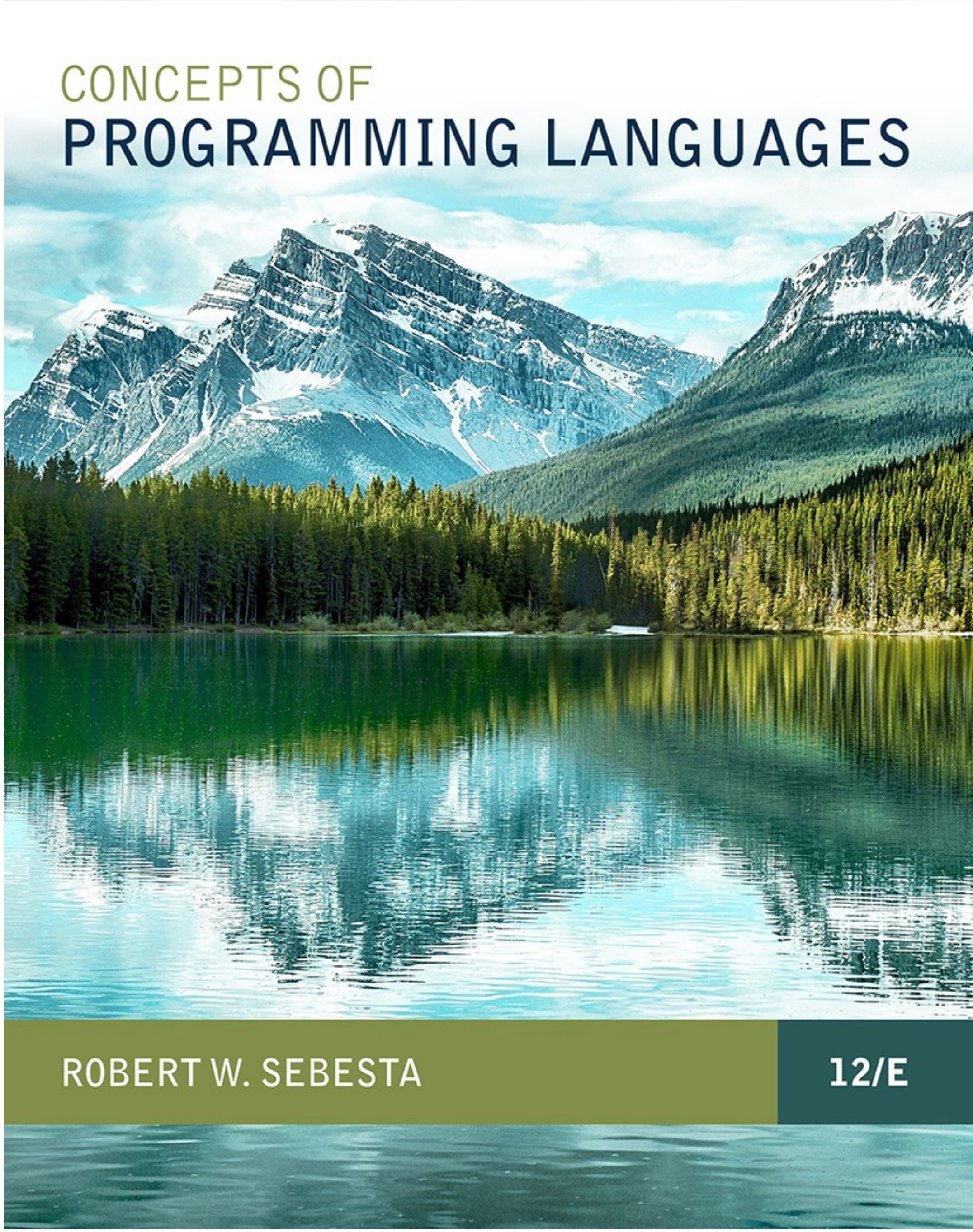
Chapter 15

Functional Programming Languages



ISBN 0-321-49362-1

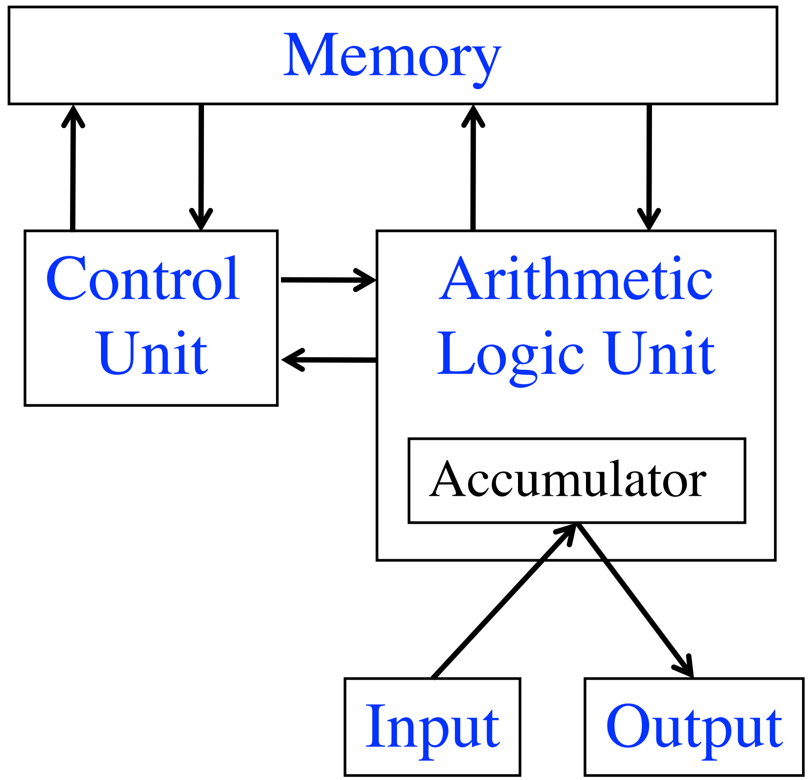
Chapter 15 Topics

- 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 theoretical basis unconcerned with the architecture of the machines on which programs will run.

von Neumann Architecture



ref: http://en.wikipedia.org/wiki/Von_Neumann_architecture

Mathematical Functions

- A mathematical function is a *mapping* of:
  - from a set of members/input--*domain set (x)*
  - to another set of output--*range set (y)*
- *Lambda expression*:
  - specifies the parameter(s) and the mapping of a function in the following form:

λ`(x) x * x * x` **--input:** `x` ; **output:** x^3

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

Lambda Expressions

- Lambda expressions
  - describe **nameless** functions
  - apply parameter(s) by placing the parameter(s) after the expression
- e.g., `(λ(x) x * x * x)` **(2)**
- result ?

which evaluates to **8**.

Functional Forms

- A *higher-order* function or *functional form*:
  - takes functions as parameters
  - yields a function as its result

- Two common functional forms:
  - **Function Composition**
  - **Apply-to-All**.

Function Composition

- A functional form
  - takes two functions as parameters
  - yields a function whose value is the first actual parameter function applied to the application of the second
- Form: $h \equiv f \circ g$

  $\circ$: operator

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

Form: α

For `h(x) ≡ x * x`

$\boldsymbol{\alpha}$`(`**`h`**`, (2, 3, 4))`

yields `(4, 9, 16).`

Fundamentals of Functional Programming Languages (FPL)

- The objective of the design of a FPL
  - mimic mathematical functions to the greatest extent possible
- 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 input parameters.

Lisp: Data Types and Structures

- *Data object types*
  - originally only atom and list
- *List form*
  - parenthesized collections of sublists and/or atoms

  e.g., `(A B (C D) E)`
- Lisp lists are stored internally as single-linked lists.

