Course Outcomes

11/01/19

CO1 : Be familiar with functional languages (IMP 20-40)

CO2 : Be exposed to logic programming (20)

CO3 : Lambda Calculus (how recursion supported, how OO lang ~~practices~~ implemented in Lambda calculus) (20)

CO4 : Be able to map design concepts of Lambda calculus to Prog^n languages

CO5 : Design issues of OOP

CO6 : Apply proper programming paradigm

{ = may be merged together i.e. single ques comprising 2 CO

Books:

Louden, Lamberten → Principles of Prog lang

Theoritical part (abstraction, prog lang efficiency, theory of func prog, logic prog)

Scanned by CamScanner

4 Programming Paradigms (style)

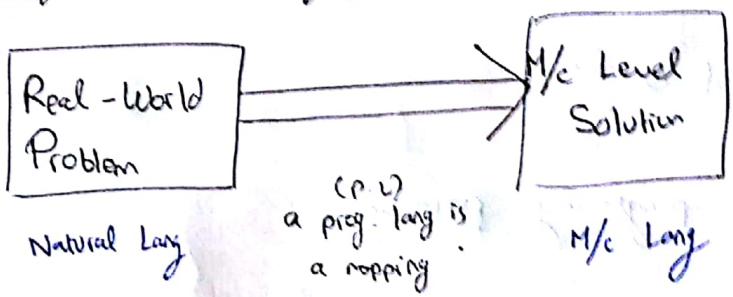
① Imperative Paradigm ② Object-Oriented Paradigm

③ Functional Paradigm ④ Logic Paradigm (Rule-based programming)

$f: x \to y$

$y = f(x)$

Programming Language Abstractions



As PL has become sophisticated, it abstracts the "M/c Level Sol^n" and has come closer to natural lang.

But natural lang can't be used because of its inherent ambiguity

→ we need unambiguous compact str to get one meaning out of it

Abstraction is the data hiding that helps us to focus on our task

P.L Abstractions
- → Data Abstrac (related to the data)
- → Control Abstrac (the code we are writing)

Data Abstraction → **Basic Level:** Hide m/c specific tasks, primitive level

Hides internal representation

(allowing to view a primitive data type as a single variable. No need to think about representation formats, no. of bytes it takes & operations allowed on it)

Eg: float f = 12.34; 2's complement

we don't need to know whether single/double precision format of IEEE

we are abstracting → data representation

Scanned by CamScanner

other abstraction we are hiding is f

basically a mem. loc^n

but we don't write in form of mem. address

② variable / identifier

③ size (not recursive)

size the variable is going to take.

Eg: --- no. of bytes from starting posn

③ type

Eg: the types of operation allowed on it.

Structured: (collection of primitive items & treat them as one thing)

① collection of similar data items

Eg: array

operations not only on single data items but also on each and every element of array

② collection of dis similar data items

Eg: data structure

③ text-file (many diff. data types together as one entity)

Structured Abstraction

→ This also allows to view each of the components of the structure, i.e. this view is not atomic unlike Basic Level

→ Allowing you to group the components together

④ User-defined types

→ based on the primitive data types.

Unit Data Abstraction: Data & the procedures (behaviour) that can be executed on the data & grouped as one unit

① ADT : allows the operations that can be done, but not how they are implemented.

Scanned by CamScanner

ADT → class in Java (small scale)

ADT → package in Java / module in python (large scale)

group classes together & specify access restriction (public, private, protected)

→ So in case of ADT

→ ① information hiding is done
(bcos how things are implemented is not of concern to programmer)

→ ② reusability
(by using language library mechanisms; standardizing use of other codes in your code itself)

→ standarized mech. of using other components

→ ③ interoperability

→ starts from reusability
(bcos way things are reused are standardized)

→ don't need to mention again how components are going to be used

---

Control Abstraction

① **Basic**

Eg: float f = a * b ;

if we were writing on m/c lang → { first 2 load instruc
mul (if supported)
store in loc^n

This enables to look at it as atomic abstraction

Eg: x += 10 ; (syntactic Sugar)

used to refer to any mechanism that allows a programmer to replace a complex notation with a simpler short-hand notation

Scanned by CamScanner

So Basic Abstraction → ① algebraic expression/assignment expression
→ ② syntactic sugar

② Structured Abstraction : Group of instructions
→ ① loops
→ ② branching
→ ③ functions
→ ④ iterator
→ ⑤ procedures

Eg:

```
for (i=0; i<n; i++)
    S.o.p (arr[i])
```

JAVA

```
for (int i : arr)
    S.o.p (i);
```

This is also a syntactic sugar for loop

④ spcl kind of obj that helps to traverse any kind of collection (abstraction)

⑤ procedures → hv certain i/p variables & certain o/p variables
→ mostly focussed on how things are going to be executed
Eg: in FORTRAN

But in ③ functions : more closely related to mathematical definition of a func^n

Eg:

```
int f (int x) {
    return x^2;
}
g = f(x)
```

So procedure invocation & func^n invocation are similar.
(both also need declaration)

In func^n, this closely relates mathematical func^n

A func^n can not only hv parameter as a func^n, but it can return a func^n as well

Eg: $y = g(f(x))$ we can club func^n

a parameter may not be a func^n, it can also be a variable

Eg: $f'(x) = g(f'(x))$ its returning a fun

Scanned by CamScanner

map ⇒ is a higher order func^n

⇒ parameters : a func^n & a list.

⇒ applies the func^n on each member of the list

Eg: name_lengths = map (len, ["Kushal", "Kanti", "Saulib"])

(In Python)

(= [6, 5, 6]) ⇒ as o/p

print name_lengths.

⇒ map applies something on a list, but returns a list

But if we require a single number out of operation on a list, we need to apply reduce (traverses throughout the list & gives one result)

(applies on first two, gets a no. in accumulator, then this & 3rd ----)

Eg: we hv a no. of elements & we want a summation.

parameters going to used in the method.

to be executed & returns result.

reduce (lambda a, x : a+x , [0,1,3,5,8])

accumulator that stores the result

iterator that traverses the list

$a = 0, x = 1$

$a = a + x = 1$

⋮

we hv not specified a name here, as we hv specified the method

This is a Lambda Expression

In imperative lang,

f() {

every func^n has a name

a = arr[0]

for (x=1 ; x < -- ; x++) {

a = a + arr[x];

} return a; }

func^n + data

③ **Unit Abstraction** → module / package

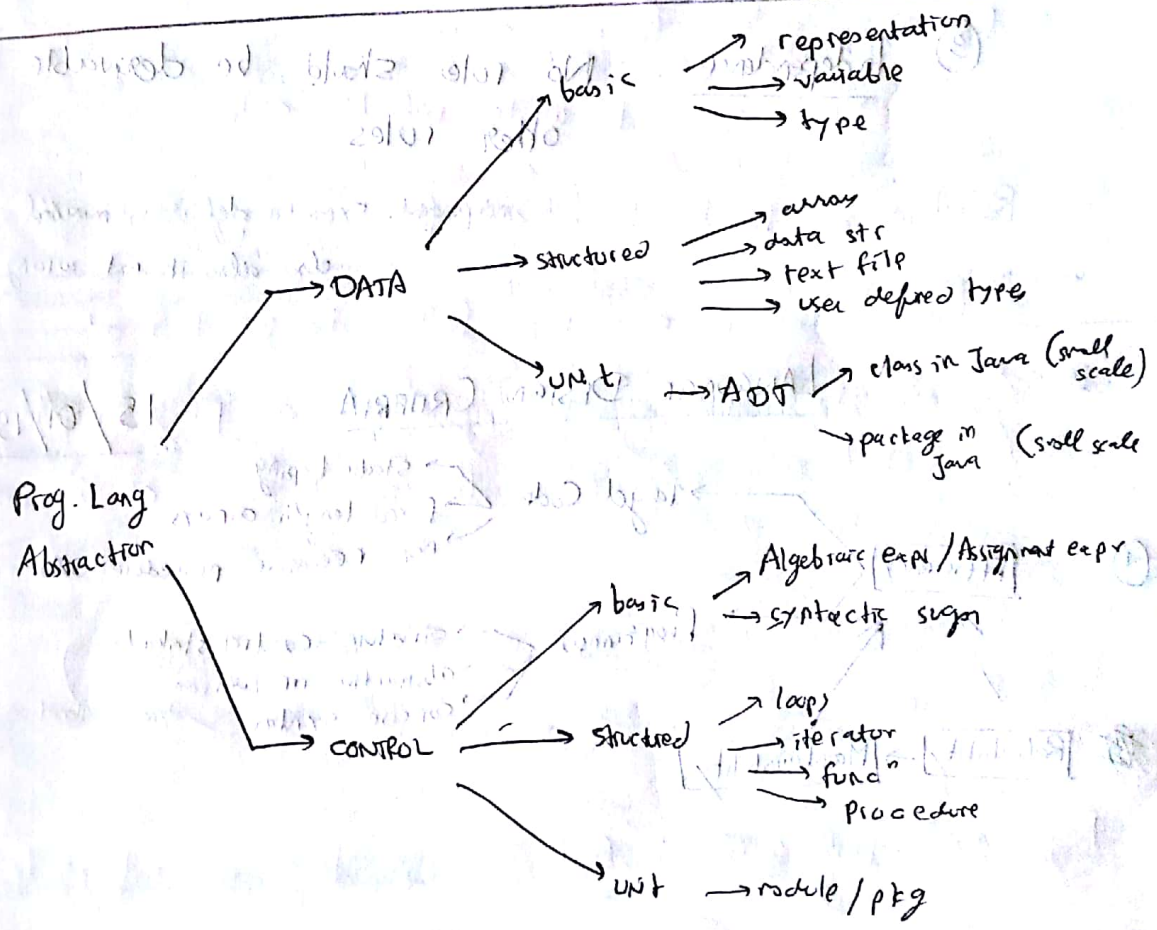
(Similar to unit abstraction in data abstrac (data + func^n))

