Chapter 15
Functional Programming Languages
Introduction

- The design of the imperative languages is based directly on the von Neumann architecture

  - Efficiency (at least at first) 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

- So what...?

  Provides 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

- Definition: 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.

\[ \lambda(x) x \times x \times x \]

for the function cube \((x) = x \times x \times x\)
Mathematical Functions (Continued)

- Lambda expressions describe nameless/“anonymous” functions

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

  e.g. \((\lambda(x) x * x * x)(3)\)

  which evaluates to 27

- Functional Forms

  Def: A higher-order function, or functional form, is one that either takes functions as parameters or yields a function as its result, or both
1. Function Composition

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

Form: \( h \equiv f \circ g \)
which means \( h(x) \equiv f(g(x)) \)
Mathematical Functions (Continued)

2. Construction

A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter

Form: \([f, g]\)

For \(f(x) = x \times x \times x\) and \(g(x) = x + 3\),
\([f, g](4)\) yields \((64, 7)\)

Q: Is this somewhat similar to the Python map() function?
A: Almost…
Mathematical Functions (Continued)

3. **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 $h(x) \equiv x \times x \times x$
   $\alpha(h, (3, 2, 4))$ yields $(27, 8, 64)$

   This looks like `map( )`
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, not so in a functional programming language.
- Management of variables is a constant concern and source of complexity for imperative programming
  - Globals, aliases, locals, parameters, statics, etc.

- In a FPL, variables are not necessary, as is the case in mathematics

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

- This is called referential transparency

Question: Doesn’t that work for imperative PL too?!
Answer: Nope, think statics and globals and you’ll see some problems
LISP

- **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

Note: Many students despise LISP because of the number of parentheses being used and trying to match them up leading LISP = Lots of insidious parentheses
LISP (Continued)

- Lambda notation is used to specify functions and function definitions, function applications, and data all 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
Intro to Scheme

- A mid-1970s dialect of LISP, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of LISP

- Uses only static scoping

- Functions are first-class entities:
  - They can be the values of expressions and elements of lists
  - They can be assigned to variables and passed as parameters
Intro to Scheme (Continued)

- **Primitive Functions**

  1. **Arithmetic:** +, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX

     e.g., (+ 5 2) yields 7

  2. **QUOTE** - takes one parameter; returns the parameter without evaluation

     - **QUOTE** is required because the Scheme interpreter, named EVAL, always evaluates parameters to function applications before applying the function. **QUOTE** is used to avoid parameter evaluation when it is not appropriate.
- **QUOTE** can be abbreviated with the apostrophe prefix operator e.g., '(A B) is equivalent to (QUOTE (A B))

3. **CAR** takes a list parameter; returns the first element of that list

   e.g., (CAR '(A B C)) yields A
   (CAR '((A B) C D)) yields (A B)

4. **CDR** takes a list parameter; returns the list after removing its first element

   e.g., (CDR '(A B C)) yields (B C)
   (CDR '((A B) C D)) yields (C D)
Intro to Scheme (Continued)

5. \texttt{CONS} takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result.

\texttt{e.g., (CONS 'A '(B C)) returns (A B C)}

6. \texttt{LIST} - takes any number of parameters; returns a list with the parameters as elements.
Intro to Scheme (Continued)

- **Lambda Expressions**

  - Form is based on $\lambda$ notation

    e.g.,
    
    \[
    \text{(LAMBDA (L) (CAR (CAR L)))}
    \]

    $L$ is called a *bound variable*

  - Lambda expressions can be applied

    e.g.,
    
    \[
    \text{((LAMBDA (L) (CAR (CAR L))) ' ((A B) C D))}
    \]
Intro to Scheme (Continued)

- A Function for Constructing Functions

**DEFINE** - Two forms:

1. To bind a symbol to an expression
e.g.,
   (DEFINE pi 3.141593)
   (DEFINE two_pi (* 2 pi))

2. To bind names to lambda expressions
e.g.,
   (DEFINE (cube x) (* x x x))

- Example use:

   (cube 4)
Intro to Scheme (Continued)

- **Evaluation process (for normal functions):**

  1. Parameters are evaluated, in no particular order
  2. The values of the parameters are substituted into the function body
  3. The function body is evaluated
  4. The value of the last expression in the body is the value of the function

(Special forms use a different evaluation process)

- **Examples:**

  (DEFINE (square x) (* x x))

