

Principles of Programming Languages

<http://www.di.unipi.it/~andrea/Didattica/PLP-14/>

Prof. Andrea Corradini

Department of Computer Science, Pisa

Lesson 25

- Functional programming languages
- Introduction to Haskell

Historical Origins

- The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
 - different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics
- These results led Church to conjecture that *any* intuitively appealing model of computing would be equally powerful as well
 - this conjecture is known as *Church's thesis*

Historical Origins

- Turing's model of computing was the *Turing machine* a sort of pushdown automaton using an unbounded storage "tape"
 - the Turing machine computes in an imperative way, by changing the values in cells of its tape like variables just as a high level imperative program computes by changing the values of variables

Historical Origins

- Church's model of computing is called the *lambda calculus*
 - based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter λ , hence the notation's name)
 - allows one to define mathematical functions in a constructive/effective way
 - Lambda calculus was the inspiration for functional programming
 - computation proceeds by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

Functional Programming Concepts

- Functional languages such as **Lisp**, **Scheme**, **FP**, **ML**, **Miranda**, and **Haskell** are an attempt to realize Church's lambda calculus in practical form as a programming language
- The key idea: do everything by composing functions
 - no mutable state
 - no side effects

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
 - 1st class and high-order functions
 - recursion
 - Takes the place of iteration
 - powerful list facilities
 - Recursive function exploit recursive definition of lists
 - serious polymorphism
 - Relevance of Container/Collections
 - fully general aggregates
 - Data structures cannot be modified, have to be re-created
 - structured function returns
 - garbage collection
 - Unlimited extent for locally allocated data structures

Recursion vs. Iteration

- Recursion can be used in place of iteration

```
x := 0; i := 1; j := 100;
while i < j do
    x := x + i*j; i := i + 1;
    j := j - 1
end while
return x
```

becomes

```
f(0,1,100), where
f(x,i,j) == if i < j then
f (x+i*j, i+1, j-1) else x
```

- Thinking about recursion as a mechanical replacement for iteration is wrong
- One should learn to think in recursive style

Other Related Concepts

- **Lisp** also has some features that are not necessary present in other functional languages:
 - programs are data
 - self-definition
 - read-evaluate-print interactive loop
- Variants of LISP
 - (Original) Lisp: purely functional, dynamically scoped as early variants
 - Common Lisp: current standard, statically scoped, very complex
 - Scheme: statically scoped, very elegant, used for teaching

Other functional languages: the ML family

- Robin Milner (Turing award in 1991, CCS, Pi-calculus, ...)
- Statically typed, general-purpose programming language
 - “Meta-Language” of the LCF theorem proving system
- Type safe, with type inference and formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
 - Expression-oriented
 - Higher-order functions
 - Garbage collection
 - Abstract data types
 - Module system
 - Exceptions

Other functional languages: Haskell

- Designed by committee in 80's and 90's to unify research efforts in lazy languages
 - Evolution of Miranda
 - Haskell 1.0 in 1990, Haskell '98, Haskell' ongoing
- Several features in common with ML, but **some differ**:
- Types and type checking
 - Type inference
 - Parametric polymorphism
 - **Ad hoc polymorphism (aka overloading)**
- Control
 - **Lazy vs. eager evaluation**
 - Tail recursion and continuations
- Purely functional
 - **Precise management of effects**
 - Rise of multi-core, parallel programming likely to make minimizing state much more important

Applicative and Normal Order evaluation

- Applicative Order evaluation
 - Arguments are evaluated before applying the function – aka *Eager evaluation*
- Normal Order evaluation
 - Function evaluated first, arguments if and when needed
 - Sort of parameter passing by name
 - Some evaluation can be repeated
- Church-Rosser
 - If evaluation terminates, the result (*normal form*) is unique
 - If some evaluation terminates, normal order evaluation terminates

β -conversion

$$(\lambda x.t) t' = t [t'/x]$$

Applicative order

$$\begin{aligned} & (\lambda x.(+ x x)) (+ 3 2) \\ & \rightarrow (\lambda x.(+ x x)) 5 \\ & \rightarrow (+ 5 5) \\ & \rightarrow 10 \end{aligned}$$

