Functional Programming and Elixir
CH 11 From Scott
Ch 1-5 “Learn Functional Programming with Elixir”
https://elixir-lang.org/getting-started/introduction.html and subsections from there 1-10

Issues like binding times, names and scope still apply.

Why Functional Programming
“Imperative languages have shared mutating values”

1. A change in one place can effect others
2. Concurrency
    - Java (and lots of other langs) has a data types to address this issue
        - Atomic[Integer,Long, Float…]
            - simple test of AtomicLong — ~20x slower
            - see atomic_java/Atomic.java
        - Also locks, thread safe classes (for instance Vector is not, ArrayList is)

Scott “One can write in a function style in many imperative languages, and many functional languages include imperative features ….”

Common features of functional languages

- Per Scott: “heavy use of polymorphism “

Remember “polymorphism”

Ad hoc polymorphism, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.
There are two subkinds of ad hoc polymorphism.

Overloading refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at compile time based on the declared types of the entities. They exhibit early binding.
Java overloads a few operator symbols, such as using the + symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

Subtyping (also known as subtype polymorphism, inclusion polymorphism, or polymorphism by inheritance) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits late binding.
The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply polymorphism.

Parametric polymorphism, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

The object oriented programming community often calls this type of polymorphism generics or generic programming. The functional programming community often calls this simply polymorphism.

FROM https://john.cs.olemiss.edu/~hcc/csci450/2016fall/notes/Fundamentals/Polymorphism.html

IT IS PARAMETRIC POLYMORPHISM THAT FP USES.

Claim:

Polymorphism offers the following advantages –

It helps the programmer to reuse the codes, i.e., classes once written, tested and implemented can be reused as required. Saves a lot of time.

Single variable can be used to store multiple data types.

Easy to debug the codes.

https://www.tutorialspoint.com/functional_programming/functional_programming_polymorphism.htm

NOTE THIS IS CONTRADICTORY TO CLAIMS MADE IN STRONGLY TYPED LANGUAGES

Back to Scott and features of functional programming languages

lists (rather than arrays)
RECURSION
Lots of temporary variables so garbage collection
first class functions
“structured function returns”

immutable values
functions are ONLY dependent on their arguments
functions ONLY effect is its return value

function programming and top down thinking

“top down programming” is what you have been taught.
start with statement of problem, design classes, design function interfaces, write …
Linked to a method of software development “waterfall”
functional programming is more naturally “bottom up”.
start by writing a program to do one little piece of task. Make sure it works.
write another function, that may use the result of first function, to do another piece
The final program ends up being a fairly simple assembly of the pieces.
You know it will work, because all of the pieces are easily and independently testable because each function depends only on its parameters
In functional programming you ALWAYS have something that works.
May not do everything, but it does things correctly

Why Elixir? — almost always shows in the top 3 FPLs

Elixir
dynamically typed (late binding)
static scope
can be either interactive or batch (much like python)
interactive: start: iex
end: ctrl-C ctrl-C (yeas twice)
I do not do this much, but the book does.
batch: UNIX> elixir abc.ex
will find / use other .ex files in current directory
Compiled: UNIX> elixirc abc.ex
results in *.beam files , one for every “module” in compiled file
beam files are found and auto used (equivalent to java .class files) when you run either elixir or iex
I rarely do this

Elixir types:
integer, float, boolean, “atom”, string, anonymous function, list, tuple
Atom is a constant whose value is its own name.
Atoms start with “:”
Strings— UTF-8
so byte_size(“string”) may not equal String.length(“String”)
Anon Func:
Go has these also
Write a function and bind it to a variable — lots of uses, but difficult/impossible to do recursion

```
iex(12)> a = fn -> 7 end    # anonymous function — no args
#Function<43.3316493/0 in :erl_eval.expr/6>
iex(13)> a.()      # call to anonymous function   Note the “.” it is required
7
iex(14)> b = fn (x) -> 7*x end     # anonymous function  — one arg
#Function<42.3316493/1 in :erl_eval.expr/6>
iex(15)> b.(7)
49
iex(16)> c = fn (x,y) -> z=x+2; z*y end     # anon fun —  2 args
#Function<41.3316493/2 in :erl_eval.expr/6>
iex(17)> c.(3,4)
20
iex(18)> d = fn (x,y,z) -> zz = z+3    # anon fun — 3 args  — on multiple lines!
...(18)> x*y*zz
...(18)> end
#Function<40.3316493/3 in :erl_eval.expr/6>
iex(19)> d(2,3,4)
** (CompileError) iex:19: undefined function d/3 (there is no such import)    # no “.”

iex(19)> d.(2,3,4)
42
```

