Introduction to Functional Programming

Zsók Viktória, Ph.D.

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary
zsv@elte.hu

Overview

Motivation

Functional programming:

- allows programs to be written clearly, concisely
- has a high level of abstraction
- supports reusable software components
- encourages the use of formal verification
- enables rapid prototyping
- has inherent parallel features

What is functional programming?

- the closest programming style to mathematical writing, thinking
- which one should be the first programming language?
- the basic element of the computation is the function
- basically function compositions are applied
- running a program is called evaluation

Syntax

The syntax of a programming language is the set of rules applied to describe a problem.

f(a) => `f a`
f(a,b) + cd => `f a b + c * d`
f(g(b)) => `f (g b)`
f(a)g(b) => `f a * g b`

History

- Lisp - list processor, in early 60s John McCarthy
- operates on lists, functions can be arguments to other functions
- type checking, ability to check programs before running them
- ML, Miranda, Haskell, Clean
- lazy functional programming

Writing functional programs is FUN

- to motivate you to write functional programs
- to get involved in working with FP
- to have FUN by learning FP

The Clean compiler can be downloaded from:
`http://clean.cs.ru.nl/Clean`
unzip, start IDE, open examples.icl create a project file examples.prj and run, only one active Start expression!!

```
module examples
import StdEnv    // needed for standard functions
Start = 42 // 42
```

Clean - Start

- Some start expressions:

```
Start = 4*6+8

Start = sqrt 2.0

Start = sin x

Start = sum [1..10]
```

- constants pi = 3.1415926

Program evaluation

- reduction steps
- redex
- normal form

```
f x = (x + 8) * x

Start = f 2

Start
→ f 2
→ (2 + 8) * 2
→ 10 * 2
→ 20
```

Reduction steps, redex

- the process of evaluation is called *reduction*
- replacing a part of expression which matches a function definition is called *reduction step*
- *redex* = reducible expression
- when a function contains no redexes is called *normal form*

Lazy and eager evaluation

- lazy = the expression is not evaluated until is not needed
- opposite is eager evaluation = all arguments are evaluated before the function's result
- Clean is pure, lazy functional language
- advantages of lazy evaluation: infinite lists, less evaluations

Standard functions

- `StdEnv` - contains all
- the name of your own functions should start with letter then zero or more letters, digits, symbols
- upper and lower case allowed but treated differently
- funny symbols, built-in function names can not be used

Some predefined operators / functions on numbers

- integers 18, 0, -23 and floating-point numbers 1.5, 0.0, 4.765, 1.2e3 1200.0
- addition +, subtraction -, multiplication *, division /
- for Int some standard functions abs, gcd, sign
- for Real sqrt, sin, exp
- for Bool type True, False (George Boole eng.math. 1815-1864)
- boolean operators
  >, <=, == (equal), <> (not equal), && (and), || (or)
- comments // or /* ... */

Getting started

Simple examples of Clean functions:

```
inc1 :: Int → Int
inc1 x = x + 1
double :: Int → Int
double x = x + x
quadruple :: Int → Int
quadruple x = double (double x)
factorial :: Int → Int
factorial n = prod [1 .. n]
```

Using them:

```
Start = 3+10*2 // 23
Start = sqrt 3.0 // 1.73...
Start = quadruple 2 // 8
Start = factorial 5 // 120
```

Definitions by cases

The cases are guarded by Boolean expressions:

```
abs1 x
| x < 0 = ~x    // tilde x
| otherwise = x
Start = abs1 -4    // two cases, the result is 4

abs2 x
| x < 0 = ~x    // tilde x
= x
Start = abs2 4    // otherwise can be omitted, 4

// more then two guards or cases
signof :: Int -> Int
signof x
| x > 0 = 1
| x == 0 = 0
| x < 0 = -1
Start = signof -8  // -1
```

Definitions by recursion

Examples of recursive functions:

```
factor :: Int → Int
factor n
| n == 0 = 1
| n > 0 = n * factor (n - 1)
Start = factor 5 // 120
```

```
power :: Int Int → Int
power x n
| n == 0 = 1
| n > 0 = x * power x (n - 1)
Start = power 2 5 // 32
```

