Programming Languages & Paradigms
PROP HT 2011

Lecture 13

Functional Programming II – Introcuction to Clojure

Beatrice Åkerblom
beatrice@dsv.su.se

Reminder

- Strong typing vs. static typing

x=x+1

Programming Without State

- Imperative style:

```
int n = 5;
int a = 1;
while ( n>0 ) {
   a = a*n;
   n = n-1;
}
```

- Declarative (functional) style:

```
fac n =
   if n==0 then 1
   else n * fac (n-1)
```

- Programs in pure functional languages have no explicit state. Programs are constructed entirely by composing expressions.

Pure Functional Languages

- Imperative Programming:
  - Program = Algorithms + Data
- Functional Programming:
  - Program = Functions Functions

- What is a Program?
  - A program (computation) is a transformation from input data to output data.

Key features of pure functional languages

- All programs and procedures are functions
- There are no variables or assignments — only input parameters
- There are no loops — only recursive functions
- The value of a function depends only on the values of its parameters
- Functions are first-class values

Merits of Functional Programming

- By avoiding variables and assignments, we gain:
  - Clearer semantics, since programs correspond closer to abstract mathematical objects
  - More freedom in implementation, e.g. for parallelisation
- By using more flexible function, we gain:
  - Conciseness and elegance
  - Better parameterisation and modularity of programs
  - Convenient ways of representing infinite data

Functional Programming Imperative Language

```
int sum (int i, int j)
{ int k, temp;
  temp = 0;
  for (k = i; k <= j; k++)
    temp += k;
  return temp;
}

int sum (int i, int j)
{ if (i > j) return 0;
  else return i + sum(i+1, j);
}
```

Tail Recursion

```
int sum1(int i, int j, int sumSoFar)
{
  if (i > j) return sumSoFar;
  else
    return sum1(i+1, j, sumSoFar+i);
}

int sum(int i, int j)
{
   return sum1(i, j, 0);
}
```


Clojure

- Java + Lisp

Lisp, with Fewer Parentheses

- Clojure generalizes Lisp's physical list into an abstraction called a **sequence**. This preserves the power of lists, while extending that power to a variety of other data structures.
- Clojure's reliance on the JVM provides a **standard library** and a deployment platform with great reach.
- Clojure provides a convenient **literal syntax** for a wide variety of data structures besides just lists. These features make Clojure code less "listy" than most Lisps.
- In Clojure, unlike most Lisps, **commas are whitespace**. Adding commas can make some data structures more readable.

Homoiconic Languages

- Clojure is homoiconic, that is, Clojure code is composed of Clojure data.
- When you run a Clojure program, a part of Clojure called the reader reads the text of the program in chunks called forms and translates them into Clojure data structures.
- Clojure then compiles and executes the data structures.
- Examples: Lisp (Clojure), Prolog, PostScript, Io, R

Forms

```
Form       Example(s)
---------------------
Boolean    true, false
Character  \a, \newline
Keyword    :tag, :doc
List       (1 2 3), (println "foo"), (\a \b \c)
Map        {:name "Bill", :age 42}, {:x 1 :y 2}
Nil        nil
Number     1, 4.2, 27/2
Set        #{:snap :crackle :pop}
String     "hello"
Symbol     user/foo, java.lang.String
Vector     [1 2 3]
```


Truth

- Every value looks like true to if, except for false and nil

```
(if true :truthy :falsey)  ;=> :truthy

(if [] :truthy :falsey)    ;=> :truthy

(if nil :truthy :falsey)   ;=> :falsey

(if false :truthy :falsey) ;=> :falsey
```


Scalars

```
user=> 42
42
user=> +9
9
user=> -107
-107
user=> 127 ;decimal
127
user=> 0x7F ;hexadecimal
127
user=> 0177 ;octal
127
user=> 32r3V ;radix-32
127
user=> 2r01111111 ;binary
127
user=> 1.17
1.17
user=> +1.22
1.22
user=> -2.
-2.0
```


Scalars, cont'd

```
user=> 366e7
3.66E9
user=> 10.7e-3
0.0107
user=> 22/7
22/7
user=> 7/22
7/22
user=> 10287983002976367676874090288 72/88829897008789478784
12859978753720459596092612860 9/11103737126098684848
user=> -103/4
-103/4
user=> user/foo
CompilerException java.lang.RuntimeException: No such var: user/foo,
compiling:(NO_SOURCE_PATH:0)
user=> user/helloo
#<user$helloo user$helloo@3c2c7ac5>
user=> helloo
#<user$helloo user$helloo@3c2c7ac5>
```

Scalars, cont'd

```
user=> :chumby
:chumby
user=> :2
:2
user=> :?
:?
user=> :aKeyword
:aKeyword
user=> "This is a string"
"This is a string"
user=> "Another
string"
"Another\nstring"
user=> \a          ; The character lowercase a
\a
user=> \A
\A
user=> \u0042
\B
user=> \\          ; The backslash character
\\
user=> \u30DE      ; The unicode katakana character
\?
```


