**Module 3**

**Declarative Programming Paradigm: Functional Programming**

| **Content** | * Introduction to Lambda Calculus * Functional Programming Concepts, * Evaluation order * Higher order functions * I/O-Streams and Monads. |
| --- | --- |

**Introduction to Lambda Calculus**

* Lambda calculus was introduced by Alonzo Church in the 1930s and is, essentially, a way of expressing computation through the use of functions
  + Church’s model of computing is called the *lambda calculus*
  + Lambda calculus was the inspiration for functional programming
  + one uses it to compute by substituting parameters into expressions, just as one computes in a high-level functional program by passing arguments to functions
* The λ-calculus can be called the smallest universal programming language in the world.
* The λ-calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to Turing machines.
* However, the λ-calculus emphasizes the use of symbolic transformation rules and does not care about the actual machine implementation. It is an approach more related to software than to hardware.

(/x.x+1)

**Lambda Calculus’ Syntax**

Everything in Lambda Calculus is an [expression](https://www.quora.com/Whats-the-difference-between-a-statement-and-an-expression-in-Python-Why-is-print-%E2%80%98hi%E2%80%99-a-statement-while-other-functions-are-expressions), which means that everything must evaluate to a value.

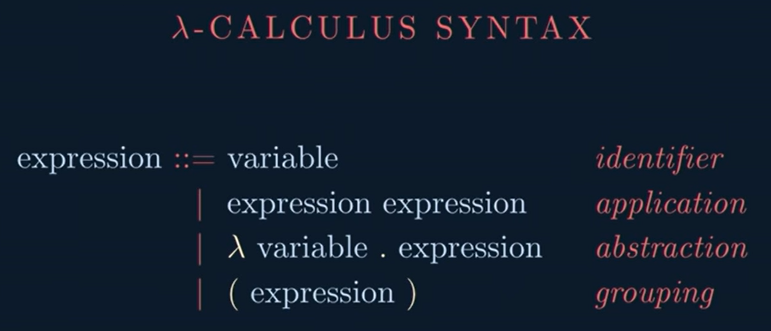
Here's an example of a simple lambda expression that defines the "plus one" function:

λx.x+1

λx.x2.

function of one argument, whose formal parameter is named 'x'. The function body is: "x+1"

There are, however, four different forms of expressions (which I’ll call **E**). An E can be either:





* ID - **Identifier**
* λID.E - **Abstraction**.
* E E - **Application**
* (E) - **Grouping**

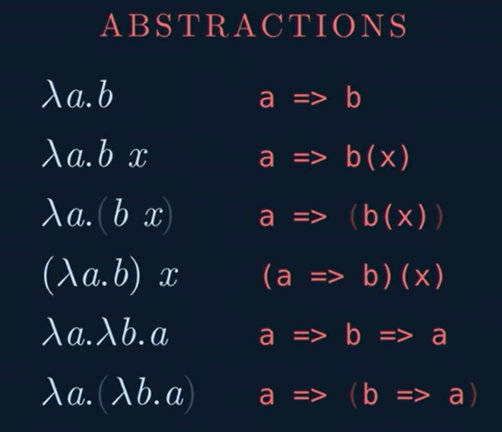
**1)Identifiers** are simply that: identifiers. They **identify certain values by giving them a “name”**, just like our modern programming languages do.

**const** x = 10

x *// Identifier equivalent*

**2) Abstractions** are perhaps the most iconic kind of lambda expression, they **define what we call functions** or, more adequately, lambdas: which are just anonymous functions.

(x) => x \* x *// Abstraction equivalent*



**3)Applications denote function invocation**. If you have a function A you can say you’re calling it with B by writing A B.