Compositions, function parameters

```
// function composition
twiceof :: (a → a) a → a
twiceof f x = f (f x)
Start = twiceof inc 0 // 2

// Evaluation:
twiceof inc 0
→ inc (inc 0)
→ inc (0+1)
→ inc 1
→ 1+1
→ 2

Twice :: (t→t) → (t→t)
Twice f = f o f
Start = Twice inc 2 // 4

f = g o h o i o j o k   is nicer than   f x = g(h(i(j(k x))))
```

Definition

- data structures - store and manipulate collections of data
- list - sequence of elements of the same type
- elements of a list can be of any type
- they are written between [ ] brackets
- coma separates the elements
- considered recursive data type

Lists in Clean

- lists in Clean are regarded as linked lists - a chain of boxes referring to each other
- empty list is []
- every list has a type, the type of the contained elements
- no restrictions on the number of elements
- singleton list with one element [False], [[1,2,3]]
- special constructor is [1:[2,3,4]] is equivalent to [1,2,3,4]
  [1,2,3] is equivalent to [1:[2:[3:[]]]]

Defining lists

One of the most important data structures in FP is the list: a sequence of elements of the same type

```
l1 :: [Int]
l1 = [1, 2, 3, 4, 5]
l2 :: [Bool]
l2 = [True, False, True]
l3 :: [Real→Real]
l3 = [sin, cos, sin]
l4 :: [[Int]]
l4 = [[1, 2, 3], [8, 9]]
l5 :: [a]
l5 = []
l6 :: [Int]
l6 = [1..10]
l7 :: [Int]
l7 = [1..]
```

Generating lists

```
Start =
    [1..10]        // [1,2,3,4,5,6,7,8,9,10]
    [1,2..10]      // [1,2,3,4,5,6,7,8,9,10]
    [1,0.. -10]    // [1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
    [1.. -10]      // []
    [1..0]         // []
    [1..1]         // [1]
    [1,3..4]       // [1,3]
    [1..]          // [1,2,3,4,5,6,7,8,9,10,...
    [1,3..]        // [1,3,5,7,9,11,13,15,...
    [100,80..]     // [100,80,60,40,20,0,-20,-40,...
```

Operations with lists

```
Start =
    hd [1, 2, 3, 4, 5]          // 1
    tl [1, 2, 3, 4, 5]          // [2, 3, 4, 5]
    drop 2 [1, 2, 3, 4, 5]      // [3, 4, 5]
    take 2 [1, 2, 3, 4, 5]      // [1 , 2]
    [1, 2, 3] ++ [6, 7]         // [1, 2, 3, 6, 7]
    reverse [1, 2, 3]           // [3, 2, 1]
    length [1, 2, 3, 4]         // 4
    last [1, 2, 3]              // 3
    init [1, 2, 3]              // [1, 2]
    isMember 2 [1, 2, 3]        // True
    isMember 5 [1, 2, 3]        // False
    flatten [[1,2], [3, 4, 5], [6, 7]]     // [1, 2, 3, 4, 5, 6, 7]
```

Definition of some operations

```
take :: Int [a] → [a]
take n [] = []
take n [x : xs]
| n < 1 = []
| otherwise = [x : take (n-1) xs]

drop :: Int [a] → [a]
drop n [] = []
drop n [x : xs]
| n < 1 = [x : xs]
| otherwise = drop (n-1) xs

Start = take 2 []                 // []
Start = drop 5 [1,2,3]            // []
Start = take 2 [1 .. 10]          // [1,2]
Start = drop ([1..5]!!2) [1..5]   // [4,5]
```

Definition of some operations

```
reverse :: [a] → [a]
reverse [] = []
reverse [x : xs] = reverse xs ++ [x]

Start = reverse [1,3..10]            // [9,7,5,3,1]
Start = reverse [5,4 .. -5]          // [-5,-4,-3,-2,-1,0,1,2,3,4,5]
Start = isMember 0 []                // False
Start = isMember -1 [1..10]          // False
Start = isMember ([1..5]!!1) [1..5]  // True
```

