Functional Programming Languages (FPL)

1. Definitions

- Functional programming languages were originally developed specifically to handle symbolic computation and list-processing applications.

- In FPLs the programmer is concerned only with functionality, not with memory-related variable storage and assignment sequences.

- FPL can be categorized into two types;

  - ✓ PURE functional languages, which support only the functional paradigm (Haskell), and
  - ✓ Impure functional languages that can also be used for writing imperative-style programs (LISP).

2. Applications

- AI is the main application domain for functional programming, covering topics such as:

  expert systems

  knowledge representation

  machine learning

  natural language processing

  modelling speech and vision

    - In terms of symbolic computation, functional programming languages have also proven useful in some editing environments (EMACS) and some mathematical software (particularly calculus)

- Lisp and its derivatives are still the dominant functional languages (we will consider one of the simpler derivatives, Scheme, in some detail).

3. Examples

- Programming Languages:
    - Lisp, Scheme, Miranda, Sisal, Haskell, APL, ML

- Code Examples: Compute the sum of n integers
    - A C implementation:

```
Sum=0;
for(i=1;i<=n;++i)
    sum +=i;
```

        - Computations is done by assignment.

    - A Haskell implementation:

```
sum [1..10]
```

        - Computations is function application.

- A Python implementation:

```
>>> sum([1,2,3,4])
10
>>>
```

  - Computations is function application.

4. FPL Characteristics:

- Functional programming languages are modeled on the concept of mathematical functions, and use only conditional expressions and recursion to effect computation.
- In the purest form they use neither variables nor assignment statements, although this is relaxed somewhat in most applied functional languages.
- The concept of side effects is also alien to purely functional programming: a function is given values and returns a value, there are no variables to manipulate and hence no possibility for side effects.
- Programs are constructed by composing function applications - the values produced by one or more functions become the parameters to another.
- For reasons of efficiency (because the underlying machine is, in fact, imperative) most functional languages provide some imperative-style capabilities, including variables with

assignment, sequences of statements, and imperative style loop structures.

- Note that the functional paradigm can also be used with some imperative languages - e.g. C has both a conditional expression and support for recursion - so the factorial function code be coded in functional style in C (or C++ or Java) as follows: int fact(int x){ return (x == 0) ? 1 : x * fact(x - 1); }

- Three primary components:
    - A set of ***data object:*** A single, high-level data structure like a list
    - A set of ***built-in functions*** for object manipulation: Building, deconstructing, and accessing lists
    - A set of ***functional forms*** for building new functions: Composition, reduction, etc.

5. Lambda calculus (LC)

- A method of modeling the computational aspects of functions
- It helps us understand the elements and semantics of functional programming languages independent of syntax

5.1. LC expressions forms

- **There are three LC expressions forms:**

- **e1**: A ***single identifier*** (such as **x**, or **3**)

- **e2**: A ***function definition*** of the form **(λx.e)**
  :The expression **e**, with **x** being a bound variable
  - ✓ **e** is the body of the function, **x** is a parameter
  - ✓ **e** may be any of the three types of expressions
  - ✓ **square(x)** would be written as **(λx.x*x)**
- **e3**: A ***function application*** of the form **e1 e2**
  Meaning **e1** applied **e2**
  square applied to **2** would be **((λx.x*x) 2)**

- **Free** and **Bound** Variables:
  A variable appearing in a function **F** is said to be ***free*** if it is not bound in **F**
  Bound variables are like formal parameters, and act like local variables
  Free variables are like non-local variables that will be bound at an outer level:
  In the function **λx.xk**, **x** is bound and **k** is free

- Substitution: Applying a function

  - To apply a function, we rewrite the function, substituting all occurrences of the bound variable by the argument

- We use ***substitution*** to replace all occurrences of an identifier with an expression:
  - ✓ **[e/x]y** means "substitute **e** for all occurrences of **x** in expression **y**"

5.2. Semantic of Functional Computations:

- We define the result of a function application in terms of the following:
  - Rewriting the definition
  - Replacing bound variables with the corresponding arguments

- Rewrite rules:
  - **r1**: ***Renaming***
    - $\lambda x_i.e \Leftrightarrow \lambda x_j.[x_j/x_i]e$, where x_j is not free in **e**
    - We can replace all occurrences of the name of a bound variable with another name without changing the meaning
  - **r2**: ***Application***
    - $(\lambda x.e1)e2 \Leftrightarrow [e2/x]e1$
    - Replace the bound variable with the argument to the application

- **r3**: ***Redundant function elimination***
  - **λx.(e x) ⇔ e**, if **x** is not free in **e**

- An expression that can no longer be reduced is said to be in normal form

- **Examples:**

  (λx.(λy.x + y) 5)((λy.y * y) 6) =
  (λx.x + 5)((λy.y * y) 6) =
  (λx.x + 5)(6 * 6) =
  ((6 * 6) + 5)

  (λx. λy.x + y) 3 4
  λy.(3 + y) 4
  (3 + 4)
  7

5.3. Python Lambda Expression

- Python's lambda creates anonymous functions

```
>>> myf = lambda z: z * 42
>>> f(3)
126
```

- Only one expression in the lambda body; its value is always returned.

