2]   →(M>L←L+1)/1+⍴⍴J←J,N⍴ 0 1 ∘.=(
      2*L)⍴1
[3]   Z←X[;(L←0)+(⌽LL)+.×J←(M,N)⍴J]
[4]   X← 2 1 ∘.○○(-O-K)÷¯1↑LL
[5]   Z←Z[;K-,LL[L]×J[L;]]+(⍴Z)⍴(-⌿X[;D]×
      Z[;C]),+⌿X[;D←O+N⍴LL[E+M-L]×-O-⍳
      2×LL[L]]×⊖Z[;C←K+,LL[L]×0=J[L;]]
[6]   →((M+O)>L←L+1)/5
    ∇
```

# Applications of Functional Languages (Continued)

- LISP is used for artificial intelligence
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
- Scheme is used to teach introductory programming at a significant number of universities

# *ML*

- A static-scoped functional language
- Uses type declarations, but also does *type inferencing* to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

# *ML Specifics*

- The `val` statement binds a name to a value (similar to `DEFINE` in Scheme)

- Function declaration form:

```
fun name (parameters) = body;
```

e.g., `fun cube (x : int) = x * x * x;`

# Comparing Functional and Imperative Languages

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed

- Functional Languages:
  - Simple semantics
  - Simple syntax
  - Inefficient execution
  - Programs can automatically be made concurrent

# Features of ML

- A pure functional language
  - serious programs can be written without using variables
- Widely accepted
  - reasonable performance (claimed)
  - syntax not as arcane as LISP

# *On this Course,*

- We use Standard ML of New Jersey
  - http://www.smlnj.org/
- Runs on PCs, and lots of other platforms
- See various ML documentation at
  - http://www.standardml.org/

# *Running SML on Windows*

- On Windows, it's invoked from the Programs menu under the Start button
- Also possible to run from MS-DOS prompt,
  - e.g. `C: sml\bin\sml-cm <foo.sml`
  - note that a set of function definitions can be read in this way automatically
- Use control z to exit interpreter

# *Hello, world in SML*

```
Standard ML of New Jersey,
- print("Hello world\n");
Hello world
val it = () : unit
-
```

# *Arithmetic in ML*

- Copy and paste the following text into a Standard ML window

```
2+2;            (* note semicolon at end*)
3*4;
4/3;            (* an error! *)
6 div 2;        (* integer division *)
7 div 3;
```

# *It should look like this*

RUNX86~1

```
Standard ML of New Jersey, Version 110.0.3, January 30, 1998 [CM&CMB]
- 2+2;         (* note semicolon at end of each *)
val it = 4 : int
- 3*4;
val it = 12 : int
- 4/3;          (* an error! *)
stdIn:14.2 Error: overloaded variable not defined at type
  symbol: /
  type: int
- 6 div 2;    (* integer division *)
val it = 3 : int
- 7 div 3;
val it = 2 : int
-
```

# Declaring Constants

- Constants are *not* exactly the same as variables
  - once set, they can't be modified
  - they can be redefined, but existings uses of that constant (e.g. in functions) aren't affected by such redefinition

```
val freezingFahr = 32;
```

# Declaring Functions

- A function takes an input value and returns an output value
- ML will figure out the types

```
fun fahrToCelsius f = (f -freezingFahr) * 5 div 9;
fun celsiusToFahr c = c * 9 div 5 + freezingFahr;
```

RUNX86~1

```
  type: int
- 6 div 2;   (* integer division *)
val it = 3 : int
- 7 div 3;
val it = 2 : int
- val freezingFahr = 32;
val freezingFahr = 32 : int
- fun fahrToCelsius f = (f -freezingFahr) * 5 div 9;
val fahrToCelsius = fn : int -> int
- fun celsiusToFahr c = c * 9 div 5 + freezingFahr;
val celsiusToFahr = fn : int -> int
- fahrToCelsius 0;
val it = ~18 : int
- fahrToCelsius 32;
val it = 0 : int
- GC #0.0.0.0.1.5:   (0 ms)
fahrToCelsius 212;
val it = 100 : int
- celsiusToFahr 0;
val it = 32 : int
- celsiusToFahr 100;
val it = 212 : int
- celsiusToFahr 30;
val it = 86 : int
-
```

# *Notes*

- ML is picky about not mixing types, such as int and real, in expressions
  - Basic types of ML: integer, real, string, char, boolean
  - From basic types, we can construct objects using tuples, lists, functions and records
- The value of “it” is always the last value computed
- Function arguments don’t always need parentheses, but it doesn’t hurt to use them

# *Types of arguments and results*

- ML figures out the input and/or output types for simple expressions, constant declarations, and function declarations

- If the default isn't what you want, you can specify the input and output types, e.g.