Lisp: Interpretation

- Lambda notation is used to specify functions and function definitions.
  - e.g., `(λ(x) x * x * x)` **(2)**
- Function and data have the same form.
  - If the list **`(A B C)`** is interpreted as data:
    - it is simple list of three atoms--A, B, and C
  - If it is interpreted as a function:
    - 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.

Origins of Scheme

- A mid-1970s dialect of Lisp
  - designed to be a cleaner, more modern, and simpler version than contemporary dialects of Lisp
- Uses only static scoping
- *Functions* are *first-class entities*/objects
  - They can be the values of expressions and elements of lists
  - They can be assigned to variables, passed as parameters, and returned from functions

Ref: first-class entity: can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable.

Scheme: Interpreter

- In interactive mode, the Scheme interpreter is an infinite Read-Evaluate-Print Loop (REPL)
  - This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function **`EVAL`**
- Literals evaluate to themselves.

Scheme: Evaluation

- Parameters are evaluated in **no** particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the **function**.

Scheme: Primitive Functions

- Primitive Arithmetic Functions
  - `ABS`, `SQRT`, `REMAINDER`, `MIN`, `MAX`, `+`, `-`, `*`, `/`

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

  `(+ 5 4 6 2)` yields `17`
- 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)` yields `49`

- `LAMBDA` expressions can have any number of parameters

  e.g., `(LAMBDA (a b c x) (+ (* a x x) (* b x) c))`

```
((LAMBDA (a b c x) (+ (* a x x) (* b x) c)) 1 2 3 4)
```
yields `27`

Scheme: Function *DEFINE*

- `DEFINE` – Two forms:
  1. To bind a symbol to an expression

     (DEFINE symbol Expression)

     e.g., `(DEFINE pi 3.141593)`

     `(DEFINE two_pi (* 2 pi))`

     These symbols are not variables – they are like the names bound by Java's **`final`** declarations
  2. To bind names to lambda expressions (`LAMBDA` is implicit)

     (DEFINE (function_name parameters) (Expression))

     e.g., `(DEFINE (square x) (* x x))`

     `(square 5)` yields `25`

Note: The evaluation process for **`DEFINE`** is different! The first parameter is **never** evaluated. The second parameter is evaluated and bound to the first parameter--square.

Scheme: 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 have side effects.

Scheme: Numeric Predicate

- `#T` (or `#t`) is true and `#F` (or `#f`) is false (sometimes `()` is used for false)
- **`=` , `<>` , `>` , `<` , `>=` , `<=`**
- **`EVEN?` , `ODD?` , `ZERO?` , `NEGATIVE?`**

  `(is it even?, odd? zero?, negative?)`

- The `NOT` function inverts the logic of a Boolean expression.

Scheme: Control Flow

- Selection– the special form, **IF**

  (**IF** predicate then_exp else_exp)

  e.g., (**IF (<> count 0)**
  (/ sum count)
  0)

- COND function: returns the value of the expression in the first pair whose predicate evaluates to true

```
(DEFINE (leap? year)
   (COND
      ((ZERO? (MODULO year 400)) #T)    // pred1 expr1
      ((ZERO? (MODULO year 100)) #F)    // pred2 expr2
      ((ZERO? (MODULO year 4))   #T)    // pred3 expr3
      (ELSE                      #F)    // else  exprn
))
```

Scheme: Function – `QUOTE, CONS`

- **`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))`
- **`CONS`** takes (concatenates) two parameters
  - the first: either an atom or a list
  - the second: 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.

Scheme: Function - `CAR, CDR`

- **`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)`**

- **`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)`** .

Scheme: Function Examples

- Examples:

  (CAR '((A B) C D)) returns **(A B)**

  (CAR 'A) returns **error**

  (CDR '((A B) C D)) returns **(C D)**

  (CDR 'A) returns **error**

  (CDR '(A)) returns **()**

  (CONS '() '(A B)) returns **(() A B)**

  (CONS '(A B) '(C D)) returns **((A B) C D)**

  (CONS 'A 'B) returns **(A . B) a dotted pair.**

Note: ref: https://classes.soe.ucsc.edu/cmps112/Spring03/languages/scheme/SchemeTutorialA.html

Scheme: Function – `LIST`

- `LIST` is a function for building a list from any number of parameters

  e.g., `(LIST 'apple 'orange 'grape)` returns

  `(apple orange grape)`.

