Where are we?

- We're used to Structured and OO
- What are those?
- Remember Structured programming emphasizes goto-less programming
  - goto still happen behind the scene, hidden from us
- OO emphasizes encapsulation and polymorphism
What’s Functional Programming?

- It’s a different way of programming
- It is
  - Assignment-less
  - Higher level of abstraction
  - Expressions with no side effects
  - Enables massive parallelism due to execution order independence

But, Why FP?

- We’re being dragged into it
- Dual core processors
- Multi-core processors
- High performance demand
- We need software to
  - function correctly
  - Take performance advantage of hardware capabilities
  - Need higher level of abstraction
What about familiar languages?

- What about languages like Ruby, Groovy, ...?
- These are OO languages (we will look at these, but keep in mind, these are not functional langs)
- They have some features that have been borrowed from functional programming (like closures)
- So, you’re already using it to some extent
- These certainly bring the power, but we need more to tackle the evolving complexities

But, What’s really FP?

- At the core of Functional Programming is higher order functions
- Works with Functional Abstraction and Mathematical Logic
What's a Function?

- Function maps input to output
- But, what about algorithm, performance,...
- OK, function is what it does and how it does it
- A pure function does not perform any assignment operations—implicitly or explicitly
- Independent evaluation order of subexpressions allow us to exploit multiprocessors
- Referential Transparency—allows elimination of common subexpressions

Equations

- Imperative programs define variables explicitly—set values to variables—easier to program
- Functional programming languages do so implicitly—easier for formal specification
- \( a \equiv 3 \) Explicitly defines \( a \) to be 3
  - \( a \) can now be substituted by 3
- \( x = 2x - 3 \) Implicitly defines \( x \) to be 3
- \( y = 2a + 5 \Rightarrow 2 \ast 3 + 5 \Rightarrow 11 \)
- \( x = 2a - 7 \) and \( a = x - 7 \); \( x \) and \( a \) implicitly defined using each other—\( x \) is 21 and \( a \) is 14.
Functions

- Functional programming languages try to express functions as recursions.
- LISP showed how significant programs can be expressed as pure functions on list structures.
- Promotes passing functions as arguments to functions—Higher Order Functions.
- Functions operate on other functions without assignments or side effects.

Higher Order Functions in Erlang

```erlang
%% We are sending an anonymous method (fun) to map method.
%% map calls that method on each element in sequence.

[4,8,18,14,2]
```
Similar Concept in Groovy

```groovy
lst = [2, 4, 9, 7, 1]
println lst.collect { it * 2 }
```

Similar Concept in Scala

```scala
val lst = List(2, 4, 9, 7, 1)
println(lst.map(_ * 2))
```

List(4, 8, 18, 14, 2)
### Expressing Functions As Recursion

#### Scala

```scala
def fact(n: Int) : Int = n match {
  case 0 => 1
  case _ => n * fact(n-1)
}

println(fact(5))  // 120
```
Expressing Functions As Recursion

```scala
#!/usr/bin/env escript
main(_) ->
    io:format("-p", [fib_seq(10)]).

fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

fib_seq(0) -> [Fib(0)];
fib_seq(N) ->
    fib_seq(N-1) ++ [Fib(N)].
%Series where each element is the sum of previous two elements...

[0,1,1,2,3,5,8,13,21,34,55]
```

Expressing Functions As Recursion: Scala

```scala
def fib(n:Int) : Int = n match
    { case 0 => 0
      case 1 => 1
      case _ => fib(n-1) + fib(n-2)
    }
def fib_seq(n:Int) : List[Int] = n match
    { case 0 => List(fib(0))
      case _ => fib_seq(n-1) ::: List(fib(n))
    }
println("[" + fib_seq(10).mkString(" ") + "]")
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```
What's Going on Here?

• Pattern Matching is at work
• When you call fib(3), it calls fib(N)
• When fib(N) calls fib(1), it ends up in fib(1) method instead of fib(N)

Let’s Explore Pattern Matching Further

```erlang
#!/usr/bin/env escript
main(_) ->
    io:format("Max is ~p\n", [max(1, 2)]),
    io:format("Max is ~p\n", [max(2, 1)]).
max(A, B) when A > B -> A;
max(_, B) -> B.
```

Max is 2
Max is 2
Pattern Matching and Recursion at Play

```
#!/usr/bin/env escript
main(_) ->
io:format("Max is ~p\n", [max([1, 2, 5, 2, 6, 1, 8, 1, 5])]).
max([[]]) -> H;
max([[[]]]) -> max2(H, max(T)).
max2(A, B) when A > B -> A;
max2(_, B) -> B.
```

Max is 8

Pattern Matching and Recursion at Play: Scala

```
def max2(a:Int, b:Int) : Int = if (a > b) a else b

def max(lst: List[Int]) : Int =
{
    lst.tail.isEmpty match
    {
        case true => lst.head
        case _ => max2(lst.head, max(lst.tail))
    }
}
println(max(List(1, 2, 5, 2, 6, 1, 8, 1, 5))) 8
```
Another Solution: Scala

