301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

AP-13: *Functional Programming*

Functional Programming - Outline

- Historical origins
- Main concepts
- Languages families: LISP, ML, and Haskell
- Core concepts of Haskell
- Lazy evaluation

Functional Programming: 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

- Church's model of computing is called the ***lambda calculus***
  - based on the notion of *parameterized expressions* (parameters introduced by letter ***λ***)
  - 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
- We shall see later the basics of lambda-calculus

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**
    - Functions can be denoted, passed as arguments to functions, returned as result of function invocation
    - Meaningful because new functions can be defined
  - **Recursion**
    - Takes the place of iteration (no "control variables")
  - **Powerful list facilities**
    - Recursive functions exploit recursive definition of lists
  - **Polymorphism** (typically universal parametric implicit)
    - Relevance of Containers/Collections

Functional Programming Concepts

- **Fully general aggregates**
  - Wide use of tuples and records
  - Data structures cannot be modified, have to be re-created
- **Structured function returns**
  - No side-effects, thus the only way for functions to pass information to the caller
- **Garbage collection**
  - In case of static scoping, unlimited extent for:
    - locally allocated data structures
    - locally defined functions
  - They cannot be allocated on the stack

The LISP family of languages

- **LISP** (**LIS**t **P**rocessor) was designed in 1958 by John McCarty (Turing award in 1971) and implemented in 1960 by Steve Russel
- Only FORTRAN is older…
- Main programming language for **AI**
- It includes some features that are not necessary present in other functional languages:
  - Programs (S-expressions) are data (lists)
    - `(func arg1 arg2 … argn)`
  - Self-definition
    - A LISP interpreter can be written in few LISP lines
  - Read-evaluate-print interactive loop

The LISP family of languages

- Variants of LISP
  - (Original) LISP
    - purely functional
    - strong dynamic type checking
    - dynamically scoped
  - **Common Lisp**: current standard
    - statically scoped
    - very rich and complex
  - **Scheme**:
    - statically scoped
    - essential syntax
    - very elegant
    - widely used for teaching

Other functional languages: the ML family

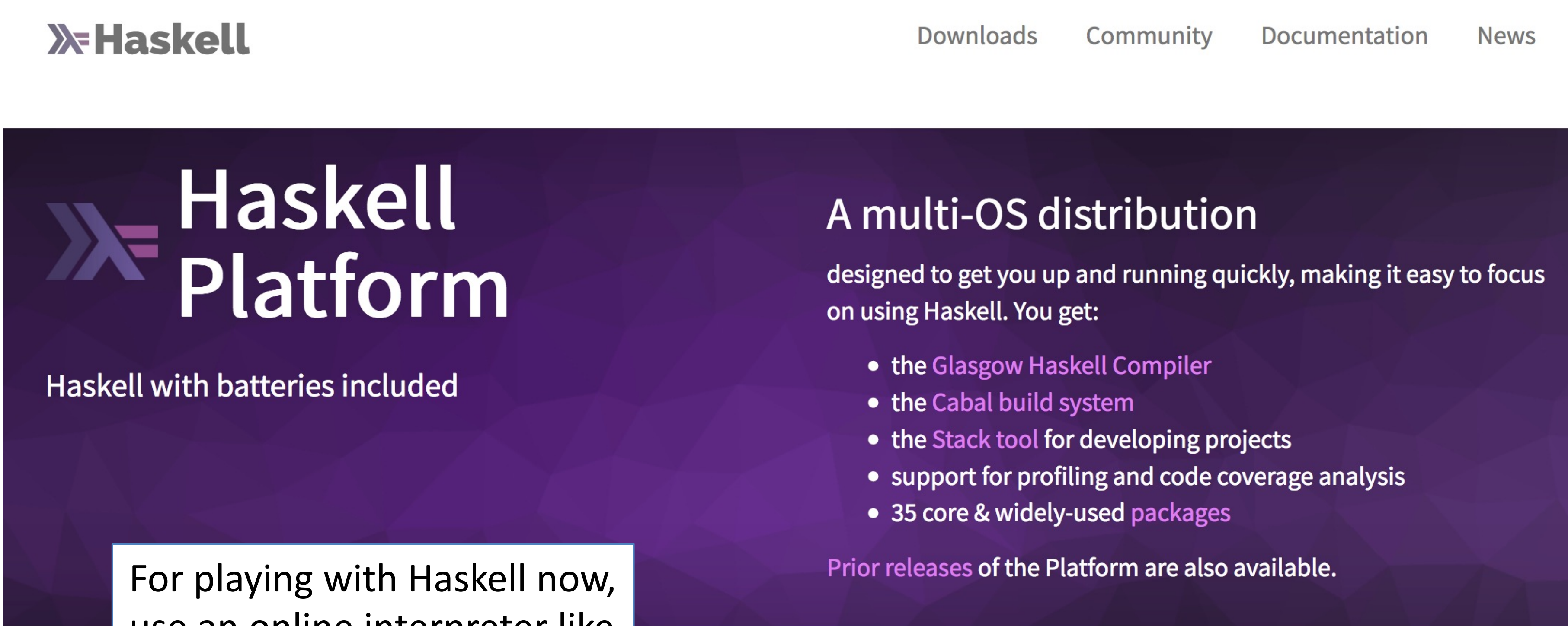
- Robin Milner (Turing award in 1991, CCS, Pi-calculus, …)
- Statically typed, general-purpose programming language
  - "Meta-Language" of the LCF theorem proving system
- Type safe, with **type inference** and **formal semantics**
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- Impure: it allows side-effects
- Members of the family: Standard ML, Caml, OCaml, $F^{\#}$

