	Haskell	Scala	Clojure
Practicalities			
install	brew install stack https://docs.haskellstack.org/en/stable/README/	brew install sbt brew install scala http://docs.scala-lang.org/getting-started.html	brew install clojure
docs	https://www.haskell.org/hoogle/	https://www.scala-lang.org/api/current/	http://clojuredocs.org/
src	https://hackage.haskell.org/package/base-4.10.1.0/docs/src/		
hyperpolyglot	http://hyperpolyglot.org/ml	http://hyperpolyglot.org/rust	http://hyperpolyglot.org/lisp
terminology	expression, reduce, redex, normal form, bindings, parameter, argument, type constructor, data constructor		form, expression, evaluate
build tooling	Stack	SBT	Leiningen lein new lein run -m pkg.core // or :main

repl	$ stack ghci
	$ stack repl
	Prelude> ctrl-d
		$ sbt
		sbt> console
		sbt> consoleQuick
		$ scala
		scala> :quit
		or ctrl-d or ctrl-c
			$ lein repl

repl	-- load file Prelude> :l foo.hs -- reload file Foo> :r -- type Foo> :t map -- kind Foo> :k Num	scala> :load foo.scala	
multiline repl	:{ :}		
interpreter	n/a	$ scala Foo.scala	$ java -cp clojure.jar clojure.main foo.clj
compile to executable	$ ghc foo.hs $ ./foo	$ scalac Foo.scala $ scala -cp . Foo	lein uberjar
default import	Prelude	java.lang._ scala._ scala.Predef._	

	Haskell	Scala	Clojure
import	import Foo.Bar -- all functions in Bar now in scope without qualification	import foo.Bar import foo.{Bar, Baz} import foo._	(require '[clojure.string :refer [split]]) (ns foo.bar (:require [clojure.string :as st]))
Basic Syntax			

main method	main = do
	  putStrLn("a")
		object Hello {
		  def main(args: Array[String]) {
		  println("a")
		  }
		}
			(defn -main [] ...)

ordered operation	do
	  expr1
	  expr2
		// by default	(do
			  (expr1)
			  (expr2))

EOL comment	-- comment	//	; adding
multiline comment	{- comment another comment -}	/* */	(quote (do ...)) #_(...) wrapped in commented out form

line sep / statement terminator	next line has equal or less indentation, or ;	; or sometimes newline
		
		A newline does not terminate a statement:
		  (1) inside ( ) or [ ],
		  (2) if the preceding line is not a complete statement,
		  (3) if following token not legal at start of a statement.
			whitespace and commas

block	offside rule or { } f x = x + r where r = 42	{ }	n/a
print	putStrLn . show $ "a"	println("a")	(println "a")
explicit type definition	1 :: Int	1 : Int	n/a clojure.spec, plumatic/schema
naming conventions	Haskell book 4.11		snakecase
Functions			

	Haskell	Scala	Clojure

function definition	f :: Int → Int
	f x = x
	let f x = x
		def f(x: Int):Int = x	(defn f [x] x)
			(def f (fn [x] x))
			(defn f
			  "Doc string goes here"
			  ([] 0)
			  ([x] x)
			  ([x & ys] x))

function destructure	f :: [Char] → [Char]
	f [] = []
	f x:'':xs = [x]
	f x:xs = xs
		n/a	(defn f
			  [[first-x second-x & rest-xs]]
			  (* first-x second-x))
			
			(defn f
			  [[first-x second-x :as all-xs]]
			  (* first-x second-x))
			
			(defn f
			  [{a :a b :b :as args}]
			  (* a b))

lambda / anonymous function	\x → x + 1 \x y → x + y	x ⇒ x + 1 (x, y) ⇒ x + y _ + _	#(+ % 1) #(+ %1 %2) #(f %1 %&) (fn [x] x)
curryable function	by default	def f(a: Int)(b: Int): Int	
curry a function	f_2args 1 (*) 1	f_2args.curried(1)	n/a
partial application	n/a	f_2args(1, _:Int)	(partial f_2args 1)
function composition	f . g	f compose g g andThen f	(comp f g) (comp (partial f_2args 1) g)
function invocation	f 1	f(1)	(f 1)
function application	f $ 1	f.apply(1)	(apply + '(1 2 3 4))
example	putStrLn . show . take 3 . reverse $ "Hello"		(defn f [& args] (apply str args))
infix in prefix	(*) 42 42	n/a	
prefix in infix	42 `f` 42	n/a	
infix/method		a + b a.+(b) a.foo(b) a foo b	
recur	n/a	last action is tail call, with @tailrec for validation	(loop [totals 0 vals vals] (recur args) // as last expr in function

