# Functional Programming Languages

## Chapter 14



## Introduction

- Functional programming paradigm
- History
- Features and concepts
- Examples:
  - Lisp
  - ML



## Functional Programming

- The Functional Programming Paradigm is one of the major programming paradigms.
  - FP is a type of declarative programming paradigm
  - Also known as *applicative programming* and *value-oriented programming*
- Idea: everything is a function
- Based on sound theoretical frameworks (e.g., the lambda calculus)
- Examples of FP languages
  - First (and most popular) FP language: Lisp
  - Other important FPs: ML, Haskell, Miranda, Scheme, Logo



## Functional Programming Languages

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

- Def: A mathematical function is a mapping of members of one set, called the *domain set*, to another set, called the *range set.*
- The lambda calculus is a formal mathematical system devised by Alonzo Church to investigate functions, function application and recursion.
- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

  ?x . x * x * x

  for the function cube (x) = x * x * x
- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  (?x . x * x * x) 3 => 3*3*3 => 27

  (?x,y . (x-y)*(y-x)) (3,5) => (3-5)*(5-3) => -4



## Characteristics of Pure FPLs

Pure FP languages tend to

- Have no side-effects
- Have no assignment statements
- Often have no variables!
- Be built on a small, concise framework
- Have a simple, uniform syntax
- Be implemented via interpreters rather than compilers
- Be mathematically easier to handle



## Importance of FP

- In their pure form FPLs dispense with notion of assignment
  - claim is: it's easier to program in them
  - also: easier to reason about programs written in them
- FPLs encourage thinking at higher levels of abstraction
  - support modifying and combining existing programs
  - thus, FPL's encourage programmers to work in units larger than statements of conventional languages: "programming in the large"
- FPLs provide a paradigm for parallel computing
  - absence of assignment (or single assignment) } provide basis
  - independence of evaluation order } for parallel
  - ability to operate on entire data structures } functional programming



## Importance of FP

- FPLs are valuable in developing *executable specifications* and *prototype implementations*
  - Simple underlying semantics
    - » rigorous mathematical foundations
    - » ability to operate on entire data structures

      ideal vehicle for capturing specifications
- FPLs are very useful for AI and other applications which require extensive symbol manipulation.
- Functional Programming is tied to CS theory
  - provides framework for viewing decidability questions
    - » (both programming and computers)
  - Good introduction to Denotational Semantics
    - » functional in form

# Expressions

- Key purpose of functional programming: to extend the advantages of expressions (over statements) to an entire programming language
- Backus* has said that expressions and statements come from two different worlds.
  - expressions: (a + b) * c arithmetic
    (a + b) = 0 relational
    ¬(a ? b) boolean
  - statements: the usual assortment with assignment singled out
  - *assignments alter the state of a computation* (ordering is important)
    e.g. a:= a * i; i:= i + 1
- In contrast, ordering of expressions is not side-effecting and therefore not order dependent (Church-Rosser property /Church Diamond)

* *"Can programming be liberated from the von Neumann style?"*, John Backus, Communications of the ACM, 21, 8, pp.613-641, 1978.)



# Church-Rosser Theorem

- We can formally model the process of evaluating an expression as the application of one or more *reduction rules*.
- E.g., lambda-calculus includes the *beta-reduction* rule to evaluate the application of a lambda abstraction to an argument expression.
  - A copy of the body of the lambda abstraction is made and occurrences of the bound variable replaced by the argument.
  - E.g. (? x . x+1) 4 => 4+1
- The CR theorem states that if an expression can be reduced by zero or more reduction steps to either expression M or expression N then there exists some other expression to which both M and N can be reduced.
- This implies that there is a *unique normal form* for any expression since M and N cannot be different normal forms because the theorem says they can be reduced to some other expression and normal forms are irreducible by definition.
- It does not imply that a normal form is reachable, only that if reduction terminates it will reach a unique normal form.