Scheme: Predicate Function – `EQ?`

- **`EQ?`** takes two expressions as parameters (usually two atoms);
  - returns `#T` if both parameters have the **same** pointer value—pointing to the same atom or list;
  - otherwise `#F`
  - **`EQ?`** does **not** work for numeric atoms

```
(EQ? 'A 'A) yields #T
(EQ? 'A 'B) yields #F
(EQ? 'A '(A B)) yields #F
(EQ? '(A B) '(A B)) yields #F
(EQ? 3.4 (+ 3 0.4)) yields #F
(define x `(a b)) (define y `(b a))
(define x y) (EQ? x y) yields #T
```

Scheme: Predicate Function – `EQV?`

- **`EQV?`** is like **`EQ?`**
- except that it works for both symbolic and numeric atoms;
- it is a *value* comparison, not a pointer comparison

`(EQV? 3 3)` yields **`#T`**

`(EQV? 'A 3)` yields **`#F`**

`(EQV? 3.4 (+ 3 0.4))` yields **`#T`**

`(EQV? 3.0 3)` yields **`#F`**

(floats and integers are different).

Scheme: Predicate Functions

- `LIST?` takes one parameter;
  - returns `#T` if the parameter is a list;
  - otherwise `#F`
  - e.g.,

    `(LIST? '())` yields `#T`
- `NULL?` takes one parameter;
  - returns `#T` if the parameter is an **empty** list;
  - otherwise `#F`
  - e.g.,

    `(NULL? '(()))` yields `#F`

Scheme: Function – `member`

- **`member`** takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
(DEFINE (member atm a_list)
(COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    ((ELSE (member atm (CDR a_list)))
))
```

e.g.,

`(member `A `(A B C)))` yields `#T`

Scheme: Function – `equalsimp`

- **`equalsimp`** takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

```
(DEFINE (equalsimp list1 list2)
(COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp(CDR list1)(CDR list2)))
    (ELSE #F)
))
```

e.g.,

`(equalsimp `(A B)`(A B))` yields **#T**

Scheme: Function – `equal`

- **`equal`** takes two general lists as parameters; returns `#T` if the two lists are equal; `#F` otherwise

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1))(EQ? list1 list2))
    ((NOT (LIST? list2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
          (equal (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

e.g.,

`(equal `(A (B C D)) `(A (B C D))` yields **`#T`**

Scheme: Function – `append`

- **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 list1 list2)
  (COND
   ((NULL? list1) list2)
   (ELSE (CONS (CAR list1)
               (append (CDR list1) list2)))
))
```

e.g.,

`(append `(A B) `(C D))` yields **(A B C D)**

Scheme: Function – LET

- General form:

```
(LET (
    (name_1  expression_1)
    (name_2  expression_2)
    ...
    (name_n  expression_n)
    )
    body )
```

- LET is actually shorthand for a LAMBDA expression applied to a parameter

  `(LET ((alpha 7))(* 5 alpha))`

  is the same as:

  `((LAMBDA (alpha) (* 5 alpha)) 7)`

Scheme: `LET` Example

```
(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)))
    )
   (LIST (+ minus_b_over_2a root_part_over_2a)
         (- minus_b_over_2a root_part_over_2a))
))
```

e.q., (-b + sqrt(b^2-4ac))/2a, (-b - sqrt(b^2-4ac))/2a

`(quadratic_roots 1 4 4)` yields **(-2 -2)**.

Scheme: Tail Recursion

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language systems convert all tail recursive functions to use iteration.