See also **anon_ex/anon.ex**

Most of this example but in a file

Lists
stored as linked list.
so access is linear in length of list.
[“a”, 1,2,3]
[104, 101, 108, 108, 111]
‘hello’
A character list — which is NOT a string

Map

```
iex(1)> aa = %{a: 3, b: 4}
%{a: 3, b: 4}
iex(2)> bb = %{"a"=>"c", 12=>42}
%{12 => 42, "a" => "c"}
iex(3)> bb[12]
42
iex(4)> aa[:a]
3
```

Tuples
{1, “hello”}
access to members if fast, change is slow.
Most common usage is to get multiple returns from functions
tuples in Exlir are real things unlike “tuple assignment” in Go

To get the type of a variable

```
iex> i(bb)
```

ALL TYPES ARE IMMUTABLE

for instance
++ is list concatenation operator

```
iex> aa=[1,2,3]
[1,2,3]
iex> aa++[4]
[1,2,3,4]
iex> aa
[1,2,3]
iex>bb=aa++[4]   # when append, the thing appended should be a list
[1,2,3,4]
iex>aa==bb
false
```

different from Java / Go!!!
In Go vars are always mutable. However strings are immutable.
Question how are strings actually stored in Go?
For instance [3]string. The point is that this should take up a fixed amount of space. But strings have variable length!!!
Answer: indirection. Actually store a struct containing pointer and length
similar to indirection for Slices just harder to see because of pass-by-value and immutability
see immut_go/imm.go

Strings also immutable in Java. Most Number are immutable. Anything declared “final” is immutable. In edge cases with generics and covariance you can have immutable items. We might get there.

Even integers “variables” are immutable in Elixir!!
How to prove this statement!!
pass a var into a function??
could just be pass by value rather than pass by reference/ sharing
Really need closures to prove!!
See immut_ex/imm.ex and imm.go

Tuple

iex> tuple = {:ok, "hello"}

```
{:ok, "hello"}
iex> elem(tuple, 1)
"hello"
```

Most common use of tuples is multiple return from a function

OPERATIONS & OPERATORS

+,-,*,/, div, rem
elixir does do type coercion
++, - - List operators, append two lists, Remove items in right list from left list

```
aa=[1,2,3,4,3,2,1]
[1, 2, 3, 4, 3, 2, 1]
iex(2)> aa--[2]
[1, 3, 4, 3, 2, 1]
iex(3)> aa--[2,1]
[3, 4, 3, 2, 1]
iex(4)> aa--[2,2,1]
[3, 4, 3, 1]
```

<> string concatenation
and, or, not boolean operators, both things must be boolean
&&, ||, ! Boolean operators anything other than false and nil are true
==, ===, !=, !==, >, <, >=, <=

```
iex(5)> 1==1.0
true
iex(6)> 1===1.0
false
```

same value and same type

|> “Pipeline”

Idea, that the output of one function and make it the first argument to the next function. NOT a necessary part of the language but it can make the code a LOT easier to read. Without it, you sort of have to start reading lines on the right and work your way left.

Much the same a UNIX pipes
see **upcase_ex/upcase.ex**

Modules and named functions

named functions can only exist within a module

```
defmodule Agt do
```

```
def ggt (val) do
 42+val
end
end

IO.puts Agt.ggt(1)
```

Closures

as with Go, closures in elixir hold the items in scope at the time the func was defined.
Unlike Go, because vars are immutable, change to variable after function is defined has no effect
see closu_ex/clos.ex
see closure_go/closure.go

PATTERN MATCHING

used in most “modern” FP languages, certainly in elixir
Matching is everywhere is Elixir
= is NOT assignment, it is match

iex(1)> a=2 # with var on left, the var can take any value; therefore it matches 2 and assignment happens
2
iex(2)> a=3
3
iex(3)> 2=a #with var on right, constant on left, they must match or it a MatchError
** (MatchError) no match of right hand side value: 3

iex(3)> 3=a #Since a had a value of 3, these match
3

Deeper: that in Go 2=a will not compile as 2 cannot be converted into an l-value
In Elixir, the problem is that = is the match operator, not the assignment operator
So “2=a” fails because the value stored in a cannot be matched to 3
OTOH 3=a is fine

pattern matching applies in lots of places

Strings

iex(18)> "a"<>b="an inconvient truth"
"an inconvient truth"
iex(19)> b
"n inconvient truth"