Define $\Omega = (\lambda x.x x)$

Then

$$\begin{aligned} \Omega\Omega &= (\lambda x.x x) (\lambda x.x x) \\ &\rightarrow x x [(\lambda x.x x)/x] \\ &\rightarrow (\lambda x.x x) (\lambda x.x x) = \Omega\Omega \\ &\rightarrow \dots \textit{non-terminating} \end{aligned}$$

$$\begin{aligned} & (\lambda x. 0) (\Omega\Omega) \\ & \rightarrow \{ \textit{Applicative order} \} \\ & \dots \textit{non-terminating} \end{aligned}$$

$$\begin{aligned} & (\lambda x. 0) (\Omega\Omega) \\ & \rightarrow \{ \textit{Normal order} \} \\ & 0 \end{aligned}$$

Normal order

$$\begin{aligned} & (\lambda x.(+ x x)) (+ 3 2) \\ & \rightarrow (+ (+ 3 2) (+ 3 2)) \\ & \rightarrow (+ 5 (+ 3 2)) \\ & \rightarrow (+ 5 5) \\ & \rightarrow 10 \end{aligned}$$

The Glasgow Haskell Compiler [GHC]

www.haskell.org/platform

Current release: 2014.2.0.0

New GHC: 7.8.3
Major update: OpenGL and GLUT

[Prior releases](#)
[Future schedule](#)

[Problems?](#)
[Documentation](#)
[Library Doc](#)

The Haskell Platform



Windows



Mac



Linux

Comprehensive

The Haskell Platform is the easiest way to get started with programming Haskell. It comes with all you need to get up and running. Think of it as "Haskell: batteries included". [Learn more...](#)

Robust

The Haskell Platform contains only stable and widely-used tools and libraries, drawn from a pool of thousands of Haskell packages, ensuring you get the best from what is on offer.

Cutting Edge

The Haskell Platform ships with advanced features such as multicore parallelism, thread sparks and transactional memory, along with many other technologies, to help you get work done.

Basic Overview of Haskell

- Interactive Interpreter (ghci): read-eval-print
 - ghci infers type before compiling or executing
 - Type system does not allow casts or similar things!
- Examples

```
Prelude> (5+3) -2
6
it :: Integer
Prelude> if 5>3 then "Harry" else "Hermione"
"Harry"
it :: [Char]      -- String is equivalent to [Char]
Prelude> 5==4
False
it :: Bool
```

Overview by Type

- Booleans

```
True, False :: Bool$  
if ... then ... else ...    --types must match
```

- Integers

```
0, 1, 2, ... :: Integer  
+, * , ...   :: Integer -> Integer -> Integer
```

- Strings

```
"Ron Weasley"
```

- Floats

```
1.0, 2, 3.14159, ...    --type classes to disambiguate
```

Simple Compound Types

- Tuples

```
(4, 5, "PLP") :: (Integer, Integer, String)
```

- Lists

```
[] :: [a] -- NIL, polymorphic type  
1 : [2, 3, 4] :: [Integer] -- infix cons notation  
[1,2]++[3,4] :: [Integer] -- concatenation
```

- Records

```
data Person = Person {firstName :: String,  
                      lastName  :: String}  
hg = Person { firstName = "Hermione",  
            lastName   = "Granger"}
```

More on list constructors

```
ghci> [1..20]           -- ranges
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> [3,6..20]        -- ranges with step
[3,6,9,12,15,18]
ghci> [7,6..1]
[7,6,5,4,3,2,1]
```

```
ghci> take 10 [1..]     -- (prefix of) infinite lists
[1,2,3,4,5,6,7,8,9,10]
ghci> take 10 (cycle [1,2])
[1,2,1,2,1,2,1,2,1,2]
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```


Patterns and Declarations

- Patterns can be used in place of variables
 <pat> ::= <var> | <tuple> | <cons> | <record> ...
- Value declarations
 - General form: <pat> = <exp>
 - Examples

```
myTuple = ("Foo", "Bar")
(x,y)   = myTuple   -- x = "Foo", y = "Bar"
myList  = [1, 2, 3, 4]
z:zs    = myList    -- z = 1, zs = [2,3,4]
```

- Local declarations