Scheme: Simple Iteration

```
(DEFINE (list-count n )
   (LET loop (( i   n ))                           ;; i = n
     (if ( > i 0 )                                 ;; loop () body
            (cons i (loop (- i 1)) )
            `()
     )
   )
)
  ;; (list-count 10) yields (10 9 8 7 6 5 4 3 2 1).
```

Scheme: Tail Recursion

- rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
  (IF (<= n 0)
      1
      (* n (factorial (- n 1)))
 ))
```

e.g., `(factorial 3)` yields **6**

Tail recursive:

```
(DEFINE (facthelper n factpartial)
  (IF (<= n 0)
      factpartial
      (facthelper((- n 1) (* n factpartial)))
)
(DEFINE (factorial n)
  (facthelper n 1))
```

Scheme: Functional Form – Composition

- Composition
  - If `h` is the composition of `f` and `g`, `h(x) = f(g(x))`

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE (h x) (+ 2 (* 3 x)))   ;; the composition
```

  - In Scheme, the functional composition function **compose** can be written:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g (x))))
```

e.g., `((compose CAR CDR) '((a b) c d))` yields **c**

```
(DEFINE (third a_list)
    ((compose CAR (compose CDR CDR)) a_list))
```

is equivalent to **CADDR**

e.g., `(third '(a b c d))` yields **c**

Scheme: Functional form Apply-to-All

- Apply to All - one form in Scheme is **map**
  - Applies the given function to all elements of the given list;

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list))
                (map fun (CDR a_list))))
))
```

```
e.g., (map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

yields ? **(27 64 8 216)**

```
;; fun == (LAMBDA (num) (* num num num)).
```

Scheme: Functions That Build Code

- 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`**.

Scheme: Adding a List of Numbers

```
(DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (EVAL (CONS '+ a_list) user-
initial-environment))
))
```

- e.g., `(adder '(3 4 6))` **Builds (+ 3 4 6) = 13**
- The parameter is a list of numbers to be added; **`adder`** inserts a **+** operator and evaluates the list:
  - Use `CONS` to insert the atom **+** into the list of numbers.
  - Be sure that **+** is quoted to prevent evaluation
  - Submit the new list to **`EVAL`** for evaluation
  - user-initial-environment is required in scheme v4 or later
  - Reference manual: https://groups.csail.mit.edu/mac/ftpdir/mit-scheme/7.7/7.7.1/doc-pdf/scheme.pdf

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

- 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**

(**defmacro** *name* (*parameters*)
"optional documentation string."
body-form)

e.g., (**defmacro** mna (a b) "multiplies and adds" `(+ ,a (* ,b 3)))

Common Lisp

- Backquote operator (`` ` ``)
  - Similar to the Scheme's `QUOTE`, except that some parts of the parameter can be unquoted by preceding them with commas

    `` `(a (* 3 4) c) `` evaluates to `(a (* 3 4) c)`

    `` `(a ,(* 3 4) c) `` evaluates to `(a 12 c)`

Common Lisp

- Common Lisp has a symbol type (similar to that of Ruby)
  - The reserved words, e.g., *nil*, are symbols that evaluate to themselves
  - Symbols have attributes:
    - name, package, property list, value, function

Ref: http://www.lispworks.com/documentation/HyperSpec/Body/t_symbol.htm#symbol

- Symbols are used for their object identity to name various entities in Common Lisp, including (but not limited to) linguistic entities such as variables and functions.
- Symbols can be collected together into packages. A symbol is said to be interned in a package if it is accessible in that package; the same symbol can be interned in more than one package. If a symbol is not interned in any package, it is called uninterned.

ML (MetaLanguage)

- 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
- Does not have imperative-style variables
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

ML: Function

- Function declaration form:

  fun *name* (*formal parameters*) = *expression*;

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

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

    `fun cube(x : int) : int = x * x * x;`

  - With no type specified, it would default to `int` (the default for numeric values)

  - User-defined overloaded functions are not allowed, so if we wanted a `cube` function for float parameters, it would need to have a different name

ML: if-else, Pattern match

- if-then-else

  if *expression* **then** *then_expression*
   else *else_expression*

  where expression must evaluate to a Boolean value

  e.g.,
```
fun fact n = if n = 0 then 1
                   else n * fact (n-1)
```

- Pattern matching is used to allow a function to operate on different parameter forms

  e.g.,
```
fun fact(0) = 1
 |   fact(1) = 1
 |   fact(n : int) : int = n * fact(n - 1)
```
  Note: | : OR

ML: List

- Lists
  Literal lists are specified in brackets
  e.g., `[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 **`hd`** `(head)`
- `CDR` is the unary operator **`tl`** `(tail)`

e.g.,
```
fun length([]) = 0
  |   length(h :: t) = 1 + length(t);
  (* note: h::t is head and tail *)
fun append([], list2) = list2
|   append(h :: t, list2)=h :: append(t, list2);
```

ML: val

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