- Examples:

```
>>> def f (x): return x**2
...
>>> print f(8)
64
>>>
>>> g = lambda x: x**2
>>>
>>> print g(8)
64

>>> f = lambda y: y * y
>>> g = lambda y: f(x) + y
>>> x=6
>>> f(6)
36
>>> g(5)
41
>>>

>>> myLamFunction = lambda a, b: a+b
>>> myLamFunction(2,3)
5
>>>
```

6. **Functions in FPLs**

- In a functional language, the basic unit of computation is the FUNCTION.
- The function definitions typically include a name for the function, its associated parameter list, and the expressions used to carry out the computation.
- A function computes a single value based on 0 or more parameters.

  - ✓ Though the parameters of a function look like variables in an imperative language, they are different in that they are not subject to having their value changed by assignment - i.e. they retain their initial value throughout the computation of the function.

  - ✓ Pure functional languages don't need an assignment statement.

- **Function construction**: given one or more functions as parameters, as well as a list of other parameters, construction essentially calls each function and passes it the list of "other" parameters.
- **Function composition**: applying one function to the result of another. E.g. square_root(absolute_value(-3))

- **Apply-to-all functions**: takes a single function as a parameter along with list of operand values. It then applies the function to each parameter, and returns a list containing the results of each call.
- Example:

  suppose applyall carried this out with the function square and the data list (1 2 3).

  The result would be a list with the values from square(1), square(2), and square(3), i.e. (1 4 9)

- Example: A LISP factorial function, illustrating use of conditional expressions and recursion for iteration

```
(DEFUN FACT (X)
  (IF (= X 0)
     1
     ( *      X      (FACT (- 1 X))    )
  )
)
```

7. IPL vs. FPL

- Note that in imperative programming we concern ourselves with both the computation sequence and maintaining the program state (i.e. the collection of current data values).
- Unlike IPLs, purely functional languages (no variables and hence no assignments) have no equivalent concept of state: the programmer focuses strictly on defining the desired functionality.
- Iteration is not accomplished by loop statements, but rather by conditional recursion.
- Functional programmers are concerned only with functionality. This comes at a direct cost in terms of efficiency, since the code is still translated into something running on Von Neuman architecture.

8. Scheme overview

8.1. Get your own Scheme from MIT

swissnet.ai.mit.edu/projects/scheme/index.html

8.2. General overview

Scheme is a functional programming language

Scheme is a small derivative of LISP:

LISt **P**rocessing

Dynamic typing and dynamic scooping

Scheme introduced static scooping

- Data Objects
  - An expression is either an ***atom*** or a ***list***
  - An atom is a string of characters

    A
    Austria
    68000

- As in Lisp, a Scheme program is a set of expressions written in prefix notation:

  to add 2 and 3, the expression is (+ 2 3)

  to subtract 2 from 3, the expression is (- 3 2)

  to use the built-in function max to determine the maximum value from 2, 3, and 17, the expression is (max 2 3 17)

8.3. Data Typing

- Scheme uses dynamic typing (data types are associated with values rather than with variables) and uses static scoping for determining the visibility of non-local variables.

8.4. Comments

- Comments begin with a semi-colon

- Example:

  For instance, showing > as the prompt for user input, a session might look like:
  >; First some commentary, which won't get evaluated
  ; below we will provide the postfix for

```
; 2+3, and then for (2+3)+6
; and finally for (2+3)-(2*2)
; we'll start the statements to be evaluated
;  on the next line
(+ 2 3)

; Value: 5

>(+ (+ 2 3) 6)

; Value: 11

>(- (+ 2 3) (* 2 2))

; Value: 1
```

8.5. Recursion Instead of Iteration

- Since we are expressing the entire computation as a composition of functions into a single function, recursion is usually used rather than iteration
- Example:

  >; the first line is the header for the Fibonacci function:

```
(define Fibonacci (lambda (n)
        ; next is the termination case
( if (< n 3) 1
                ; and the recursive cal
                (+ (Fibonacci (- n 1)) (Fibonacci (- n 2))))))

> (Fibonacci 6)
```

; Value: 8

8.6. Evaluation

- The functional approach sometimes requires us to take a "bottom-up" view of the problem: creating functions to compute the lowest layer of values, then other functions taking those as operands.