Definitions by patterns

Various patterns can be used:

```
// some list patterns
triplesum :: [Int] → Int
triplesum [x, y, z] = x + y + z
Start = triplesum [1,2,4] // 7 [1,2,3,4] error

head :: [Int] → Int
head [x : y] = x
Start = head [1..5] // 1

tail :: [Int] → [Int]
tail [x : y] = y
Start = tail [1..5] // [2,3,4,5]

// omitting values
f :: Int Int → Int
f _ x = x
Start = f 4 5 // 5
```

Definitions by patterns

```
// patterns with list constructor
g :: [Int] → Int
g [x, y : z] =  x + y
Start = g [1, 2, 3, 4, 5] // 3

// patterns + recursively applied functions
lastof [x] = x
lastof [x : y] = lastof y
Start = lastof [1..10] // 10
```

Definitions by recursion 2

```
// recursive functions on lists
sum1 x
| x == [] = 0
| otherwise = hd x + sum1 (tl x)

sum2 [] = 0
sum2 [first : rest] = first + sum2 rest
Start = sum1 [1..5] // 15 the same for sum2

// recursive function with any element pattern
length1 [] = 0
length1 [_ : rest]= 1 + length1 rest
Start = length1 [1..10] // 10
```

Warm-up exercises

Evaluate the following expressions:

1. `(take 3 [1..10]) ++ (drop 3 [1..10])`
2. `length (flatten [[1,2], [3], [4, 5, 6, 7], [8, 9]])`
3. `isMember (length [1..5]) [7..10]`
4. `[1..5] ++ [0] ++ reverse [1..5]`

Solutions

1. `(take 3 [1..10]) ++ (drop 3 [1..10])`
2. `length (flatten [[1,2], [3], [4, 5, 6, 7], [8, 9]])`
3. `isMember (length [1..5]) [7..10]`
4. `[1..5] ++ [0] ++ reverse [1..5]`

```
1. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2. 9
3. False
4. [1, 2, 3, 4, 5, 0, 5, 4, 3, 2, 1]
```

init, last, flatten

`init` selects everything but the last element (compare with `last`).

```
init :: [a] → [a]
init [x] = []
init [x : xs] = [x : init xs]

last :: [a] → a
last [x] = x
last [x : xs] = last xs

flatten :: [[a]] → [a]
flatten [] = []
flatten [x : xs] = x ++ flatten xs
```

Comparing and ordering lists

Equality of lists (operators are also functions written between the arguments)

```
(==) :: [a] [a] -> Bool | == a
(==) [] [] = True
(==) [] [y : ys] = False
(==) [x : xs] [] = False
(==) [x : xs] [y : ys] = x == y && xs == ys
```

Ordering lists

Lexicographical ordering (dictionary ordering)
E.g. [2, 3] < [3, 0] or [10, 1] < [10, 2]

```
(<) :: [a] [a] → Bool | <, == a
(<) [] [] = False
(<) [] _ = True
(<) _ [] = False
(<) [x : xs] [y : ys] = x < y || ( x == y && xs < ys)
```

Other comparisons

Once we have < and == all others can be defined.

```
(<>) x y = not ( x == y )
(>) x y = y < x
(>=) x y = not (x < y)
(<=) x y = not (y < x)
```

Compositions, function parameters

```
// function parameters
filter :: (a → Bool) [a] → [a]
filter p [] = []
filter p [x : xs]
| p x = [x : filter p xs]
| otherwise = filter p xs

Start = filter isEven [1..10] // [2,4,6,8,10]

odd x = not (isEven x)
Start = odd 23 // True

Start = filter (not o isEven) [1..100] // [1,3,5,..,99]
```

Partial parameterization

Calling a function with fewer arguments than it expects.

```
plus x y = x + y
successor :: (Int → Int)
successor = plus 1
Start = successor 4 // 5
succ = (+) 1
Start =  succ 5 // 6

// the function adding 5 to something
Start = map (plus 5) [1,2,3] // [6,7,8]

plus :: Int → (Int→Int)
accepts an Int and returns the successor function of type Int→Int

Currying: treats equivalently the following two types
Int Int → Int    and    Int → (Int → Int)
```