Thus, OOP can be done by these listed abstractions

But no such single abstraction that can support Parallel Prog Paradigm

→ we can use some features from Structured & Unit Abstraction

---



Any prog Lang : Syntax & Semantics

① Operational Semantics :

② Denotational Semantics : functional def^n (map where to where)

Eg : + : Integer × Integer → Integer

Scanned by CamScanner

How to define a programming lang (4)
① Syntax → lexical str.
② Semantics → acc. to meaning

③ Axiomatic Semantics : specifies axioms

Eg: assert in Java
↓
follows axiomatic semantics
→ allows to denote semantics of a func^n

While defining semantics, look at 3 properties:

(*) but these two must be satisfied

① Completeness : Every correct terminating program must have associated semantics given by the rules

② Consistent : Same program cannot be given two different conflicting semantics

③ Independence : No rule should be derivable from other rules

(If independent, semantic def^n is minimal else also its not wrong) (*)

LANGUAGE DESIGN CRITERIA. 15/01/19

① Efficiency
- → Target Code
  - → static typing
  - → fixed length arrays
  - → non-recursive procedure calls
- → Programmers
  - → structured control statements
  - → abstraction mechanisms
  - → Concise syntax — dynamic allocation

Reliability → Maintainability

non-recursive procedure calls are better for Target Code efficiency

→ This is bcos in case of rec. call, we don't know how many bytes will be reqd. for procedure calls.

→ FORTRAN supports both

Programmers efficiency

But nowadays with multi-core m/c, even if target code is somewhat less efficient, but now programmers are given some level of comfort.

→ Making code more readeable i.e. **Structured Control Statements**

And,

Abstraction Mechanism : way we group our arrays
→ makes code more readeable

Concise Syntax : Recursion provides concise syntax, though target code efficiency may be hampered

Scanned by CamScanner

Reliability of the code: The probability that the code will work correctly at some pt. of time

If we can easily find out at which place modification reqd. (Enhancement of code easily)

If there is any error, whether lang. is helping me to debug that particular error quickly

We will go for a lang that will help to maintain easily, even if it takes longer to write the code

---

→ Efficiency is the foremost concern
↳ will allow to write reliable code

Reliability & Maintainability are based on programer's efficiency

- Maintainability is most imp feature to be look at.
- We should look at those aspects of prog's efficiency that will help us in the maintainability of the code

---

LANG. DESIGN CRITERIA (contd....)

(2) **Regularity**: How well the different features of a lang are integrated

Eg:

```
int i;
i = 5;      → literal
i = j;      → variable
i = f(x);   → function
```

In the same way, can we write:

$f(x) = i$ → In C, this is not possible so a restriction is put

$f(x) = f'(y)$

If prog. lang we are using is allowing to use all

Scanned by CamScanner

the 5 constructs. Then we can say lang allows regularity wrt assignment operator

Regular Lang → allow u to hv fewer unusual restrictions on the use of any particular construct (GENERALITY)

3 sub-criterias of regularity

→ Fewer strange interactions betn constructs. (ORTHOGONALITY)

Eg: In JAVA, int i;
Integer i;

kind of operations allowed on each not same
So way u can club things together for obj.
JAVA not allow to club for primitive data type

→ Fewer surprises in general in the way lang features behave (UNIFORMITY)

i.e. similar things should look similar
diff things should look diff

Eg class & func look same. But totally diff

Regularity
- generality ① → func^n, operators, constants
- orthogonality ② → func^n return type, → placement of variable declaration, → primitive types v/s reference types
- uniformity ③ → Placement of semicolon after class/func^n

① 3 aspects of Generality: ⓐ Procedures & Functions:

In C,
Procedure can be nested (But in a func^n, we can't hv func^n def^n)

So not totally general in C.

In functional prog., a func^n is like a mathematical def^n

$f: X \to Y$

eg $x = 5$ in fun^n lang is treated as a equality as in Maths, not an assignment

$f(x)$ depends on x

But $f()$ is a const.

So now we can assign like data ie $f() = i$

So now generalized.

Scanned by CamScanner

(b) Generality wrt operator : Meaning given to an operator, irrespective of datatype will be same or not

Eg: ==

if (i == j) primitive data type → value checking

Object → addr. checking

So JAVA does not show generality wrt == operator

For generality,
the way == operator operates on an array should work similar on members as well

Eg: Hascale is a lang that provides complete generality wrt operators

(c) Constants

If a lang shows all the 3 ways of assigning values to a const,

i = 5
i = j + k
i = f(x)

then it shows complete generality

Eg: ADA shows complete generality wrt constants

---

(2) Orthogonality aspects

(a) Func^n return types : In case of C/C++, func^n return type can be ① primitive type (like pointers, not arrays. Again to check size is headache of programmer,)
② structured type,
③ function

Any functional lang Eg Python, ADA, Java 8 onwards, can show orthogonality wrt func^n return types.

Scanned by CamScanner

(b) Placement of variable declaration : Eg. In C we hv to list variables at the top

(c) Primitive types v/s reference types : In JAVA, primitive types are treated as scalar types and they follow value semantics

i = j ; (Value will be copied)

i → [▨]
j → [▨]

Even though they refer to diff. mem. loc^n

But in case of object types in JAVA, it doesn't follow value semantics, it follows reference semantics

Eg: i' = j' ;

Thats why, when we compare 2 objects, their addresses are compared (i' == j')

Thats why wrt data types, JAVA does not show orthogonality

In Python, all data types are object types. So always reference semantics followed. So wrt this criteria, Python shows better orthogonality than JAVA

---

(3) Uniformity :

(a) Placement of semicolon after class or function

class { ⋮ } ;

struct { ⋮ } x;

To achieve programmer's efficiency, uniformity is bit compromised

Eg: ALGOL-68 shows highest amt of Regularity (3-concepts) But none use it

Scanned by CamScanner

So lang design efficiency should be a well - balance betn all the lang. design criterias

---

Design Criteria

18/01/19

art Efficiency

① **Regularity** → Generality
→ Orthogonality
→ Uniformity

But even if prog is regular not the best prog. lang.
Gives more importance to Programmers' efficiency than target code efficiency
Regular code → easily to understand, we are fast to learn

② **Security**

more concerned with target code efficiency. But we can say prog. eff. also → as we can say that we will not get seg fault ⇓ easier to debug the error

when we write a prog & execute, prog should never fail

Security deals with types, type declaration, variable } makes code more lengthy (Static typing)

Idea of security: is to maximise the no. of errors that could not be made (prog. lang is restricting the programmer to make these errors)

C compiler → ensures code is syntactically safe

Python, Java, Lisp → - - - - semantically safe → there could not be any error at runtime, that can violate the def^n of the lang

(C, C++ are not semantically safe because : ...)

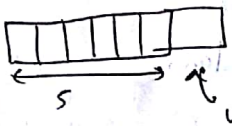
Scanned by CamScanner

① Garbage collection (we hv to explicitly say tht we want to delete the obj. In C, C++,)

In Java, automatic garbage collection present. Prevents mem. leak etc)

② In Java, Python, we hv a runtime environment dedicated which will take care of execution of prog. → makes prog. secure and ensuring semantic safety

Eg:



we try to access this

In C++, C, its not possible to track these accesses

For these, we need some runtime env. that takes care of such things by looking closely

But ① & ② takes extra time to execute

So Java, Py, slower than C++

Better to use type declaration → easier to track errors

↓

but hampers explicitness of lang

↙

as it results in more no. of lines in code

```
int i;
i=5;
```

~~i = "Hi"~~

Static typing → any type error should be detected by compiler

dynamic " → — — — — — by runtime env.