# More Expressions

With Church-Rosser
- reasoning about expressions is easier
- order independence supports fine-grained parallelism
- Diamond property is quite useful

Referential transparency
- In a fixed context, the replacement of a subexpression by its value is completely independent of the surrounding expression
  - » having once evaluated an expression in a given context, shouldn't have to do it again.
- Alternative: referential transparency is the universal ability to substitute equals for equals (useful in common subexpression optimizations and mathematical reasoning)



# FPLs address C.A.R. Hoare's Principles of Structuring

1) Transparency of meaning
   - Meaning of whole expression can be understood in terms of meanings of its subexpressions.
2) Transparency of Purpose
   - Purpose of each part consists solely of its contribution to the purpose of the whole. => No side effects.
3) Independence of Parts
   - Meaning of independent parts can be understood completely independently.
   - In E + F, E can be understood independently of F.
4) Recursive Application
   - Both construction and analysis of structure (e.g. expressions) can be accomplished through recursive application of uniform rules.
5) Narrow Interfaces
   - Interface between parts is clear, narrow (minimal number of inputs and outputs) and well controlled.
6) Manifestness of Structure
   - Structural relationships among parts are obvious. e.g. one expression is subexpression of another if the first is textually embedded in the second. Expressions are unrelated if they are not structurally related.

Hoare, Charles Antony Richard. "*Hints on programming language design.*", In SIGACT/SIGPLAN Symposium on principles of programming languages, October 1973.

## Properties of Pure Expressions

- Value is independent of evaluation order
- Expressions can be evaluated in parallel
- Referential transparency
- No side-effects (Church Rosser)
- Inputs to an expression are obvious from written form
- Effects of operation are obvious from written form
  => Meet Hoare's principles well
  => Good attributes to extend to all programming (?)



## What are some FPLs?

- LISP – the first FPL, ~1958
- Pure FPLs have no side effects
  – Haskell and Miranda are the two most popular examples
- Some FPLs try to be more practical and do allow some side effects
  – Lisp and it's dialects (e.g. Scheme)
  – ML (Meta Language) and SML (Standard ML)



## Lisp

- Defined by John McCarthy* ~1958 as a language for AI.
- Originally, LISP was a typeless language with only two data types: atom and list
- LISP's lists are stored internally as single-linked lists
- Lambda notation was used to specify functions.
- Function definitions, function applications, and data all have the same form
  If the list (A B C) is interpreted as data it is a simple list of three atoms, A, B, and C but if interpreted as a function application, it means that the function named A is applied to the two parameters, B and C.
- Example (early Lisp):
  (defun fact (n) (cond ((lessp n 2) 1)(T (times n (fact (sub1 n))))))
- Common Lisp is the ANSI standard Lisp specification

** Recursive Functions of Symbolic Expressions and their Computation by Machine (Part I)*, John McCarthy, CACM, April 1960. http://www-formal.stanford.edu/jmc/recursive.html



## Scheme

- In the mid 70's Sussman and Steele (MIT) defined Scheme as a new LISP-like Language
- Goal was to move Lisp back toward it's simpler roots and incorporate ideas which had been developed in the PL community since 1960.
  – Uses only static scoping
  – More uniform in treating functions as first-class objects which can be the values of expressions and elements of lists, assigned to variables and passed as parameters.
  – Includes the ability to create and manipulate *closures* and *continuations*.
    » A *closure* is a data structure that holds an expression and an environment of variable bindings in which it's to be evaluated. Closures are used to represent unevaluated expressions when implementing FPLs with lazy evaluation.
    » A *continuation* is a data structure which represents "the rest of a computation"
- Example: (define (fact n) (if (< n 2) 1 (* n (fact (- n 1)))))
- Scheme has mostly been used as a language for teaching Computer programming concepts where as Common Lisp is widely used as a practical language.

## ML

- ML (Meta Language) is a strict, static-scoped functional language with a Pascal-like syntax that was defined by Robin Milner et. al. in 1973.
- It was the first language to include statically checked polymorphic typing.
  - Uses type declarations, but also does type inferencing to determine the types of undeclared variables
  - Strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types, garbage collection and a formal semantics.
- Most common dialect is Standard ML (SML)
- Example:
  fun cube (x : int) = x * x * x;



## Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing)
- Different from ML and most other FPLs in that it is **purely** functional -- no variables, no assignment statements, and no side effects of any kind
- Some key features:
  - Uses lazy evaluation (evaluate no subexpression until the value is needed)
  - Has "list comprehensions," which allow it to deal with infinite lists
- Example:
  fib 0 = 1
  fib 1 = 1
  fib (n + 2) = fib (n + 1) + fib n