Higher order functions

```
map :: (a→b) [a] → [b]
map f [] = []
map f [x:xs] = [f x : map f xs]

Start = map inc [1, 2, 3]          // [2, 3, 4]
Start = map double [1, 2, 3]       // [2, 4, 6]

// lambda expressions
Start = map (λx = x*x+2*x+1) [1..10] // [4,9,16,25,36,49,64,81,100,121]

// mapfun [f,g,h] x = [f x, g x, h x]
mapfun [] x = []
mapfun [f : fs] x = [f x : mapfun fs x]

Start = mapfun [inc, inc, inc] 3 // [4, 4, 4]
```

Filtering

```
filter p [] = []
filter p [x : xs]
| p x = [x : filter p xs]
| otherwise = filter p xs
Start = filter isEven [2,4,6,7,8,9] // [2, 4, 6, 8]

takeWhile :: (a→Bool) [a] → [a]
takeWhile p [] = []
takeWhile p [x : xs]
| p x = [x : takeWhile p xs]
| otherwise = []
Start = takeWhile isEven [2,4,6,7,8,9] // [2, 4, 6]

dropWhile p [] = []
dropWhile p [x : xs]
| p x = dropWhile p xs
| otherwise = [x : xs]
Start = dropWhile isEven [2,4,6,7,8,9] // [7, 8, 9]
```

Folding and writing equivalences

```
foldr op e [] = e
foldr op e [x : xs] = op x (foldr op e xs)

foldr (+) 0 [1,2,3,4,5] → ( 1 + ( 2 + ( 3 + ( 4 + ( 5 +  0 )))))

Start = foldr (+) 10 [1, 2, 3]    // 16

product [] = 1
product [x:xs] = x * product xs

and [] = True
and [x:xs] = x && and xs

product = foldr (*) 1
and = foldr (&&) True
sum = foldr (+) 0
```

Iteration

```
// compute f until p holds
until p f x
| p x = x
| otherwise = until p f (f x)
Start = until ((<)10) ((+)2) 0 // 12

// iteration of a function
iterate :: (t → t) t → [t]
iterate f x = [x : iterate f (f x)]
Start = iterate inc 1 // infinite list [1..]
```

Tuples

```
(1,'f')            :: (Int,Char)
("world",True,2)   :: (String,Bool,Int)
([1,2],sqrt)       :: ([Int],Real→Real)
(1,(2,3))          :: (Int,(Int,Int))
// any number 2-tuples pair, 3-tuples, no 1-tuple (8) is just integer

fst :: (a,b) → a
fst (x,y) = x
Start = fst (10, "world") // 10

snd :: (a,b) → b
snd (x,y) = y
Start = snd (1,(2,3))      // (2,3)

f :: (Int, Char) → Int
f (n, x) = n + toInt x
Start = f (1,'a') // 98
```

Tuples

```
splitAt :: Int [a] → ([a],[a])
splitAt n xs = (take n xs, drop n xs)

Start = splitAt 3 ['hello'] // (['h','e','l'],['l','o'])

search :: [(a,b)] a → b | == a
search [(x,y):ts] s
| x == s = y
| otherwise = search ts s

Start = search [(1,1), (2,4), (3,9)] 3 // 9
```

Zipping

```
zip :: [a] [b] → [(a,b)]
zip [] ys = []
zip xs [] = []
zip [x : xs] [y : ys] = [(x , y) : zip xs ys]

Start = zip [1,2,3] ['abc'] // [(1,'a'),(2,'b'),(3,'c')]
```

List comprehensions

```
Start :: [Int]
Start = [x * x \\ x ← [1..10]]  // [1,4,9,16,25,36,49,64,81,100]

// expressions before double backslash
// generators after double backslash
// i.e. expressions of form x <- xs x ranges over values of xs
// for each value value the expression is computed

Start = map (λx = x * x) [1..10]  // [1,4,9,16,25,36,49,64,81,100]

// constraints after generators

Start :: [Int]
Start = [x * x \\ x ← [1..10] | x rem 2 == 0]   // [4,16,36,64,100]
```