} it is not related with whether we state int i; → its just a bool

We want to type less, but we want errors to be detected by compiler

↙

ideal sol^n (ideal balance bet^n Regularity & Security)

(balance bet^n expressiveness & security)

Eg: New version of Java provides this type of security

Scanned by CamScanner

③ **Extensibility** : ~~You can extend the~~

syntax behavior | packages & modules | extend built-in features through versions.

→ Extending the basic syntax behaviour

↓ new keywords in LISP

→ Not only a lang allows u to define a new func^n, but also allows to group a no. of func^n together as a header file / module / class.

⇓

i.e. create your own pkgs / modules ⇒ this feature provided by most of the modern High Level Lang

→ when the lang designers are providing diff version of lang, they are not only providing syntax behaviour, pkgs & modules, but also provides:

→ new keyword to lang

→ new meaning to syntaxes.

→ when a new version comes up, designers ensure that

→ it has to be <u>bckwd - compatible</u> with prev ones

(∵ all prev codes already be written)

→ some features are identified, which are no longer reqd (<u>depricating features</u>)

They are removed in more further newer versions

<u>But for LISP</u> :

Scanned by CamScanner

```
(do()
condition → ((= 0 b))
body of loop → ( let (temp b))
               ⋮
```

BUT we are more comfortable with while loop ⇒

new keyword we want to add

```
(while (condition))
    (let (temp b))
    ⋮
```

body

So if we wanna do it, we can define a Macro as:

```
(defmacro while (cond^y & rest body)
    (do ()
        ((not, condition))
        /@ body ))
```

these syntax will be mapped

Through macro facility, LISP allows you to extend the basic behaviour of the language i.e. to give new keywords in LISP

But if these 3 features of extensibility are ensured, does it restrict programmer to write bad pieces of code?

Eg. write everything in main

Eg. a func^y having a lot (20-30) parameters

Eg. 2 func^y running back & forth

(ideally, one func is a tight piece of code that is least dependent on other

Modules should have → ① high cohesion ② low coupling

If it is low cohesion & high coupling → bad piece of code

but no prog lang stops u from doing it

Eg. go to in C

Scanned by CamScanner

So if your lang has so many good features, it is upto programmer to write good pieces of code/use these good features. Language don't enforce any programs

FLON'S AXIOM : states that, there does not now, nor will there ever, exist a programming language in which it is least bit hard to write bad programs.

i.e there can never be a bad language, but there can be bad programs

---

In C, use of pointers give flexibility → but we can write very bad prog with pointers which are unreadable to others.

But in Java, Py → no pointers

So it actually restricts to write bad program

TURING TAR-PIT : It states that all computing languages OR computers can compute anything in theory, but nothing of practical interest is easy

Eg. Diff prog lang hv a commonality → most of these lang are complete → every complete prog lang enables u to do whatever problems for which a Turing m/c will halt

So whats achievable in C are achievable in Java too

↓

Bcos all lang are complete wrt Turing m/c

↓

From perspective of functionality, each are equally powerful (Turing complete)

So, if we can develop a lang with least keyword which is also Turing complete, its easier to learn → Eg. BrainFuck has 8 keywords

⇓

but very difficult to write

→ Don't make your lang so powerful that makes it very difficult to solve a particular problem.

But to make it useful as a tool that can be customised to solve a class of problems. (Eg. Perl)

Python → aims to bridge gap betn Shell Scripting & prog. lang like C

Simplicity : Py provides small set of primitive instruc, that u need to know to hv your prog running

Regularity : (Ensuring syntax are regular)

Functional lang related with regularity, that naturally allows u to follow Generality, Ortho, Uni

Since Py, func prog → so naturally provides some sort of reg

Extensibility : → use pkg

→ API support very strong

→ It is simple, but its powerful. Some non-primitive data types are also provided like basic data type

Eg: dictionary → simple, but powerful

Scanned by CamScanner

Interactivity & Portability :

Execute your prog in 2 ways

more interactive :
→ write a prog & execute it
→ write in a module and then execute it

(Eg don't hv to write public class ...)

→ execute your code to any other platform ..
→ platform independent like Java
(hv a PVM (Python virtual M/c))
runtime environment

→ attracts diverse set of audience (dedicated API for diff zone of programming)

→ Since it is dynamic typing, so lesser no. of code wrt static typing

---

22/01/19

① imperative
② functional
③ Obj-oriented
④ logic

Scanned by CamScanner

In functional (every variable is a const.) ~~imperative~~ programming, no assignment. Use a new variable on LHS of assignment, every time

Eg: $(x = x + 1)$ ✗ $(y = x + 1)$ ✓

But, in for-loops, while-loops there is a counter that gets incremented and assigned every time Eg:

```
for (i=0; i<5; i++){
}
```

But in functional prog., we can't hv something like this.

Solution is to use recursion (rec. stops when variable reaches certain value)

→ All types of loops hv to be replaced by recursion in functional prog.

Eg:

In imperative programming →

```
void gcd (int u, int v, int *x){
    int y, t, z;
    z = u, y = v;
    while (y != 0){
        t = y
        y = z % y
        z = t;
    }
    *x = z;
}
```

Scanned by CamScanner

In functional programming :

```
int gcd (int u , int v){
    if (v==0)
        return u;
    else
        return gcd (v, u%v)
}
```

(*)

can't use pointer ~~to return the~~ store value at particular loc^n also.

So use int instead of void

Mathematically, $gcd(u, v) = \begin{cases} u & \text{if } v == 0 \\ gcd(v, u \bmod v) & \text{otherwise} \end{cases}$

⇓

Can't map this def^n directly to imperative prog but can do so to func^nal prog..

(*) In func^nal prog eg above, we are returning an int

⇓

dependent only on parameters & not on any internal variable

⇓

always return same value based on parameters only

⇓

possible only bcos we hv removed assignment property

output of func^n depends on :

1. its arguments
2. any non-local variables

(Eg. i=5 , a const outside func^n & returned value depends on i)

3. should not depend on the order in which the arguments are evaluated

(3 criterias for referential transparency)

If any func^n follows all these, we can say the func^n is referentially transparent

NOTE: All methods can't be made referentially transparent Eg: random no. generator.

Scanned by CamScanner

Advantage of functional prog: (ref. transparency)

① If you make a func^n referentially transparent,

as far as x & y are const., z will not chng

$z = f(x, y)$ ⇓ f always returns same value

⇓ can treat $f(x, y)$ as a 1st class data variable

A referentially transparent func^n can be treated as a first-class data value. ⟹ With data-value we can call func^n (ie. use as an argument)

A func^n call call a data value

On the otherhand, method can't be used as parameter/return-type to any other method

Eg: $z = f(f'(x), y)$

$f'(x) \Downarrow x'$

We can also return a method itself

Eg: $g(x) = f(x, y)$ ⟹ we can pass behaviour

② help in parallel prog.

⇓

if thread uses any of those variable shared by other threads

⇓

hw to ensure no other thread is writing ~~at that~~ / modifying that variable

In func^n prog.

variables are non-mutable

→ easily ~~chng~~ use all those shared variables bcos these variables don't need any synchronization

⇓

only bcos of ref. transparency ⟸ immutable

Scanned by CamScanner

③ func^n composition for higher order functions

Eg: $f: X \rightarrow Y$

$g: Y \rightarrow Z$

$g \circ f: X \rightarrow Z$

$(g \circ f)(x) = g(f(x))$

Here by composition operator, we are clubbing 2 func^n together

one func^n provided as i/p to other func^n

possible bcos of referential transparency

If your func^n returns a func^n / takes a func^n as argument (if both/ any one are true), we call the func^n a Higher-Order Function

How to make a func^n to return a constant value } ⇒ if your func^n doesn't hv any argument, it will always generate same value wherever called.

Eg:
```
int func(){
    S.o.p("Hello World");
    return 5;
}
```

always give you 5 wherever called

So, int i = 5 (OR) int i = func() } Basically these 2 are same

⇓

no argument func ≡ const value

Main characteristic of functional prog

→ No assignment ; variables are immutable

→ Loops are replaced by recursive calls

Scanned by CamScanner

→ func^n are refeentially transparent

→ func^n can be treated as First-Class Data values.

Eg. of func. prog: ML, Hascale, Lisp, Clojure

As h/w cheaper, philosophy of func. prog. very imp in todays world

So Java, CPP supports multiple paradigms like imperative, obj-oriented, func prog

(Multi-Paradigm Lang.)

JAVA

There is a button & we want to print "Hello World" wherever the button gets clicked

```
button.addActionListener(new ActionListener(){
    void actionPerformed(ActionEvent event){ (*)
        S.o.p("Hello World");
    }
});
```

Here we are passing a behaviour, not data, to the Widget

Since in OOP there is no way of passing behaviour, so the concept is to build Anonymous Class.

We don't hv the name of the class that has implemented ActionListener.

What func. prog can give u?

→ Since we hv refeentially transparent prog. (*) (this prog) will always give same result.

