Topics

- ML
- Haskell
- F#

ML (Meta Language)

- 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:

  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 = 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 real parameters, it would need to have a different name

ML Specifics (continued)

- ML selection

  `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

```
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 `hd`
  `CDR` is the unary operator `tl`

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

fun append([], lis2) = lis2
|   append(h :: t, lis2) = h :: append(t, lis2);
```

ML Specifics (continued)

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

```
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

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

ML Specifics (continued)

- `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`**

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

This returns `[25, 1, 50]`

ML Specifics (continued)

- `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

```
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 Specifics (continued)

- Function Composition
  - Use the unary operator, ∘

```
val h = g ∘ f;
```

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

```
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

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


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


  square x = x * x
```

- Because Haskell support 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]`

Haskell (continued)

- Pattern Parameters

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

  - Factorial:

```
fact n = product [1..n]
```

- List Comprehensions (Chapter 6)

```
[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]
```

The backticks specify the function is used as a binary operator

Quicksort

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

Illustrates the concision of 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(a:x) b=(a == b)||member x b
squares = [n * n | n ← [0..]]
member squares 16
```

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
```

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

```
let x = seq {1..4};;
```

  - 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 to create sequences

```
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`**

```
(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

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

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

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

F# (continued)

- 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# (continued)

- Functional Operators (continued)
  - Composition (>>)
    - Builds a function that applies its left operand to a given parameter (a function) and 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))

F# (continued)

- 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

Announcements

- Exam #2
  - Write two small programs (one in C++, and the other in Smalltalk)
  - Similar to Assignments 5&6, but on a small scale.
  - Take-home exam
- Assignment #7
  - Same shape project in Scheme
  - Due on April 12, Monday
  - Weight: 6%