Introduction to Functional Programming

Functional programming (FP): a style of programming in which computation is accomplished via the evaluation of functions*

*Mathematical functions, not procedures.

Why FP is important

- Elements of FP languages, such as lambda (i.e., nameless) functions and high-order functions, are steadily being incorporated into commonly-used languages, including:
  - C# and VB.NET
  - Java
  - Python
  - Ruby

Why FP is important

- The FP style inherently solves many of the issues that arise when writing programs that execute concurrently
- Purely functional programs tend to be more concise than their counterparts in other paradigms.

What is FP?

Programs written in a FP language are *declarative*; they specify the overall computation that needs to be performed, but not the order in which to solve it.

```
def f(x,y):
    print (x + y)

f(3,4)
```

What is FP?

In contrast, programs written in *imperative* languages, such as C, C++, C#, and Java, are a series of instructions to the computer:

```
int x = 3;          //First, set x to the value 3
int y = 4;          //Next, set y to the value 4
int z = x + y;      //Then, set z to the result of
                    //adding x to y
printf(“%i”, z);    //Finally, follow the steps in
                    //the printf subprocedure
```

What is FP?

Example: Return the n^th^ element of a linked list

Imperative (C)

```
int f (ListNode *head, int n) {
      ListNode *cur = head;
      while (n != 0 && cur != NULL)
{
            cur = cur->next;
            n = n - 1;
      }
      if (cur != NULL)
            return cur->data;
      else {
            //handle the error
      }
}
```

Declarative (Python)

```
def f (node, n) :
      if (node is None):
            //handle the error
            pass
      elif n == 0:
            return node.data
      else:
            return f(node.next,
n-1)
```

What is FP?

Example: Return all elements of a list greater than a threshold

Imperative (C)

```
ListNode * query (ListNode *head,
                  int threshold) {
       ListNode *cur = head;
       ListNode *tmp, *results = NULL;
       while (cur != NULL) {
              if (cur->data > threshold) {
                     tmp = new ListNode();
                     tmp->data = cur->data;
                     tmp->next = results;
                     results = tmp;
              }
       }
       return results;
}
```

Declarative (SQL)

```
SELECT value
FROM table
WHERE value > threshold;
```

What is FP?

Summary:

- Programs in declarative languages describe what computation needs to be performed.
- Programs in imperative languages describe how to perform computation.

Common Elements of Functional Languages

1. Side-effects (or lack thereof)

2. Iteration (or lack thereof)

3. Higher-order functions

Common Elements of Functional Languages

1. Side-effects

A subprocedure is said to have *side effects* when executing the subprocedure causes a change in state or some other observable interaction (such as I/O).

```
int numInvoks = 0;
int foo(void) {
  numInvoks++;
  if (numInvoks < 5)
    return 1;
  else
    printf("No foo");
}
```

```
int bar(int x, int y, int *ans)
{
       if (y == 0)
              return 0;
       else {
              *ans = x / y;
              return 1;
       }
}
```

Common Elements of Functional Languages

1. Side-effects

A subprocedure is said to have *side effects* when executing the subprocedure causes a change in state or some other observable interaction (such as I/O).

```
int numInvoks = 0;
int foo(void) {
  numInvoks++;
  if (numInvoks < 5)
    return 1;
  else {
    printf("No foo");
    return 0;
  }
}
```

```
int bar(int x, int y, int *ans)
{
      if (y == 0)
            return 0;
      else {
            *ans = x / y;
            return 1;
      }
}
```

Common Elements of Functional Languages

1. Side-effects

Side effects are the reason that imperative programs have to be executed in order; if a program does not have any side effects, it will evaluate to the same value regardless of how it is executed.

Common Elements of Functional Languages

1. Side-effects

Code that is side-effect free is helpful when:

- programming concurrently
- optimizing via memoization
- unit testing

Common Elements of Functional Languages

1. Side-effects

Imperative programming heavily depends on side effects; imagine writing a C program that never assigns to a variable!

On the other hand, side effects are actively avoided in FP:

- Most data are immutable—once created, their value will not change.
- Assignment expressions are a rarity, if they are used at all; most functions are stateless.