**const** a = (x) => x \* x

**const** b = 10

a(b)

const a sum(2,3)=>2+3

square(a)

4) **Grouping exists for the sake of disambiguation**. We use these parentheses around the expressions we want to group to make it clear which ones of them we want to apply to each other.(2+3)\*5

2+3\*5

## Evaluating Lambda Calculus

Here's an example of a simple lambda expression that defines the "plus one" function:

λx.x+1

λx.x2.

function of one argument, whose formal parameter is named 'x'. The function body is: "x+1"

## Bound and Free Variables

Free variables are variables defined outside of the function’s scope**. They’re**the opposite of bound variables**, which are variables that only exist inside of a function’s scope.**

Now, the same examples above, but in using Lambda Calculus’ syntax:

1. λx. x - x is a bound variable
2. λx. x y - x is a bound variable and y is a free variable
3. λx. y z - y and z are both free variables
4. λx. λy. x(y) - Both x and y are bound variables

## Currying

In Lambda Calculus, each abstraction cannot take more than one argument, which means that we have to do [currying](https://en.wikipedia.org/wiki/Currying) in order to be able to work with multiple variables in the body of a function.

* succ 7 \* 8

write **succ 7 \* 10** means it would get the successor of 7, which would then be multiplied by 8.

* succ (7 \* 8)

**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
* Necessary features, many of which are missing in some imperative languages
  + - 1st class and high-order functions
    - serious polymorphism
    - powerful list facilities
    - structured function returns
    - fully general aggregates
    - garbage collection
* So how do you get anything done in a functional language?
  + - Recursion (especially tail recursion) takes the place of iteration
    - In general, you can get the effect of a series of assignments  
       x := 0 ...  
       x := expr1 ...  
       x := expr2 ...  
      from f3(f2(f1(0))), where each f expects the value of x as an argument, f1 returns expr1, and f2 returns expr2

**A Bit of Haskell**

* 2 + 3
* 2 – 3
* 2 \* 3
* succ 7 \* 8
* succ (7 \* 8)
* min 4 9
* mod 3 2
* reverse “hello”

min :: **ord a=>**a->a->a

**What is a Type?**

**A type is a name for a collection of related values. For example, in Haskell the basic type**

**Bool-contain 2 logical value**

**True false**

**Type Errors:Applying a function to one or more arguments of the wrong type is called a type error.**

**Example:** **> 1 + False**

**error ...**

**Types in Haskell**

* **If evaluating an expression e would produce a value of type t, then e has type t, written**

**e::t**

* **Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.**
* **All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.**
* **In GHCi, the :type command calculates the type of an expression, without evaluating it:**

**Basic Types in Haskell**

**Tuple Types**

**A tuple is a sequence of values of different types:**

**((“Abc”,10,”SE5”)(“xyz”,11,”SE5”))**

**(False,’a’,True) :: (Bool,Char,Bool)**

fst takes a pair and returns its first component.

ghci> fst ("Wow", False)

"Wow"

**Function Types**

**A function is a mapping from values of one type to values of another type:**

**not :: Bool → Bool**

**:t not**

## An intro to lists

In Haskell, lists are a **homogenous** data structure. It stores several elements of the same type. That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.

**List**

* **[]**
* **[[]]**
* **[1,2,3,4]**
* **[“a”,”b”,”c”,”d”]**
* **[‘a’,’b’,’c’,’d’]**

**sum::Int a=> a->a->a //function declaration**

**or**

**sum::Int->Int->Int**

**A common task is putting two lists together. This is done by using the ++ operator**.

**[1,2,3,4] ++ [9,10,11,12]**

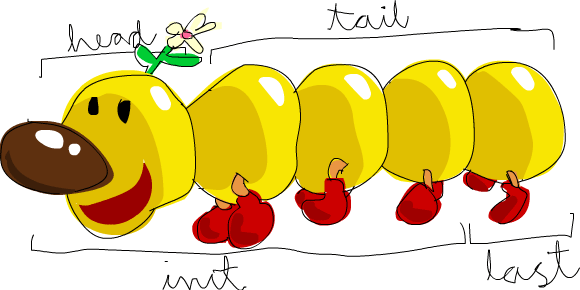
**ghci> "hello" ++ " " ++ "world"**

**"hello world"**

If you want to get an element out of a list by index, use **!!**. The indices start at 0.

