If syntax tree can be made, then it is a valid lambda term. Otherwise, it is invalid.

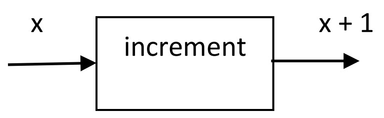
Lecture 2 **24.1.2025**

Today's agenda:

Foundations of functional programming:

View of a function: a function transforms an input to an output. It just says what it does and not how it does.

Example 1:



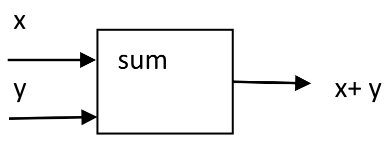
f: x → x + 1 (mapping)

λx. x + 1

Fig 1 blackbox

we are not allowed to see what happens inside the blackbox. there is no hidden state.

Example 2:



f: (x,y) → x+y

Fig 2

(x,y) means the two inputs are fed together

In the foll., the terms input, arguments, and parameters have the same meaning. Output and result have the same meaning.

anonymous form of a function

--**function name is irrelevant** (anonymous form) $f(x,y) \equiv g(x,y)$

--**names of arguments of a function are irrelevant** $f(x,y) \equiv f(z1,z2)$

Partial application of a function: What happens when *sum* is applied to **one** argument, say, 2?

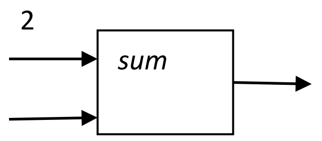


Fig 3

After 2 is consumed, there is still one input arrow to the function. Pictorially, we have

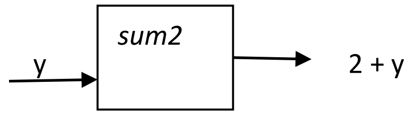
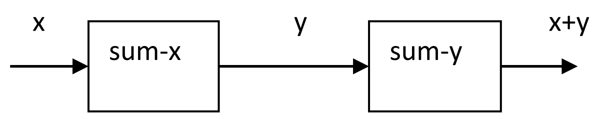


Fig 4

Currying: is a technique for transforming a function of n arguments to a chain of n functions where each function has exactly one argument. It uses the concept of **partial application of a function**. i.e., it applies one argument at a time.

The *sum* function (which is uncurried) after applying Currying is given as



function applied to x,y

Fig 5 λx. λy. x + y

We will write f(x,y) as f x y as in sin θ [to be read as application of sine to theta] or log x

Example:

```
int sum (int x, int y)                this code corresponds to fig 2
{
        return x + y;

}
```

Let us just specify what this function is doing. So we get

sum x y = x + y

Can be rewritten as: f x y = x + y or f u v = u + v

In anonymous mode we can write this as:

(x y) → x + y // function of 2-argument (1)

Now we need to understand how currying works: **partial application**

x is passed as input and output is a function which takes one argument.

Can be rewritten as:

x → (y → x + y) // function of 1-argument (2)

this is a function of 1-argument with argument x and it returns another function [shown in red] In this way, any function of more than 2 arguments can be transformed to several functions of 1-argument each.

such a function (2) is an example of a higher-order function (HOF).

Functions which have other functions as arguments or results are called HOFs.

See carefully the difference between (1) and (2). In (2) the body of the function is also a function of 1-argument. In (1) or in the code, we must supply the actual parameters of both the arguments together.

Let us now see how to use such functions:

Now if we supply 3 for the argument x , [in (2)] $x \rightarrow (y \rightarrow x + y)$

We get $y \rightarrow 3 + y$

Similarly if we supply 4 for the argument y,

We get 3 + 4

A special feature of FP is partial application of a function, which is not allowed in Imperative PLs.

Thus functions are first-class citizens in FP (LC). They can be assigned to variables, passed as parameters to functions, can also be returned by a function. A function in FP is a HOF, in general.

We will study the design of a generic Functional PL called Lambda Calculus.

LC has just three constructs (these are also integral to any PL):

--1. variables
--2. function creation (definition)
--3. Function application

```
Eg, int sum(int x, int y)
    {
          return  x + y;
    }


main(){ ….. sum(5,3);….}
```

-

Recall that procedures can be expressed in anonymous mode.

So we can write the above procedure as: $\lambda x.\ \lambda y.\ \ x + y$ [fig 2]

Example of another function: $\lambda x.\ \ x + 1$ [fig 1]

To be read as:

λ denotes function abstraction (definition) $\longrightarrow$ lambda starts a new abstraction

λx means a function of one argument (x)—variable-(name)

after the **dot** the body of the function is written—what the function is supposed to do.

the previous example can now be read as a function of two variables (x,y) and the body is x + y

Now we want to work with the functions; pass values to the variables or names or placeholders. This is called **function application**.

This is done as: (λx. x*x) 5

the above says, apply the function where the placeholder takes the value 5.

this means that we evaluate the function with x=5 so we get 5*5 = 25.

similarly for the other example we have

$$((\ \underline{\cancel{\lambda x}.\ \lambda y.\ x*x + y*y}\qquad)\ 3)\ 4$$

$$= \qquad (\cancel{\lambda y}.\ 3*3 + y*y)\ 4$$

$$= \qquad 3*3 + 4*4$$

Now λx. λy. x*x + y*y is the same as λy. λx. x*x + y*y

Let us work out the above

$$((\cancel{\lambda y}.\ \lambda x.\ x*x + y*y)\ 4)\ 3$$

$$= \quad (\cancel{\lambda x}.\ x*x + 4*4)\ 3 \quad = \quad 3*3 + 4*4$$

More examples: what will these functions return?

1. λx. x (λx. x) n
2. λx. y (λx. y) n
3. λx. λy. x ((λx. λy. x) n) m)

Summary of key concepts: HOF, partial application of a function, function abstraction, function application.
==

So we have only three constructs:

--variables

--function abstraction

--function application [**highest precedence**]

highest precedence will be of function application.
Function application is left associative.
Lambda will extend as much right as possible

Pure Lambda calculus: there are no constant symbols (0,1,2,....), there are no operators like +, *,..

This means that λx. x*x λx. x + 1 are not valid terms of pure LC.

But there exist pure lambda terms for 0,1,2,..... and +, *, ..., We shall develop these terms later.

Syntax of Pure Lambda calculus:

M ::= x	variables		
| (λ x. M)	abstraction		
| (M N)	application		$(M_1\ M_2)$

instead of (M N) we can write $(M_1\ M_2)$.

Syntactic Convention:

f g h is equiv to (f g) h since function application is **left associative**.

$\lambda\, x. M\ N$ is equivalent to $\lambda x. (M\ N)$

$\lambda x. \lambda y. \lambda z. M$ is equivalent to $\lambda\, xyz. M$

How is a function application done?

(λ x. 5x + 2) 2

= 5.2 + 2 = 10 + 2 = 12

(λ x. M) N means **we look for appropriate places** in M for x that can be replaced or substituted by N

If x does not occur in M, the output of the above is M, eg, λ x. 0 = 0 for any N

By appropriate place we mean that x is not captured by another lambda term in M.

that is, x **occurs free in** M.

Eg, (λ x. λ x. x + 1) 1 how would this be evaluated?

Here M = λ x. x +1

The outer x is captured by the inner λ x, this means that the outer x is not visible inside the inner λ x, so we can rewrite it as λ x. λ y. y + 1, so M does not contain x, so the answer would be M i.e., λ y. y+1 which by replacement is λ x. x +1

End of lecture