```
let (x,y) = (2, "FooBar") in x * 4
```

Functions and Pattern Matching

- Anonymous function

```
\x -> x+1      --like Lisp lambda, function (...) in JS
```

- Function declaration form

```
<name> <pat1> = <exp1>
```

```
<name> <pat2> = <exp2> ...
```

```
<name> <patn> = <expn> ...
```

- Examples

```
f (x,y) = x+y      --argument must match pattern (x,y)
```

```
length [] = 0
```

```
length (x:s) = 1 + length(s)
```

Higher Order functions: Map Function on Lists

- Apply function to every element of list

```
map f [] = []  
map f (x:xs) = f x : map f xs
```

```
map (\x -> x+1) [1,2,3]
```



```
[2,3,4]
```

More Functions on Lists

- Apply function to every element of list

```
map f [] = []  
map f (x:xs) = f x : map f xs
```

```
map (\x -> x+1) [1,2,3]           [2,3,4]
```

- Reverse a list

```
reverse [] = []  
reverse (x:xs) = (reverse xs) ++ [x]
```

```
reverse xs =  
  let rev ( [], accum ) = accum  
      rev ( y:ys, accum ) = rev ( ys, y:accum )  
  in rev ( xs, [] )
```

List Comprehensions

- Notation for constructing new lists from old:

```
myData = [1,2,3,4,5,6,7]

twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]

twiceEvenData = [2 * x | x <- myData, x `mod` 2 == 0]
-- [4,8,12]
```

- Similar to “set comprehension”
 $\{ x \mid x \in \text{Odd} \wedge x > 6 \}$

More on List Comprehensions

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20] -- more predicates

ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110] -- more lists

length' xs = sum [1 | _ <- xs] -- anonymous (don't care) var

-- strings are lists...
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Datatype Declarations

- Examples

```
data Color = Red | Yellow | Blue
```

elements are Red, Yellow, Blue

```
data Atom = Atom String | Number Int
```

elements are Atom "A", Atom "B", ..., Number 0, ...

```
data List = Nil | Cons (Atom, List)
```

elements are Nil, Cons(Atom "A", Nil), ...

Cons(Number 2, Cons(Atom("Bill"), Nil)), ...

- General form

```
data <name> = <clause> | ... | <clause>  
<clause> ::= <constructor> | <constructor> <type>
```

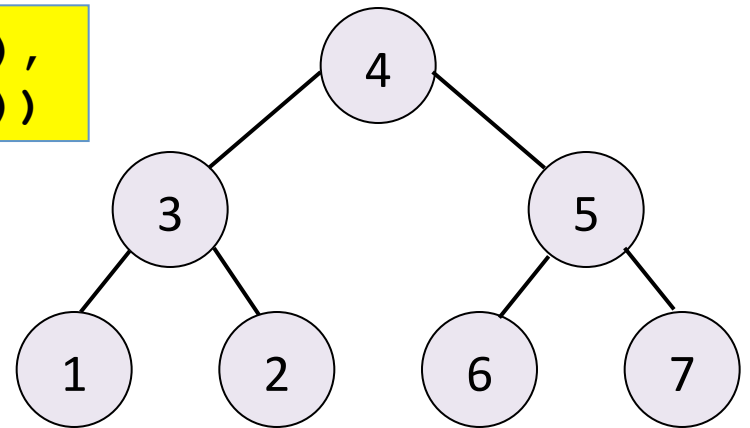
– Type name and constructors must be Capitalized.

Datatypes and Pattern Matching

- Recursively defined data structure

```
data Tree = Leaf Int | Node (Int, Tree, Tree)
```

```
Node (4, Node (3, Leaf 1, Leaf 2),  
      Node (5, Leaf 6, Leaf 7))
```



- Constructors can be used in Pattern Matching
- Recursive function

```
sum (Leaf n) = n  
sum (Node (n, t1, t2)) = n + sum(t1) + sum(t2)
```


Case Expression

- Datatype

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)
```

- Case expression

```
case e of
  Var n -> ...
  Const n -> ...
  Plus (e1, e2) -> ...
```

Indentation matters in case statements in Haskell.