**[9.4,33.2,96.2,11.2,23.25] !! 1**

**33.2**



**Head()**

**Tail()**

**Init()**

**Last()**

**Length**

**Null()**

**Reverse()**

**Take()**

**Drop()**

**Minimum**

**Maximum**

**Range**

* **[‘a’..’z’]**
* **[’J’..’M’]**
* **[1..10]**

**[0,2..10]**

* **Cycle**
* **Repeat**
* **Example**

**• x:xs**

**• xs = [1,2,3,4]**

**xs!!2**

**0:xs**

**[0,1,2,3,4]**

**xs**

**• ys = [5,6,7,8]**

**• zs = xs ++ ys**

**• 0:xs**

**• ys ++ [5]**

**• xs !! 3**

**• head xs**

**• tail xs**

**• init xs**

**• last xs**

**• take 3 [1,2,3,4,5]**

**• drop 3 xs**

**• null [1,2,3,4,5]**

**• minimum [1,2,3,4]**

**• maximum [1,2,3,4]**

**• sum xs**

**:t not**

**not:Int->Bool**

**not 7**

**• product xs**

**• length xs**

**• 6 `elem` [3,4,5,6]**

**elem 6 [3,4,5,6]**

list comprehension

A basic comprehension for a set that contains the first ten even natural numbers is

set notation.

The part before the pipe is called the output function,

**x** is the variable,

**N** is the input set

and **x <= 10** is the predicate.

**[ x+3 | x<- [1..20] ,even x]**

**x=[1,2,3..20],odd x**

**1,3,5,7..19**

**4,6,**

**x=[1,2..20]**

**x=[2,4,6,8,]**

**5,7,..**

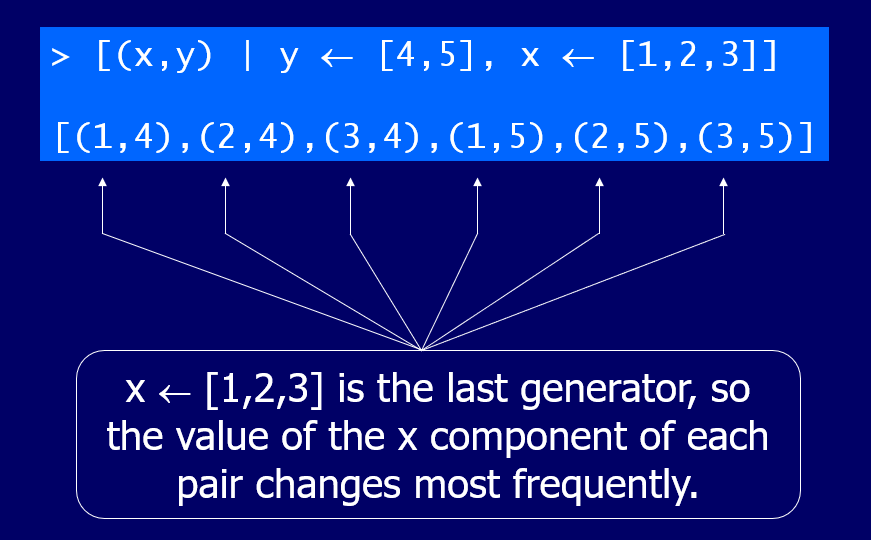
**2,4,6,8,10**

[ x | x<- [1..20],x\*2<20 ]

