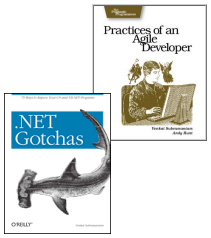
Functional Programming for Java Programmers

spkr.name = 'Venkat Subramaniam'
spkr.company = 'Agile Developer, Inc.'
spkr.credentials = %w{Programmer Trainer Author}
spkr.blog = 'agiledeveloper.com/blog'
spkr.email = 'venkats@agiledeveloper.com'



Abstract

- Most interest around Functional Programming (FP) has been academic until recently. Recent commercial languages are beginning to exploit FP features. Knowing more about FP will not only help us make better use of these features, but to exploit those. In this session we will take a close look at FP.

- We will look at What is FP, Strength and weakness of FP, FP languages for Java programmers, Examples that you can use today, Thinking in FP

Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

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 massing parallelism due to execution order independence

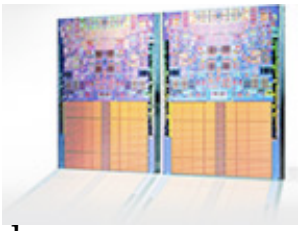
Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

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

Intel Announced Qurd-core (By popular Demand!)



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

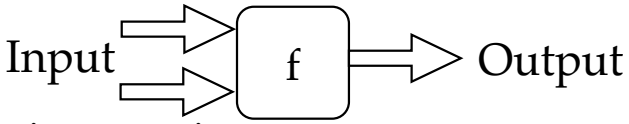
Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

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 ≡ 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 => 2 * 3 + 5 => 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

map.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    Result = double([2, 4, 9, 7, 1]),
5    io:format("~p", [Result]).
6
7  double(L) -> lists:map(fun(E) -> E * 2 end, L).
8
9  %We are sending an anonymous method (fun) to map method.
10 %map calls that method on each element in sequence.
11
```

```
[4,8,18,14,2]
```

Similar Concept in Groovy

map.groovy

```groovy
lst = [2, 4, 9, 7, 1]

println lst.collect { it * 2 }
```

```
[4, 8, 18, 14, 2]
```


Expressing Functions As Recursion

fact.erl

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

main(_) ->
  io:format("Factorial is ~p", [fact(5)]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

```
Factorial is 120
```

Expressing Functions As Recursion

fib.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    io:format("~p", [fib_seq(10)]).
5
6  fib(0) -> 0;
7  fib(1) -> 1;
8  fib(N) -> fib(N-1) + fib(N-2).
9
10 fib_seq(0) -> [fib(0)];
11 fib_seq(N) ->
12   fib_seq(N-1) ++ [fib(N)].
13 %Series where each element is the sum of previous two elements...
14
```

```
[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

max_two.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    io:format("Max is ~p\n", [max(1, 2)]),
5    io:format("Max is ~p\n", [max(2, 1)]).
6
7  max(A, B) when A > B -> A;
8  max(_, B) -> B.
9
```

```
Max is 2
Max is 2
```


Pattern Matching and Recursion at Play

max_many.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    io:format("Max is ~p\n", [max([1, 2, 5, 2, 6, 1, 8, 1, 5])]).
5
6  max([H|[]]) -> H;
7  max([H|T]) -> max2(H, max(T)).
8
9  max2(A, B) when A > B -> A;
10 max2(_, B) -> B.
11
```

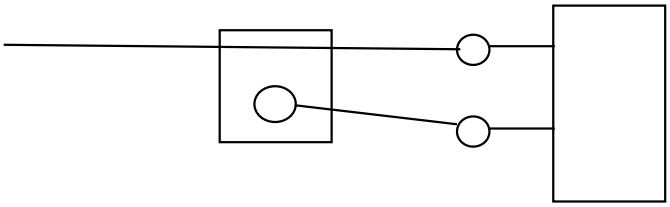
```
Max is 8
```

Curried Function

- Transforming a function that takes multiple arguments into a function that takes a single argument
- f(X, Y) → Z is transformed into

  curry(f): X → (Y → 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



curry.groovy

