Lambda-Calculus

06016415 Functional Programming

Outline

- Expression
- Free and Bound Variables
- Reduction
  - α Equivalence (alpha-conversion)
  - β Reduction (beta-reduction)
  - η-reduction (eta-reduction)
- Application
  - Athematic
  - Logic / Boolean
  - Relational Operators

Imperative style VS Declarative style (functional)

Imperative style

```
def factorial_iter(n:Int) : Int = {
  var fact = 1
  for(i<-1 until n+1) {
      fact = fact * i
    }
    fact
  }
```

Declarative style

```
def factorial_recursive(n:Int) : Int =
  if(n==0) 1 else n * factorial_recursive(n-1)
```

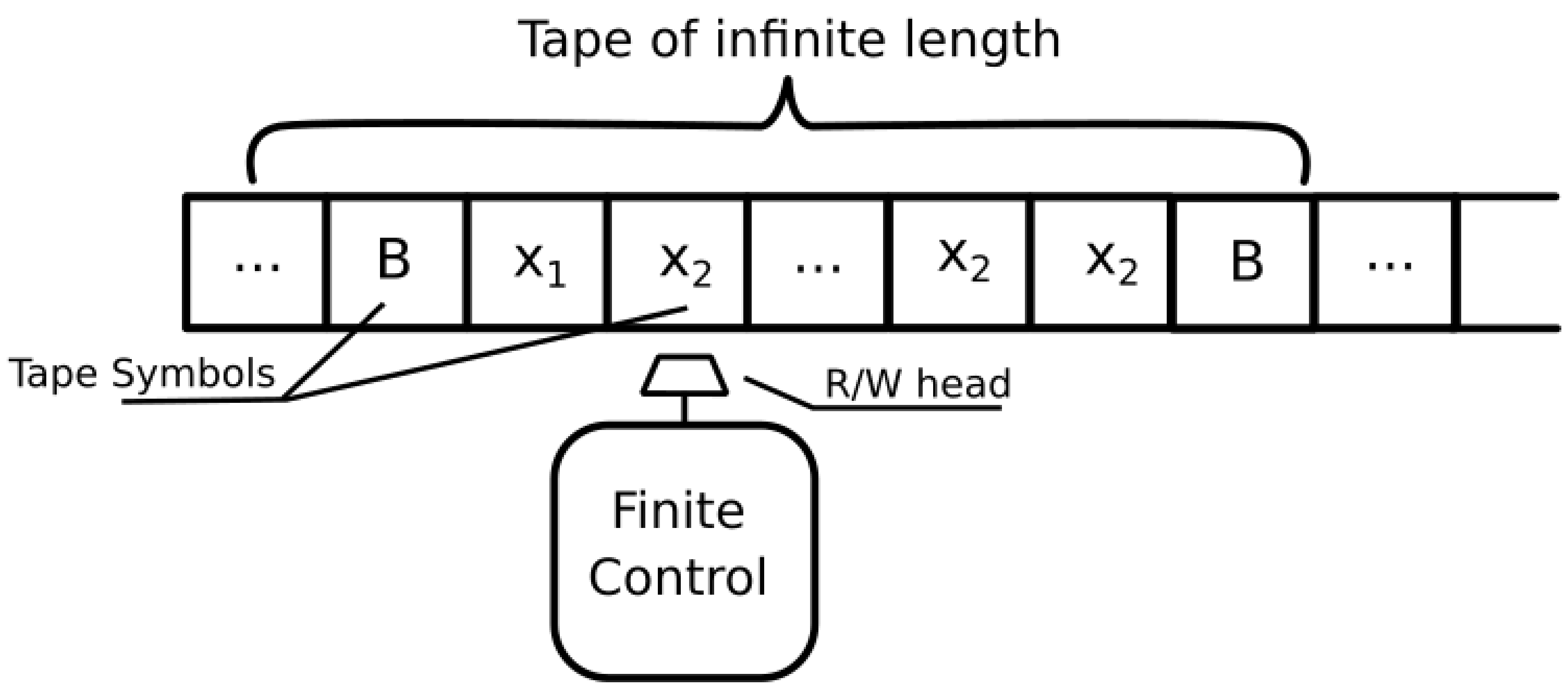
focus on result

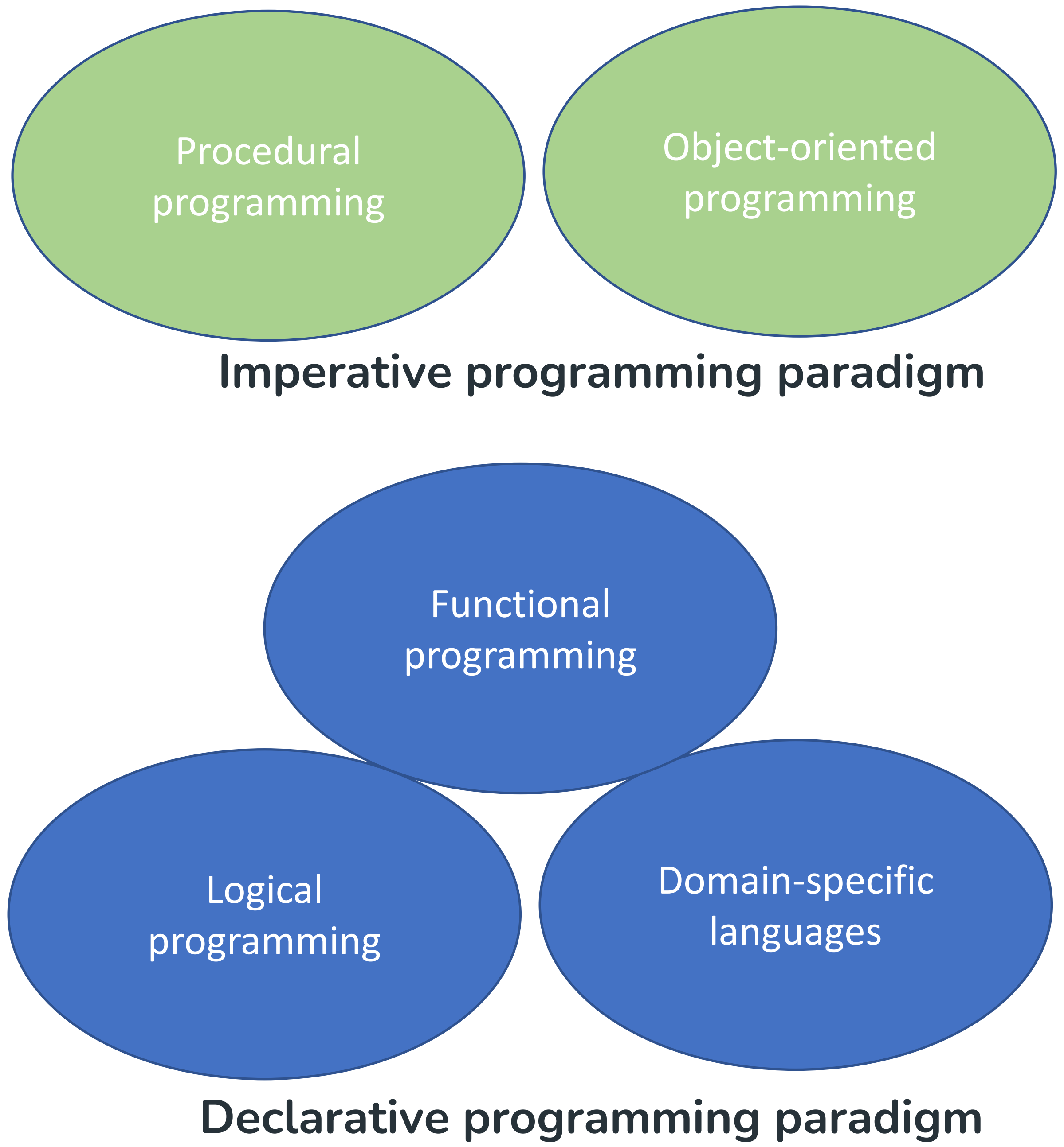
Result

```
scala> (0 to 5).foreach(i => println(s"$i: ${factorial_recursive(i)}"))
0: 1
1: 1
2: 2
3: 6
4: 24
5: 120

scala> (0 to 5).foreach(i => println(s"$i: ${factorial_iter(i)}"))
0: 1
1: 1
2: 2
3: 6
4: 24
5: 120
```