- Example: Design a code to compute (a + b + c) / (x + y + z)

- Compute the numerator and denominator separately,

```
; for the numerator
(+ a b c)
; for the denominator
(+ x y z)
```

and then decide how to apply division with those two functions as operands, i.e.:

```
(/ (+ a b c) (+ x y z))
```

8.7. Storing and using Scheme code

The load function is available to load a Scheme program stores in a an text file, e.g.:

```
> (load "myfile.txt")

; Loading "myfile.txt" -- done
```

8.8. Variables

- Variables are always *bound* to values
- To declare and initialize a variable, we use the built in define command, giving it the variable name and the value it is to be initialized with (the value may be an expression)

Examples:

```
> (define x 3)

; Value:x

> (define foo (+ 4 7))

; Value: foo
```

Check the content of a variable:

```
>x

; Value: 3

>foo

; Value: 11
```

8.9. Data types

Literals are described as *self-evaluating*, in that evaluating the literal returns the value they represent. (E.g. evaluating 3 returns the integer value 3.)

The primitive types are:

- characters
- strings (in double-quotes)
- Booleans:
  - True: #t
  - False: The empty set for false or #f (see example below).
- Integers
- rational numbers
- real numbers
- complex numbers.

<u>List:</u> There is also a composite data type, called the ***list,*** which is a fundamental part of Scheme. Lists are considered in detail in a later section.

- Numbers

There are integers, rationals, reals, and complex numbers.

In general, Scheme will return as exact an answer as it can (i.e. it will give an exact integer or rational over a real approximation).

Examples:

Let's see the results of some basic arithmetic:

```
>(/ 3.2 1.6)
```

```
; Value: 2.

>(/ 16 10)

; Value: 8/5
```

Suppose we were to try some comparisons:

```
>(< 2 3)

; Value: #t

>(< 4 3)

; Value: ()
```

8.10. Arithmetic functions

There are many built-in arithmetic functions. Some of the commonly used ones include:

max, min

+, *, -, /

quotient, modulo, remainder

ceiling, floor, abs, magnitude, round, truncate

gcd, lcm

exp, log, sqrt

sin, cos, tan

There are also a number of comparison operators returning Boolean values

<, >, =, <=, >=

real?, number?, complex?, rational?, integer?

Example:

(complex? 4+3i)

;Value: #t

zero?, positive?, negative?, odd?, even?, exact?

Examples:

>; does 7 divided by 3 produce an integer result?
(integer? (/ 7 3))

; Value: ()

>; does 7 divided by 3 produce an exact result?
(exact? (/ 7 3))

; Value: #t

(Note that rational values *are* considered exact.)

- Boolean functions

  and, or, not

  equal?, Boolean?

  E.g., check to see if three is less than seven and two is not equal to four

  >(and (< 3 7) (not (= 2 4)))

  ; Value: #t

8.11. Selection functions

- Selection in a functional language still controls the choice between different computations, but is expressed by returning the results of functions representing the different computations.

- The two major Boolean control operations are:

  IF
  COND.

  - IF:

    For example, suppose if x is less than 0 we want to return y - x: (if (< x 0) (- y x))

    Now suppose that if x is less than 0 we want to return 0, otherwise we want to return the value x - 1:

    ```
    (if (< x 0) 0
                (- x 1))
    ```

  - COND statement is somewhat like the C switch statement, allowing a series of conditions to test for (with corresponding functions to evaluate and return) and a default case:

    ```
    (cond ((= x y) 0)
          ((> x y) 1)
          (else   -1)
    )
    ```

- Lists

    - ✓ Lists are the main composite data type in Scheme.

    - ✓ Lists are composed of a series of elements, enclosed in brackets.

    - ✓ **Implementation note:** the typical implementation format for lists is to represent each element in a list using two pointers:

        - One points to the actual implementation of the element (hence allowing us to use anything we like as a list element, the pointer can refer to a primitive data element, a list, a string, etc)
        - The other points to the next element in the list

    - ✓ Example:

        - (a b c d) has the four elements a, b, c, and d.
        - The empty list is denoted ()

    - ✓ Examples of lists include

```
'(a)     ; a list with a single element
'(a b c) ; a list with three elements
'()      ; an empty, or null, list
'((a b)) ; a list with a single element, which
           happens to be another list
```

'("blah" 3.7 () (a b) c) ; a list with 5 elements of a variety of types

Head:

The front element of the list

It is always an element

Tail:

The list of the remaining elements.

