Higher-Order Programming: The Essence of Functional Programming

- Higher-Order programming is defined as
  - Programming with functions as arguments to other functions or functions as return values from functions.
Higher-Order Programming: A Cornerstone of Functional Programming

- It is a natural outgrowth from the lambda calculus where
  - a) lambda expressions can be passed to other lambda expressions, and
  - b) new lambda expressions can be computed by lambda expressions

- E.g.

  a) $(\lambda y. y + 1)(\lambda x. x + 1) \Rightarrow 2$

  b) $(\lambda x. (\lambda y. x + y)) 1 1 \Rightarrow 2$
Modifying Behavior of a Function

- We can use this to write generic functions which we can then make specific by passing in desired behavior via a function.

- Note: this is NOT programming with generics
  - Generics are generic with respect to types
  - Higher-order functions are generic with respect to behavior!
Modifying Behavior of a Function

- The ‘filter’ function is generic with regards to the ordering predicate.

```plaintext
function filter
  with ([],_,_) do
    []
  with ([e|rest],pivot,fcmp) do
    [e] + filter (rest,pivot,fcmp)
    if fcmp (e,pivot)
      else filter (rest,pivot,fcmp)
  end
```

If `e` is kept or discarded depends on the passed in function – the filter function has a generic filtering capability which is made specific by the passed in predicate.
Modifying Behavior of a Function - QuickSort

function filter
  with (e, [], fcmp) do
    []
  with (e, [a | rest], fcmp) do
    [a] + filter(e, rest, fcmp)
    if fcmp(a, e)
      else filter(e, rest, fcmp)
end

function qsort
  with [] do -- empty list
    []
  with [pivot | rest] do -- head-tail operator
    let less = filter(pivot, rest, lambda with (x, y) do x < y).
    let more = filter(pivot, rest, lambda with (x, y) do x >= y).
    qsort less + [pivot] + qsort more.
end
Modifying Behavior of a Function

- The Asteroid sort module is another example,
  \[
  \text{sort } @\text{sort}(p:\%\text{function}, l:\%\text{list})
  \]

```plaintext
1 load system sort.
2
3 let sl = sort @sort(
4 (lambda with (x,y) do true if x<y else false),
5 [10,5,110,50]).

6 assert (sl == [5,10,50,110]).
```
Dispatch Tables

- We can also associate behavior with appropriate keys in a dispatch table.
- We can then dispatch (lookup) desired behavior given specific keys.
- Example: A generic ‘calculate’ function that takes two values and a key symbol and then performs the appropriate computation.
Dispatch Tables

```
1 load system hash.
2
3 let dispatch_table = hash @hash ()
4
5 dispatch_table @insert [ 
6     ("+", lambda with (a,b) do a + b),
7     ("-", lambda with (a,b) do a - b),
8     ("*", lambda with (a,b) do a * b),
9     ("/", lambda with (a,b) do a / b)
10 ]
11
12 function calculate with (operator,a,b) do
13     dispatch_table @get operator (a,b)
14 end
15
16 -- Example usage
17 assert (calculate("+", 3, 5) == 8)
18 assert (calculate("-", 7, 2) == 5)
19 assert (calculate("*", 2, 4) == 8)
20 assert (calculate("/", 10, 2) == 5)
```
Map & Reduce

- The ‘map’ and ‘reduce’ functions are functions that take a function and apply the given function to ALL the elements of a list.

- Both functions are higher-order functions that come straight out of the functional programming tradition.
Below is Asteroid code that explains the behavior of the map function.

Beware that map is not required to apply the function \( f \) in the sequential manner shown here.

For example, it is free to exploit threads to apply the function \( f \) in parallel to the elements of the list.

```
list @map f

-- is equivalent to --

let r = [].
for e in list do
  r @append(f e).
end
let list = r.
```

The function argument to \( f \) must be of the same type as the list elements.
One interesting application of map is the transformation of a simple list constructor into any kind of list. Here we compute a list of alternating 1’s and -1’s.

```
load system math.

let a = [1 to 10] @map(lambda with x do math @mod (x,2))
     @map(lambda with x do 1 if x else -1).

assert (a == [1,-1,1,-1,1,-1,1,-1,1,-1,1,-1]).
```
Most modern languages support some form of ‘map’ since it is such a powerful programming tool.

Python

```python
l = [x for x in range(1,10+1)]
it = map(lambda x : x % 2, l)
a = list(map(lambda x : 1 if x else -1, it))
assert(a == [1,-1,1,-1,1,-1,1,-1,1,-1])
```

In012/map.py

Rust

```rust
use std::vec::Vec;

fn main() {
    let a : Vec<i32> = vec![1,2,3,4,5,6,7,8,9,10].iter().map(|x| x % 2).map(|x| if x == 0 { -1 } else { 1 }).collect();
    assert_eq!(a, vec![1, -1, 1, -1, 1, -1, 1, -1, 1, -1]);
}
```

In012/map.rs
Reduce

- Whereas ‘map’ applies a function to a list producing another list, the ‘reduce’ function applies a function to a list so that the list gets reduced to a single value.

  - In functional languages this is often called ‘fold’ – folding the list into a single value
Reduce

- For example, the reduce function lets us sum the elements of a list without a loop.

```plaintext
let value = [1,2,3] @reduce (lambda with (x,y) do x+y).
assert(value == 6).
```

The argument of the reduce function must be a pair where each component of the pair is of the element type of the list.
The reduce function gives us an interesting way to implement the factorial of an integer.

```plaintext
1   -- factorial with reduce
2   function fact with x do
3     [1 to x] @reduce (lambda with (i,j) do i*j)
4   end
5
6   assert (fact 3 == 6).
```
Reduce

- The Asteroid code below illustrates the behavior of the ‘reduce’ function
  - Notice the function application to a pair of values!
  - The **first value of the pair acts like an accumulator** containing the partially reduced value at each function application
    - In the default setting it is initialized to the first element of the list.

```
1  list @ reduce f
2
3  -- is equivalent to --
4
5  let value = list@0.
6  for i in range(len(list)) do
7  |     let value = f (value, l@i).
8  end
9  -- value has now the reduced value of the list
```
Reduce

Python

```python
1 from functools import reduce
2
3 value = [1, 2, 3]
4 result = reduce(lambda x, y: x + y, value)
5
6 assert result == 6
```

Rust

```rust
fn main() {
    let value: i32 = (1..10).reduce(|acc, e| acc + e).unwrap();
    assert_eq!(value, 45);
}
```