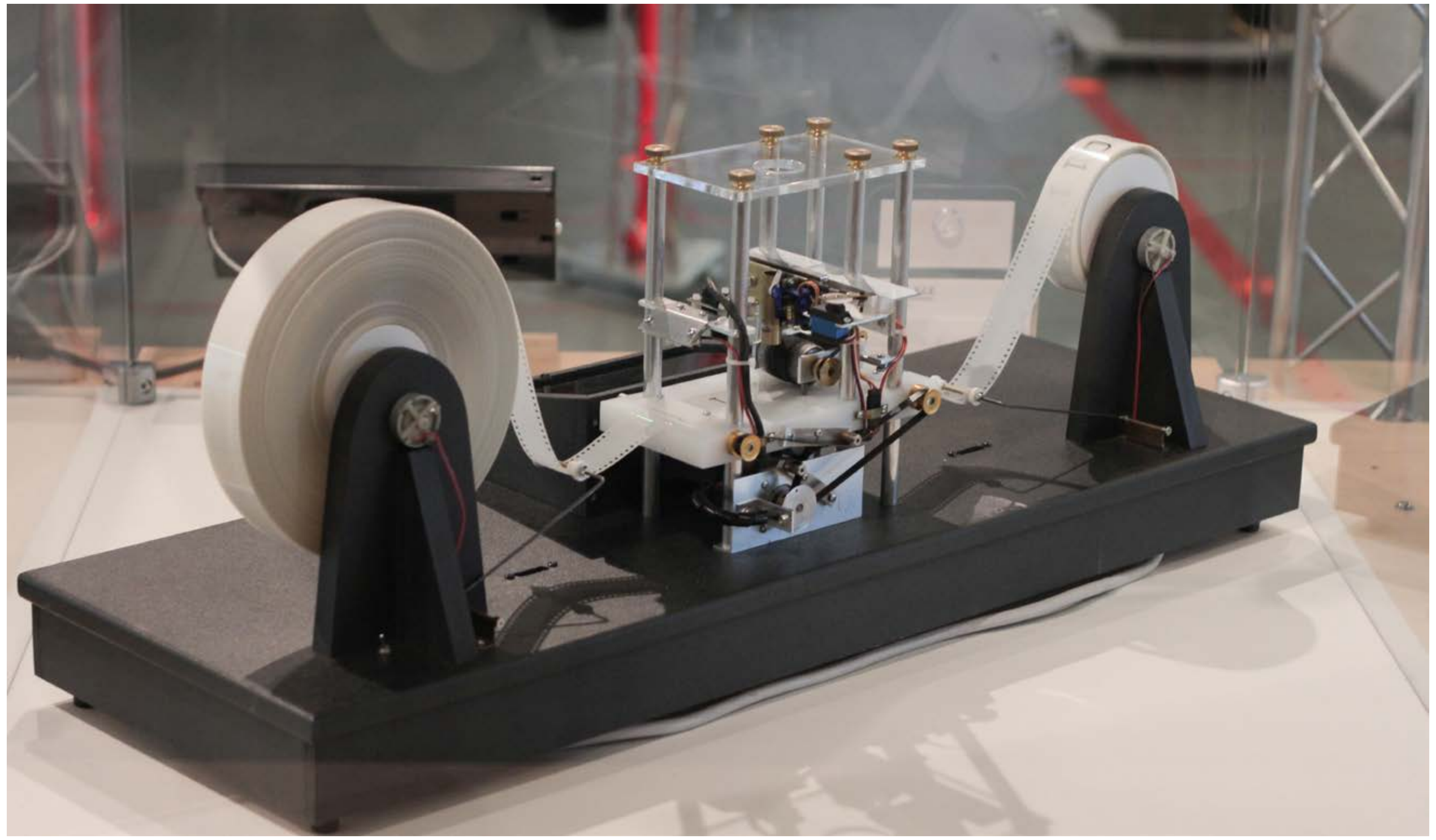
- **Functional Programming (FP)** languages are based on the lambda-calculus.





Lambda calculus (λ-calculus), originally created by Alonzo Church(1930s), is the world's smallest programming language.