It is always a list

Single quote:

It is used to denote elements which are actually lists (see the examples in list functions below)

- List functions

  - Constructing lists:

    - ✓ (list a b c d): creates a list of the given elements (a b c d)
    - ✓ (append '(a b) '(c d)): joins the two lists to create list (a b c d)
    - ✓ (cons a '(b c d)): adds the first operand at the head of the other list to create a new list (a b c d)
    - ✓ Note that (cons '(a) '(b c d)) would add the *list* (a) as the head element, giving ((a) b c d)
    - ✓ CAR function returns the head element of a list, i.e. (car '(a b c d)) gives a
    - ✓ CDR function returns the tail of a list, i.e. (car '(a b c d)) gives (b c d)

  - Examples:

```
>(define mylist (list 1 2 3 4 5))
; Value: mylist
>mylist
; Value: (1 2 3 4 5)
>(length mylist)
; Value: 5
>(reverse mylist)
; Value: (5 4 3 2 1)
>mylist
; Value: (1 2 3 4 5)
>(reverse (cdr mylist))
; Value: (4 3 2 1)
```

- ✓ Observe that the functions applied to mylist are NOT altering the list itself - they are returning manipulated copies of the list.

8.12. Iteration

- Scheme do expression is similar to a C for loop.

```
(do
        (( variable init step)...)
        ( test test-expression ...)
                body-expression ...
)
```

- Example:

```
(do                 ; for(i=1;i<10;i++)
        ( (i 1 (+ i 1)))
        ((> i 10))
                (write i)
                (write-char #\newline)
)
```

- *step* part may be omitted

```
(do
        (       ; for(i=10,sum=0;i!=0;i--)
                (i 10 (- i 1))
                (sum 0)
        )
        ((= i 0)
                (write-char #\newline)
                (write "The sum is:")
                (write sum)
        )
        (set! sum (+ sum i))
)
```

8.13. Defining functions

- User-defined functions can be created through the use of the lambda operator as follows:

  **(define functionname (lambda (functionparameters)
  (expression) (expression) ... (expression)))**

  NOTE: The value returned by the function is the value of the last expression in the list

  ✓ Example: For example, the function below calculates factorials:

```
>(define factorial (lambda (n)
        ( if (< n 3) n
             (* n (factorial (- n 1))))
))

; Value: factorial

>(factorial 3)

; Value: 6
```

9. 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 (See Chapter 4)

- 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 operations

- 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;`

10. Haskell

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

- 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*:
    - List functions Uses lazy evaluation (evaluate no sub-expression until the value is needed)
    - Has "list comprehensions," which allow it to deal with infinite lists

- *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)
    ```

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]

- Catenation is with ++

  e.g., [1, 3] ++ [5, 7]

  results in [1, 3, 5, 7]

- CAR and CDR via the colon operator (as in Prolog)

  e.g., 1:[3, 5, 7]

  results in [1, 3, 5, 7]

11. Functional Programming Using Python

- A list is a collection of objects.
- List constants are surrounded by square brakets and the elements in the list are separated by commas.
- Lists are "**mutable**" - we can change an element of a list using the index operator
- Examples:

```
myFriendsList = [ 'Paul', 'Mary', 'Sally' ]
myNums = [1,2,3,4]
myList = []
```

- A list can element of another list:

```
>>> myList2 = [1, 2]
>>> myList3 = ['a', myList2, 50]
>>> myList2
[1, 2]
>>> myList3
['a', [1, 2], 50]
>>>
```

11.1. List Manipulation

- Building a list from scratch:

```
>>> myNewList = []
>>> myNewList.append(1)
```

```
>>> myNewList
[1]
>>> myNewList.append(2)
>>> myNewList
[1, 2]
>>>
```

- Generate a list by tokenizing a text:
  - Use **split(delimiter)** function:
    - When delimiter is omitted, space(s) is used.
  - Examples:

```
>>> myPhrase = "The quick brown fox jumps over the moon"
>>> myPhraseList = myPhrase.split()
>>> myPhraseList
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'moon']
>>>
```

- Does an element exist in a list?
  - Python provides two operators that let you check if an item is in a list:
    - in
    - not in
  - These operators do not change the content of the list.
  - Examples:

```
>>> myList = [1, 9, 21, 10, 16]
>>> 10 in myList
```

```
True
>>> 2 in myList
False
>>> 9 not in myList
False
>>>
```

- Append to list:

  - Add an element to a list without changing the list:

