Principles of Programming Languages

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

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 27

- Introduction to Hakell

The origins: ML programming language

- Statically typed, general-purpose programming language
  - "Meta-Language" of the LCF theorem proving system
- Type safe, with 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

Haskell

- Haskell programming language is
  - Similar to ML: general-purpose, strongly typed, higher-order, functional, supports type inference, interactive and compiled use
  - Different from ML: lazy evaluation, purely functional core, rapidly evolving type system
- 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

Some intersting features of Haskell

- Types and type checking
  - General issues in static and dynamic typing
  - 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

The Glasgow Haskell Compiler [GHC]
www.haskell.org/platform

Basic Overview of Haskell

- Interactive Interpreter (ghci): read-eval-print
  - ghci infers type before compiling or executing
  - Type system does not allow casts or other loopholes!
- 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, "Griffendor") :: (Integer, Integer, String)
```

- Lists

```
[] :: [a]                          -- polymorphic type
```

```
1 : [2, 3, 4] :: [Integer]    -- infix cons notation
```

- 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 = (“Flitwick”, “Snape”)
(x,y)   = myTuple
myList = [1, 2, 3, 4]
z:zs   = myList
```

  - Local declarations

```
let (x,y) = (2, “Snape”) in x * 4
```

Functions and Pattern Matching

- Anonymous function

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

- Function declaration form

<name> <pat$_1$> = <exp$_1$>

<name> <pat$_2$> = <exp$_2$> …

<name> <pat$_n$> = <exp$_n$> …

- Examples

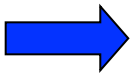
```
f (x,y) = x+y      --argument must match pattern (x,y)
length [] = 0
length (x:s) = 1 + length(s)
```

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]
```

- Compare to Lisp

```
(define map
    (lambda (f  xs)
      (if   (eq? xs ())   ()
       (cons (f  (car xs))  (map f  (cdr xs)))
   )))
```

More Functions on Lists

- Append lists

```
append ([], ys) = ys
append (x:xs, ys) = x : append (xs, ys)
```

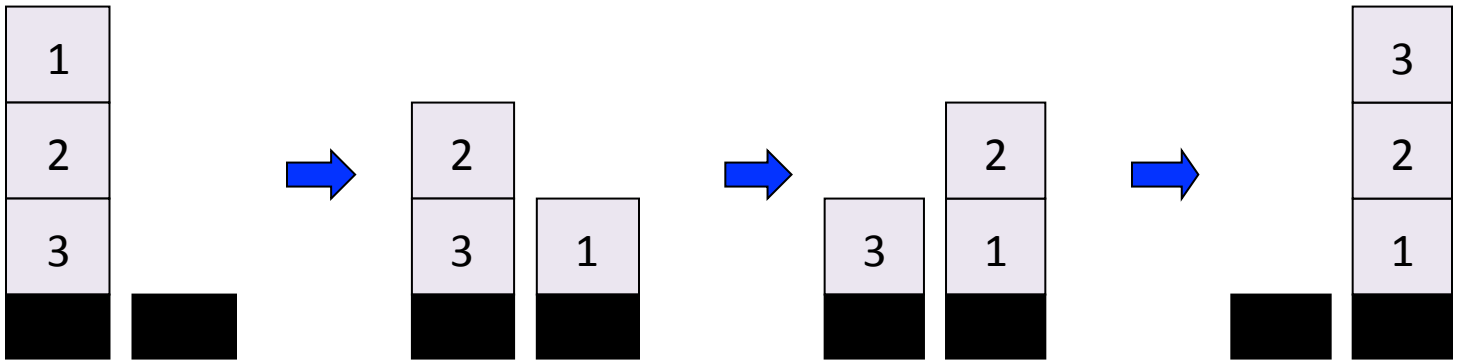
- Reverse a list

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

- Questions
  - How efficient is reverse?
  - Can it be done with only one pass through list?

More Efficient Reverse

```
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> | <contructor> <type>
```

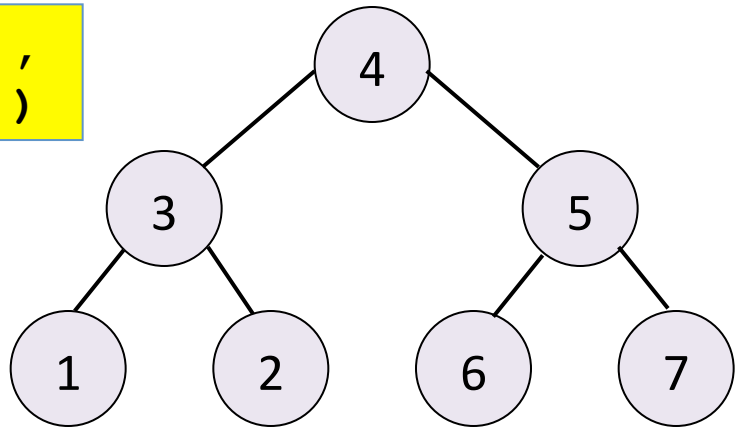
– Type name and constructors must be Capitalized.

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))
```



- Recursive function

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

Example: Evaluating Expressions

- Define datatype of expressions

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

write (x+3)+ y as Plus(Plus(Var 1, Const 3), Var 2)

- Evaluation function

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

- Examples

```
ev(Plus(Const 3, Const 2))   ⇒   Const 5
```

```
ev(Plus(Var 1, Plus(Const 2, Const 3)))   ⇒
                          Plus(Var 1, Const 5)
```

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.

Evaluation 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 -> 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;
}
```

Exploiting Laziness

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

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

type String = [Char]

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

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