CMPS 3500

Programming Languages

Dr. Chengwei Lei
CEECS
California State University, Bakersfield
Chapter 15 Topics

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
- The First Functional Programming Language: Lisp
- Introduction to Scheme
- Common Lisp
- ML
- Haskell
- F#
- Support for Functional Programming in Primarily Imperative 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
A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set.

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

\[ \lambda(x) \ x \ast x \ast x \]

for the function \( \text{cube}(x) = x \ast x \ast x \)
Lambda Expressions

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

  e.g., \((\lambda(x) \ x \ * \ x \ * \ x)(2)\)

  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.
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
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: \( h = f \circ g \)

which means \( h(x) \equiv f(g(x)) \)
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 \( g(x) \equiv 3 \times x \),

\( h \equiv f \circ g \) yields ????
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 \( g(x) \equiv 3 \times x \),

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

$\alpha(h, (2, 3, 4))$ yields
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 $h(x) \equiv x \times x$

$\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$
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.
Lisp Data Types and Structures

- **Data object types**: originally only atoms and lists
- **List form**: parenthesized collections of sublists and/or atoms
e.g., `(A B (C D) E)`
- Originally, Lisp was a typeless language
- Lisp lists are stored internally as single-linked lists
Lisp Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.

  e.g., If the list \((A \ B \ C)\) is interpreted as data it is a simple list of three atoms, \(A\), \(B\), and \(C\)

  If it is interpreted as a function application, it means that the function named \(A\) is applied to the two parameters, \(B\) and \(C\)

- The first Lisp interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation
Common Lisp

- A combination of many of the features of the popular dialects of Lisp around in the early 1980s
- A large and complex language--the opposite of Scheme
- Features include:
  - records
  - arrays
  - complex numbers
  - character strings
  - powerful I/O capabilities
  - packages with access control
  - iterative control statements
Common Lisp (continued)

- Macros
  - Create their effect in two steps:
    - Expand the macro
    - Evaluate the expanded macro
- Some of the predefined functions of Common Lisp are actually macros
- Users can define their own macros with `DEFMACRO`
Common Lisp (continued)

- Reader Macros
  - Lisp implementations have a front end called the reader that transforms Lisp into a code representation. Then macro calls are expanded into the code representation.
  - A reader macro is a special kind of macro that is expanded during the reader phase.
  - A reader macro is a definition of a single character, which is expanded into its Lisp definition.
  - An example of a reader macro is an apostrophe character, which is expanded into a call to `QUOTE`.
  - Users can define their own reader macros as a kind of shorthand.
Common Lisp (continued)

- Common Lisp has a symbol data type (similar to that of Ruby)
  - The reserved words are symbols that evaluate to themselves
  - Symbols are either bound or unbound
    - Parameter symbols are bound while the function is being evaluated
    - Symbols that are the names of imperative style variables that have been assigned values are bound
    - All other symbols are unbound
ML

- A static-scoped functional language with syntax that is closer to Pascal than to Lisp
- 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
- Does not have imperative-style variables
- Its identifiers are untyped names for values
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations
ML Specifics

- A table called the evaluation environment stores the names of all identifiers in a program, along with their types (like a run-time symbol table)

- Function declaration form:

  \[
  \text{fun } \text{name} \ (\text{formal parameters}) \ = \ \text{expression};
  \]

  e.g., \[
  \text{fun } \text{cube} \ (x : \text{int}) \ = \ x \ * \ x \ * \ x;
  \]

  - The type could be attached to return value, as in:

    \[
    \text{fun } \text{cube} \ (x) : \text{int} \ = \ x \ * \ x \ * \ x;
    \]

  - With no type specified, it would default to \[
  \text{int} \ (\text{the default for numeric values})
  \]

  - User-defined overloaded functions are not allowed, so if we wanted a \text{cube} function for real parameters, it would need to have a different name
ML Specifics (continued)

- ML selection
  
  ```ml
  if expression then then_expression
  else else_expression
  ```

  where the first expression must evaluate to a Boolean value.

- Pattern matching is used to allow a function to operate on different parameter forms
  
  ```ml
  fun fact(0) = 1
  | fact(1) = 1
  | fact(n : int) : int = n * fact(n - 1)
  ```
ML Specifics (continued)

- Lists
  Literal lists are specified in brackets
  
  \[3, 5, 7\]
  
  \([]\) is the empty list
  
  CONS is the binary infix operator, ::
  
  \[4 :: [3, 5, 7]\], which evaluates to \[4, 3, 5, 7\]
  
  CAR is the unary operator \texttt{hd}
  
  CDR is the unary operator \texttt{tl}
  
  \begin{verbatim}
  fun length([]) = 0
  | length(h :: t) = 1 + length(t);
  
  fun append([], lis2) = lis2
  | append(h :: t, lis2) = h :: append(t, lis2);
  \end{verbatim}
ML Specifics (continued)

