EDAF/N40: Functional Programming Other Programming Languages

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

May 24th, 2018

Some functional languages

- ERLANG
- ML (SML, OCaml)
- F#
- Curry (= Haskell + constraints)
- Lisp (Common Lisp)
- Scala
- Clojure
- ...

Idioms and design patterns

What are we looking for?

- Immutable objects
- Higher order functions
  - taking functions (pointers) as arguments
  - returning functions (pointers) as results
  - possibly with their environment
- Lazy evaluation
  - encapsulated in objects
  - infinite data structures
- (Monadic) encapsulation, e.g. parser combinators

Clojure

"Clojure is a dynamic programming language that targets the *Java Virtual Machine* (and *CLR* (Common Language Runtime = Microsoft .Net), and *JavaScript*).

It is designed to be a general-purpose language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming.

Clojure is a compiled language - it compiles directly to JVM bytecode, yet remains completely dynamic. Every feature supported by Clojure is supported at runtime."

`clojure.org`

Clojure, examples

```
user=> (map #(list %1 (inc %2)) [1 2 3] [1 2 3])
((1 2) (2 3) (3 4))

user=> (map (fn [x y] (list x (inc y))) [1 2 3] [1 2 3])
((1 2) (2 3) (3 4))

user=> (map #(list % (inc %)) [1 2 3])
((1 2) (2 3) (3 4))

user=> (map (fn [x] (list x (inc x))) [1 2 3])
((1 2) (2 3) (3 4))
```


Clojure, examples

```
user=> (def nums (iterate inc 0))
#'user/nums
user=> (take 5 nums)
(0 1 2 3 4)

user=> (def s (for [x nums :when (zero? (rem x 4))] (inc x)))
#'user/s
user=> (take 5 s)
(1 5 9 13 17)
```


Clojure, examples

```
user=> (def nums (iterate inc 0))
#'user/nums
user=> (def s (map inc (filter (fn [x] (zero? (rem x 4))) nums))
#'user/s
user=> (take 5 s)
(1 5 9 13 17)
user=>
```


XSLT

XSLT = XSL Transformations

- XSL - The Extensible Stylesheet Language
- XML - standard notation to represent annotated trees - Extensible Markup Language
- XSLT - tree transformation language, XML $\rightarrow$ XML

XSLT Syntax

```
<xsl:stylesheet>
   <xsl:template match="/">
      <H1><xsl:value-of select="title"/></H1>
      <H2><xsl:value-of select="author"/></H2>
   </xsl:template>
</xsl:stylesheet>
```

Input:

```
<xslTutorial>
   <title>XSL</title>
   <author>John Smith</author>
</xslTutorial>
```

Output:

```
<H1>XSL</H1>
<H2>John Smith</H2>
```


XSLT Templates

```
<xsl:stylesheet>
   <xsl:template match="bold">
      <P><B><xsl:value-of select="."/></B></P>
   </xsl:template>
   <xsl:template match="red">
      <P style="color:red"><xsl:value-of select="."/></P>
   </xsl:template>
   <xsl:template match="italic">
      <P><i><xsl:value-of select="."/></i></P>
   </xsl:template>
</xsl:stylesheet>
```

Input:

```
<xslTutorial>
   <bold>Hello, world.</bold>
   <red>I am </red>
   <italic>fine.</italic>
</xslTutorial>
```

Output:

```
<P><B>Hello, world.</B></P>
<P style="color:red">I am </P>
<P><i>fine.</i></P>
```


XSLT with control flow

Imperative style also possible:

- if
- for-each
- call-template
- . . .

XSLT

Two distinct styles/approaches:

- Shape of result tree determined by the program
  - XSLT program is imperative, with traditional control flow constructs
- Shape of result tree determined by shape of input tree
  - XSLT is declarative, using fp-style pattern matching

Python

An interpretive, interactive object-oriented language. Combines power with clear syntax.

> *"Python has been an important part of Google since the beginning, and remains so as the system grows and evolves." (Peter Norvig, director at Google Inc.)*
>
> *"Python plays a key role in our production pipeline. Without it a project the size of Star Wars: Episode II would have been very difficult to pull off. From crowd rendering to batch processing to compositing, Python binds all things together" (Tommy Burnette, Senior Technical Director, Industrial Light and Magic)*

Python - First class functions

```
>>> def increment(n):
        return n+1

>>> def decrement(n):
        return n-1

>>> def incOrDec(p):
        if p: return increment
        else: return decrement

>>> x=3

>>> incOrDec(x>5)(x)
2
```


Python - Lambda forms

```
>>> def make_incrementor(n):
        return lambda x: x + n

>>> f = make_incrementor(42)

>>> f(0)
42

>>> f(1)
43
```


Python - Standard list functions

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]

>>> def cube(x): return x*x*x
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]

>>> def add(x,y): return x+y
>>> reduce(add, range(1, 11))
55
```

Python - List comprehensions

```
>>> vec = [2, 4, 6]

>>> [3*x for x in vec]
[6, 12, 18]

>>> [3*x for x in vec if x > 3]
[12, 18]

>>> [3*x for x in vec if x < 2]
[]

>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```


Lazy Python - Generators

```
>>> def fib():
        a, b = 0, 1
        while 1:
            yield b
            a, b = b, a+b
>>> x = fib()
>>> for i in range(7):
        print x.next()
1
1
2
3
5
8
```


Scala

A modern, object-oriented, functional, imperative language :-)
Scala was developed starting in 2003 by Martin Odersky's group at EPFL, Lausanne, Switzerland.
Scala is designed to integrate easily with applications that run on modern virtual machines, primarily the Java virtual machine (JVM). The main Scala compiler, scalac, generates Java class files that can be run on the JVM. However, another Scala compiler exists that generates binaries that can be run on the .NET CLR, as Scala is designed to integrate with both the Java and .NET worlds.