List comprehensions

// nested combination of generators
// coma combinator - generates every possible combination of the
// corresponding variables, last variable changes faster
// for each x value y traverses the given list

```
Start :: [(Int,Int)]
Start = [(x,y) \\ x ← [1..2], y ← [4..6]]
```
// [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6)]

// parallel combinator of generators is &

```
Start = [(x,y) \\ x ← [1..2] & y ← [4..6]]
```
// [(1,4),(2,5)]

// multiple generators with constraints

```
Start = [(x,y) \\ x ← [1..5], y ← [1..x] | isEven x]
```
// [(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]

List comprehensions - equivalences

```
mapc :: (a→b) [a] → [b]
mapc f l = [ f x \\ x ← l ]

filterc :: (a→Bool) [a] → [a]
filterc p l = [ x \\ x ← l | p x ]

zipc :: [a] [b] → [(a,b)]
zipc as bs = [(a , b) \\ a ← as & b ← bs]

Start = zipc [1,2,3] [10, 20, 30] // [(1,10),(2,20),(3,30)]

// functions like sum, reverse, isMember, take
// are hard to write using list comprehensions
```

Warm-up exercises 2

Write a function or an expression for the following:

1. compute 5! factorial using `foldr` => 120
2. rewrite `flatten` using `foldr` (for the following list [[1,2], [3, 4, 5], [6, 7]] => [1,2,3,4,5,6,7])
3. using `map` and `foldr` compute how many elements are altogether in the following list [[1,2], [3, 4, 5], [6, 7]] => 7
4. using `map` extract only the first elements of the sublists in [[1,2], [3, 4, 5], [6, 7]] => [1,3,6]

Solutions 2

Write a function or an expression for the following:
1. compute 5! factorial using `foldr` => 120
2. rewrite `flatten` using `foldr` (for the following list [[1,2], [3, 4, 5], [6, 7]] => [1,2,3,4,5,6,7])
3. using `map` and `foldr` compute how many elements are altogether in the following list [[1,2], [3, 4, 5], [6, 7]] => 7
4. using `map` extract only the first elements of the sublists in [[1,2], [3, 4, 5], [6, 7]] => [1,3,6]

```
1. Start = foldr (*) 1 [1..5]
2. Start = foldr (++) [] [[1,2], [3, 4, 5], [6, 7]]
3. Start = foldr (+) 0 (map length [[1,2], [3, 4, 5], [6, 7]])
4  Start = map hd [[1,2], [3, 4, 5], [6, 7]]
```

Sorting lists

```
Start = sort [3,1,4,2,0] // [0,1,2,3,4]

// inserting in already sorted list
Insert :: a [a] → [a] | Ord a
Insert e [] = [e]
Insert e [x : xs]
| e ≤ x = [e , x : xs]
| otherwise = [x : Insert e xs]
Start = Insert 5 [2, 4 .. 10] // [2,4,5,6,8,10]

mysort :: [a] → [a] | Ord a
mysort [] = []
mysort [a:x] = Insert a (mysort x)
Start = mysort [3,1,4,2,0] // [0,1,2,3,4]

Insert 3 (Insert 1 (Insert 4 (Insert 2 (Insert 0 [] ))))
```

Mergesort

```
merge :: [a] [a] → [a] | Ord a
merge [] ys = ys
merge xs [] = xs
merge [x : xs] [y : ys]
| x ≤ y = [x : merge xs [y : ys]]
| otherwise = [y : merge [x : xs] ys]

Start = merge [2,5,7] [1,5,6,8] // [1,2,5,5,6,7,8]
Start = merge [] [1,2,3] // [1,2,3]
Start = merge [1,2,10] [] // [1,2,10]
Start = merge [2,1] [4,1] // [2,1,4,1]
Start = merge [1,2] [1,4] // [1,1,2,4]
```