	Haskell	Scala	Clojure
comprehension		for (i ← 0 to 42 if i % 2 == 0) yield i	

pattern matching		x match {
		  case 0 ⇒ "zero"
		  case _ ⇒ "many"
		}	

case classes		https://docs.scala-lang.org/tour/pattern-matching.html	
Basic Types			
type	:type foo		(type 1)
global keyword			:foo (keyword "foo")
namespaced keyword			::foo
symbol			a (quote a)
tuple	(1, "foo", True) (,) 1 "foo" True , is an infix operator!	(42, "foo", true)._1 nested pattern matching!	Tuple or just use vector
tuple operators	fst t snd t swap t		
vector / array		Vector(1, 2, 3) x +: xs // front xs :+ x // end	[1 2 3] (vector 1 2 3 4)
set		Set(1, 2, 3)	#{1 1 2 3 4} (hash-set 1 1 2 3 4)

map/dictionary		Map("a" → 1, "b" → 2)
		map("c") // NSEE
		map get "c" // None
			{:a 1 :b 2}
			(hash-map :a 1 :b 2)
			(array-map :a 1 :b 2)
			(into {} '([:a 1] [:b 2]))

		m withDefaultValue "foo"	
		m + ("a", 1) m + ("a" → 1)	
range		List.range(1, 10) 1 until 10 1 to 10 by 2	
tabulate		List.tabulate(5)(n ⇒ n * n)	
optional		Option : None Some(x)	
List operations			

	Haskell	Scala	Clojure
empty list	`[]`	`Nil` `List()`	`'()`
is empty list?	`null xs`	`xs == Nil`	`(empty? xs)`

list	`[1, 2, 3, 4]`	`1 :: 2 :: 3 :: Nil`
		`List(1,2,3)`
			`'(1 2 3 4)`
			`(quote (1 2 3 4))`
			`(list 1 2 3 4)`

first / car	`head xs`	`xs.head`	`(first xs)`
rest / cdr	`tail xs`	`xs.tail`	`(rest xs)`
last	`last xs`	`xs.last`	`(last xs)`
all but last	`init xs`	`xs.init`	
length/size	`length xs`	`xs.length`	
nth	`xs !! 1`	`xs(n)`	`(nth xs 1)`
min/max	`minimum xs` `maximum xs`	`xs.min` `xs.max`	
take	`take 1 xs`	`xs take n`	`(take 1 xs)`
drop	`drop 1 xs`	`xs drop n`	`(drop 1 xs)`

concat	`xs ++ xs2`
	`concat xs xs2`
		`xs ++ xs2`
		`List.concat(xs, xs2)`
		`xs ::: xs2`
			`(concat xs xs2)`

reverse list	`reverse xs`	`xs.reverse`	`(reverse '(1 2 3 4))` `(apply str (reverse "foo"))`
xs w/ x in position n		`xs updated (n, x)`	
cons (add to head)	`42 : xs`	`4 :: xs`	`(cons 1 xs)`
add to tail	`xs ++ [x]`	`xs :+ x` `xs ::: List(x)`	
conj (add in most natural way)			`(conj xs 1)`

add to middle	`let (ys,zs) = splitAt n xs`
	`  in ys ++ [new_element]`
	`++ zs`		

exists		`xs exists p`	
forall		`xs forall p`	
contains		`xs contains x`	`(contains? xs 1)`
index of		`xs indexOf x`	
take-while		`xs takeWhile p`	`(take-while p xs)`

	Haskell	Scala	Clojure
drop-while		xs dropWhile p	(drop-while p xs)
span		xs span p	
is empty list?		xs == Nil	(empty? xs)
into			(into [] (f xs)) (into {} (f m))
complement			(complement pred)
boolean	data Bool = True | False	true false	true false
null		null scala.Null	nil
bottom value	undefined	scala.Nothing	
falsey		false	false nil
null test		x == null	(nil? x)
number types	Int (Int32) Integer Float Double Rational Scientific		
disj (set)			(disj xs 1)

for-expression		for (x ← xs if x > 42) yield x * x
		for {
		  x ← xs
		  y ← ys
		  if x + y == 0
		} yield (x, y)	

sort with		xs sortWith p	
sort		xs.sorted	
group by		xs groupBy f	
Custom Types			
type definition	data Foo = Bar | Baz Corge deriving Grault		
type alias	type Name = String		
Control Structures			