Common Elements of Functional Languages

2. Iteration

In most imperative languages, for loops and while loops heavily depend on side-effects:

```
ListNode *cur;
for (cur = head;
     cur != NULL;
     cur = cur->next)
{
     processNode(cur);
}
```

```
ListNode *cur;
cur = head;
while (cur != NULL)
{
     processNode(cur);
     cur = cur->next;
}
```

Question: How can you iterate without side effects?

Common Elements of Functional Languages

2. Iteration

Answer: Recursion!

Recursion is the process of defining a function in terms of itself. A recursive function typically handles a large problem by first finding a solution to a slightly smaller version, and then building from this solution to solve the original problem. Note that the slightly smaller problem will be solved in the same way!

Example: Factorial Function

```
n! = 1, if n == 0
     n * (n-1)!, otherwise
```

Common Elements of Functional Languages

2. ~~Iteration~~ Recursion

Typically, a recursive solution to a problem will have a *base case* that is trivial to solve, and an *inductive case* that shows how to solve larger versions.

Example: Factorial Function

```
n! = 1, if n == 0                 Base case
     n * (n-1)!, otherwise        Inductive case
```

Common Elements of Functional Languages

2. Recursion

Example: Reverse a linked list.

```
def reverse(x):
      if x:
            #x has at least 1 element; call it y
            #the reverse of x will be the result of
            #appending y to the reverse of x without y
            return reverse(x[1:]) + [x[0]]
      else:
            #x must be an empty list.
            #the reverse of an empty list is an empty list
            return []
```

Note: This example makes use of Python's list-slicing syntax. The expression x[1:] should be read as “All elements of the list x, starting from index 1.”

Common Elements of Functional Languages

2. Recursion

Example: Create a list of n increasing integers, e.g. [0,1,2,3].

Sometimes, recursive calls need more parameters than the function interface should require. In these cases, *helper functions* can be used to pass the extra data.

```
def helper(n, x):
	if x == n:
		return []
	else:
		return [x] + helper(n, x
+ 1)

def counter(n):
	return helper(n, 0)
```

Common Elements of Functional Languages

2. Recursion

Example: Create a list of n increasing integers, e.g. [0,1,2,3].

Sometimes, recursive calls need more parameters than the function interface should require. In these cases, *helper functions* can be used to pass the extra data.

Alternative Solution

```
def helper(n, x):
     if len(x) == n:
          return x
     else:
          return helper(n, x + 
[len(x)])

def counter(n):
     return helper(n, [])
```

Common Elements of Functional Languages

2. Recursion

In imperative languages:

- Looping is easy and fast
- Recursion has overhead; each call creates a stack frame

In functional languages:

- Looping is heavily discouraged
- Can be efficient via advanced data structures outside of the scope of this class

Common Elements of Functional Languages

3. Higher-order functions

Write two functions: one that adds 1 to every integer in a list, and another that multiplies every integer in a list by 2.

Common Elements of Functional Languages

3. Higher-order functions

Write two functions: one that adds 1 to every integer in a list, and another that multiplies every integer in a list by 2.

```
void addOne(ListNode *head) {
      ListNode *cur = head;
      while (cur != NULL) {
            cur->data = cur->data +
1;
            cur = cur->next;
      }
}
```

```
void multTwo(ListNode *head) {
      ListNode *cur = head;
      while (cur != NULL) {
            cur->data = cur->data *
2;
            cur = cur->next;
      }
}
```

Notice that all the looping code gets duplicated!

Common Elements of Functional Languages

3. Higher-order functions

A better way: refactor the looping code common to the `addOne` and `multTwo` functions into a single function.

```
def each(node, f):
      if node:
            node.data =
f(node.data)
            each(node.next, f)

def addOne(node):
      each(node, lambda x: x +
1)

def multTwo(node):
      each(node, lambda x: x *
2)
```

Common Elements of Functional Languages

3. Higher-order functions

Higher-order function: a function that accepts a function as a parameter, or returns a function as its result

```
def each(node, f):
     if node:
          node.data =
f(node.data)
          each(node.next, f)

def addOne(node):
     each(node, lambda x: x +
1)

def multTwo(node):
     each(node, lambda x: x *
2)
```

Common Elements of Functional Languages

3. Higher-order functions

First-class functions: functions that can be dynamically created, assigned to variables and be passed as arguments to (or returned as values from) other functions, like any other kind of data.

```
def each(node, f):
     if node:
          node.data = f(node.data)
          each(node.next, f)

def addOne(node):
     each(node, lambda x: x + 1)

def multTwo(node):
     each(node, lambda x: x * 2)
```

Common Elements of Functional Languages

3. Higher-order functions

First-class functions: functions that can be dynamically created, assigned to variables and be passed as arguments to (or returned as values from) other functions, like any other kind of data.