- The `val` statement binds a name to a value (similar to `DEFINE` in Scheme)
  ```plaintext
  val distance = time * speed;
  ```
- As is the case with `DEFINE`, `val` is nothing like an assignment statement in an imperative language
- If there are two `val` statements for the same identifier, the first is hidden by the second
- `val` statements are often used in `let` constructs
  ```plaintext
  let
      val radius = 2.7
      val pi = 3.14159
  in
      pi * radius * radius
  end;
  ```
**Haskell**

- Similar to ML (syntax, static scoped, strongly typed, type inferencing, pattern matching)
- Different from ML (and most other functional languages) in that it is purely functional (e.g., no variables, no assignment statements, and no side effects of any kind)

Syntax differences from ML

```haskell
fact 0 = 1
fact 1 = 1
fact n = n * fact (n - 1)
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```
Function Definitions with Different Parameter Ranges

\[
\text{fact } n \\
\quad | \quad n == 0 = 1 \\
\quad | \quad n == 1 = 1 \\
\quad | \quad n > 0 = n \times \text{fact}(n - 1)
\]

\[
\text{sub } n \\
\quad | \quad n < 10 = 0 \\
\quad | \quad n > 100 = 2 \\
\quad | \quad \text{otherwise} = 1
\]

\[
\text{square } x = x \times x
\]

- Because Haskell supports polymorphism, this works for any numeric type of \( x \)
Haskell Lists

- List notation: Put elements in brackets
e.g., directions = ["north", "south", "east", "west"]

- Length: #
e.g., #directions is 4

- Arithmetic series with the .. operator
e.g., [2, 4..10] is [2, 4, 6, 8, 10]

- Catenation is with ++
e.g., [1, 3] ++ [5, 7] results in [1, 3, 5, 7]

- CONS, CAR, CDR via the colon operator
e.g., 1: [3, 5, 7] results in [1, 3, 5, 7]
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
  ```fsharp
  let x = seq {1..4};;
  ```

- Generation of sequence values is lazy
  ```fsharp
  let y = seq {0..10000000};;
  Sets y to [0; 1; 2; 3;...]
  ```

- Default stepsize is 1, but it can be any number
  ```fsharp
  let seq1 = seq {1..2..7}
  Sets seq1 to [1; 3; 5; 7]
  ```

- Iterators – not lazy for lists and arrays
  ```fsharp
  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`
    ```fsharp
    (fun a b -> a / b)
    ```
  - No difference between a name defined with `let` and a function without parameters
  - The extent of a function is defined by indentation
    ```fsharp
    let f =
      let pi = 3.14159
    let twoPi = 2.0 * pi;;
    ```
F# (continued)

- Functions (continued)
  - If a function is recursive, its definition must include the `rec` reserved word
  - Names in functions can be outscoped, which ends their scope
    
    ```fsharp
    let x4 =
    let x = x * x
    let x = x * x
    ```

    The first `let` in the body of the function creates a new version of `x`; this terminates the scope of the parameter; The second `let` in the body creates another `x`, terminating the scope of the second `x`
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
Support for Functional Programming in Primarily Imperative Languages

- Support for functional programming is increasingly creeping into imperative languages
  - Anonymous functions (lambda expressions)
    - JavaScript: leave the name out of a function definition
    - C#: \( i \mapsto (i \% 2) == 0 \) (returns true or false depending on whether the parameter is even or odd)
    - Python: `lambda a, b : 2 * a - b`
Support for Functional Programming in Primarily Imperative Languages (continued)

- Python supports the higher-order functions filter and map (often use lambda expressions as their first parameters)

  ```
  map(lambda x : x ** 3, [2, 4, 6, 8])
  Returns [8, 64, 216, 512]
  ```

- Python supports partial function applications

  ```
  from operator import add
  add5 = partial(add, 5)
  (the first line imports add as a function)
  Use: add5(15)
  ```
Support for Functional Programming in Primarily Imperative Languages (continued)

- **Ruby Blocks**
  - Are effectively subprograms that are sent to methods, which makes the method a higher-order subprogram
  - A block can be converted to a subprogram object with `lambda`
    
    ```ruby
    times = lambda { |a, b| a * b }
    Use: x = times.(3, 4) (sets x to 12)
    ```
  - Times can be curried with
    
    ```ruby
    times5 = times.curry.(5)
    Use: x5 = times5.(3) (sets x5 to 15)
    ```
Comparing Functional and 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
Summary

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution.
- 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 that uses type inference.
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- F# is a .NET functional language that also supports imperative and object-oriented programming.
- Some primarily imperative languages now incorporate some support for functional programming.
- Purely functional languages have advantages over imperative alternatives, but still are not very widely used.