**Guards:** **List comprehensions can use guards to restrict the values produced by earlier generators.**

**[x\*2 | x <- [1..10],even x]**

**Note:** **The expression x ← [1..10] is called a generator, as it states how to generate values for x.**



**[ if x < 10 then "BOOM!" else "BANG!" | x <- [7..13], odd x]**

**x=7,8,9,10,11,12,13**

**7,9,11,13**

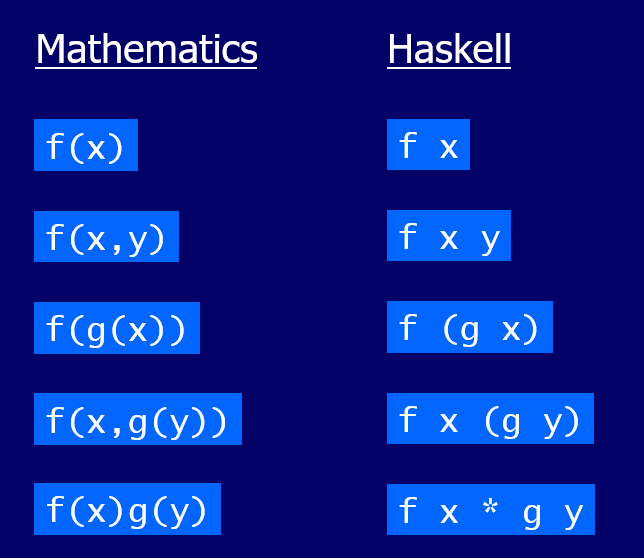
**boomBangs xs =**

**[ if x < 10 then "BOOM!" else "BANG!" | x <- [7..13], odd x]**

**ghci> boomBangs [7..13]**

**ghci> [ x | x <- [50..100], x `mod` 7 == 3]**

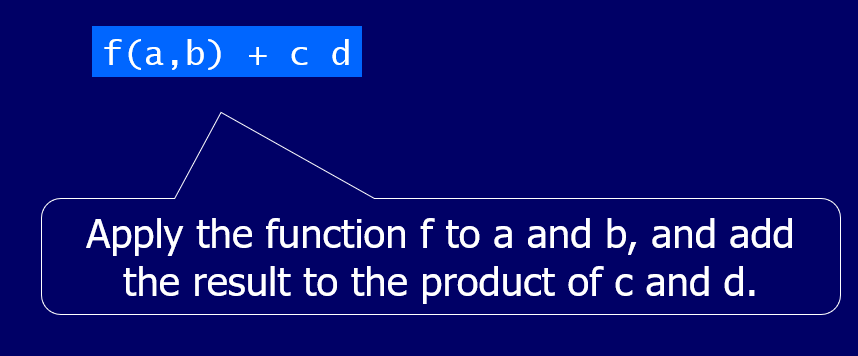
**Function Application**



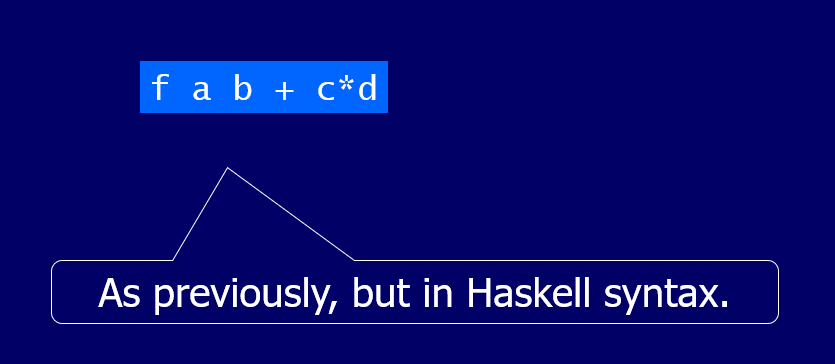
take 5(+2)[1,2,3,4,5,6]6

succ (6 \*7)