Strings

- Clojure Strings are Java Strings
- Strings can be created using the `str` function

```
user=> (str 1)
"1"

user=> (str 1 2 nil 3)
"123"

user=> (doc str)
-------------------------
clojure.core/str
([] [x] [x & ys])
  With no args, returns the empty string. With one arg x, returns
  x.toString().  (str nil) returns the empty string. With more than
  one arg, returns the concatenation of the str values of the args.
nil
```


Strings

- Clojure does not wrap most of Java's string functions like with `str`. Instead, you can call them directly using Clojure's Java interop forms

```
user=> (.toUpperCase "hello")
"HELLO"
```

- The dot before toUpperCase tells Clojure to treat it as the name of a Java method instead of a Clojure function

Symbols

- Symbols name all sorts of things in Clojure:
  - Functions like `str` and concat
  - "Operators" like + and –, which are, after all, just functions
  - Java classes like `java.lang.String` and `java.util.Random`
  - Namespaces like `clojure.core` and Java packages like `java.lang`
  - Data structures and references

```
user=> cons
#<core$cons clojure.core$cons@3c32fb80>
user=> str
#<core$str clojure.core$str@5e3d5149>
user=> +
#<core$_PLUS_ clojure.core$_PLUS_@2b8f73cb>
user=> java.lang.String
java.lang.String
user=> 1
1
user=> [2 3 4]
[2 3 4]
```

But...

```
user=> (2 3 4)
ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn
user/eval320 (NO_SOURCE_FILE:406)
```

- That's Clojure telling us that an integer (the number 2 here) can't be used as a function...

Prevent Evaluation

```
user=> (quote (2 3 4 5))
(2 3 4 5)

user=> '(3 4 5 6)
(3 4 5 6)

user=> (quote (pos? 3))
(pos? 3)
```

- Remember that ' affects all of its argument, not only top level

```
user=> [1 (+ 2 3)]
[1 5]

user=> '(1 (+ 2 3))
(1 (+ 2 3))
```


Basics About Collections

```
user=> ()  ;empty list, not nil
()
user=> (1 2 3 4)
ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn
user/eval120 (NO_SOURCE_FILE:110)
user=> (+ 1 2 3)
6
user=> '(1 2 3 4)
(1 2 3 4)
user=> '("hello" 1 2.3 \b)
("hello" 1 2.3 \b)
user=> [1 2 :a :b :c]
[1 2 :a :b :c]
user=> []
[]      ;empty vector, not nil
user=> {1 "one", 2 "two", 3 "three"}
{1 "one", 2 "two", 3 "three"}
user=> #{1 2 "three" :four 0x5 1}
IllegalArgumentException Duplicate key: 1
clojure.lang.PersistentHashSet.createWithCheck (PersistentHashSet.java:
68)
user=> #{1 2 "three" :four 0x5}
#{1 2 5 :four "three"}
```


Functions

- Function call:

```
(+ 1 2 3)
;=> 6
```

- Function definition using the special form `def`, which can be used to assign a symbolic name to a piece of Clojure data

```
user=> (def mk-set
         (fn
           ([x y] #{x y})))
#'user/mk-set

user=> (mk-set 2 3)
#{2 3}
```

Functions, cont'd

- What we really did was capturing and naming a function:

```
user=> (fn mk-set [x y] #{x y})
#<user$eval150$mk_set__151 user$eval150$mk_set__151@1b72290f>

user=> ((fn [x y] #{x y}) 1 2)
#{1 2}
```

- There are still even better ways:

```
user=> (defn mk-set
   ([x y] #{x y}))
#'user/mk-set

user=> (mk-set 3 4)
#{3 4}
```


Functions, cont'd

- And even better, with documentation and different numbers of arguments:

```
user=> (defn make-a-set
  "Takes a number of values and makes a set from them"
  ([x]       #{x})
  ([x y]     #{x y})
  ([x y & z] (set (conj z x y)))
)
#'user/make-a-set

user=> (make-a-set 2 3 4 5)
#{2 3 4 5}

user=> (doc make-a-set)
-------------------------
user/make-a-set
([x] [x y] [x y & z])
  Takes a number of values and makes a set from them
nil
```


Vars

- A Var is named by a symbol and holds a single value.
- Using `def` creates a "root binding"—a binding that's the same across all threads, unless rebound in a specific thread:

```
user=> (def x 42)
#'user/x
user=> x
42

user=> (def y)
#'user/y
user=> y
#<Unbound Unbound: #'user/y>

user=> (def mk-set
  (fn
    ([x y] #{x y})))
#'user/mk-set
user=> (mk-set 2 3)
#{2 3}

user=> (def make-a-set-2 make-a-set)
#'user/make-a-set-2
user=> (make-a-set-2 2 3 4 5 )
#{2 3 4 5}
```


Functions are First Class Objects

- Functions are first-class objects that can be:
  - dynamically created at any time during runtime
  - used in the same way as any value
  - stored in Vars, held in lists and other collection types
  - passed as arguments to and returned as the result of other functions

```
user=> (def my-funcs [make-a-set make-a-set-2 print-down-from])
#'user/my-funcs

user=>  (nth my-funcs 0)
#<user$make_a_set user$make_a_set@44755866>

user=> ((nth my-funcs 0) 2 3 4 5)
#{2 3 4 5}
```