Note here that the variable b gets the assignment of the part of the string (on right) contained in a that does not match to “a”

Tuples

iex(7)> a={1,2,:OK}
{1, 2, :OK}
iex(8)> {b,c,d}=a
{1, 2, :OK}
iex(9)> b
1
iex(10)> {e,f,:ok} =a

** (MatchError) no match of right hand side value: {1, 2, :OK}
Tons of other ways to get match errors

```
iex(10)> {e,f,:OK} =a
{1, 2, :OK}
iex(11)> f
2
iex(12)> {_,_,:OK}=a
{1, 2, :OK}
```

8: binds the values of b,c,d to the parts of the tuple
9: shows that b has a value of 1
10: given that the third element in the tuple is the atom :ok, bind e, f to the first 2
this fails because atoms are case sensitive
11: As with 10, but success
12: When you do not case about binding vars just an _ or _ preceding a name will match anything

LISTS

Can match everything just like tuples, but more commonly head and rest

```
a=[1,2,3,4,5,6,7,8,9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
iex(14)> [b|c] = a
[1, 2, 3, 4, 5, 6, 7, 8, 9]
iex(15)> b
1
iex(16)> c
[2, 3, 4, 5, 6, 7, 8, 9]
```

Note that the | operator can also be used to add to lists

```
iex(17)> d=[10|a]
[10, 1, 2, 3, 4, 5, 6, 7, 8, 9]
iex(18)> d
[10, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Again, important to keep in mind that = may look like assignment and often behave that way, but it is NOT.

MATCHING in functions (chapter 4)

First, order matters in elixir. See **funmatch_ex/funmatch.ex**
function arguments can be pattern matching expressions!!!!!!
see **recur_go/recur.go**. Very simple linked list and recursive printer in Go
Now **recur_ex/recur.ex**

prnt_lst is exactly equivalent to recur.go
he(x) gives first element of list, tl(x) rest of list
match_prnt_lst same thing using matching!!!

matching and guard clauses

recur_go/recur.go toLimByMFromS
problem matching does not do well here because the stopping conditions could be met in an unbounded number of ways
so “guard clauses”
guards must be very simple (e.g., numeric comparisons) and without side effects

TAIL Recursion

see **tailrec/tailrec.ex**

elixir does have tail call optimization

problem compute the sum of the number 1..n (brute force) using recursion

A non-tr implementation

100,000,000 takes about 11.7 sec

Max ~145,000,000

TR implementation

100,000,000 takes 0.63 sec (18x faster)

No detectable max

STRUCTS

much the same reason and logic as structs in go.

much the same syntax as maps

implementing protocols in particular String.Chars — defimpl

see struct_ex/struct.ex

Note that this also implements an interface and to get structs into strings you need to do this

Higher Order Functions (ch 5 of elixir book)

Idea have a function that takes a function as one of its arguments. Therefore can do something fairly general.

Often can use pre-packaged higher-order functions instead of writing a recursion

Done some of this before … kind of

consider adding the elements in a list of numbers

see **higher_ex/higher.ex rsum_tc**

now consider passing in a function to do that addition. A little harder for the simple addition task, see **higher_ex/higher.ex rsum_tc_f**

BUT consider sum of squares, sum of cubes, …

Note that the passed in function needs to take two parameters!!

What if I just want to transform items in a list

see t_map_tc_f

Note that for both of these, the efficient tail call version reverses the order

Or filtering a list for particular items

see

filter_tc_f

Many of the higher order functions are defined in Enum module.

Enum can be applied to anything that implements the Enumerable protocol

Enum.reduce

takes a list, a starting point and a function and returns a single thing. the sum of the elements, a string of the elements

```
aa=[1,2,3,4,5,6,7,8,9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
iex(3)> Enum.reduce(aa, 0, fn (itm, acc) -> itm+acc end)
45
iex(5)> Enum.reduce(~w[a s d f g h], "", fn (itm, acc) -> itm<>acc end)
"hgfdsa"
```

Enum.map

list and a function, returns a list of each of the original list with the function applied to it

```
iex(6)> Enum.map(aa, fn (itm) -> itm*itm end)
[1, 4, 9, 16, 25, 36, 49, 64, 81]
iex(7)> Enum.map(aa, &(&1*&1))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Practical matters — file reading and command line parameters

see **commread_ex/commread.ex**

that this program also shows use of case command in a couple of error handling situations