So we can write this as:

Scanned by CamScanner

event → S.o.p ("Hello World")

We can then rewrite the statement as.

defn of func (deals with HOW)

button.addActionListener (event → S.o.p ("Hello World"))

parameter or arg. of func (deals with WHAT)

behaviour we are passing as parameter to addActionListener.

more readable

(Here we don't send an obj as before we just sent the behavior)

This construct is known as Lambda expression

event is basically a parameter. ← hv to be ref. transparent.

Compiler will take the event as ActionEvent

NOTE: we hv not mentioned dataType, but compiler understands its dataType

As per Java API, addActionList... takes an obj of type ActionList... So whatever we hv written will be related to A...L... class. But compiler sees its an interface.

→ Only one parameter related to Action performed.
→ Exactly one abstract method related to the interface.

p.void actionPerformed (ActionEvent)

No ambiguity, thus.

Here its dynamic typing (prog. hv not written, but compiler recognizes)

Since Java supports multiple paradigm, so these things possible

Another eg: Runnable interface (an interface having only one method)

So if we make an obj. of Runnable interface,

Scanned by CamScanner

Runnable r = () → s.o.p ("Hello World");

This will be tedious to write by oop.

only one method that should be defined for Runnable

since it doesn't take any argument so this is also mapped

So no argument.

We can have multiple statements as well, but in that case hv to put braces

Eg: Runnable r = () → {sop ("Hello World");
s.o.p ("Hi BCSE");};

If we hv an interface that has two variables, then

(x, y) →

System automatically understands the ordering maintaining that of method declaration

Eg: Interface1 iff;

int add(x, y)

interface Interface1 method of the int add(int x, int y); Interface1 iff (declared exactly one method for this interface)

Interface1 iff = (x, y) → x+y

OR

Interface1 iff = (x, y) → x*x + y*y

There is an interface called Binary Operator

Eg: BinaryOperator add = (x, y) → x+y;

This will not work

We hv to specify the data-type as it takes Generic

Eg: BinaryOperator<Long> add = (x, y) → x+y;

But there is a confusion whether x, y are considered as long or int.

Although automatic cast from int to long

Scanned by CamScanner

Eg: BinaryOperator <Long> add = (Long x, Long y) → x + y;

Eg: of the flavour of dynamic typing in JAVA

int [] arr = { 5, 6, 7, ... }

String [] arr = { "Sti", "abc" }

↓

types of these depends on what is written here

Eg: NULL (depends on whether it's string, int)

→ You can use lambda expressions for those interfaces that has exactly one abstract method

→ may use more than 1 parameter & return a value

→ may use generics

→ Signature of the interface is the same as that of the abstract method

---

Action Event → Action Listener

25/01/19

```
{ String name = userName;
  button.addActionListener(new Ac.... () {

        public void ActionPerformed(A.E... e) {
            S.O.P("Hello" + name);
        }
  });
  name = "BCSE III"
}
```

Scanned by CamScanner

→ output of A.P. ( ) doesn't depend on argument

→ also depends on non-local variable, 'name'.

→ method is not ref. trans. bcos we are using a variable that not const, its mutable

Even we are using non-local variable, we must ensure that its value don't change ⟶ implicitly final / effectively final

So for name = "BCSE III", compiler ~~gives~~ throws error.
↙ value changed

In JAVA,

instance variables → stored in a heap & shared among threads

local " → not shared among threads

→ makes a local copy & works on it

→ chng on one thread might not reflect on other thread

→ o/p may not be consistent

JAVA → strongly typed lang → compiler never lets u run any prog that gives erroneous o/p.
not described in lang description

So doesn't allow a mutable variable to be used in a ref. trans. prog

→ value copied

→ can't trace its value

It will only allow to use immutable non-local variable

→ Lambda Functions are effectively closed around values (rather than variables)

⇓

Since it always gives same value whatever tem it is run

In OOP, methods are closed under variables ⟷ (a paradigm shift)

Scanned by CamScanner

Q What do you mean by Lambda expression ?

A. Lambda expression is a func^n w/o a name i.e. used to pass around behaviour as if it were data.

In JAVA, we usually say, { class / method / variable → no method w/o any class

① But for lambda exp, it will get the class according to the context where its used.

func^n

bcos not closely bound to any class.

☆ corresponding Lambda exp ...

button.addA — (event → sop("Hello" + name));

body of func^n

② w/o a name

③ pass around behaviour

data pass around → we can assign to or assign from it

pass around behaviour → used as argument to a method

OR can be stored in a variable

So, we can actually return a func^n

Eg: ① Lambda exp to create object

() → new Employee () ; ⇒ an obj created & returned

② Lambda exp to chck if list empty / not

(list) → list.isEmpty () ; ⇒ chck some cond^n. Return back a Boolean value

Scanned by CamScanner

③ used to extract info from an obj

(string s) → s.length( )

④ compare/combine things

(string s) → { return s + "ABC"; }

↳ if u are writing return, write within { }

OR

→ s + "ABC"

Ways of writing Lambda exp.

w/o Lambda exp

Runnable r1 = new Runnable () {

}

with Lambda

Runnable r1 = () → - - - -

↳ created but not invoked. we need someone to invoke it.

P.S.V process(Runnable r) {

r.run();

}

⇒ invocation

But also if we straightaway write something like,

process(() → s.o.p. ("Thread 2"));

it is also valid

⇓

Eg. of Pass-Around Behaviour (directly pass the behaviour)

Scanned by CamScanner

Predicate <T>

⇓

a functional interface having exactly 1 abstract method called test, that will take i/p as a generic & o/p a boolean

① T → Predicate <T> → boolean

test i/p (no o/p)

② T → Function <T, R> R

apply i/p o/p

(takes one type of generic & o/p another generic Eg: int → long)

③ → Binary Operator <T, T, T> →

i/p i/p o/p

(all generic are of same type)

Eg:

$x \to x+1$ } Function

$(x, y) \to x+y$ } Binary Operator

$x \to x==1$ } Predicate

Functional Interfaces:

Callable <Integer>

PriviledgedAction <Integer>

} Don't Accept any argument (i/p) but gives back a generic ⇒ mapped to integer in this eg.

Eg: c = () → 42;

p = () → 42;

} same Lambda exp used for diff interfaces

→ Lambda exp. can be mapped differently, depending on the context in which it is used (i.e. type of c & p)

Scanned by CamScanner

→ it will be mapped only if valid signature

→ ambiguity in mapping ⇒ compiler error.

Another problem in generic : **Boxing**

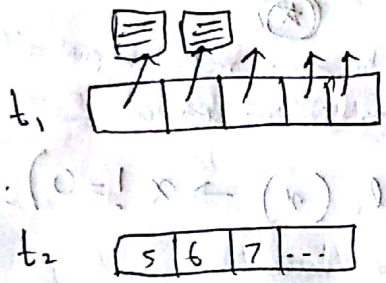
Generic

Primitive Type

Reference Type

Thread [ ] t1Arr ;

int [ ] t2Arr ;

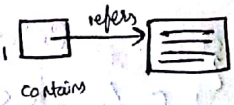


for every member of array it will point to diff loc

int t2 ;
t2 = 5

t2 [5]

Thread t1 ;



So Integer array takes more space than int array

⇓

also takes more time as extra mapping is used (go to 3rd posn in array then use the mapping)

So for generic like Integer, ... one extra memory read to access ⇒ (*) [run-time overhead in numerical ~~problems~~ algo]

So special Predicate for int, float & double (but not for others) (to reduce (*))

Eg: Predicate to check whether no. is even/odd

IntPredicate evenN = (int x) → x % 2 == 0 → no overhead here

valid Boolean expression

Predicate<Integer> evenN = (x) → x % 2 == 0 ;

⇓

overhead of autoboxing

takes more time than prev.

less efficient.

Scanned by CamScanner

Sometimes can print even nos., odd nos., multiples of 3, etc.

Eg:

```
Public class No.Test {

    P.S.V main(S.a_) {

        List<Integer> L[] = Arrays.asList(1, ....);
```

(1)

takes Predicate Integer

```
P.S.V evaluate(List<Integer> L,
              Predicate<Integer> P)
{
  for (i : L) {
      if (P.test(i) {
          S.O.P(i);
      }
  }
}
```

single method in Predicate

print nos. according to behaviour passed

```
// lets say we want to print multiples of 3
evaluate(l1, (x) -> x%3==0);

// if we want the same method to print all nos. that are not 0
evaluate(l1, (x) -> x != 0);
```

But in OOP, we would hv to write seperate methods for all / use overloading (then only difficult as if we pass) : then even no. 2 : odd no. ⋮)

advantage of pass around behaviour

After everything hv been fixed & code gone to production, we can easily chng the line of evaluate to print something else we want

(2)

Suppose we overload the evaluate func^n

takes Predicate Int

```
P.S.V evaluate(List<I...> L2,
               Int Predicate P) {
    ⋮
}
```

Problem is when we are calling evaluate, which version to call

l_1 : always mapped to List<Integer>

⇓

no problem

⇓

But don't help to select which version

⇓

So problem not resolved also.