Conditionals

- Clojure doesn't have local variables, it does have locals but they can't vary:

```
user=> (if (= 1 1)
   "Math works")
"Math works"

user=> (if (= 1 3)
   "Math broken"
   "Math works")
"Math works"

user=> (defn weather-judge
       [temp]
       (cond
           (< temp 20) "It's cold"
           (> temp 25) "It's hot"
           :else "It's comfortable"))

#'user/weather-judge
user=> (weather-judge 12)
"It's cold"
user=> (weather-judge 22)
"It's comfortable"
```


Locals

- Clojure doesn't have local variables, it does have locals but they can't vary:

```
user=> (let [a 2 b 3] (+ a b))
5

user=>  (let
        [r 5 pi 3.14 r-sq (* r r)]
        (println "radius is" r)
        (* pi r-sq))
radius is 5
78.5
```


Locals, cont'd

- Locals can be useful for making code more readable, e.g the function

```
user=> (defn seconds-to-weeks
       "Converts seconds to weeks"
       [seconds]
       (/(/(/(/ seconds 60) 60) 24) 7))
#'user/seconds-to-weeks
```

- Can be rewritten in a more

```
user=> (defn seconds-to-weeks
       "Converts seconds to weeks"
       [seconds]
       (let [minutes (/ seconds 60)
             hours (/ minutes 60)
             days (/ hours 24)
             weeks (/ days 7)]
       weeks))
#'user/seconds-to-weeks
```


Loops

- Clojure encourages the use of immutable data, and to support this looping is done by recursion (like in other functional languages)
  - Function arguments are used to store and modify computational progress

```
user=> (defn power
   "Calculates a number to the power of the provided exponent"
   [number exponent]
   (if (zero? exponent)
     1
          (* number (power number (- exponent 1)))))
#'user/power

user=> (power 2 2)
4
```

```
user=> (defn avg
  "Returns the average of two arguments"
  [a b]
  (/ (+ a b) 2))
#'user/avg

user=> (defn good-enough?
  "Tests if a guess is close enough to the real square root"
  [number guess]
  (let [diff (- (* guess guess) number)]
    (if (< (abs diff) 0.001)
      true
      false)))
#'user/good-enough?

user=> (defn sqrt
  "Returns the square root of the supplied number"
  ([number] (sqrt number 1.0))
  ([number guess]
     (if (good-enough? number guess)
       guess
       (sqrt number (avg guess (/ number guess))))))
#'user/sqrt

user=> (sqrt 25)
5.000023178253949
user=> (sqrt 10000)
100.00000025490743
```



```
user=> (java.lang.Math/sqrt 25)
5.0
```


Limits for Recursion

- There is a limit on the number of possible nested functions that is machine-specific
  - Imperative languages solves this through not using recursion for everything
  - Functional languages solves this through tail-call-optimisation
  - Recursion from a tail position is in many ways like a structured goto, and has more in common with an imperative loop than it does with other kinds of recursion
  - Clojure does not automatically use tail-call-optimisation, but we can explicitly ask for it using `recur`

Functional

```
user=> (defn add-up
         "adds all the numbers below a given limit"
         ([limit] (add-up limit 0 0))
         ([limit current sum]
           (if (< limit current)
              sum
              (add-up limit (+ 1 current) (+ current sum)))))
#'user/add-up

user=> (add-up 3)
6

user=> (add-up 10000)
StackOverflowError   java.lang.Number.<init> (Number.java:32)
```



```
user=> (defn add-up
         "adds all the numbers below a given limit"
         ([limit] (add-up limit 0 0))
         ([limit current sum]
            (if (< limit current)
                sum
                (recur limit (+ 1 current) (+ current sum)))))
#'user/add-up

user=> (add-up 3)
6

user=> (add-up 10000)
50005000

(fn [x] (recur x) (println x))
; java.lang.UnsupportedOperationException:
;    Can only recur from tail position
```


Side Effects

- Clojure avoids side effects, but some tasks are, by nature, side effects, e.g.:
  - IO
  - Explicit state management
  - Java interaction
- Using `do`, all the expressions will be evaluated, but only the last one will be returned:

```
user=> (do
         (println "hello")
     (println "from")
         (println "side effects")
         (+ 5 5))
hello
from
side effects
10
```


Side Effects, cont'd

- Functions can also be written to contain side effects, by providing multiple expressions instead of just one as the body of the function

```
user=> (defn square
   "Squares a number, with side effects"
   [x]
   (println "Squaring" x)
   (println "The return value will be" (* x x))
   (* x x))
#'user/square

user=> (square 8)
Squaring 8
The return value will be 64
64
```

- This can also be done for loops

The End

References

- Sebesta, R., “Concepts of Programming Languages”
- Biancuzzi, F. and Warden, S.,“Masterminds of Programming – Conversations with the Creators of Major Programming Languages”, 2009
- Fogus, M. and Houser, C., “The Joy of Clojure”, 2011
- Halloway, S., “Programming Clojure”, 2009
- VanderHart, L. and Sierra, S., “Practical Clojure”, 2010
- clojure.org

Stockholm University