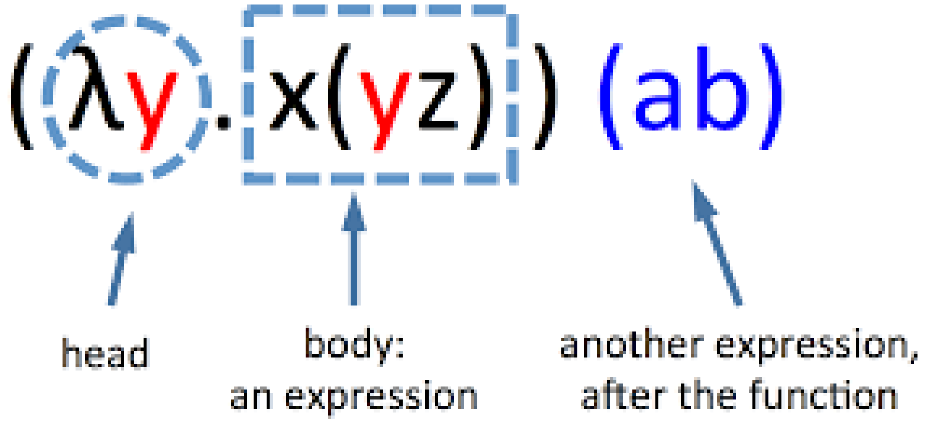
Despite not having numbers, strings, booleans, or any non-function datatype, lambda calculus can be used to represent any Turing Machine!



A physical Turing machine model. A true Turing machine would have unlimited tape on both sides; however, physical models can only have a finite amount of tape.

Lambda calculus is composed of 3 elements:
variables, functions, and applications.

Name	Syntax	Example	Explanation
Variable	<name>	x	a variable named "x"
Function	λ<parameters>.<body>	λx.x	a function with parameter "x" and body "x"
Application	<function><variable or function>	(λx.x)a	calling the function "λx.x" with argument "a"



λx.x equivalent to **f(x) = x**

Abstraction - λx.M

- λx . λy . x + y ≡ λxyz.x+y
- λu . λt . λa . ut + $(1/2)\, at^2$

Application - (M1 M2)

"Apply expression M1 with expression M2"

- f(x) = x * 10 -------> λx . x * 10 5

function application

- parking_fee(hour) = hour * 10 -------> λx . x * 10 5

Function Declaration

focus on result

Abstract: $\lambda x.M$

f(x) = x * 10 -------> $\lambda x . x * 10$

parking_fee(hour) = hour * 10 -------> $\lambda x . x * 10$

issarapong@it.kmitl.ac.th

In the function

- **λx.x**, “x” is called a bound variable.
  - because it is both in the body of the function and a parameter.

- **λx.y**, “y” is called a free variable.
  - because it is never declared before hand.

λx . λy . x + y

λu . λt . λa . ut + (1/2)at^2

$$f(x, c) = \frac{x^2}{c \cdot x}$$

Call Function

Application: (M1 M2)

"***Apply*** expression **M1** with expression **M2**“

(λx.x*10) 5   or (λx.x*10)a^b+100

```
//declare function f
function f(x){
    return x * 10;
}
//call f
f(5);

(function(x){
    return x * 10;
})(5);
```

	+=1
Athematic	+=1 -=1 • Successor / Predecessor • Addition • Subtraction • Multiplication • Exponentiation
Logic / Boolean	• NOT • AND • OR
Relational Operator	• IsZero • Less Than or Equal to (LEQ) • Equality

- **Successor:** To obtain the successor of n,

$$SUCC\ \overline{n} = \overline{n+1}$$

A successor function, which takes a Church numeral n and returns n + 1 by adding another application of f, the function 'f' is applied n' times on 'x'.

- **Predecessor**: To obtain the predecessor of n,

$$PRED\ \overline{n} = \overline{n-1}$$

The predecessor function defined by PRED n = n − 1 for a positive integer n and PRED 0 = 0 is considerably more difficult. The formula

- **Addition :**

like times

PLUS = λmn.(m SUCC n)

plus --> use (succ n) for m times

- **Multiplication**

MULT = λmn.m (PLUS n) 0

Since multiplying m and n is the same as repeating the add n function m times and then applying it to zero.