Question: Does C++ have first-class functions?

Common Elements of Functional Languages

3. Higher-order functions

First-class functions: functions that can be dynamically created, assigned to variables and be passed as arguments to (or returned as values from) other functions, like any other kind of data.

Question: Does C++ have first-class functions?

Answer: No. Function pointers can mimic some of these behaviors, but there is no way to create new functions dynamically.

Common Elements of Functional Languages

3. Higher-order functions

Functional language: a language that has first-class functions.

```
def each(node, f):
    if node:
        node.data = f(node.data)
        each(node.next, f)

def addOne(node):
    each(node, lambda x: x + 1)

def multTwo(node):
    each(node, lambda x: x * 2)
```

Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

map: Given a function for transforming data, and a list of data to be transformed, return a list of the transformed data.

Examples:

```
> map (lambda x: x + 2, [1, 2, 3])
[3, 4, 5]
> map (lambda x: chr(x), [104, 101, 108, 108, 111])
['h', 'e', 'l', 'l', 'o']
```

Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

map: Given a function for transforming data, and a list of data to be transformed, return a list of the transformed data.

```
def map (f, xs):
    if xs:
        return [f(xs[0])] + map (f, xs[1:])
    else:
        return []
```

Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

filter: Given a function deciding whether an item should be kept, and a list of items, return a list containing only items that should be kept.

Examples:

```
> filter (lambda x: x % 2 == 1, [1, 2, 3])
[3, 5]
> filter (lambda x: x.isupper(), “My Name Is Hal”)
['M', 'N', 'I', 'H']
```

Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

filter: Given a function deciding whether an item should be kept, and a list of items, return a list containing only items that should be kept.

```
def filter (f, xs):
    if not xs:
        return []
    elif f(xs[0]):
        return [xs[0]] + filter(f, xs[1:])
    else:
        return filter(f, xs[1:])
```

Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

reduce: Given a function for combining a list item and a seed value into a new seed value, a list of items, and an initial seed value, recursively apply the function to the list items to arrive at a single value

Examples:

```
> reduce (lambda seed,x: seed + x, [1, 2, 3], 0)

6

> reduce (min, [5,8,4,2,1,7,6,3,9], 10)

1

> reduce (min, [5,8,4,2,1,7,6,3,9], 0)

0
```

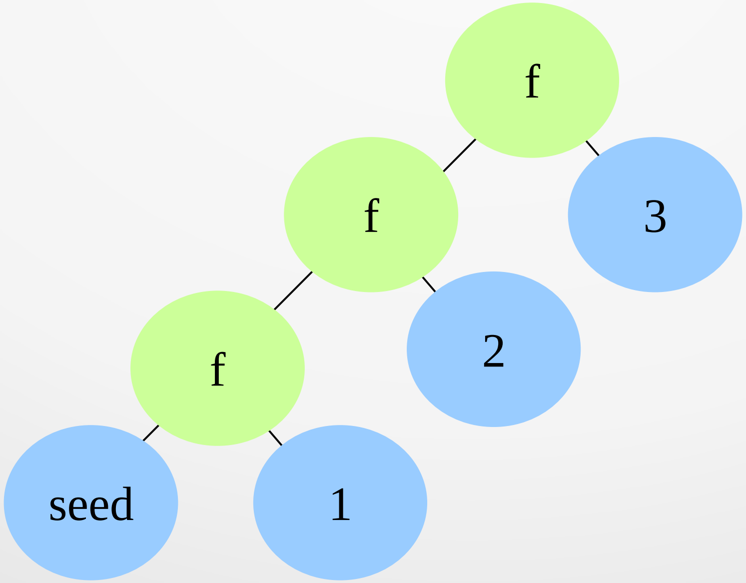
Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

reduce: Given a function for combining a list item and a seed value into a new seed value, a list of items, and an initial seed value, recursively apply the function to the list items to arrive at a single value

```
reduce(f, [1,2,3], seed)
```