Scanned by CamScanner

for x, it maps to both Int Predicate & Predicate <Integer>
(∵ no most specific type)

(2 books mixed)

So for type-Inference, 3 rules

both equally probable ⇓ ambiguous ⇓ compiler throws error

① if there is a single specific target type (say any, one among ① or ②)

Javac infers the type from the corresponding argument in the functional interface

② if there are ~~more~~ several specific target types, the most specific type is inferred

③ if several specific types & no most specific type, the programmer should manually provide the type

(today) Java 8 Lambdas 3rd & 5th chapt)

Lambda Expression (contd.)

See first of 01/02/19

29/01/19

Method Reference : System.out::println (class::method)

Eg: forEach is a method & takes a functional interface as a parameter

FUNCTIONAL INTERFACE : Interface having one abstract method

Eg: getLowCalorieDishesNamesInJava8(menu).forEach(System.out::println)
will print every element corresponding to functional interface

Defn: Filter is a method that takes a predicate & returns a stream

→ Stream <T> filter (Predicate <T>);

→ List <String> strList1 = Stream.of("1.Shikha", "2.Anuran", "3.Koustav", "Prithwish")
.filter(S → Character.isDigit(s.charAt(0))
.collect(toList());

Scanned by CamScanner

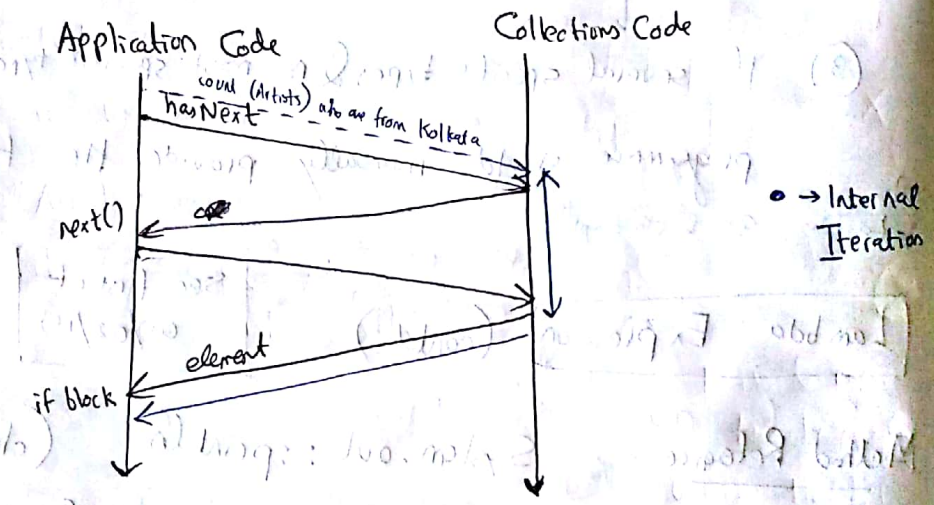
→ interface BufferedReaderProcessor {
    p. String process (BufferedReader b) throws IO Exception
}

BufferedReaderProcessor bp = (b) → b.readLine();

→ throws has to be written in declaration or object creation should be placed in try block

Stream:

new concept introduced in java (java.util.stream)



ArrayList allArtists;
⋮
int count = 0;

external & internal iteration

for (Artist artist : allArtists) {
    if (artist.isFrom("Kolkata")) {
        count++;
    }
}

⇓

while (itr.hasNext()) {
    Artist a = itr.next()
    if (a.isFrom(...))

Artist
String city;
String band;
String name;
boolean isFrom(String)

code is not naturally scalable since it uses external iteration

Scanned by CamScanner

In Internal iteration, using JAVA Stream (java.util.stream), we can write :

```
long count = allArtists.stream().filter(a -> a.isFrom("Kolkata")).count();
```

criteria

stream() → method that makes your collection ready for internal iteration

a → Artist object

We don't need any iterator.

T (generic) → Predicate → boolean } So filter is actually using Predicate

Stream is not like a Data Structure

In case of Collection, we have an entire Data Str stored in the memory

But entire thing never exists in case of stream (Philosophical diff. betn stream & collection)

→ filter is actually working on stream, not filter

→ also gives back a stream

.count() takes a stream & returns a value.

.stream() - - - - . collect(toList())

⇓

will return a list of artists who are from Kolkata.

.count() & .collect(toList()) } consume the stream & gives back other type of data

filter() is (1) one type of method ⇒ take a stream & gives back a stream

⇓

so we can chain these type of methods

Eg:

```
allArtists.stream().filter(a -> a.isFrom("Kolkata"))
                   .map(a -> a.getName())
                   .collect(toList());
```

It gives the names of the artists who are from Kolkata (& not the artist obj itself, as before)

Scanned by CamScanner

filter()
map()
} → takes stream, returns stream → can be chained
one type (Lazily Evaluated)

~~could collect~~

methods will not be evaluated unless you hv an eagerly evaluated method

collect
count
} 2nd type (Eagerly Evaluated)

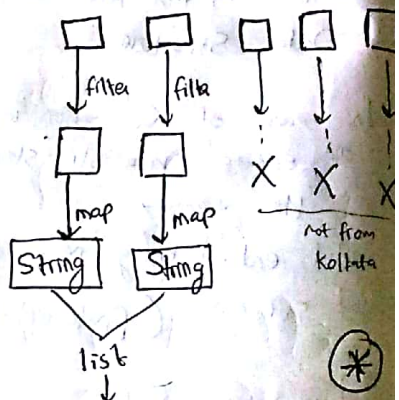
Eg: Stream<Artist> str
= allArtists.stream().
filter(a → a.isFrom("Kolkata"))

System will not execute/evaluate this method

We can check it by writing

------ = allArtists.stream().
filter({ s.o.p(...);
a → a.isFrom ...});

If we hv a collection of items :



(*)

But since Filter & map are lazily evaluate

we can internally break the types

first 2 elem in 1 core, Nxt 3 in ot core

(☆) helps in parallelism ←

Eg: first 3 artists from collection who are from Kolkata

List<String> names = allArtists.stream().filter(a → a.isFrom("kolkata"))
.map(a → a.getName())
.limit(3)
.collect(toList());

Adv. of Lazy Evaluation

(☆) → once it gets 3 artists who are from Kol, others are not evaluated at all.

So efficient

→ (*) since last 3 don't match filter cond^n, So they are not evaluated at all in nxt steps. (Efficient)

you hv all config first
↓
then you hv build method

Builder Design Pattern

(jvm)
Eg: automatically build threads for you
= allArtists.parallelStream().filter() ..... collect()

Stream (Defn) → can be defined as a sequence of elements from a source that supports data processing operations

Eg: collection is a valid source
I/O resources, array can also be source
(those which preserve ordering

becos stream maintains the ordering in which the collection is organized (not pickup elements randomly)

supports both DataBase-like operations & functional programming operations to manipulate data
→ operations can be executed in parallel / in sequence

Adv. of Stream
→ It is declarative (in ext. iteration, we show How of the program Here it is just what we want)
→ composable (flexibility) (filter, map, then again filter, etc.—)
→ parallelizable (programer doesn't need to think how to design it to be parallel betn cores)

2 main characteristics of stream data processing

① Internal Iteration
② Pipeling (only those which returns stream i.e. is Lazily Evaluated)
⇓
possible bcos of Laziness
I can process a limited set of elements based on certain methods like .limit() ⇒ short-circuiting Operator

Eg: first 50 prime nos. from an inf. stream of prime no.

allows you to work with infinite streams (But there's no concept of infinite collection)

(if I can predict result from first condn/elem, then need not proceed further)

Stream v/s Collection

Similarity → Both are interfaces that preserve the sequence of elements stored in a data structure
(gives ability to visualize as a seq. of elem)

Scanned by CamScanner

Diffence → ① Every element in a collection is Eagerly Evaluated

② ---- Stream ---- Lazily Evaluated

For Stream → entire thing never exists in mem at same time (eagerly collected func^n)

→ Processing of a stream is dependent on method eg. collect

→ Consumer-driven (I want to execute whenever I need)

For Collection → entire data str stored in memory

→ supply-driven (whether you want it or not, it will happen)

③ A stream can be consumed exactly once & after its consumed it can't be used for another operation

Eg: Stream<Artist> str = allArtists.stream()
.filter({sop(---);
a → a.isFam---});

str.collect(toList());

we hv already consumed in prev. line. So str will lose its significance after it is consumed. So error → str.collect(toList());

④ (Stream) (coll^n)
Internal v/s External Iteration.