    ```
    >>> myList = ['tot', 1, 3, 'foo', 'python', 'c']
    >>> myList
    ['tot', 1, 3, 'foo', 'python', 'c']
    >>> myList + ['new']
    ['tot', 1, 3, 'foo', 'python', 'c', 'new']
    ```

  - Change the content of the list:

    ```
    >>> myList
    ['tot', 1, 3, 'foo', 'python', 'c']
    >>> myList.append('Java')
    >>> myList
    ['tot', 1, 3, 'foo', 'python', 'c', 'Java']
    >>>
    ```

- Merge two lists:

  ```
  >>> myList1 = myList + myList
  >>> myList1
  ['tot', 1, 3, 'foo', 'python', 'c', 'Java', 'tot', 1, 3, 'foo',
  'python', 'c', 'Java']
  >>>
  ```

```
>>> myList = ['a', 'b', 100]
>>> myList1 = 3*myList
>>> myList1
['a', 'b', 100, 'a', 'b', 100, 'a', 'b', 100]
>>>
```

- List Operations:
  - Accessing a specific element of a list:
    ```
    >>> mylist
    [1, 2, 3, 4]
    >>> mylist.index(3)
    2
    ```
  - Removing elements:
    ```
    >>> mylist.remove(2)
    >>> mylist
    [1, 3, 4]
    ```
  - Counting Occurences of an element:
    ```
    >>> mylist.count(1)
    1
    >>> mylist = [1,2,3,3,3,4,5]
    >>> mylist.count(3)
    3
    >>> mylist = [2,2,2,4,4,5,6]
    >>> mylist.count(2)
    3
    ```

- Sorting a List:

```
>>> myList.sort()
>>> myList
[1, 9, 10, 16, 21]
>>>

>>> myList = ['z', 'b', 'o', 'b', 'p', 'a']
>>> myList.sort()
>>> myList
['a', 'b', 'b', 'o', 'p', 'z']
>>>
```

11.2. Car & CDR

- Car/cdr Function:
  - Example:

```
>>> myList =['tot', 1, 3, 'foo', 'python', 'c']
>>> myList[0]        # like (car myList)
'tot'
>>> myList[1:]       # like (cdr myList)
[1, 3, 'foo', 'python', 'c']
>>>
```

  - Example:
    - In Lisp

```
(DEFINE (member atm lis)
  (COND
    ((NULL? lis) '())
```

```
((EQ? atm (CAR lis)) #T)
((ELSE (member atm (CDR lis)))
))
```

- In Python:

```
def member(atm, lis):
    if len(lis) == 0:
        return (False)
    elif atm == lis[0]:
        return(True)
    else:
        return(member(atm, lis[1:]))
```

12. Functions

- Functions can be used as any other datatype.
- They can even be used as arguments to functions
    - Example:

```
>>> >>> def square(x): return x*x
>>> def useFunction(anyfunction, x): return anyfunction(x)

>>> useFunction(square,4)
16
>>>
```

12.1. Composition

```
>>> def powerOf2 (n):
        return (2**n)

>>> powerOf2(2)
4
>>> def doublePower(n):
        return(powerOf2(powerOf2(n)))

>>> doublePower(2)
16
>>>
```

12.2. Apply-to-all functions

- Max/Min Function:

```
>>> myNums
[1, -1, 200, 12, 33, 400, 0]
>>> print max(myNums))
SyntaxError: invalid syntax
>>> print (max (myNums))
400
>>>
>>> print (min (myNums))
-1
>>>
```

- Length Function:

```
>>> myList = [1,2,3,4,5,6]
>>> print (len(myList))
6
>>>
```

- **Map, Reduce, Filter Functions**
  - Let us now look at 3 higher order functions which are used a lot in functional
  - **Map**:
    - Map takes a function and a list and applies the function to each elements in the list
    - Example:

```
>>> g = lambda x: x**2
```

```
>>> mylist = [1,2,3,4]
>>> print (list (map(g, mylist)))
[1, 4, 9, 16]
>>>
```

- **Reduce**:
  - The function reduce(func, seq) continually applies the function func() to the list. It returns a single value.
  - Example:

```
>>> from functools import reduce
>>> myList = [1,2,3,4,5,6]
>>> k = lambda x,y: x+y
>>> print  (reduce(k,mylist))
99
>>>
```

- **Filter**:
  - Filter takes a function (what type) and a list, and returns items which pass the test
  - Example:

```
>>> mylist = [10,4,6,9,11,7,22,30]
>>> h = lambda x: x % 3 == 0
>>> print (list (filter(h, mylist)))
[6, 9, 30]
>>> h = lambda x: x % 2 == 0
>>> print (list (filter(h, mylist)))
[10, 4, 6, 22, 30]
```