```scala
def max(lst: List[Int]) : Int =
{
  (lst(0) /: lst) {((max, e) => if (e > max) e else max)}
}

println(max(List(1, 2, 5, 2, 6, 1, 8, 1, 5)))
```

Curried Function

- Transforming a function that takes multiple arguments into a function that takes a single argument

- $f(X, Y) \rightarrow Z$ is transformed into
  
  $$\text{curry}(f) : X \rightarrow (Y \rightarrow Z)$$

- Makes it easier to express

- Some languages have these built in

- Named after Haskell B. Curry

- Actual work by Moses Schönfinkel and Friedrich Ludwig Gottlob Frege
Curried Closure in Groovy

```groovy
def doWork(val, closure)
{
  curried = closure.curry("preset_value")
  curried(val * 2)
  curried(val * 10)
}
doWork(3) { p1, p2 -> println "Received ${p1} and ${p2}" }
```

Received preset_value and 6.
Received preset_value and 30.

λ-Calculus

- Formal system, introduced by Alonzo Church and Stephen Kleene, for function definition, function application, and recursion
- `<λ-term> ::= <variable>`
  - `| (<variable>< λ-term>)` function definition
  - `| (λ<term>< λ-term>)` function application
- `<variable> ::= x | y | z ...`
- For example `(λx(λy((+x)y)))` for `f(x, y) = x + y`
- Formalization for computability using transformations and substitutions
- Lead to Church-Turing theorem that it is impossible to decide algorithmically if general statements in arithmetic are true or false (Entscheidungsproblem or decision problem)
- "A little bit of syntax sugar helps you to swallow the λ-calculus"—Peter J. Landin
Types

- Atoms are numbers, booleans, strings, non-composites, indivisible

- Sequences are composites and dividable
  - \( \{x_1, x_2, ..., x_n\} \in T \) if \([x_1, x_2, ..., x_n] \in [T]\)

- An abstract types is specified in terms of abstract values without regard to any specific concrete implementation

Operations on Sequences

- first: \([x_1, x_2, ..., x_n] \rightarrow x_1\)

- rest: \([x_1, x_2, ..., x_n] \rightarrow [x_2, ..., x_n]\)

- prefix: \(x, [y_1, y_2, ..., y_n] \rightarrow [x, y_1, y_2, ..., y_n]\)

- ...

...
Operations on Sequences

Operations on Sequences: Scala

```scala
def countEven(lst: List[Int]) : Int =
{
  (0 :: lst) { (even, e) => even + (1 - (e % 2)) }
}

println(countEven(List(1, 2, 5, 2, 6, 1, 8, 1, 5)))
```
Assignments in FPLs

- X = 3
- Appears like assignment, but it's not
- X is first unbound. When you set it first time, it is bound
- What is Z = X + 1?
  - If Z is not bound, Z is given value of 4.
  - If Z is bound, checks if Z is equal to 4.
- \{W, vapor\} = \{water, vapor\}.
- W (variable) is bound to water (atom) in this case

List Comprehension

- Pattern matching helps make code concise
- List comprehension takes that one step further
- Here is an example to double elements in an array
  - \([X \times 2 \mid X \leftarrow L]\)
  - Says double element X where X is a member of sequence L
  - Has generators and filters
  - Generator helps generate sequence, filter helps limit or select elements
Count Even Using List Comprehension

```erlang
%!usr/bin/env escript

main(_) ->
    io:format("p", [count_even([1, 2, 9, 2, 3, 4, 6, 8, 7])]).

count_even(L) ->
    OnlyEven = [X || X <- L, X rem 2 == 0],
    length(OnlyEven).
```

Pick Even-Odd Using List Comprehension

```erlang
%!usr/bin/env escript

main(_) ->
    List = [1, 2, 7, 9, 7, 9, 7, 9, 2, 3, 6],
    {Even, Odd} = get_even_odd(List),
    io:format("p\n", [Even]),
    io:format("p\n", [Odd]).

get_even_odd(L) ->
    get_even_odd_acc(L, [], []).

get_even_odd_acc([], Even, Odd) ->
    case (H rem 2) of
        1 -> get_even_odd_acc(T, Even, [H|Odd]);
        0 -> get_even_odd_acc(T, [H|Even], Odd)
    end;
get_even_odd_acc([], Even, Odd) ->
    {Even, Odd}.  
```

```erlang
[6,2,2]
[3,9,7,9,7,9,7,1]
```
Sort Using List Comprehension

```erlang
#!/usr/bin/env escript
main(_) ->
    io:format("p", [sort([2, 1, 4, 8, 2, 9, 1, 9, 12, 3])]).
sort([]) -> [];
sort([H|T]) ->
    sort([E || E <- T, E < H]) ++ [H] ++ sort([E || E <- T, E >= H])
        ++ sort([E || E <- T, E < H])
[[1,1,2,2,3,4,8,9,9,12]]
```

Count Primes Using List Comprehension

```erlang
#!/usr/bin/env escript
main(_) ->
    io:format("Number of primes ~p\n", [prime_count(114)])
prime_count(2) -> 1;
prime_count(N) ->
    is_prime(N) + prime_count(N-1)
is_prime(N) ->
    DivisibleList = [X || X <- lists:seq(2,N-1), N rem X =:= 0],
    if
        length(DivisibleList) > 0 -> 0;
        true -> 1
    end.
Number of primes 30
```