```
let
   val radius = 2.7
   val pi = 3.14159
 in
   pi * radius * radius
 end;
```

Ref: https://en.wikipedia.org/wiki/Standard_ML

ML: filter

- **`filter`**
  - A higher-order filtering function for lists
  - Takes a predicate function as its parameter, often in the form of a lambda expression
  - Lambda expressions are defined like functions, except with the reserved word **`fn`**

e.g.,

```
filter(fn(x) => x < 100, [25, 1, 711, 50, 100]);
```

This returns `[25, 1, 50]`

ML: map

- **`map`**
  - A higher-order function that takes a single parameter--a function
  - Applies the parameter function to each element of a list and returns a list of results

    e.g.,
    ```
    fun cube x = x * x * x;
         val cubeList = map cube;
         val newList = cubeList [1, 3, 5];
    ```

    This sets `newList` to `[1, 27, 125]`
  - Alternative: use a lambda expression

```
val newList = map (fn x => x * x * x, [1, 3, 5]);
```

ML: operator ∘

- Function Composition
  - Use the operator, ∘

```
val h = g o f;
```

ML: Currying

- Currying
  - make a function in several "stages", each taking an input and producing a new function

e.g.,

```
    fun add a = fn b => a+b; == fun add a b = a + b;
    when call add 2, get fun b => 2 + b retuned
    when call add 2 3, it actually calls (add 2) 3
    which yields 5
```

ML: Partial Evaluation

- Partial Evaluation
  - Curried functions can be used to create new functions by partial evaluation
  - Partial evaluation means that the function is evaluated with actual parameters for one or more of the leftmost actual parameters

```
fun add5 x add 5 x;
```

  - takes the actual parameter `5` and evaluates the `add` function with `5` as the value of its first formal parameter.
  - returns a function that adds `5` to its single parameter

```
val num = add5 10;   (* sets num to 15 *)
```

Haskell

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

Syntax differences from ML

```
fact 0 = 1
fact 1 = 1
fact n = n * fact (n - 1)


fib 0 = 0
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
-- Fibonacci sequences
```

ML:

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

Haskell: Function Definitions

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

sub n
  | n < 10        = 0
  | n > 100       = 2
  | otherwise     = 1


square x = x * x
```

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` yields **`4`**
- Arithmetic series with `..` operator
  e.g., `[2, 4..10]` yields **`[2, 4, 6, 8, 10]`**
- Concatenation is with `++` `for two lists`
  e.g., `[1, 3] ++ [5, 7]` yields **`[1, 3, 5, 7]`**
- CONS via `:` colon operator for a head element + list
  e.g., `1:[3, 5, 7]` yields **`[1, 3, 5, 7]`**

Haskell: Pattern Parameters

- Pattern Parameters

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

  - Factorial: `fact n = product [1..n]`

- List Comprehensions

  set notation: [body | qualifiers]

  e.g., `[n * n * n | n <- [1..50]]`

  The qualifier in this example has the form of a *generator*. It could be in the form of a test

```
factors n = [i | i <- [1..n `div` 2], n `mod` i == 0]
```

  `e.g., factors(10)  yields` **[1, 2, 5]**

  note: the backticks `` specify function as a binary operator.