  (DEFINE (hypotenuse side1 side2)
    (SQRT (+ (square side1) (square side2))))
Intro to Scheme (Continued)

- **Predicate Functions**: (\#T and () are true and false)

1. **EQ?** takes two symbolic parameters; it returns \#T if both parameters are atoms and the two are the same

   e.g., (EQ? 'A 'A) yields \#T
   (EQ? 'A ' (A B)) yields ()

   Note that if EQ? is called with list parameters, the result is not reliable
   Also, EQ? does not work for numeric atoms

2. **LIST?** takes one parameter; it returns \#T if the parameter is an list; otherwise ()
Intro to Scheme (Continued)

3. **NULL?** takes one parameter; it returns #T if the parameter is the empty list; otherwise ()

   Note that **NULL?** returns #T if the parameter is ()

4. Numeric Predicate Functions
   - =, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?, NEGATIVE?

- Output Utility Functions:
  - (DISPLAY expression)
  - (NEWLINE)
Intro to Scheme (Continued)

- Control Flow

- 1. Selection - the special form, IF

\[
(\text{IF} \ \text{predicate} \ \text{then}\_\exp \ \text{else}\_\exp )
\]

E.g.,
\[
(\text{IF} \ (\neq \ \text{count} \ 0)
  \n  (\text{/} \ \text{sum} \ \text{count})
  \n  0
  \n)
\]

- 2. Multiple Selection - the special form, COND

- General form:

\[
(\text{COND}
  \n  (\text{predicate}_1 \ \text{expr} \ \{\text{expr}\})
  \n  (\text{predicate}_1 \ \text{expr} \ \{\text{expr}\})
  \n  \ldots
  \n  (\text{predicate}_1 \ \text{expr} \ \{\text{expr}\})
  \n  (\text{ELSE} \ \text{expr} \ \{\text{expr}\})
  \n)
\]

Returns the value of the last expr in the first pair whose predicate evaluates to true
Intro to Scheme (Continued)

- Example of `cond`:

```scheme
(define (compare x y)
  (cond
    ((> x y) (display "x is greater than y"))
    ((< x y) (display "y is greater than x"))
    (else (display "x and y are equal")))
)
```

Example Scheme Functions:

- `member` - takes an atom and a list; returns `#t` if the atom is in the list; `()` otherwise

```scheme
(define (member atm lis)
  (cond
    ((null? lis) '())
    ((eq? atm (car lis)) #t)
    (else (member atm (cdr lis)))))
```
Intro to Scheme (Continued)

Example Scheme Functions: (continued)

- 2. equalsimp - takes two simple lists as parameters; returns \( \texttt{#T} \) if the two simple lists are equal; () otherwise

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) '())
    ((eq? (car lis1) (car lis2))
     (equalsimp (cdr lis1) (cdr lis2)))
    (else '())))
```
- 3. equal - takes two lists as parameters; returns 
   #t if the two general lists are equal;
   () otherwise

(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) '())
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) '())
    ((equal (CAR lis1) (CAR lis2))
      (equal (CDR lis1) (CDR lis2)))
    (ELSE '()))
)
Intro to Scheme (Continued)

Example Scheme Functions: (continued)

- 4. **append** - takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(DEFINE (append lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1)
           (append (CDR lis1) lis2)))))
```
Intro to Scheme (Continued)

- The \texttt{LET} function

- General form:

\[
\text{LET} \ (\\ \\
(name_1 \ \text{expression}_1) \\
(name_2 \ \text{expression}_2) \\
\ldots \\
(name_n \ \text{expression}_n) ) \\
\text{body}
\]

- \textit{Semantics}: Evaluate all expressions, then bind the values to the names; evaluate the body - kind of a trick for local variables but more like aliases a la Python
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c)))
          (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
    (DISPLAY (+ minus_b_over_2a
               root_part_over_2a))
    (NEWLINE)
    (DISPLAY (- minus_b_over_2a
               root_part_over_2a)))
))
Intro to Scheme (Continued)

Functional Forms

- 1. Composition
   - The previous examples have used it

- 2. Apply to All - one form in Scheme is mapcar
   - Applies the given function to all elements of the given list; result is a list of the results

\[
\text{(DEFINE (mapcar fun lis)} \\nonumber \\
\quad \text{(COND)} \nonumber \\
\quad \quad (\text{((NULL? lis) '())}) \nonumber \\
\quad \quad (\text{ELSE (CONS (fun (CAR lis))}) \nonumber \\
\quad \quad \quad \text{(mapcar fun (CDR lis))))}) \nonumber \\
\text{)}
\]
Intro to Scheme (Continued)