```
1  def doWork(val, closure)
2  {
3    curried = closure.curry("preset_value")
4
5    curried(val * 2)
6    curried(val * 10)
7  }
8
9  doWork(3) { p1, p2 -> println "Received ${p1} and ${p2}." }
10
```

```
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>

  | (λ<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. Lardin

Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

Types

- Atoms are numbers, booleans, strings, non-composites, indivisible
- Sequences are composites and dividable
  - if {x1, x2, ..., xn} $\in$ T, [x1, x2, ..., xn] $\in$ [T]
- An abstract types is specified in terms of abstract values without regard to any specific concrete implementation

Operations on Sequences

- first: [x1, x2, ..., xn] $\rightarrow$ x1
- rest: [x1, x2, ..., xn] $\rightarrow$[x2, ..., xn]
- prefix: x, [y1, y2, ..., yn] $\rightarrow$[x, y1, y2, ..., yn]
- ...

Operations on Sequences

count_even.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    io:format("~p", [count_even([1, 2, 9, 2, 3, 4, 6, 8, 7])]).
5
6  count_even([]) -> 0;
7  count_even([H|T]) ->
8    IsEven = if
9      (H rem 2) =:= 0 -> 1;
10     true -> 0
11   end,
12   IsEven + count_even(T).
13 % Wait till you see list comprehension later
14
```

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

Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

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 * 2 | | X <- 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

count_even_using_list_comprehension.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    io:format("~p", [count_even([1, 2, 9, 2, 3, 4, 6, 8, 7])]).
5
6  count_even(L) ->
7    OnlyEven = [X || X <- L, X rem 2 =:= 0],
8    length(OnlyEven).
9
```

```
5
```


Pick Even-Odd Using List Comprehension

pick_even_odd_using_list_comprehension.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    List = [1, 2, 7, 9, 7, 9, 7, 9, 2, 3, 6],
5    {Even, Odd} = get_even_odd(List),
6    io:format("~p\n", [Even]),
7    io:format("~p\n", [Odd]).
8
9  get_even_odd(L) ->
10  get_even_odd_acc(L, [], []).
11
12 get_even_odd_acc([H|T], Even, Odd) ->
13   case (H rem 2) of
14     1 -> get_even_odd_acc(T, Even, [H|Odd]);
15     0 -> get_even_odd_acc(T, [H|Even], Odd)
16   end;
17 get_even_odd_acc([], Even, Odd) ->
18  {Even, Odd}.
```

```
[6,2,2]
[3,9,7,9,7,9,7,1]
```

Sort Using List Comprehension

sort_using_list_comprehension.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    io:format("~p", [sort([2, 1, 4, 8, 2, 9, 1, 9, 12, 3])]).
5
6  sort([]) -> [];
7  sort([H|T]) ->
8    sort([E || E <- T, E < H]) ++ [H] ++ sort([E || E <- T, E >= H]).
9
```

```
[1,1,2,2,3,4,8,9,9,12]
```


Count Primes Using List Comprehension

prime_using_list_compurehension.erl

```
1  #!/usr/bin/env escript
2
3  main(_) ->
4    io:format("Number of primes ~p\n", [prime_count(114)]).
5
6  prime_count(2) -> 1;
7  prime_count(N) ->
8    is_prime(N) + prime_count(N-1).
9
10 is_prime(N) ->
11   DivisibleList = [X || X <- lists:seq(2,N-1), N rem X =:= 0],
12   if
13     length(DivisibleList) > 0 -> 0;
14     true -> 1
15   end.
16
```

```
Number of primes 30
```

Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

How can it help Concurrency?