- **Exponentiation**

$EXP\ \overline{m}\ \overline{n} = \overline{m}^{\overline{n}}$, EXP = λb.e e b

Exponentiation has a rather simple rendering in Church numerals

- **Subtraction**:

SUB = λmn.(m PRED n)

If-Then-Else

ifthenelse=λc.λx.λy.(cxy)

ifthenelse T ab =a

if TRUE return a
if False return b

ifthenelse F ab =b

TRUE = λx . λy . x
FALSE = λx . λy . y

issarapong@it.kmitl.ac.th

NOT = λx . (x FALSE TRUE)

NOT Truth Table

A	B
0	1
1	0

Example

x y.x

NOT TRUE

where we have to calculate

λx . x FALSE TRUE

....

put TRUE into x
then it will be TRUE FALSE TRUE
TFT equal to F

ex. NOT FALSE
FALSE(FALSE TRUE)
F(FT) equal to F(F)
F(F) equal to T

so NOT FALSE is TRUE

AND = λx . λy . (x y FALSE)

Example

AND x y

AND TRUE y

(T y F) or prof wrote T(y, F)

....

AND FALSE y

(F y T)

...

AND Truth Table

(in equation) x y

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

AND = λx . λy . (x y FALSE)

and' x y = if x then y else False

OR = λx . λy . (x TRUE y)

or' x y = if x then True else y

Example

AND x y

AND TRUE y

....

AND FALSE y

...

AND Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

- **Is zero?**

  isZero =λn.n FALSE not FALSE

- **Less Than**

  LEQ=λmn.**isZero**(sub mn)

- **Equality**

  EQ=λmn.(AND(**LEQ** mn)(**LEQ** nm))

Assignments

- **Individual work (5 points)**
- **Choose 1 or more functions**
- **Describe**
  - **In functional abstraction (lambda syntax)**
  - **Create a program in Scala**
- **Summarize the idea of Reduction**

Submit .docx or word file only!

Level	Weight	Lists
Basic	**1**	• Successor • Predecessor • NOT
Medium	**2**	• Addition • Subtraction • Multiplication • AND • OR • IsEven • IsOdd • Square
Advance	**3**	• Exponentiation • IsZero • Less Than or Equal to (LEQ) • Equality

Lambda is defined by how expressions can be reduced.

Reduction	
α-conversion *(Equivalent, Renaming)*	• Changing bound variables. • Two expressions are α-equivalent, if they can be α-converted into the same expression.
β-reduction	• Applying functions to their arguments.
η-reduction	• Captures a notion of extensionality. • Judge objects to be equal if they have the same external properties.

α equivalence states that any bound variable is a placeholder and can be replaced (renamed) with a different variable, provided there are no clashes.

Example

- $\lambda x.x$ and $\lambda y.y$ are α equivalent
- $\lambda x.(\lambda x.x)$:
  - is α equivalent to $\lambda y.(\lambda x.x)$
  - But not to $\lambda y.(\lambda x.y)$

In programming languages with static scope, α-conversion can be used to make name resolution simpler by ensuring that no variable name masks a name in a containing scope (α-renaming).

β-reduction captures the idea of function application. It tells us how <u>simplifications</u> of abstractions work

Example

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

$$\mathbf{(\lambda x.x)y} \to^{\beta} \boldsymbol{y}$$

$$(\lambda x.x)(\lambda y.y) \to^{\beta} \ ???$$

In programming languages, β-reduction can be used to make the process of calculating a result from the application of a function to an expression.

Expressions can be thought of as programs in the language of lambda calculus.

```
x = 5
y = 2 – x

ƒ(x, y) = x + (y × 3)
```

```
ƒ(x, y)
= ƒ(x, 2 - x) // replace y
= ƒ(5, 2 - 5) // replace x
= ƒ(5, -3)    // ƒ(x, y)
= 5 + (-3 × 3)
= 5 + (-9)
= -4
```

```
y = 2x+3

λx.2x+3          // Function Abstraction

λx.2x+3 a²

2a²+3            //beta-reduction
```

$(\lambda n.n*2)7 \rightarrow^{\beta}$???

η-reduction captures a notion of extensionality.

Principle of Extensionality: Two functions are the same if and only if they give the same result for all arguments (the same mapping).

Example

$$\lambda x.(Mx) \rightarrow^{\eta} M$$

If x is a variable and does not appear free in M

In programming languages, η-reduction is to drop an abstraction over a function to simplify it.