	Haskell	Scala	Clojure

if	`if c then e1 else e2`		`(if (predicate)`
			`  true-expr`
			`  false-expr)`

if-not	`if not c then e2 else e1`		`(if-not (predicate)`
			`  false-expr`
			`  true-expr)`

when			`(when (predicate)` `  ... list of expressions, implied do)`
when-not			`(when-not (predicate)` `  ... list of expressions, implied do)`

multiple expressions			`(do`
			`  (println 'foo')`
			`  :return-val)`

case			`(case x`
			`  "a" :a`
			`  "b" :b`
			`  :nothing)`

cond			`(cold`
			`  (= x "foo") :foo`
			`  (= (apply str (reverse x)) "rab") :bar`
			`  :otherwise :nothing)`

loop/recur			loop/recur **todo**
destructuring			
assoc			`(assoc {:a 1} :b 2)`
assoc-in			`(assoc-in {:l1 {:l2a 1 :l2b 2}} [:l1 :l2a] 3)`
update-in			`(update-in {:l1 {:l2a 1 :l2b 2}} [:l1 :l2a] f)`

get			`(:a m)`
			`(m :a)`
			`(get xs :a)`

map		`xs.map(f)`	`(map f xs)`
fold left	`foldl`	`xs.foldLeft(init)(f)` `(xs foldLeft init)(f)`	
fold right	`foldr`	`xs.foldRight(init)(f)`	n/a
reduce		`xs reduce op` `xs reduceLeft op`	`(reduce f xs)` `(reduce f init xs)`

	Haskell	Scala	Clojure
filter		xs filter p	(filter pred xs)
filter not		xs filterNot p	
partition		xs partition p	
flatten		xss.flatten	(flatten nested_xs)
flat map		xs flatMap f	
zip		xs zip ys	(map vector xs ys)
unzip		xs.unzip	
any true			(some pred xs) // first truthy value
sort			(sort xs)
sort-by			(sort-by f xs)
filter not		xs filterNot p	
sum		xs.sum	(import 'java.util.Date) (Date.) // constructor
product		xs.product	
comparators	== /= > < >= <=		=
boolean operators	&& || not	&& || !	or and not
boolean pred			true? false?
other functions			inc
			(declare down)

threading expressions			(->> (f 10) (f2 inc) (f3 5) (reduce f4)) // last arg
			(-> (f 10) (f2 inc) (f3 5) (reduce f4)) // first arg
			(->> (range 10) (map inc) (interpose 5) (reduce +))

fun function names			hex->rgb
repeat			(repeat 42)

	Haskell	Scala	Clojure
repeatedly			(repeatedly f)
I/O			
read file			(spit file-path "foo")
write file			(slurp file-path)
open file			(with-open [reader (io/reader file-path)] …)
lines			(line-seq reader)
DB			clojure/java.jdbc
Strings			
literal	"foo"	"foo" """multiline foo"""	"foo"
regex			#",+"
string conc.		"a" + "b"	(str '(1 2 3))
format string		"foo %s %d %.2f".format("bar", 7, 3.1415)	(format "foo %s" "bar")
casing		"foo".toUpperCase "FOO".toLowerCase	import Data.Char map toUpper "foo" map toLower "FOO"
trim		" foo ".trim	
type conversion		x.toString toInt toFloat	
join		xs.mkString(",")	
split		"a b c".split(" ")	
length		"foo".length	
substring		"foo".substring(0, 1)	
Operators			
relational operators		== != < > <= >=	= /= < > <= >=
Variables			
write-once variable	x = 1 let x = 1	val x = 1 lazy val x = 1	(def x 1)

	Haskell	Scala	Clojure
scoped write-once variable		{ val x = 1 }	(let [x "a"] ...)
mutable variable		var x = 1	(declare ^:dynamic *foo*) (binding [*foo* "I exist!"] (println *foo*))
ADT/Class	data Person = Person String String Int		(deftype Foo) (defrecord Foo)
Multimethods			(defmulti foo) (defmethod ...) (derive ...)
Interface			(defprotocol ...) (extend-protocol ...)
arithmetic	+ - * / div mod quot rem		

mixins		trait Foo {
		  ...
		}	

comprehension	[a | a ← xs, a ⩽ n]		

https://github.com/dhinojosa/language-matrix

f - some function
p - predicate -- reduces to boolean

Scala

companion object with 'apply' method can use class like a function
StringOps (like Apache StringUtils)
1.to(10) or 1 to 10 -- single method infix no parens

{case p1 ⇒ e1 ... case pn ⇒ en}
x ⇒ match x { case ... }

Vector.fill(3)("*").updated(1, "+").mkString

implicit scope

```
def foo[A:Num](a:A):A
def foo[A](a:A)(implicit evidence: Num[A]):A =
```

Scala with Style

catz discipline

http://eed3si9n.com/herding-cats/