Mergesort 2

```
msort :: [a] → [a] | Ord a
msort xs
| len ≤ 1 = xs
| otherwise = merge (msort ys) (msort zs)
where
    ys = take half xs
    zs = drop half xs
    half = len / 2
    len = length xs

Start = msort [2,9,5,1,3,8] // [1,2,3,5,8,9]
```

Quick sort

```
qsort :: [b] → [b] | Ord b
qsort [] = []
qsort [a : xs] = qsort [x \\ x ← xs | x <  a] ++ [a] ++
                 qsort [x \\ x ← xs | x >= a]

Start = qsort [2,1,5,3,6,9,0,1] // [0,1,1,2,3,5,6,9]
```

Generating infinite list

```
// generating infinite list
Start = [2..] // [2,3,4,5,...]
Start = [1,3..] // [1,3,5,7,...]

fromn :: Int → [Int]
fromn n = [n : fromn (n+1)]

Start = fromn 8 // [8,9,10,...]

// intermediate result is infinite
Start = map ((^)3) [1..]

// final result is finite
Start = takeWhile ((>) 1000) (map ((^)3) [1..])
// [3,9,27,81,243,729]
```

Infinite lists - repeat

```
// generating infinite list with repeat from StdEnv
repeat :: a → [a]
repeat x = list where list = [x:list]

Start = repeat 5 // [5,5,5,...]

repeatn :: Int a → [a]
repeatn n x = take n (repeat x)

Start = repeatn 5 8 // [8,8,8,8,8]
```

Infinite lists - iterate

```
// generating infinite list with iterate from StdEnv
iterate :: (a→a) a → [a]
iterate f x = [x: iterate f (f x)]

Start = iterate inc 5 // [5,6,7,8,9,...]

Start = iterate ((+) 1) 5 // [5,6,7,8,9,...]

Start = iterate ((*) 2) 1 // [1,2,4,8,16,...]

Start = iterate (λ x= x/10) 54321 // [54321,5432,543,54,5,0,0...]
```

Prime numbers

```
divisible :: Int Int → Bool
divisible x n = x rem n == 0

denominators :: Int → [Int]
denominators x = filter (divisible x) [1..x]

prime :: Int → Bool
prime x = denominators x == [1,x]

primes :: Int → [Int]
primes x = filter prime [1..x]

Start = primes 100 // [2,3,5,7,...,97]
```

Sieve

```
sieve :: [Int] → [Int]
sieve [p:xs] = [p: sieve [ i \\ i ← xs | i rem p ≠ 0]]

Start = take 100 (sieve [2..])
```

Some more examples

```
qeq :: Real Real Real → (String,[Real])
qeq a b c
 | a == 0.0      = ("not quadratic",[])
 | delta < 0.0   = ("complex roots",[])
 | delta == 0.0  = ("one root",[¬b/(2.0*a)])
 | delta > 0.0   = ("two roots", [(¬b+radix)/(2.0*a),
                                  (¬b-radix)/(2.0*a)])
 where
     delta = b*b-4.0*a*c
     radix = sqrt delta

Start = qeq 1.0 2.0 1.0

Start = qeq 1.0 5.0 7.0
```

Warm-up exercises 3

Write a function for the following:

1. fibonnacci n
2. count the occurrences of a number in a list
3. write a list comprehension for the doubles of a list

Solutions 3

```
fib :: Int → Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
Start = fib 5  // 8

fib2 :: Int → Int
fib2 n = fibAux n 1 1

fibAux 0 a b          = a
fibAux i a b | i > 0 = fibAux (i-1) b (a+b)

Start = fib2 8
```

Solutions 3

```
CountOccurrences :: a [a] → Int | == a
CountOccurrences a [x : xs] = f a [x : xs] 0
where
      f a [] i = i
      f a [x : xs] i
        | a == x = f a xs i+1
                 = f a xs i

Start = CountOccurrences 2 [2, 3, 4, 2, 2, 4, 2, 1] // 4

Start = [2*x \\ x ← [1..5]] // [2, 4, 6, 8, 10]
```

Conclusions

The goal was:

- to give an introduction to functional programming
- to present important data structures in fp
- to get familiarized with basic and higher order functions
- to practice by examples in order to acquire the programming paradigm