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation

- This is possible because the interpreter is a user-available function, `EVAL`

  e.g., suppose we have a list of numbers that must be added together

  ```scheme
  ((DEFINE (adder lis)
    (COND
      ((NULL? lis) 0)
      (ELSE (EVAL (CONS '+ lis))))
  )
  ```
Intro to Scheme (Continued)

- The parameter is a list of numbers to be added; `adder` inserts a + operator and interprets the resulting list

- *Scheme includes some imperative features:*
  1. `SET!` binds or rebinds a value to a name
  2. `SET-CAR!` replaces the car of a list
  3. `SET-CDR!` replaces the cdr part of a list
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

- Includes:
  - records
  - arrays
  - complex numbers
  - character strings
  - powerful i/o capabilities
  - packages with access control
  - imperative features like those of Scheme
  - iterative control statements
Common LISP (Continued)

- **Example** (iterative set membership, member)

```
(DEFUN iterative_member (atm lst)
  (PROG ()
    loop_1
    (COND
      ((NULL lst) (RETURN NIL))
      ((EQUAL atm (CAR lst)) (RETURN T))
    )
    (SETQ lst (CDR lst))
    (GO loop_1)
  ))
```
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, we’ll see this in our ML studies

- 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 operation
ML (Continued)

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

- Function declaration form:
  ```
  fun function_name (formal_parameters) =
  function_body_expression;
  ```

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

- Functions that use arithmetic or relational operators cannot be polymorphic--those with only list operations can be polymorphic

- Full syntax covered in our other slide decks
Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing)

- 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)

- Most Important Features
  - Uses lazy evaluation (evaluate no subexpression until the value is needed), this can get tricky at times.

  - Has “list comprehensions,” which allow it to deal with infinite lists
Haskell (Continued)

Examples

1. Fibonacci numbers (illustrates function definitions with different parameter forms)

   fib 0 = 1
   fib 1 = 1
   fib (n + 2) = fib (n + 1) + fib n

2. Factorial (illustrates guards)

   fact n
   | n == 0 = 1
   | n > 0 = n * fact (n - 1)

   The special word otherwise can appear as a guard
3. List operations

- 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]
Haskell (Continued)

- Catenation is with `++`
  - e.g., 
    
    \[
    [1, \ 3] \++ \ [5, \ 7] \qquad \text{results in} \qquad [1, \ 3, \ 5, \ 7]
    \]

- CAR and CDR via the colon operator
  - e.g., 
    
    \[
    1: [3, \ 5, \ 7] \qquad \text{results in} \qquad [1, \ 3, \ 5, \ 7]
    \]

- Examples:

```
product [] = 1
product (a:x) = a * product x

fact n = product [1..n]
```
Haskell (Continued)

4. List comprehensions: set notation
e.g.,
\[ n \times n \mid n \leftarrow [1..20] \]

defines a list of the squares of the first 20 positive integers

factors \( n = [i \mid i \in [1..n \div 2], n \mod i == 0] \)

This function computes all of the factors of its given parameter
Haskell (Continued)

Quicksort:

sort [] = []
sort (a:x) = sort [b | b ← x; b <= a]
    ++ [a] ++
    sort [b | b ← x; b > a]
Haskell (Continued)

5. Lazy evaluation

- Infinite lists
  e.g.,

  positives = [0..]
squares = [n * n | n ← [0..]]

(only compute those that are necessary)

  e.g.,

  member squares 16

  would return True
Haskell (Continued)

The `member` function could be written as:

```haskell
member [] b = False
member (a:x) b = (a == b) || member x b
```

However, this would only work if the parameter to `squares` was a perfect square; if not, it will keep generating them forever. The following version will always work:

```haskell
member2 (m:x) n
| m < n = member2 x n
| m == n = True
| otherwise = False
```
Applications of Functional Languages:

- 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. This was a choice after Pascal and others choose Java for better or worse.
Haskell (Continued)

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, or so they say

Bottom Line though: FPL’s haven’t lived up to their hype but they still contribute greatly if you think carefully about Python, Ruby, and JavaScript