  [1..n `div` 2] = [1,2,3,4,5], for n = 10.

Haskell: Quicksort

```
sort [] = []
sort (h:t) =
      sort [b | b ← t, b <= h]
    ++ [h] ++
      sort [b | b ← t, b > h]
```

```
e.g., sort [3,1,5,2,4] yields [1,2,3,4,5]
```

Illustrates the concision of Haskell:

- shorter and simpler than imperative programming language

Haskell: Lazy Evaluation

- A language is *strict* if it requires **all** actual parameters to be fully evaluated
- A language is *nonstrict* if it does not have the strict requirement
- Nonstrict languages are more **efficient** and allow some **interesting** capabilities – *infinite lists*
- Lazy evaluation – Only compute those values that are necessary
- Positive numbers

```
positives = [0..]
```

- Determining if 16 is a square number

```
member b [] = False
member b (a:x) = (a == b) || member b x
-- note:  a:x  head:tail;  || = Or
squares = [n * n | n ← [0..]]
member 16 squares  yields True
```

Haskell: Member Revisited

- The member function could be written as:

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

- 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:

```
member2 n (m:x)
      | m < n = member2 n x
      | m == n = True
      | otherwise = False
      -- note: check if 'n' is a square number
member2 15 squares  yields  False
```

Ref: https://www.haskell.org/

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

Notes: OCaml: https://ocaml.org
discriminated unions: type checking

F#: Sequences

- 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 a list `[(1, 1); (2, 8); (3, 27); (4, 64)]`.

F#: Functions

- Use fun with lambda expressions

  e.g.,

  ```
  fun a b -> a / b
  ```

- Use let with indentation

  e.g.,

  ```
  let f =
      let pi = 3.14159
      let twoPi = 2.0 * pi
      twoPi
  ```

F#: Functions

- Recursive function must include **rec** reserved word

  e.g., let rec fact x =

  if x <= 1 then 1

  else n * fact (n-1)

- Names in functions can be out-scoped:

  e.g.,

```
let x4 x =
    let x = x * x
    let x = x * x
    x
```

- The first **let** in the body of x4 function creates a new version of `x`
- the second **let** in the body creates another `x`, terminating the scope of the `x` in the previous **let**.

F#: Functional Operators

- Pipeline (|>)
  - A binary operator that sends the value of its left operand to the last parameter of the call (the right operand)

```
let myNums = [1; 2; 3; 4; 5]
let evensTimesFive = myNums
     |> List.filter (fun n -> n % 2 = 0)
     |> List.map (fun n -> 5 * n)
```

The return value is `[10; 20]`

F#: Functional Operators

- Composition (>>)
  - builds a function that applies its left operand to a given parameter (a function)
  - then passes the result returned from the function to its right operand (another function)
  - the F# expression **`(f >> g) x`** is equivalent to the mathematical expression g(f(x))
- Curried Functions

```
let add a b = a + b
let add5 = add 5
```

Support for Functional Programming in Primarily Imperative Languages

- Anonymous functions (lambda expressions)
  - JavaScript: leave the **name** out of a **function** definition. e.g., (function () {var x="Hello!!"; })();
  - 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`

Support for Functional Programming in Primarily Imperative Languages

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

  e.g., `map(lambda x : x ** 3, [2, 4, 6, 8])`

  Returns `[8, 64, 216, 512]`

- Ruby Blocks
  - A block can be converted to a subprogram object with **`lambda`**

    ```
    times = lambda {|a, b| a * b}
    ```

    e.g., `x = times.(3, 4)`   (sets `x` to `12`)
  - times can be curried with

    ```
    times5 = times.curry.(5)
    ```

    e.g., `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