## Some FP concepts

- A number of interesting programming language concepts have arisen, including:
  - Curried functions
  - Type inferencing
  - Polymorphism
  - Higher-order functions
  - Functional abstraction
  - Lazy evaluation



## Curried Functions

- The logician Frege noted in 1883 that we only need functions of one argument.
  - We can replace a function f(x,y) by a new function f'(x) that when called produces a function of another argument to compute f(x,Y).
  - That is: (f'(x))(y) = f (x,y)
- Haskell Curry developed combinatorial logic which used this idea.
- We call f' a "curried" form of the function f.
- Two operations:
  - To curry : ((a,b) -> c) -> (a -> b -> c)
  - To uncurry : (a -> b -> c) -> ((a,b) -> c)

# Type Inferencing

- Def: ability of the language to infer types without having programmer provide type signatures.
- SML ex: fun min(a:real,b) = if a > b then b else a
  - type of a has to be given, but then that's sufficient to figure out the type of b and the type of min
  - What if type of a is not specified? Could be ints or bools or …
- Haskell (as with ML) guarantees type safety (if it compiles, then it's type safe)
  - Haskell ex: eq = (a = b)
  - a polymorphic function that has a return type of bool, assumes only that its two arguments are of the same type and can have the equality operator applied to them.
- Overuse of type inferencing in both languages is discouraged
  - declarations are a design aid
  - declarations are a documentation aid
  - declarations are a debugging aid



# Polymorphism

- ML:

  fun factorial (0) = 1

  = | factorial (n) = n * factorial (n - 1);

  – ML infers factorial is an integer function: int -> int
- Haskell:

  factorial (0) = 1

  factorial (n) = n * factorial (n - 1)

  – Haskell infers factorial is a (numerical) function: Num a => a -> a



# Polymorphism

- ML:

  fun mymax(x,y) = if x > y then x else y;

  – SML infers mymax is an integer function: int -> int

  fun mymax(x: real ,y) = if x > y then x else y;

  – SML infers mymax is real
- Haskell:

  mymax(x,y) = if x > y then x else y;

  – Haskell infers factorial is an Ord function



# Higher Order Functions

- Definitions:
  - *zero-order functions*: data in the traditional sense.
  - *first-order functions*: functions that operate on zero-order functions.
  - *second-order functions*: operate on first order
- In general, higher-order functions (HOFs) are those that can operate on functions of any order as long as types match.
  - HOFs are usually polymorphic
- Higher-order functions can take other functions as arguments and produce functions as values.
  - *Applicative programming* has often been considered the application of first-order functions.
  - *Functional programming* has been considered to include higher-order functions: *functionals*.

# Functional Abstraction

- Functional programming allows *functional abstraction* that is not supported in imperative languages, namely the definition and use of functions that take functions as arguments and return functions as values.
  - supports higher level reasoning
  - simplifies correctness proofs
- Simple examples: Map and filter in Scheme
  - (map square '(1 2 3 4)) => (1 4 9 16)
  - (filter prime (between 1 15)) => (1 2 3 5 7 11 13)
- (define (map f l)
  ; Map applies a function f to elements of list l returning list of the results.
  (if (null? l) nil (cons (f (first l)) (map f (rest l)))))
- (define filter (f l)
  ; (filter f l)returns a list of those elements of l for which f is true
  (if (null? L )
  nil
  (if (f (first l)) (cons (first l) (filter f (cdr l))) (filter f (cdr l)))))



# Lazy evaluation

- An evaluation strategy in which arguments to a function are evaluated only when needed for the computation.
- Supported by many FPLs including Scheme, Haskell and Common Lisp.
- Very useful for dealing with very large or infinite streams of data.
- Usually implemented using closures – data structures containing all the information required to evaluate the expression.
- Its opposite, *eager evaluation*, is the usual default in a programming language in which arguments to a function are always evaluated before the function is applied.



# Applications of Functional Languages

- Lisp is used for artificial intelligence applications
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
- Embedded Lisp interpreters add programmability to some systems, such as Emacs
- Scheme is used to teach introductory programming at many universities
- FPLs are often used where rapid prototyping is desired.
- Pure FPLs like Haskell are useful in contexts requiring some degree of program verification



# Summary: ILs vs FPLs

*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