```
fun divBy2 x:int = x div 2 : int;
fun divideBy2 (y : real) = y / 2.0;
divBy2 (5);
divideBy2 (5.0);
```

# Two similar divide functions

```
- fun divBy2 x:int = x div 2 : int;
val divBy2 = fn : int -> int

- fun divideBy2 (y : real) = y / 2.0;
val divideBy2 = fn : real -> real

- divBy2 (5);
val it = 2 : int

- divideBy2 (5.0);
val it = 2.5 : real
-
```

# *Ints and Reals*

- Note ~ is unary minus
- min and max take just two input arguments, but that can be fixed!
- Real converts ints to real
- Parens can sometimes be omitted

```
Int.abs ~3;
Int.sign ~3;
Int.max (4, 7);
Int.min (~2, 2);
Real(freezingFahr);
Math.sqrt real(2);
Math.sqrt(real(2));
Math.sqrt(real 3);
```

```
- Int.abs ~3;
val it = 3 : int
- Int.sign ~3;
val it = ~1 : int
- Int.max (4, 7);
val it = 7 : int
- Int.min (~2, 2);
val it = ~2 : int
- real(freezingFahr);
val it = 32.0 : real
- Math.sqrt real(2);
stdIn:57.1-57.18 Error: operator and operand don't agree [tycon mismatch]
  operator domain: real
  operand:         int -> real
  in expression:
    Math.sqrt real
- Math.sqrt(real(2));
val it = 1.41421356237 : real
- Math.sqrt(real 3);
val it = 1.73205080757 : real
-
```

# *Strings*

- Delimited by double quotes
- the caret mark ^ is used for string concatenation, e.g. “house”^”cat”
- \n is used for newline, as in C and C++

# Lists in ML

- Objects in a list must be of the same type
  - [1,2,3];
  - [“dog”, “cat”, “moose”];

- The empty list is written `[]` or `nil`

# *Making Lists*

- The @ operator is used to concatenate two lists of the same type
- The functions hd and tl give the first element of the list, and the rest of the list, respectively

# List Operations

```
- val list1 = [1,2,3];
val list1 = [1,2,3] : int list
- val list2 = [3,4,5];
val list2 = [3,4,5] : int list
- list1@list2;
val it = [1,2,3,3,4,5] : int list
- hd list1;
val it = 1 : int
- tl list2;
val it = [4,5] : int list
```

# Strings and Lists

- The explode function converts a string into a list of characters
- The implode function converts a list of characters into a string
- Examples:

```
- explode("foo");
val it = [#"f",#"o",#"o"] : char list
- implode [#"c",#"a",#"t"];
val it = "cat" : string
-
```

# *Heads and Tails*

- The cons operator :: takes an element and prepends it to a list of that same type
- For example, the expression 1::[2,3] results in the list [1,2,3]
- What’s the value of [1,2]::[ [3,4], [5,6]] ?

# *Functions and Patterns*

- Recall that min and max take just two arguments

- However, using the fact that, for example,
  - min(a, b, c) = min(a, min(b, c))

# Generalizing Min

- An example of ML pattern matching
  - the cons notation x::xs is both a binary constructor *and* a pattern
  - cases aren't supposed to overlap

```
fun multiMin (x: int) = x |
    multiMin (x:int, y:int) = Int.min(x, y) |
    multiMin (x::xs) = Int.min(x, multiMin(xs));

(* What's wrong with above first attempt??? *)
```

# Example

```
fun fact(n : int): int = if n = 0 then 1
                         else n * fact(n-1);
```

```
fun fact(0) = 1
|   fact(n: int): int = n * fact(n-1);
```

# List operations in ML (revisit)

- Lists are represented in square bracket with elements separated by commas
  - [1, 2, 3, 4]
- Empty list is specified by [ ] or nil
  - [1, 2, 3, 4]
- You can construct a list using :: operator
  - 1 :: [2, 3, 4]
- hd takes out head element from list and tl returns remaining list without head element
  - hd  [1 , 2,  3]
  - tl  [1 , 2,  3]

# List operations in ML

```
fun length([]) = 0
|    length(h::t) = 1 + length(t);
```

```
fun append([], list2) = list2
|    append(h::t, list2) = h :: append(t,list2);
```

# Chapter Summary

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution instead of imperative features such as variables and assignments
- LISP began as a purely functional language and later included imperative features
- Scheme is a relatively simple dialect of LISP that uses static scoping exclusively
- COMMON LISP is a large LISP-based language
- ML is a static-scoped and strongly typed functional language which includes type inference, exception handling, and a variety of data structures and abstract data types
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- Purely functional languages have advantages over imperative alternatives, but their lower efficiency on existing machine architectures has prevented them from enjoying widespread use