Stream Operations

INTERMEDIATE operations (filter, map, flatMap, limit, skip)

TERMINAL operations (collect, count, forEach)

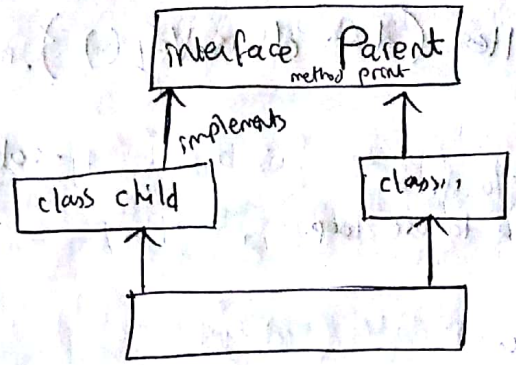
Streams() available from Java 8 onwards

In Java 8, in collection Interface there will not only be abstract methods, but also default methods

Initially, interface had only method declaration not def^n. (so that no prob. in multiple inheritance)

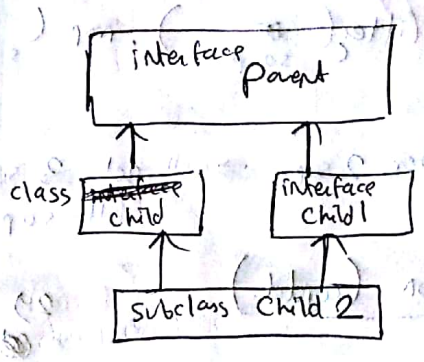
Scanned by CamScanner

C++ had the problem of multiple interface :



which method print to execute

In Java,



Rules: ① If there is a method with a body, or an abstract method, in the superclass chain, then you can ignore the default method (e.g that defined in class child ⇒ class type always wins)

② If 2 interfaces are extending each other, and they compete for a default method, sub-interface will always win

③ In case of any ambiguity, the subclass must either override/implement that method OR it may declare it as abstract.

Dish

string name;
boolean vegetarian;
int calories;
Type type;
getName() is Veg() getCalories() / String toString(); getType();

public enum Type { MEAT, FISH, OTHER }

menu.stream ()

Scanned by CamScanner

Q Names of vegetarian dishes.

A. menu.stream().filter(d -> d.isVeg()).map(d -> d.getName())
.collect(toList());

→ if we had to write this w/o stream, then had to use loop.

Q name of 3 such veg dishes

A. -- menu.stream().filter(s.o.p...).map(s.o.p.).limit(3)
.collect(toList());

try by writing these 2 sop. → will get 3 sets of filter s.o.p & map s.o.p as o/p.

Lambda Expression (contd.)

29/02/19 (first part)

F.I: obj = () -> { s.o.p.(...); }

→ Not every interface having exactly one abstract method is a functional interface

Eg: close() is a method of Closeable interface, depends on internal state of the object. close() inherantly follows reference semantics. Closeable Interface has exactly one abstract method but can't be a functional interface.

→ Functional Interfaces allow exceptions to be thrown.

Scanned by CamScanner

Method Reference: System.out::println (class::method) 1/2/19

Eg: forEach is a method & takes a functional interface as a parameter.

FUNCTIONAL INTERFACE: Interface having one abstract method

Eg: getLowCalorieDishesNamesInJava8(menu).forEach(System.out::println)

will print every element corresponding to functional interface

Def^n: Filter is a method that takes a Predicate & returns a stream

→ Stream<T> filter(Predicate<T>);

List<String> strList1 = Stream.of("1. Shithi", "2. Anuran", "3. Koustav", "Prithwish")

those which has roll no.

- filter(s → isDigit(s.charAt(0)))
- collect(toList());

Filter can: Stream<T> filter(Predicate<T>)

→ Select unique elements

T → Predicate → boolean

→ Ignore first few elements

→ Truncate a ~~string~~ stream to a given size

(try these at home)

→ serves similar purpose as selection

Eg: List<Integer> numbers = Arrays.asList(1, 2, 5, 7, 9, ...)

We wanna print event nos. from the list.

A.

this will return a stream of even nos.

numbers.stream().filter(i → i % 2 == 0)
.forEach(System.out::println);

Scanned by CamScanner

I want low calorie veg. dishes.

```
public static List<String> getLowCalDishName (List<Dish> dishes) {
return
    dishes.stream().filter(d -> { S.o.p ("Filtering " + d.getName());
    return d.getCalories() < 600 && d.isVegeterian(); ))
    .map(d -> System.out.println(d.getName()); )
    return d.getName(); ))
    .limit(3)
}   .collect(toList());
```

Map

Stream<T> map (Function<T,R>)

↓ Takes one type of generic returns same/diff type of generic

---

From eg. of ☆

```
strList1.stream().filter(s -> ch...isDigit(s.charAt(0)))
                .map(s -> s.charAt(0).toString())
                .collect(toList());
```

seeing similar purpose like "Project" in Reln. calculus

But actually need to take from 1st posn to • to handle multi-digit roll numbers.

```
strList1.stream().map(s -> s.toUpperCase()) ----
```

Collection of length of names

```
---- .map(s -> s.length()) ----
```

Scanned by CamScanner

We want list of distinct words

```
List <String> word1 = Arrays.asList ("Hi", "Hello", Hi", "Hello"
                                     "Hell") ;

List <String [] > distinctWords
        = word1.stream().distinct().collect (toList())
```

We want list of unique letters from words,

```
List < String [] > distinctLetters = word1.stream().map (
        w -> w.split("")).distinct().collect(toList())
```

no space.
So splits strings into characters.

⇒ But this gives out hashcode as o/p

This is bcos:

⇒ Stream <String>

After split, we are getting

⇒ stream <String []>

When we take distinct we get:

When we do:

S.o.p (distinctLetters)

it actually prints members of ~~distinct letters~~ distinctLetters. which are arrays (i.e. objects)

We are getting list of string array, while we ~~expecting~~ expected list of string / list of characters.

Scanned by CamScanner

We need a flat collection of the characters:

H i H e l l o ----

So we hv the method: flat Map

⇓

it takes a generic, but it gives back a Stream of Generics

It once again is of a type Function

~~flatMap~~

↳ T → Function → Stream <F>

So we write:

```
List <String> distinctLetters = word1.stream().map(w -> w.split("..))
                                .flatMap(Arrays::stream)
                                .distinct().collect(toList());
```

converts individual Arrays to a stream

Eg:

I hv a stream of nos:

```
List <Integer> numbers = Arrays.asList(1, 2, 5, 7, 9 ---)
```

I want a list that hv square of the individual nos.

A.

```
numbers.stream().map(i -> i*i).collect(toList());

List<Integer> sqr =
```

Eg: Merge to Lists (So we will use flatMap)

Eg: Given 2 lists num1, num2 form a pair such that sum of the numbers in each pair is even

A: List <Integer> num2 = Arrays.asList (2, 5, 7, 17);

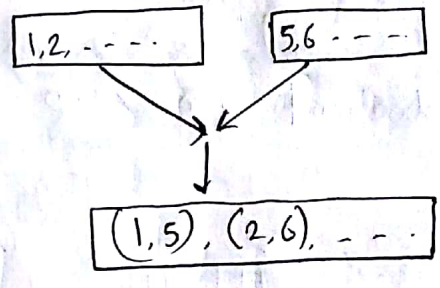
— — — num1 = — — — — (1, 2, 5, 7, 9);

We want op: $\overbrace{(\underbrace{(1,5)}_{\text{Array}}, (5,5), \ldots)}^{\text{List}}$

```
List <int[]> result = num1.stream()
                        .flatMap( i -> num2.stream()
                                  .filter(j -> (i+j)%2 == 0)
                                  .map(j -> new int[]{i,j})
                                  .collect(toList()));
```

Java 8 onwards

from 2nd list



flatmap gives this collection
we hv to select from this

Stream Operations

05/02/19

→ filter
→ map, flatMap
→ limit, distinct
→ sort.

