



Functional Programming

Lecture 1: Introduction

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Acknowledgements

This course is based on materials created by:

- Jiří Vyskočil, Michal Pěchouček
 - ČVUT, Czech Republic
- Koen Claessen and Emil Axelsson
 - Chalmers University of Technology, Sweden

What is functional programming?

Wikipedia: Functional programming is a **programming paradigm**—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions.

Goal of the course

1. Improve your programming skills!
 - master recursion
 - master problem decomposition
 - rethink side effects
 - different perspective to the same problems
2. Learn principles of functional programming
 - has clear benefits for SOME problems
 - it is used in many other languages

Why do I care?

- quickly learn new programming languages
- programming paradigms change and develop
- no side effects is great for parallelization and verification
- understanding fundamentals of computation

Does anyone use it?

- Lisp: AutoCAD, Emacs, Gimp
- Haskell: Facebook, Google, Intel
- Scala: Twitter, eBay, LinkedIn
- Erlang: Facebook, Pinterest
- Clojure: Walmart, Atlassian

Imperative vs. Declarative

- **Instructions** to change the computer's state
 - $x := x + 1$
 - `deleteFile("slides.pdf")`
- Are executed
 - have effects
- Run program by following instructions top-down
- Functions used to **declare** dependences between data values:
 - $z = g(y)$
 - $y = f(x)$
- Expressions are evaluated
 - result to a value
- Run program by evaluating dependencies

Pure functional programming

- No side effects
 - output of a function depends **only** on its inputs
 - function does not change anything in evaluation
 - can be evaluated in any order (many times, never)
- No mutable data
- More complex function based on recursion
 - no for/while cycles
 - natural problem decomposition
 - mathematical induction

Pure functional programming

- Forbids most of what you use in (C/Java)
 - we will show you do really not lose anything
 - it can be useful for many tasks
 - it often leads to more compact code !?!
- Substantially less time spent debugging
 - encapsulation, repeatability, variety of mistakes
- Focus on operations with symbols
- Easier parallelization and verification
- Generally less computationally efficient

Brief History

- Lambda calculus (1930s)
 - formal theory of computation older than TM
- Lisp = List processor (1950s)
 - early practical programming language
 - second oldest higher level language after Fortran
- ML = Meta language (1970s)
 - Lisp with types, used in compilers
- Haskell = first name of Curry (1990s)
 - standard for functional programming research
- Python, Scala, Java8, C++ 11,

What will we learn?

Lisp (Scheme)

Lambda calculus

Haskell

Why LISP?

- Extremely simple
- Reasonably popular
- Allows deriving all concepts from principles
- Directly matches lambda calculus

Why Haskell?

- Purely functional language
 - promotes understanding the paradigm
- Rich syntactic sugar (contrast to Lisp)
- Most popular functional language
- Fast prototyping of complex systems

- Why not Scala?

Course organization

- Web: cw.fel.cvut.cz/wiki/courses/fup
- Lectures + Labs
- Homework – every 2 weeks (50 %)
 - 3x10 Scheme
 - 2x10 Haskell
 - must have at least 1 point from each and ≥ 25
- Programming exam (30 %)
- Test (20 %)

Suggested literature

R. Kent Dybvig: The Scheme Programming Language, Fourth Edition, MIT Press, 2009.

<https://www.scheme.com/tspl4/>

Greg Michaelson: An Introduction to Functional Programming Through Lambda Calculus, Dover edition, 2011.

Scheme

- Dialect of Lisp (such as Common Lisp, Racket)
- Created in 1970 at MIT by Steele and Sussman
- Last standard from 2007
 - The Revised⁶ Report on the Algorithmic Language Scheme (R6RS)
- Supports imperative programming
 - we will not use it (we want to learn FP)
- DrRacket: racket-lang.org
 - text editor + REPL (read-evaluate-print loop)

Scheme syntax

Scheme program is a collection of expressions

Expression	Evaluates to
5	5
"abc"	"abc"
#t	#t
+	#<procedure: +>
a	?

