Functions

First Class

Prefix notation:

```
(predicate argument argument ...)
```

Pass a function as an argument:

```
(predicate arguments
(predicate (predicate arguments)))
```

First Class

- Homoiconicity
    - Arguments are data
    - Operators are data
- ∴ functions are treated like any other object
    - Passed as an argument
    - Returned as a result
    - Created at runtime

Pure

- Pure function: compiled → evaluated → result
- Side effects
  - Possible in Clojure
  - Do not change the program's state, unless **explicitly** made to do so
  - Example: `println`

Creating Functions

`fn`

`(fn [x] x)` - `(predicate argument argument)`

Creating Functions

Anonymous Functions

- `((fn [x] x) 2)` ⇒ `2`
    - `((predicate argument argument) argument))`
- Parameters can be functions
    - `((fn [x] (zero? x)) 0)` ⇒ `true`
    - `((fn [x] (zero? x)) 4)` ⇒ `false`

Creating Functions

Anonymous Functions

- `#()` - for short functions passed as arguments
  - It takes arguments named `%`, `%2`, `%3`, `%n` ... `%&`.
  - `(#(* 2 %) 3)` ⇒ `6`
  - `((fn [x] (even? x)) (#(* 2 %) 3))` ⇒ `true`
- An anonymous function has no name, so you don't know what to "call" it!

Creating Functions

Symbols Revisited

Remember:

- Symbols are forms. They evaluate to what they *name*.
- Example: `inc` is a symbol that *names* a function

Creating Functions

Symbols Revisited

`def`

- Defines a symbol
- `(def hello-world "Hello World!")`
- `hello-world` ⇒ `"Hello World!"`

Creating Functions

`def`

- Creates or locates a **global var** with the name of **symbol**
- Can name a scalar: `(def x 1)`: `x` ⇒ `1`
- Can name a collection: `(def x '(+ 2 3))`: `x` ⇒ `(+ 2 3)`
- Can name a function: `(def double-num (fn [x] (* x 2)))`: `(double-num 2)` ⇒ `4`

Creating Functions

The Fast Way

`defn`

```
(defn double-num [x] (* x 2))
```

≡

```
(def double-num (fn [x] (* x 2)))
```

Creating Functions

Multi-arity

```
(defn do-something
  ([] "nothing")
  ([one] "one parameter")
  ([one two] "two parameters")
  ([one two & more] "more than two parameters!"))

(do-something)
> "nothing"
(do-something 1)
> "one parameter"
(do-something 1 2)
> "two parameters"
(do-something 1 2 3 4)
> "more than two parameters"
```

Creating Functions

Multi-arity

Faster

```
(defn do-something [a & [b c]]
  (str "Required argument a is " a
       ". Optional argument b is " b
       ". C, optional, is " c "."))

(do-something 1 2)

> "Required argument a is 1.
   Optional argument b is 2. C, optional, is ."
```

Local Bindings

Special Form

`let`

- Immutable
- Bindings are sequential
- Pairs: `symbols` and `init-exprs`

```
(let [x 1
      y x]
  y)

> 1
```

Local Bindings

Special Form

`let`

`(let [double (fn [x] (* 2 x))] (double 21))` ⇒ `42`

`letfn`

`(letfn [(double [x] (* 2 x))] (double 21))` ⇒ `42`

Controlling Flow

`do`

- `let` contains an implicit `do`
- Evaluates expressions in order
- Fundmentally *imperative*
- Often used to create side effects (ex: print or i/o)

Controlling Flow

`do`

```
(if true (println "This is true: ") (+ 1 1))
> This is true:
> nil
> ;; nil is the return value
```

vs.

```
(if true (do (println "This is true: ") (+ 1 1)))
> This is true:
> 2
> ;; 2 is the return value
```

Recursion

Recursion

```
(defn factorial
 [n]
 (if (== 1 n)
   n
   (* n (factorial (- n 1)))))

(factorial 10)
> 3628800

(factorial 20000)
> ERROR: Stack Overflow
```

Recursion

Mutual recursion

```
(letfn [(is-even? [n]
          (if (zero? n)
              true
              (is-odd? (dec n))))
        (is-odd? [n]
          (if (zero? n)
            false
            (is-even? (dec n))))]
        (is-even? 42))

> true
```

Recursion

`loop` and `recur`

- `recur` must be the last expression evaluated aka the "tail position"
- Form: `loop` ≈ `let`
- Arity: the number of bindings.

```
(loop [x 10]
  (when (> x 1)
    (println x)
    (recur (- x 2))))
```

Recursion

`loop` and `recur`

```
(def factorial
  (fn [n]
    (loop [cnt n
           acc 1N]
      (if (>= 0 cnt)
          acc
        (recur (dec cnt) (* acc cnt))))))

(factorial 2000)
> 181920632023034513348...
```

Recursion

`loop` and `recur`

Tail Recursion

- Function calls are not duplicated on the stack
- Final answer obtained when the bottom of the recursive chain is reached
- No need to climb all the way back up to the top of the chain again

Recursion

`loop` and `recur`

`recur`

- The only non-stack-consuming looping construct in Clojure
- Use in tail-position is verified by the compiler
- Since Clojure uses the Java calling conventions, tail call optimization must be made explicit by `recur`

Recursion vs. Looping

Style: Declarative vs. Imperative

Imperative - uses statements that change a program's state by describing the program's flow

```
var numbers = [1,2,3]
var total = 0

for(var i = 0; i < numbers.length; i++) {
  total += numbers[i]
}
```

Note: `n` and `total` are modified in the loop

Recursion vs. Looping

Style: Declarative vs. Imperative

Declarative - the function expresses the logic of a computation without describing its control flow

```
(fn [numbers]
  (loop [n numbers
         total 0]
    (if (empty? n)
      total
      (recur (rest n) (+ total (first n))))))
```

Note: `n` and `total` are *not variables*, they are new local bindings in every recursive call

Recursion vs. Looping

Style: Declarative vs. Imperative

Equivalent Functional Solution

`(reduce + '(1 2 3))` ⇒ `6`

Note: this is more idiosyncratic to Clojure, more on this later!