Terminal Operations:

→ collect (toList())
→ forEach
→ count
→ allMatch, noneMatch, anyMatch
→ findAny, findFirst

// Boolean allMatch(Predicate)
will operate on all items of stream & return true/false

↑ can be made parallel

↑ inherently sequential
work with ordered collection

Scanned by CamScanner

→ if (menu.stream().allMatch(dl → dl.isVegetarian()))

S.o.p ("This is a strictly vegitarian menu");

else if (!(menu.stream().anyMatch(dl → (dl.getCalories<130

S.o.p ("This menu contains a low calorie dish");

(done)

Scanned by CamScanner

Stream Operations

① filter
② map, flatMap } stateless (why, stated later today)
③ limit, distinct, skip
ⓐ ⓑ ⓐ gives unpredictable result in parallel stream
stateful
④ sort ⓑ
only for ordered collection / sequential execution

Terminal Op's

is filter to map possible? & vice versa

① collect (toList())
② forEach
③ count ⓐ → stateful
④ allMatch, noneMatch, anyMatch
work on parallel stream
⑤ findAny, findFirst ← inherently sequential
stateless ☆
give us back 1st matching item.
⑥ reduce ⓐ
stateful

ⓐ: bounded states
ⓑ: unbounded states

Optional Class

doesn't hv a type of its own. works on where it is applied

(is a new class in stream API to handle NULL problem)

```
Optional <Dish> d1 = menu.stream().filter(d -> d.isVeg()).
                                    .findArray();
```

Methods in Optional Class

→ <T> findAny()
→ ifPresent (Consumer) → d1.ifPresent(d2 -> d2.getName());
→ orElse (

Action
Ppl02

will be executed only if I hv a matching case.
Then only it will return (matching obj)

→ But, d1.orElse(new Dish());

if there is no Veg item, in that case default value will be just an empty obj.

Diff betn findAny & findFirst :

☆ → There is no visible diff betn findAny & findFirst if your stream is not a parallel one.
(You should not go for parallel stream when ordering is imp)

→ If your source of Stream imposes ordering, string will hv ordered collection. If stream is itself ordered collection first find its significance

Scanned by CamScanner

Till now, we hv given some criteria. By this criteria it chks every elem & goes to nxt. Doesn't hv to remember state with prev. elem. Eg: None Match

↓ Once it gets the result, it will leave.

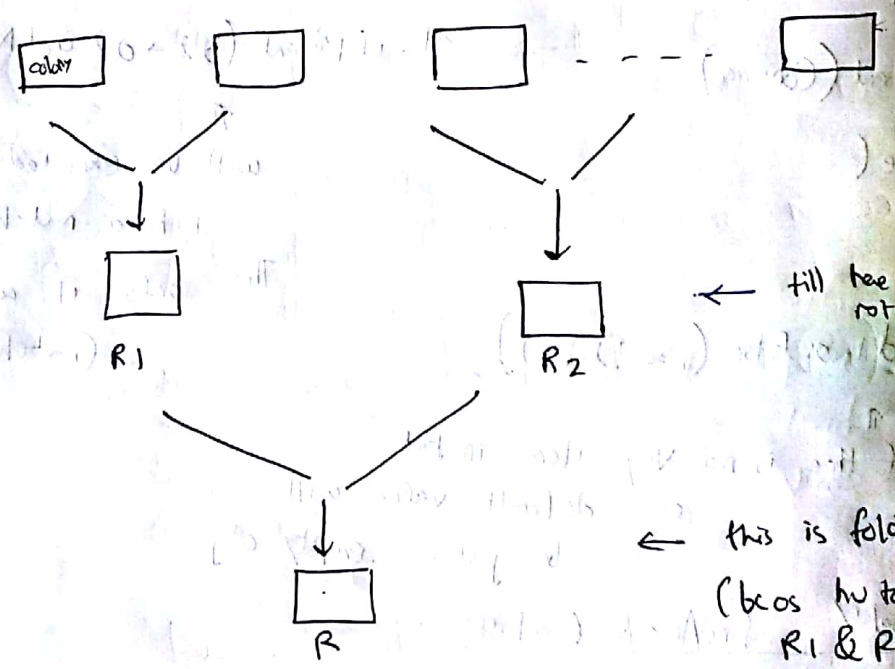
⇓ best case is, the 1st elem matches the criteria.

⑥ $\boxed{\text{REDUCE}}$: (stateful)

will also operate with each of the members of the string, but will remember what it had done with prev. → & clubs that together

- This is called FOLD in func^n prog

Eg:

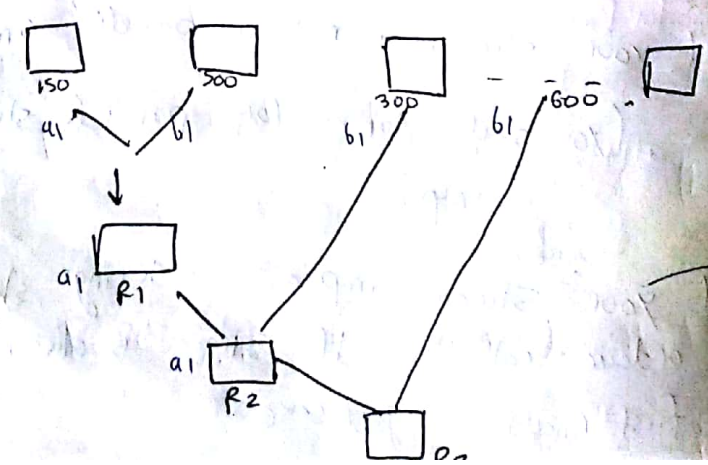


← till here its not fold

← this is fold (bcos hv to remember R1 & R2 to get R)

OR

Eg:



we can sum, avg of elements OR get dish consuming max^m calorie

→ filter / map don't club result of prev. elements with new elem. This is done actually by reduce

Type 1 :

<T> ← reduce (Binary Operator <T>)

→ this is for some initialization value of accumulator (a1)

Type 2 :

Optional <T> ← reduce (Binary Operator <T>)

another overloaded version of reduce.

⇓

Bin..Op... is a functional interface, which takes 2 generics of the same time & gives back generic of same type

T, T → [Binary Operator] → T

thats why you can merge the result.

Eg: Find name of dish with highest calorie :

.... stream().reduce(new Dish(), (a1, b1) → (s.o.p(b1.getName()); return a1.getCalorie() > b1.getCalorie() ? a1 : b1;

default value of accumulator initialized with empty Dish object

compare objects based on calorie

returns dish object itself

))).getName();

PTO

OR in short

(a1, b1) → a1.getCalorie() > b1.getCalorie() ? a1 : b1

⇓

first parameter represents an accumulator & 2nd param will indicate the index → i.e. move through the item.

find result will be in a1

Eg: to get sum of calories :

Type 1

menu.stream().map(a → a.getCalories()).reduce(0, (a1, b1) → a1 + b1);

initial value

accumulator

traversing index

150 500
600 300
950 600

first hv to map the whole stream to integer values.

→ bcos reduce takes 2 op of same type & returns of same type

Scanned by CamScanner

→ Here a1 can't be immutable (as value of acc. needs to be changed)

→ for multiplication of all elem, the initialization value for a1 should be 1.

But what if we dont specify any default value? This follows Type 2

```
Optional<Integer> totalCalorie = new.stream().map(a -> a.getCalories())
                                                  reduce((a1, b1) -> a1+b1);
```

But what if we don't initialize a1 with any value

we can initialize a1 with 1st elem, b1 with 2nd

But what if stream is empty?

→ if total calorie is three, it prints that

→ if empty string then prints 0

Rev Pg

If we didnt want to initialize a1 with new Dish()

```
_____.reduce( ) ... )).orElse(new Dish()).getName();
```

for the case if stream is empty

Numeric Streams:

Eg: numbers = {12, 13, 15, ...}

I want ① product of this list

② square of each element of this list.

For ②, numbers.stream().map(a → a*a).collect(toList())

⇓

returns a list: {144, 169, ...}

Scanned by CamScanner

For ①

```
ct = numbers.stream().reduce(1, (a1,b1) -> a1 * b1);
```

↓

12 x 13 x 15 x - - - -

If I wanted to find sum,

```
t sum = - - - - - - - .reduce(0, (a1,b1) -> a1 + b1);
```

OR

```
- - - - - - - - .reduce(0, Integer::sum);
```

(Bcos a1,b1 are Integers and Integer has a method called sum)

we can't write any method here
Eg s.o.print . . .

Bcos we are FOLDing here.

But we learnt about boxing unboxing and said though we can hv Predicate<r> but also IntPredicate

Similarly, we have methods like mapToInt.

Eg: menu.stream().map(a -> a.getCalories())

.mapToInt(a -> a.getCalories())

reduce can be applied on the directly

saves the boxing cost

gives a stream of Integers

(first read ptr where elem is stored & then points to that value)

gives you an Int Stream

(hv one less memory access/read for each element)

So when we are sure that we get a stream of Int, we can directly do func^n like sum();
avg();
⋮

No need to reduce

Eg: .... . mapToInt (a → a . getCalories()) . sum();

But this kind of specialized operation (to reduce boxing cost) is for only Int, Float, Long.

(bcos these are most frequently used & boxing overhead highest for these)

---

③ count → returns no. of elem in stream
↓
thats why its a Terminal operator.

Eg:

```
long noItems = menu.stream().filter(dl -> dl.getType() == Dish.Type.FISH).count();
```

Stateful / Stateless

→ For filter, map, flatMap we don't need reduce

Eg: filter even no. of list {12, 13, 14 ...}

```
numbers.st....filter(a -> a%2==0).collect(toList());
```

⇓

it doesnt need any memory to store the state value (& thus completely stateless operation)

(∵ don't need to remember what you've done with prev item)

Same for map & flatMap

→ For limit, skip, reduce, we need to remember how many elem we hv taken before.