Scala, example

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

Scala, example

```
class Reference[T] {
  private var contents: T = _

  def set(value: T) { contents = value }
  def get: T = contents
}
```


Scala, example

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " +
                                        (cell.get * 2))
  }
}
```


Scala, example

```
scala> List(1, 2, 3) map (_ + 1)
res28: List[Int] = List(2, 3, 4)
scala> val words = List("the", "quick", "brown", "fox")
words: List[java.lang.String] = List(the, quick, brown, fox)
scala> words map (_.length)
res29: List[Int] = List(3, 5, 5, 3)
scala> words map (_.toList.reverse.mkString(""))
res30: List[String] = List(eht, kciuq, nworb, xof)
scala> words map (_.toList)
res31: List[List[Char]] = List(List(t, h, e), List(q, u, i, c,
 k), List(b, r, o, w, n), List(f, o, x))
scala> words flatMap (_.toList)
res32: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w, n,
 f, o, x)
```


F#

F# is a multi-paradigm programming language, targeting the .NET Framework, that encompasses functional programming as well as imperative and object-oriented programming disciplines. It is a variant of ML and is largely compatible with the OCaml implementation. F# was initially developed by Don Syme at Microsoft Research but is now being developed at Microsoft Developer Division and is being distributed as a fully supported language in the .NET Framework and Visual Studio as part of Visual Studio 2010.

F#, examples

```
let rec fib n =
    match n with
    | 0 | 1 -> n
    | _ -> fib (n - 1) + fib (n - 2)

(* An alternative approach - a lazy recursive sequence
   of Fibonacci numbers *)
let rec fibs = seq {
    yield! [1; 1];
    for (x, y) in Seq.zip fibs (Seq.skip 1 fibs) -> x + y }

(* Another approach - a lazy infinite sequence *)
let fibSeq = Seq.unfold (fun (a,b) -> Some(a+b, (b, a+b)))
                                                    (1,1)
```


F#, examples

```
(* Print even fibs *)
[1 .. 10]
|> List.map     fib
|> List.filter  (fun n -> (n % 2) = 0)
|> printlist

(* Same thing, using sequence expressions *)
[ for i in 1..10 do
    let r = fib i
    if r % 2 = 0 then yield r ]
|> printlist
```


Accumulator generator

http://www.paulgraham.com/accgen.html
The problem:

- Write a function foo that takes a number n and returns a function that takes a number i, and returns n incremented by i.
- Note (a): that's number, not integer,
- Note (b): that's incremented by, not plus.

C++

```
template<typename T>
struct Acc {
  Acc(T n)
  : n(n) {}

  template<typename U>
  Acc(const Acc<U>& u)
  : n(u.n) {}

  template<typename U>
  T operator()(U i) {
     return n += i;
  }
  T n;
};

template<typename T>
Acc<T> foo(T n)
{
   return Acc<T>(n);
}
```

Dylan

```
define function foo (n)
  method (i) n := n + i end;
end function;
```


Erlang

```
foop(N)->
  receive
    {P,I}-> S =N+I, P!S, foop(S)
  end.

foo(N)->
  P=spawn(foo,foop,[N]),
  fun(I)->
    P!{self(),I},
    receive V->V end
  end.
```


Haskell

```
import IOExts
foo n = do
  r <- newIORef n
  return (\i -> do
    modifyIORef r (+i)
    readIORef r)
```


Javascript

```
function foo (n) {
  return function (i) {
           return n += i } }
```

Lisp: CL

```
(defun foo (n)
   (lambda (i) (incf n i)))
```


Lisp: Arc

```
(def foo (n) [++ n _])
```


Lisp: Scheme

```
(define (foo n)
  (lambda (i)
    (set! n (+ n i))
    n))
```


Lua

```
function foo(n)
  return function (i)
           n = n + i
           return n
         end
end
```

Maple

```
foo := proc(n)
    local s;
    s := n;
    proc(i) s := s + i
        end
    end
```


Perl 5

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```


Python

```
class foo:
   def __init__(self, n):
      self.n = n
   def __call__(self, i):
      self.n += i
      return self.n
```


Ruby

```
def foo (n)
   lambda {|i| n += i } end
```

Smalltalk

```
foo: n
  |s|
  s := n.
  ^[:i| s := s + i. ]
```


VBScript

```
Class acc
  Private n
  Public Default Function inc(i)
    n = n + i
    inc = n
  End Function
End Class

Function foo(n)
  Dim bar
  Set bar = New acc
  bar(n)
  Set foo = bar
End Function
```