Example: Evaluating expressions by Cases

```
data Exp = Var Int | Const Int | Plus (Exp, Exp)

ev ( Var n ) = Var n
ev ( Const n ) = Const n
ev ( Plus ( e1, e2 ) ) =

  case ev e1 of
    Var n -> Plus( Var n, ev e2)
    Const n -> case ev e2 of
      Var m -> Plus( Const n, Var m)
      Const m -> Const (n+m)
      Plus(e3,e4) -> Plus ( Const n,
                            Plus ( e3, e4 ))
    Plus(e3, e4) -> Plus( Plus ( e3, e4 ), ev e2)
```

Function Types in Haskell

In Haskell, $f :: A \rightarrow B$ means for every $x \in A$,

$$f(x) = \begin{cases} \text{some element } y = f(x) \in B \\ \text{run forever} \end{cases}$$

In words, “if $f(x)$ terminates, then $f(x) \in B$.”

In ML, functions with type $A \rightarrow B$ can throw an exception or have other effects, but not in Haskell

```
ghci> :t not      -- type of some predefined functions
not :: Bool -> Bool
ghci> :t (+)
(+) :: Num a => a -> a -> a
ghci> :t not
not :: Bool -> Bool
ghci> :t (:)
(:) :: a -> [a] -> [a]
ghci> :t elem
elem :: Eq a => a -> [a] -> Bool
```

Higher-Order Functions

- Functions that take other functions as arguments or return as a result are higher-order functions.
- Common Examples:
 - **Map**: applies argument function to each element in a collection.
 - **Reduce**: takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.

```
ghci> :t map
map :: (a -> b) -> [a] -> [b]
ghci> let list = [1,2,3]
ghci> map (\x -> x+1) list
[2,3,4]
ghci> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
ghci> foldl (\accum i -> i + accum) 0 list
6
```

Laziness

- Haskell is a **lazy** language
- Functions and data constructors don't evaluate their arguments until they need them

```
cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e
```

- Programmers can write control-flow operators that have to be built-in in eager languages

Short-circuiting
"or"

```
(||) :: Bool -> Bool -> Bool
True  || x = True
False || x = x
```

Searching a substring: Java code

```
static int indexOf(char[] source, int sourceOffset, int sourceCount,
                  char[] target, int targetOffset, int targetCount,
                  int fromIndex) {
    ...

    char first = target[targetOffset];
    int max = sourceOffset + (sourceCount - targetCount);

    for (int i = sourceOffset + fromIndex; i <= max; i++) {
        /* Look for first character. */
        if (source[i] != first) {
            while (++i <= max && source[i] != first);
        }

        /* Found first character, now look at the rest of v2 */
        if (i <= max) {
            int j = i + 1;
            int end = j + targetCount - 1;
            for (int k = targetOffset + 1; j < end && source[j] ==
                target[k]; j++, k++);

            if (j == end) {
                /* Found whole string. */
                return i - sourceOffset;
            }
        }
    }
    return -1;
}
```

Searching a Substring: Exploiting Laziness

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool
-- returns True if first list is prefix of the second
isPrefixOf [] x = True
isPrefixOf (y:ys) [] = False
isPrefixOf (y:ys) (x:xs) =
    if (x == y) then isPrefixOf ys xs else False
```

```
suffixes :: String -> [String]
-- All suffixes of s
suffixes [] = [[]]
suffixes (x:xs) = (x:xs) : suffixes xs
```

```
or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or [] = False
or (b:bs) = b || or bs
```

```
isSubString :: String -> String -> Bool
x `isSubString` s = or [ x `isPrefixOf` t
                        | t <- suffixes s ]
```

A Lazy Paradigm

- Generate all solutions (an enormous tree)
- Walk the tree to find the solution you want

```
nextMove :: Board -> Move
nextMove b = selectMove allMoves
  where
    allMoves = allMovesFrom b
```

A gigantic (perhaps infinite)
tree of possible moves

Core Haskell

- Basic Types
 - Unit
 - Booleans
 - Integers
 - Strings
 - Reals
 - Tuples
 - Lists
 - Records
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions