

# Programming Systems in Artificial Intelligence

# Functional Programming

Siegfried Nijssen

8/03/16



Universiteit  
Leiden  
The Netherlands

Discover the world at Leiden University

# Overview

- Foundations: lambda calculus
- Functional programming languages and their concepts
  - LISP / Scheme
  - Haskell
  - Other languages
- Foundations: monads

# Lambda Calculus

- Invented by Alonzo Church in the early 1930s
- Is a universal model of computation and equivalent to the Turing Machine (Church-Turing thesis, 1937)
- Has a different perspective on performing calculations:
  - Turing machines are built on execution instructions (imperative coding style)
  - Lambda calculus is built on rewriting function applications (declarative coding style)
- Is the formal basis for functional programming languages

# Lambda Calculus: Expressions

- An expression in lambda calculus defined recursively as follows:

$\langle \text{expression} \rangle := \langle \text{variable} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle$

$\langle \text{function} \rangle := ( \lambda \langle \text{variable} \rangle . \langle \text{expression} \rangle )$

$\langle \text{application} \rangle := ( \langle \text{expression} \rangle \langle \text{expression} \rangle )$

- Functions and variables are written down using identifiers
- Examples of expressions:

$(\lambda x. x)$

$(\lambda x. y)$

$(\lambda x. x)(\lambda y. y)$

$((\lambda x. x)y)$

# Lambda Calculus: Notation

- Brackets around applications can be removed; applications are left associative:

$$(xy)z \equiv xyz$$

- Brackets around functions can be removed; the inner expression reaches as far right as possible:

$$(\lambda x.(xy)) \equiv \lambda x.xy \not\equiv (\lambda x.x)y$$

- A sequence of lambdas is contracted:

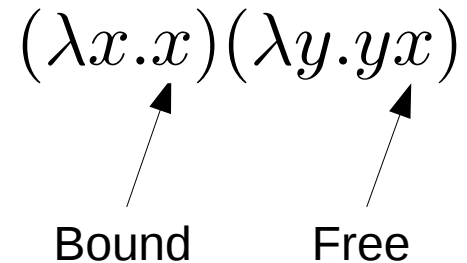
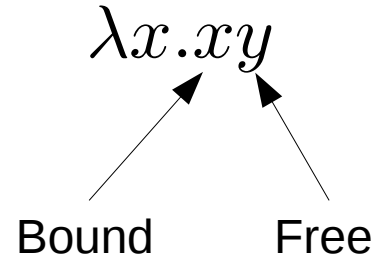
$$\lambda x.\lambda y.\lambda z.xyz \equiv \lambda xyz.xyz$$

- We can give names to expressions:

$$P \equiv (\lambda x.x) \qquad Q \equiv (\lambda y.y) \qquad PQ = (\lambda x.x)(\lambda y.y)$$

# Lambda Calculus: Free/Bound Variables

- Lambdas are similar to quantifiers in first-order logic; they bind variables



# Lambda Calculus: Substitutions

- Function applications can be rewritten using *substitutions*:

$$(\lambda x.E_1)(E_2) \equiv E_1 \theta$$

with  $\theta = \{x \mapsto E_2\}$

- Examples:

$$(\lambda x.x)y = x\{x \mapsto y\} = y$$

$$(\lambda x.x)(\lambda y.y) = x\{x \mapsto \lambda y.y\} = \lambda y.y$$

- Avoid naming conflicts by renaming variables; expressions with only different variable names are equivalent!

$$(\lambda x.(\lambda y.xy))y \neq \lambda y.yy \qquad (\lambda x.(\lambda y.xy))y \equiv (\lambda x.(\lambda t.xt))y \equiv \lambda t.yt$$

# Computations using Lambda Calculus

- Define the following expressions:

$$T \equiv \lambda xy.x$$

$$F \equiv \lambda xy.y$$

$$\wedge \equiv \lambda xy.xyF$$

- Then:

$$\wedge TF \equiv (\lambda xy.xyF)TF \equiv (\lambda y.TyF)F \equiv TFF \equiv (\lambda xy.x)FF \equiv (\lambda y.F)F \equiv F$$

$$\wedge TT \equiv (\lambda xy.xyF)TT \equiv (\lambda y.TyF)T \equiv TTF \equiv (\lambda xy.x)TF \equiv (\lambda y.T)F \equiv T$$

$$\wedge FF \equiv (\lambda xy.xyF)FF \equiv (\lambda y.FyF)F \equiv FFF \equiv (\lambda xy.y)FF \equiv (\lambda y.y)F \equiv F$$



# Computations using Lambda Calculus

- Define the following expressions:

$$T \equiv \lambda xy.x$$

$$F \equiv \lambda xy.y$$

$$\wedge \equiv \lambda xy.xyF$$

$$\vee \equiv \lambda xy.xTy$$

$$\neg \equiv \lambda x.xFT$$

- Then:

$$\neg(\vee(\wedge TF)F) \equiv \neg(\vee TF) \equiv \neg(T) \equiv F$$

# Computations using Lambda Calculus

- Define the following expressions:

$$1 \equiv \lambda sz.sz$$

$$2 \equiv \lambda sz.s(sz)$$

$$3 \equiv \lambda sz.s(s(sz))$$

$\vdots$

$$S \equiv \lambda w y x.y(w y x)$$

- Then:

$$S1 = (\lambda w y x.y(w y x))1 = \lambda y x.y(1 y x) = \lambda y x.y(y x) = \lambda sz.s(sz) = 2$$

$$S2 = (\lambda w y x.y(w y x))2 = \lambda y x.y(2 y x) = \lambda y x.y(y(y x)) = \lambda sz.s(s(sz)) = 3$$

# Computations using Lambda Calculus

- Exercise:  
Calculate the outcome of  $2S3$

$$1 \equiv \lambda sz.sz$$

$$2 \equiv \lambda sz.s(sz)$$

$$3 \equiv \lambda sz.s(s(sz))$$

$\vdots$

$$S \equiv \lambda w y x.y(w y x)$$

# Computations using Lambda Calculus

- Exercise:  
Calculate the outcome of  $2S3$

$$1 \equiv \lambda sz.sz$$

$$2 \equiv \lambda sz.s(sz)$$

$$3 \equiv \lambda sz.s(s(sz))$$

$\vdots$

$$S \equiv \lambda w y x.y(w y x)$$

$$((\lambda sz.s(sz))S)3 \equiv$$

$$(\lambda z.S(Sz))3 \equiv$$

$$S(S3) \equiv 5$$

# LISP

- List Processing languages: has a large focus on manipulating lists
- Programming language implementing ideas found in Lambda Calculus
- Development started in 1957, with many variations existing today
- Current languages strongly inspired by LISP:
  - Common Lisp (the current standard)
  - Scheme (simplified version of LISP, the basis for the Church probabilistic programming system)
  - Clojure (the basis for the Anglican probabilistic programming system, runs on the Java VM, can use Java libraries)

# Scheme

- Uses prefix notation similar to Lambda Calculus

```
(+ 3 4 )  
(* 5 6 )  
(and #T #F)  
(and (or #T #F) #T)
```

- Lambda functions can be defined

```
(lambda (x) (* x x))
```

- And applied:

```
((lambda (x) (* x x)) 7)
```

# Scheme

- Note: lists are everywhere in LISP...

<code>(+ 3 4 )</code>	←	List with +, 3, 4
<code>(* 5 6 )</code>		
<code>(and #T #F)</code>		
<code>(and (or #T #F) #T)</code>	←	List with and, (or #T #F), #T
<code>(lambda (x) (* x x))</code>	←	List with LAMBDA, (x), (* x x)
<code>( (lambda (x) (* x x) ) 7 )</code>		

- Expressions constructed using (nested) lists, such as in Scheme, are called “s-expressions”

# Scheme: Definitions

- We can give names to expressions

```
(define pi 3.14)  
(define two_pi (* 2 pi))
```

- This includes lambda expressions

```
( define f ( lambda ( x y ) ( * x y ) ) )
```

- After which one can write

```
f 3 5
```



# Scheme: Definitions

- There is a shorthand notation for function definitions:

```
( define f (lambda ( x y ) ( * x y ) ) )
```

→

```
( define ( f x y ) ( * x y ) )
```

- These definitions are 100% equivalent
- Scheme is *strict*: *all* arguments of a function call must be evaluated before the function is called, and all arguments must be specified
- These lines will give error messages for **both** definitions of **f**, whether or not lambda functions are used:

```
( f 1 )  
( f )
```

# Scheme: Control Flow

- If statements require a boolean predicate as first parameter

```
(define (factorial n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

- Predicates can be defined using functions, and have names ending with ?

```
(define (leap? year)
  (if (zero? (modulo year 400)) #T
      (if (zero? (modulo year 100)) #F
          (zero? (modulo year 4))
          )
      )
  )

(leap? 2016)
```

# Scheme: Control Flow

- This long code:

```
(define (leap? year)
  (if (zero? (modulo year 400)) #T
      (if (zero? (modulo year 100)) #F
          (zero? (modulo year 4))
        )
    )
)
```

- Can be shortened to:

```
(define (leap? year)
  (cond
    ((zero? (modulo year 400)) #T)
    ((zero? (modulo year 100)) #F)
    (else ( zero? (modulo year 4)))
  )
)
```

# Scheme: List Manipulation

- The four most important functions manipulating lists are:
  - `car L`: returns the first element of the list
  - `cdr L`: returns the tail of the list
  - `cons a L`: prepends `a` in front of the list `L`
  - `null? L`: test whether the list is empty
- Example: sum up all elements in a list:

```
(define (add L) (if (null? L) 0 (+ (car L) (add (cdr L)))))
```

```
(add '(1 3 4))
```

- Note: `'` indicates that the list after the `'` is *not* evaluated as a piece of Scheme code, but rather is stored as a list; without `'`, `(1 3 4)` would be seen as applying function `1` on arguments `3` and `4`, which yields an error

# Scheme: List Manipulation

- Given that programs are list, Scheme code can create new code and execute it using an *eval* function

```
(define (productify L)
  (cond
    ((null? L) ())
    ((list? (car L)) (cons (productify (car L)) (productify (cdr L))))
    ((eq? (car L) '+) (cons '* (productify (cdr L))))
    (else (cons (car L) (productify (cdr L)))))
  )
)
```

Treat + as a symbol, not as a function that needs to be evaluated

```
( eval ( productify '(+ 2 ( + 3 4 ) ) (the-environment) ) )
```

# Scheme: Exercise

- Implement a function *append* that takes two lists as argument and concatenates them

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))
  )
)
```

# Scheme: Map

- A common function in functional programming is the map function:

```
(define (map fun a_list)
  (if (null? a_list)
      ()
      (cons
        (fun (car a_list))
        (map fun (cdr a_list))
      )
  )
)
```

- Example application:

```
( map ( lambda (x) (* x x) ) '(1 2 3) )
```

# Scheme: Reduce

- A common function in functional programming is the reduce function:

```
(define (reduce fun a_list)
  (if (null? (cdr a_list))
      (car a_list)
      (fun (car a_list) (reduce fun (cdr a_list))
           )
  )
)
```

- Example application:

```
( reduce ( lambda (x y) (+ x y) ) '(1 2 3) )
```



# Scheme: Printing

- The `display` function writes its argument to the screen and returns *undefined*

```
(display ( + 3 4 ))
```

- How to call `display` in a function?

```
(define (add x y)
```

```
  (begin ←————— Starts a block of consecutive function calls
```

```
    (display x)
```

```
    (display y)
```

```
    (+ x y ) ←————— This is returned by the code block
```

```
  )
```

```
)
```

# Scheme: Properties

- Scheme with IO operations is not *pure*:
  - If the order of function call evaluation changes, the output of the code changes
  - This makes it hard to parallelize, optimize
- Scheme notation can be cumbersome
- Scheme does not have static types (int, float, ...)
- Scheme lacks concepts found in lambda calculus: eg. rewriting of this kind:  
$$(\lambda xy.xy)1 \equiv \lambda y.1y$$
- Up next: languages that address these weaknesses...

# Haskell

- Haskell 1.0 was defined in 1990 by a committee of researchers in functional programming
- Properties:
  - *Not strict*: not all arguments of a function call do need to be specified; this leads to *lazy* evaluation
  - *Pure*: functions do not have side effects; a different order of functional calls will never change the output; **Monads** are used to deal with side effects
  - *Infix notation*: infix notation can be used
  - *List notation*: a more convenient notation for dealing with lists
  - *Types*: variables and functions have strict types
  - *Pattern matching*: conditions can be expressed using patterns

# Haskell: Functions

- Type definition:

```
factorial :: Integer -> Integer
```

- Example using IF notation:

```
factorial n = if n > 0 then n * factorial (n-1) else 1
```

- Example using guard notation:

```
factorial n
  | n < 2      = 1
  | otherwise = n * factorial (n - 1)
```

# Haskell: Lambda Functions

- The following two statements are equivalent:

```
increment :: Integer -> Integer
```

```
increment n = n + 1
```

and

```
increment :: Integer -> Integer
```

```
increment = \n -> n + 1
```

# Haskell: Lists

- Prolog-like notation

`[ 1, 2, 4 ]`

- Pattern matching can be used to perform tests on lists, similar to Prolog
- Example:

```
add :: [Integer] -> Integer
```

```
add [] = 0
```

```
add (a:l) = a + add l
```

} Function definitions are evaluated top-to-bottom



Pattern that matches the head of the list and the tail of the list

# Haskell: Lists

- Lists can be created in a similar manner
- Example:

```
generate :: Integer -> [Integer]
```

```
generate 0 = []
```

```
generate n = n : generate (n - 1)
```

# Haskell: Currying

- The following code is not strict, as we call the function `incr` without specifying its two parameters; in that case, a function is returned

```
add :: Integer -> Integer -> Integer  
add x y = x + y
```

```
incr = add 1  
main = print (incr 2)
```

Can be read as:

`integer → ( integer → integer )`

If one integer is given as a parameter, the function returns a function in which `x` is substituted with the integer, i.e., here the result of

`add 1`

is a lambda function with argument `y` that returns `1 + y`



# Haskell: Lazy Evaluation

- Suppose we have the following program:

```
generate :: Integer -> [Integer]
generate n = n : generate (n + 1)
```

```
main = print (generate 0)
```

- This program will not terminate
- However, this program terminates:

```
take 0 L = []
take n (a:l) = a : take (n-1) l
```

```
main = print (take 2 ( generate 0) )
```

# Haskell: Lazy Evaluation

take 2 ( generate 0 )  
↓  
take 2 ( 0 : generate (0+1) )  
↓  
0 : take (2-1) ( generate (0+1) )  
↓  
0 : take 1 ( generate (0+1) )  
↓  
0 : take 1 ( (0+1) : (generate (0+1+1)) )  
↓  
0 : take 1 ( 1 : (generate (0+1+1)) )  
↓  
0 : 1 : take (1-1) (generate (0+1+1)) )  
↓  
0 : 1 : take 0 (generate (0+1+1))  
↓  
0 : 1 : []

```
generate n = n : generate (n + 1)  
  
take 0 L = []  
take n (a:l) = a : take (n-1) l  
  
main = print (take 2 ( generate 0 ) )
```

# Haskell: Lazy Evaluation

- Strictness can be forced

```
generate :: Integer -> [Integer]
```

```
generate n =
```

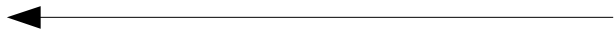
```
  let
```

```
    nplus = n + 1
```

```
  in
```

```
    seq nplus ( n : generate plus )
```

Syntax can be used to define a term that is used in the body of the function multiple times



Forces evaluation



```
main = print (generate 0)
```

# Haskell: Data Types

- Complex data types can be defined in Haskell using type & data constructors
- Example: binary trees

Type constructor      Data constructor

↓                      ↙

```
data Tree a = Tip | Node a (Tree a) (Tree a)
```

```
sumTree :: Num a => Tree a -> a
```

```
sumTree Tip = 0
```

```
sumTree (Node v a b) = v + sumTree a + sumTree b
```

As + can only be applied on numerals, sumTree is only defined for Num

```
main = print ( sumTree ( Node 3 ( Node 4 Tip Tip ) Tip ) )
```

# Haskell: IO Monad

- Haskell is pure: functions do not have side effects
- How to print?
- Conceptually, an *IO monad* can be thought of as a type

```
data IO a = RealWorld -> (RealWorld, a)
```

A conceptual data type  
that reflects the state that  
the computer is in

Over which certain operations are defined

- For instance:
  - The signature of print is:  

```
print :: String -> IO ()
```

“Empty”
  - The signature of readLn is:  

```
readLn :: IO String
```

# Haskell: IO Monad

- An example of the execution of this code:

```
main = print "Hello"
```

- The type of main is `main :: IO ()`
- The state of the world before the program is executed is `w`
- `(print "hello")` is a function with signature  
`RealWorld -> (RealWorld, ())`
- The Haskell runtime evaluates `main` by calling it with world `w` as parameter

```
main w
```

the result is a new world in which “Hello” is on the screen

# Haskell: IO Monad

- Two prints can be combined; in low-level code, as follows:

```
main :: RealWorld -> (RealWorld, ())
```

```
main world0 =  
  let  
    (world1,a) = print "Text1" world0  
    (world2,b) = print "Text2" world1  
  in (world2,())
```

- This code is cumbersome; let us define a new function to make this easier:

```
(>>) c d = \world0 =>  
  let (world1,a) = c world0  
  let (world2,b) = d world1  
  in (world2,())
```

# Haskell: IO Monad

- This code is cumbersome; let us define a new function to make this easier:

```
(>>) c d = \world0 =>  
    let (world1,a) = c world0  
    let (world2,b) = d world1  
in (( ), world2)
```

- Now we can write:

```
main :: RealWorld -> (RealWorld,())  
  
main = (>>) ( print "Text1" ) ( print "Text2" )
```

- Alternatively,

```
main = print "Text1" >> print "Text2"
```



# Haskell: IO Monad

- Alternatively,

```
main = print "Text1" >> print "Text2"
```

```
main = do  
    print "Text1"  
    print "Text2"
```

- Note:
  - in this notation a “world” object is implicitly passed from the one function call to the other function call
  - functions for which the signature does not include an IO object, can not perform IO

# Other Monads

- Alternative definitions of `>>`, for different Monads, can serve other purposes
- For example, the Maybe monad:

```
data Maybe t = Just t | Nothing
```

```
div :: Float -> Float -> Maybe Float
```

```
div x 0 = Nothing
```

```
div x y = Just (x/y)
```

```
docalc :: Maybe Float -> Maybe Float -> Maybe Float -> Maybe Float
```

```
docalc a b c = do
```

```
  x <- a
```

```
  y <- b
```

```
  z <- c
```

```
  t <- div x y
```

```
  div x t
```

# Other Functional Programming Languages

- ML, Miranda: predecessors of Haskell
- F#, Scala: multi-paradigm languages that include functional primitives

```
def addB(x:Int): Int => ( Int => Int ) = ( y => ( z => x + y + z ) )

val a = addB(1)(2)(3)
println(a)

lazy val x = { print ("foo") ; 10 }
print ("bar")
print (x)
print (x)
```

Factorie,  
Figaro,  
Oscar

Scala

```
let rec fact x =
    if x < 1 then 1
    else x * fact (x - 1)

Console.WriteLine(fact 6)
```

Infer.NET

F#

# Overview Lectures

- 8 march: End of functional programming

1) 15 march: Dyna

2) 22 march: Gringo / Clasp

3) 29 march: ECLiPSe, FO(.)

4) 5 april: Gecode, MiniZinc

5) 12 april: Markov Logic

6) 19 april: PRISM, Problog

7) 26 april: Church

8) 3 may: Factorie

9) 10 may: Tensorflow, Theano

# Overview Lectures

- 8 march: End of functional programming
  - 15 march: No lecture
- |                               |  |
|-------------------------------|--|
| 1) 22 march: Dyna             | Mats Derk, Renuka Ramgolam, -                              |
| 2) 29 march: ECLiPSe, FO(.)   | (1), (2), (3)  |
| 3) 5 april: Gecode, MiniZinc  | Jelco Burger, Anne Hommelberg, (4)                         |
| 4) 12 april: Markov Logic     | Gogou Evangelia, Stellios Paraschiakos, Nick van den Bosch |
| 5) 19 april: Markov Logic     | Mark Post, +2  |
| 6) 26 april: PRISM, Problog   | Hanjo Boekhout, (5), -                                     |
| 7) 3 may: Church, Factorie    | Raymond Parag, (6), (7)                                    |
| 8) 10 may: Tensorflow, Theano | Arthur van Rooijen, Lau Bannenberg, -                      |

# Content Presentation

- Motivation: discuss applications of the programming system
- Examples: provide one or more concrete illustrative examples of programs in the programming system, showing code
- Concepts: discuss on which fundamental concepts the programming system is based
- Execution: discuss how statements in the programming system are executed
- Results: show some experimental results reported in the literature (run time, quality, ...)