- Mutable and Non-mutable State
- Memory area that can be modified is mutable state
- Functional programming languages prefer non-mutable states
- Shared memory is not modified
- No locking is needed for multiple processors to access these
- Promotes concurrency

Spawning in Erlang

```
#!/usr/bin/env escript

main(_) ->
  process([1, 3, 9, 8, 2, 3, 1, 8]).

do_work(Caller, X) ->
  io:format("working on ~p...~n", [X]),
  Caller ! {self(), X * 2}.

process_response([]) -> void;
process_response([Pid|T]) ->
  receive
    {Pid, Val} -> io:format("Received ~p~n", [Val]),
    process_response(T)
  end.

process(L) ->
  Caller = self(),
  Pids = lists:map(fun(E) -> spawn(fun() -> do_work(Caller, E) end) end,
  process_response(Pids).
```

```
working on 1...
working on 3...
working on 9...
working on 8...
working on 2...
working on 3...
working on 1...
working on 8...
Received 2
Received 6
Received 18
Received 16
Received 4
Received 6
Received 2
Received 16
```


Cost of spawn

process_timing.erl

```
1  #!/usr/bin/env escript
2
3  main([A]) ->
4    process(list_to_integer(A)).
5
6  wait() ->
7    receive
8      _ -> void
9    end.
10
11 process(N) ->
12   statistics(runtime),
13   statistics(wall_clock),
14   Pids = lists:map(fun(_) -> spawn(fun() -> wait() end) end, lists:seq(1, N)),
15   {_, Runtime} = statistics(runtime),
16   {_, WallClock} = statistics(wall_clock),
17   lists:map(fun(Pid) -> Pid ! done end, Pids),
18   io:format("microseconds: Runtime ~p WallClock ~p~n", [Runtime * 1000/N, WallClock * 1000/N]).
19
20 % Run this with some examples as follows
21 % process_timing 5000
22 % process_timing 10000
23 % process_timing 15000
24 % process_timing 20000
```

```
process_timing.erl 10000
microseconds: Runtime 49.0000 WallClock 60.1000
```

Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

FP on JVM

- Languages like Groovy and JRuby have some features that are borrowed from Functional Programming
- Haskell is a functional programming language
- Jaskell is a version of Haskell that interoperates with Java and runs on JVM
- JSR 233 provides engine to execute Jaskell on JVM

Usign Jaskell

```
jrunscript -classpath $JASKELLPATH -l jaskell
jaskell> add a b = a + b
jaskell> add 5 6
11
jaskell> lst = java.util.ArrayList.new[]
jaskell> lst
[]
jaskell> lst.size[]
0
jaskell> arr = int[].new 1
jaskell> lst.add arr
true
jaskell> lst.size[]
1
```


Agenda

- What's FP?
- Why FP?
- Features of FP
- Pattern Matching
- List Comprehension
- Concurrency
- Running on JVM
- Conclusion

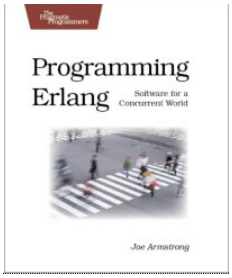
Quiz Time

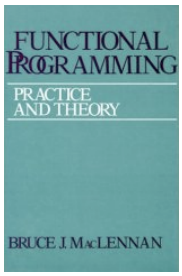
Conclusion

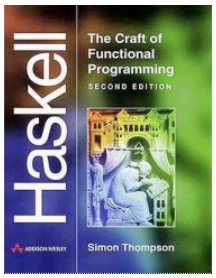
- Functional Programming has strong mathematical underpinning
- Provides higher level of abstraction
- Functions are treaded as first class citizens
- Higher order functions provide greater flexibility
- Subexpression independent evaluation orders and substitution paves way for greater flexibility for concurrency.

References

- http://www.erlang.org
- http://jaskell.codehaus.org
- http://www.haskell.org







Thank You!

You can download code examples from the link below

http://www.agiledeveloper.com — download