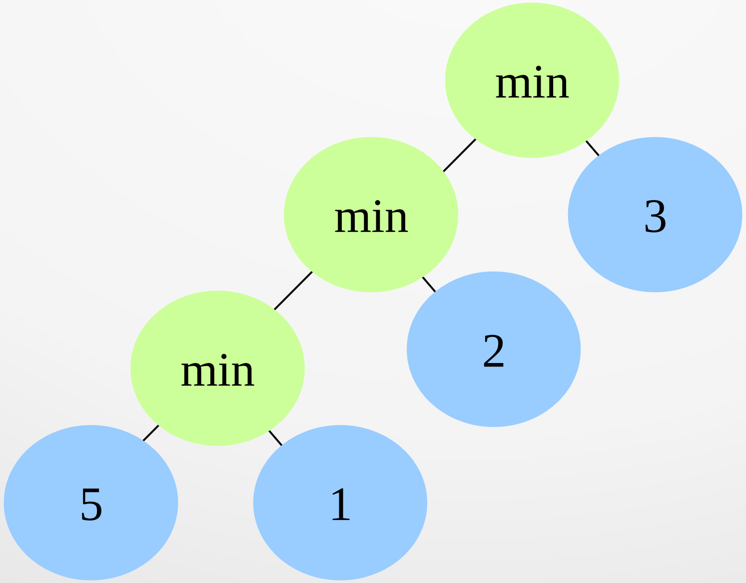
Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

reduce: Given a function for combining a list item and a seed value into a new seed value, a list of items, and an initial seed value, recursively apply the function to the list items to arrive at a single value

```
reduce(min, [1,2,3], 5)
```



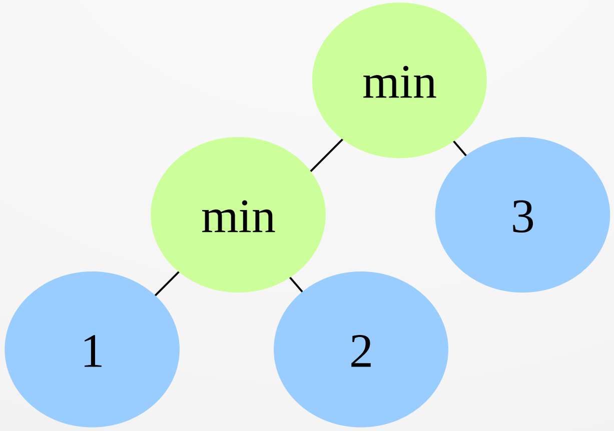
Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

reduce: Given a function for combining a list item and a seed value into a new seed value, a list of items, and an initial seed value, recursively apply the function to the list items to arrive at a single value

```
reduce(min, [1,2,3], 5)
```



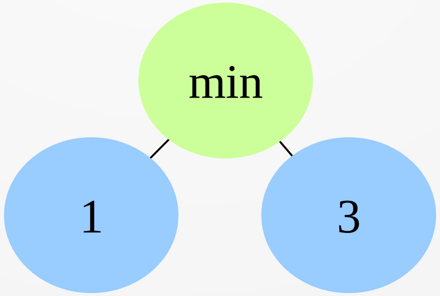
Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

reduce: Given a function for combining a list item and a seed value into a new seed value, a list of items, and an initial seed value, recursively apply the function to the list items to arrive at a single value

```
reduce(min, [1,2,3], 5)
```



Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

reduce: Given a function for combining a list item and a seed value into a new seed value, a list of items, and an initial seed value, recursively apply the function to the list items to arrive at a single value

```
reduce(min, [1,2,3], 5)
```

1

Common Elements of Functional Languages

3. Higher-order functions

Important higher-order functions:

reduce: Given a function for combining a list item and a seed value into a new seed value, a list of items, and an initial seed value, recursively apply the function to the list items to arrive at a single value

```
def reduce(f, xs, seed):
    if xs:
        return reduce(f, xs[1:], f(seed, xs[0]))
    else:
        return seed
```