Prefix notation

Infix notation

$$1+2*5$$

Prefix notation

$$+ 1 * 2 5$$

In Scheme, there are no operator preferences

$$(+ 1 (* 2 5))$$

S - expression

(fn arg1 arg2 ... argN)

(“operator of calling a function”
fn expression that evaluates to a *procedure*
argX arguments of the function
) end of function call

Conventions

Special suffixes

? for predicates

! for procedures with side effects

-> in procedures that transform a type of an object

Prefix of character / string / vector procedures

char-, string-, and vector-

Basics data types

Numbers (infinite precision, complex, etc.)

`+, -, *, /, abs, sqrt`

Logical values

`#t, #f, >, <, and, or, boolean?`

Strings

`"abc", "Hello !!!", string?, substring`

Other types

`symbol?, char?, procedure?, pair?, port?,
vector?`

Quote

Do not evaluate, just to return the argument

```
(quote exp)
```

Abbreviated by `'`

A quoted expression can be evaluated by *eval*

```
(eval (quote (+ 1 2)))
```

Evaluate part of the argument

```
(quasiquote (* 1 2 3 (unquote (+ 2 2)) 4 5))
```

Abbreviated by ``` and `,` respectively

Identifiers

Keywords, variables, and symbols may be formed from the following set of characters:

the lowercase letters a through z,

the uppercase letters A through Z,

the digits 0 through 9, and

the characters ? ! . + - * / < = > : \$ % ^ & _ ~ @

cannot start with 0-9, +, -, @

Define

Naming expressions

```
(define id exp)
```

Defining functions

```
(define (name <formals>) <body>)
```


Comments

`;` starts a comment until the end of the line
on the line before the explained expression

`::;` still start a comment until the end of the line
used to comment a function or code segment

`#|` `|#` delimit block comments

Conditional expressions

if

```
(if test-exp then-exp else-exp)
```

cond

```
(cond  
  (test-exp1 exp)  
  (test-exp2 exp)  
  (#t exp)  
  ...)
```

Recursion

A function calling itself

```
(define (fact n)
  (cond ((= 0 n) 1)
        ((= 1 n) 1)
        (#t (* n (fact (- n 1)))))
  )
)
```

Avoiding infinite recursion

1. First expression of the function is a `cond`
2. The first test is a termination condition
3. The "then" of the first test is not recursive
4. The `cond` pairs are in increasing order of the amount of work required
5. The last `cond` pair is a recursive call
6. Each recursive call brings computations closer to the termination condition

Recursion

Tail recursion

Last thing a function does is the recursive call

Analytic / synthetic

Return value from termination condition / composed

Tree recursion

Function is called recursively multiple times (qsort)

Indirect recursion

Function A calls function B which calls A

Lists

The key data structure of Lisp

S-expressions are just lists

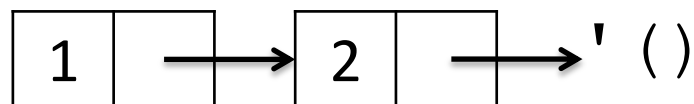
```
' (+ 1 2 3 4 5)
```

Lists can be created by a function

```
(list item1 item2 ... itemN)
```

List are linked lists of pairs with ' () at the end
sometimes abbreviated by .

```
(cons 1 2), (cons 1 (cons 2 ' ()))
```



Lists

Pairs forming the lists can be decomposed by

`car` *[car]* first element of the pair

`cdr` *[could-er]* second element of the pair

`(caddr x)` shortcut for `(car (cdr (cdr x)))`

Empty list is a null pointer

`null?` tests whether the argument is the empty list

Last

Return the last element of a list

```
(define (last list)
  (cond
    ((null? (cdr list)) (car list))
    (#t (last (cdr list))))
)
```


What have we learned?

- Functional programming is an alternative programming paradigm
 - no side effects
 - no mutable data structures
 - focus on symbols
- Recursion is the key programming method
- Lists are a key data structure