Other functional languages: **Haskell**

- Designed by committee in 80's and 90's to unify research efforts in lazy languages
  - Evolution of Miranda, name from **Haskell Curry**, logician (1900-82),
  - Haskell 1.0 in 1990, Haskell '98, Haskell 2010 (→ Haskell 2020)
- Several features in common with ML, but **some differ**:
- Types and type checking
  - Type inference
  - Implicit parametric polymorphism
  - **Ad hoc polymorphism (overloading)**
- Control
  - **Lazy evaluation**
  - Tail recursion and continuations
- Purely functional
  - **Precise management of effects**

Downloading Haskell

https://www.haskell.org/platform/



For playing with Haskell now, use an online interpreter like repl.it

Core Haskell

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

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==4
False
Prelude> :set +t  -- enables printing of types
Prelude> 'x'
'x'
it :: Char
Prelude> (5+3)-2
6
it :: Num a => a  -- generic constrained type
                   -- "type class"
Prelude> :t (+)      -- type of a function
(+) :: Num a => a -> a -> a
```

Overview by Type

- Booleans

```
True, False :: Bool

not :: Bool -> Bool

and, or :: Foldable t => t Bool -> Bool

if …  then … else …
      --conditional expression: types must match
```

- Characters & Strings

```
'a','b',';','\t', '2', 'X' :: Char

"Ron Weasley" :: [Char] --strings are lists of chars
```

Overview by Type

- Numbers

```
0,1,2,…:: Num p => p --type classes, to disambiguate

1.0, 3.1415 ::  Fractional a => a

(45 :: Integer)  :: Integer  -- explicit typing

+, * , -, …   :: Num a => a -> a -> a
-- infix  +  becomes prefix (+)
-- prefix binary op becomes infix `op`
/         :: Fractional a => a -> a -> a

div, mod :: Integral a => a -> a -> a

^ :: (Num a, Integral b) => a -> b -> a
```

Simple Compound Types

- Tuples

```
("AP",2017) ::  Num b => ([Char], b) -- pair
fst :: (a, b) -> a  -- selector: only for pairs
snd :: (a, b) -> b  -- selector: only for pairs

('4', True, "AP") :: (Char, Bool, [Char]) -- tuple
```

- Lists

```
[] :: [a]                          -- NIL, polymorphic type
1 : [2, 3, 4] :: Num a => [a]-- infix cons notation
[1,2]++[3,4] :: Num a => [a]    -- concatenation
head :: [a] -> a      -- first element
tail :: [a] -> [a]   -- rest of the list
```

- Records

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

More on list constructors

```
ghci> [1..20]          -- range
[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]        -- range with step
[3,6,9,12,15,18]
ghci> [7,6..1]
[7,6,5,4,3,2,1]
```

```
ghci> [1..]            -- an infinite list: runs forever
ghci> take 10 [1..]          -- prefix of an infinite lists
[1,2,3,4,5,6,7,8,9,10]       -- returns!
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]
```

How does it work??? Later…

Binding variables

- Variables (names) are bound to expressions, without evaluating them (because of lazy evaluation)
- The scope of the binding is the rest of the session
- Comparing OCaml and Haskell

HASKELL	OCaml
`Prelude> let a = 6 -- no output`	`# let a = 6 ;;`
	`val a : int = 6`
`Prelude> b = a + 2 --'let' optional`	`# let b = a + 2 ;;`
	`val b : int = 8`
`Prelude> b    -- now b is evaluated`	`# b ;;`
`8`	`- : int = 8`
`Prelude> a = a + 1 -- no output`	`# let a = a + 1 ;;`
`Prelude> a  -- what does it print?`	`val a : int = 7`
`^CInterrupted. - loop broken`	

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
```

Anonymous Functions (lambda abstraction)

- Anonymous functions

```
\x -> x+1   --like LISP lambda, function (…) in JS
Prelude>  (\x -> x+1)5     =>  6
Prelude>  f = \x -> x+1
Prelude> :t f
f :: Num a => a -> a
Prelude> f 7  =>  8
```

- Anonymous functions using patterns

```
Prelude> h = \(x,y) -> x+y
h :: Num a => (a, a) -> a
Prelude> h (3, 4)  =>  7
Prelude> h 3 4     =>  error

Prelude> k = \(z:zs) -> length zs
k :: [a] -> Int
Prelude> k "hello"     =>  4
```

Function declarations

- Function declaration form

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

  <name> <pat$_2$> = <exp$_2$> …

- Examples

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

length [] = 0
length (x:s) = 1 + length(s)

Prelude> len (z:zs) = length zs
len :: [a] -> Int
Prelude> len [1,2,3]  =>  2
Prelude> len []
*** Exception: <interactive>:143:5-24: Non-
  exhaustive patterns in function len
```

More Functions on Lists

- Reverse a list

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

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

- Other (higher-order) functions later