**In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.**



**In Haskell, function application is denoted using space, and multiplication is denoted using \*.**



**Example**

**Working for writing user define function**

**Function**

**add x y = x + y**

**Expression Evaluation:** **Expressions are evaluated by a stepwise process of applying functions to their arguments.**

**((x+y)+z)\*z1**

**Example :**

**fac::Int->Int**

**fac 0 = 1 //base case**

**fac n =n\* fac n-1 // inductive case**

**fact :: Int → Int**

**fact n = product [1..n]**

**fact 3**

**product[1..3]**

**product[1,2,3]**

**1\*2\*3**

**6**

**fact 5**

**5\*4\*3\*2\*1**

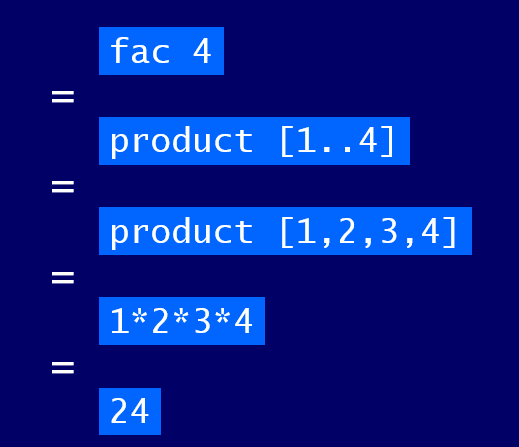
**fac 5**

**5\*4\*3\*2\*1**

**prelude>:l a.hs**

**main>fact 5**

**fac maps any integer n to the product of the integers between 1 and n**



**Recursive Functions:** **In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.**

**fac::Integer->Integer**

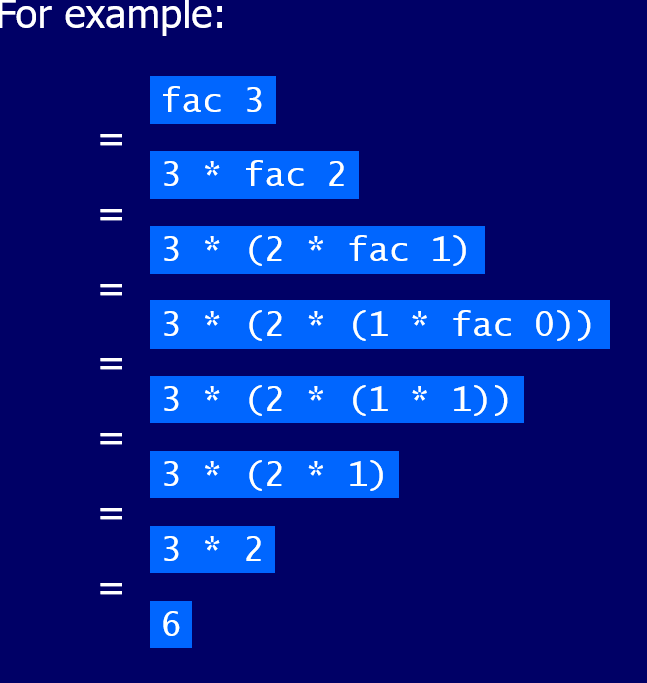
**fac 0 = 1**

**fac n = n \* fac (n-1)**

**ghci>:l fac.hs**

**main>fac 5**

**fac maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.**



**Why is Recursion Useful?**

1. Some functions, such as factorial, are simpler to define in terms of other functions.
2. As we shall see, however, many functions can naturally be defined in terms of themselves.
3. Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

product [1,2,3]

1 \* product[2,3]

1\*2 (product [3])

1\*2\*3(product[])

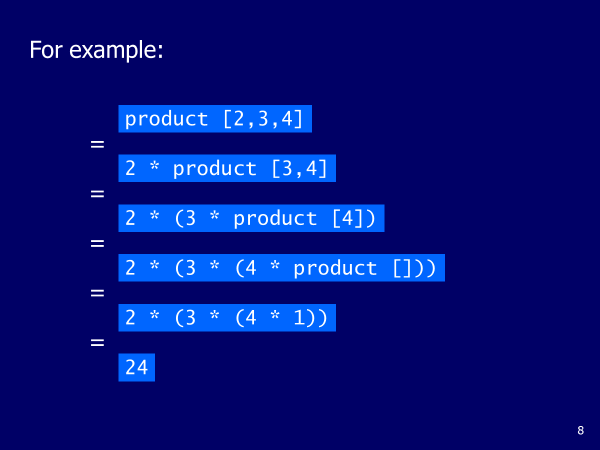
1\*2\*3(1)

[1,2,3]--->6

product::[Num]->Num

product [] = 1 // base case

product (n:ns) = n \* product ns



**Type variable:**

* Haskell has a static type system
* The type of every expression is known at compile time, which leads to safer code.
* Now we'll use GHCI to examine the types of some expressions. We'll do that by using the **:t** command which, followed by any valid expression, tells us its type.
* Functions also have types. When writing our own functions, we can choose to give them an explicit type declaration.Example

removeNonUppercase :: [Char] -> [Char]

addThree :: Int -> Int -> Int -> Int

factorial :: Integer -> Integer

**charName :: Char -> String**

**charName 'a' = "manya"**

**charName 'b' = " nida"**

**[1..]**

take 5 [1..]

1

take 4[1,2..]

1,2 take 3 [1,2,3..]

take 3 [1..]

1,2, take [1,2,3..]

sieve :: [Int] → [Int]

sieve (p:xs) =

p : sieve [x | x ← xs, mod x p /= 0

]

sieve :: [Int] → [Int]

sieve (p:xs) =

p : sieve [x | x ← xs, mod x p /= 0]