So these op. are stateful
⇓
need to maintain state

need to know what you've done with prev elem

Same goes for count → stateful as well

distinct → take unique elem from list
sort → arranges elem of list
} definitely stateful

But there's a diff. with the kind of state:

reduce & distinct needs to maintain

reduce → remember just 1 value / 1 object
(eg. 1 int value that stores summation)

Same for count → hv to remember just current value of count

→ So, here states are bounded
⇓
Though it needs to remember, but how much it needs to remem. is constant
(i.e. just 1 value for sum/count)
⇓
No need to remember individual elem

limit, skip → are also bounded
⇓
hv to remember the value of no. of items passed
(No need to remember which items passed)

Hv to remember all the distinct elem in the list, in order to ensure new elem is distinct / not

Eg: {5, 10, 15, 10, 12, 20, 21}
distinct needs to keep track of : {5, 10, 15}
so that it understands 10 is a duplicate

So, state is unbounded
(not bcos states are infinite)
But amt of data u need to store, Depends on the size of the data str what you have

In case of distinct hv to remember distinct elem

In sort, hv to remember where you hv placed the prev elem.

Eg. numbers.stream()

numbers.stream().map(a → 1).reduce(0, (a1, b1) ⇒ a1 + b1)
or can do (a1 + 1)

hv to map bcos reduce takes bin operator

→ I wana know how many elem in this stream by map & reduce only (w/o count)

Scanned by CamScanner

OR if stream is integer, no need to map

Q Implement count w/o using map(), count() HW

08/02/19

Methods
- Intermediate → stateless
- Terminal
  - stateful
    - bounded (does not depend on size & how many elem. we hv to process)
    - unbounded
  - stateless

Building Streams :

- Bounded Streams (string will always hv a fixed no. of elem. in it)
- Infinite Streams

Bounded Streams

Eg: of (a static method of Stream)

An interface not only hv abstract methods, but also default methods

Also can hv static methods

Eg: parseInt (in Integer)

We can straight away put utility methods in Interface itself

① Stream.of ("Hello", "world", ...)

② Stream() — Array, collection

since collection is bounded, these 2 methods will always generate a bounded stream

③ — Files (read content from file & put it in a stream

But for file, we hv to use nIO (non-blocking IO)

Eg: long uniquewords = 0;

splits into lines & reads from that

```
try {
    Stream<String> lines1 = Files.lines(Paths.get("abc.txt"),
                                        Charset.defaultCharset());
    uniquewords = lines1.flatMap((lines2 -> Arrays.stream(lines2.split(" "))
                                 )).distinct().count();
    S.o.p("Unique words are " + uniquewords);
}catch (Exception e){}
```

convert array of word to stream

this returns array of words.

⇓

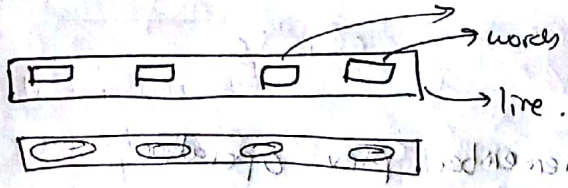
(★)

instead of space, if we give ("") then we get distinct chars.

→ If I used lines1 = Files.lines
lines1.distinct()

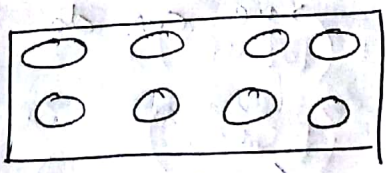
this would give distinct lines.

Similarly by PTO, we get duplicate chars



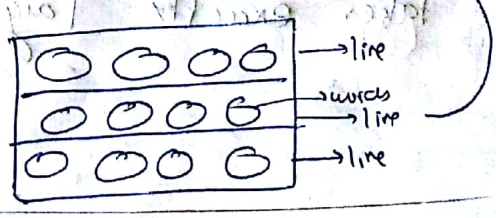
from this we need bounded collection to chck distinct words →

FlatMap takes diff stream elem, clubs them as 1 stream elem



flatMap converts it to this.

(★) this gives a stream of streams ⇒



Q I wanna find duplicate words only using stream.

A. List<String> word1 = Arrays.asList("Hi", "Hello", "Hi", "Hell")

Set<String> allItems = new HashSet<>();

Scanned by CamScanner

whenever we wanna add duplicate item to a Set, it gives False

for 2nd "Hi", this will be True

```
List<String> duplicates = word.stream()
                .filter(n -> (!allItems.add(n)))
                .distinct().collect(Collectors.toList());
S.O.P(duplicates);
```

Bcos for 5 "Hi" in word, 4 occurrences will be there after filter

Q Find duplicate words in a stream (prev pg)

count of distinct words

```
Map<String, Long> collect = words.stream().collect(groupingBy(Function.identity()));
```

Infinite Streams

Theoritically it's inf

both are static methods of stream

① iterate ← takes unary operator

② generate ← takes Supplier func. Interface

But how to limit stream using some short-cutting

iterate (unbounded, sequential bcos result dependent on prev. operation)

Iterate can remember prev. operation

So any kind of AP, list of even no., list of odd no., multiples of 10,

→ we can do with iterate

generate (doesn't apply successively a func on a newly processed value)

takes exactly 1 arg (supplier arg)

↓

functional interface

↓

$\boxed{\text{Supplier<T>}} \rightarrow T$

() → T

Eg: get()

Stream.generate (Math::random()).limit(10).

forEach(a → s.o.p(a)

OR

System.out::println);

will not remember what it has done in prev. run.

Print Fibonacci num using iterate

Iterate takes Unary Operator : T → [Unary Operator <T>] → T

But Fibo num is intercontly binary, we h/v to take 2 numbers.

0, 1, 1, 2, 3, 5, 8, 13, 21

So, stream.iterate(___ lambda expr.)

initialization of accumulator

In reduce we could do (2 var a&b) (a,b) → a+b But here we can use only one

We will modify a from simple int variable to an array having 2 elem

Stream.iterate(new int[]{0,1}, t → {t[1], t[0]+t[1]}

.limit(20).forEach(t1 → s.o.p(t1));

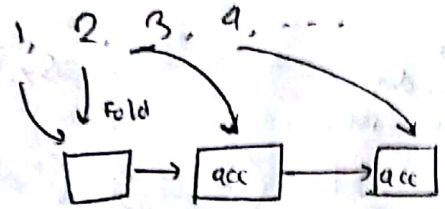
20 Fibo num.

t → {t[1], t[0]+t[1]}

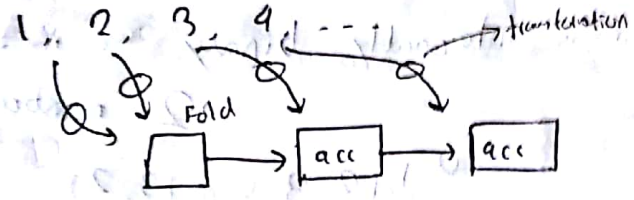
.map(t → t[0])

CT till here

Stream (reduce) 19/02/19



But with collect, we can also apply a transformation func^n and then you can apply fold



3 diff type of func^n, collect can do

→ reducing & summarizing stream elements to a single value

→ grouping elements

→ special case of grouping i.e. partitioning elements

Note, you can also have

① ..... collect (toList());

② (toSet())

③ (toCollection(TreeSet::new))

you have to instead ... (∵ it doesn't know what to expect)

Collector interface has 3 things

Collector <T, A, R>

T → used for traversing; generic type of stream elements

A → accumulator type

R → resultant type; generic

Scanned by CamScanner

reduce takes BinOp → i/p & variable used for accumulation will be of same type

But here since we can hv transformation & then fold (traverse), the accumulator can be of different type

→ when it is not applied specifically, it is identity func^n

---

① **Reducing & Summarizing** → most often returning a particular obj or int, double, float or a string — a single val

I want count no. of dishes in the menu

```
menu.stream().map(d -> 1).reduce(0,(a,b) -> a+b);
```

OR

```
menu.stream().count();
```

Using collect →

```
menu.stream().collect(counting());
```

a static method defined in the collectors class.

Collector is an interface
Collectors is a class

I want know the dish that gives you maximum calorie:

```
Optional<Dish> R = menu.stream().reduce((a,b) -> (a.getCalories() > b.getCalories() ? a : b));
```

Using collect:

We'll use maxBy. → takes a comparator

↓

```
maxBy(Comparator<T>)
```

→ a functional interface that has exactly 1 abstract method. The name of the abs. method we'll be using here is: ComparingInt (∵ we'll be comparing Integer)

Comparing is the basic method

↓

hv some overloaded func^n

↓

ComparingInt, ComparingLong, ComparingDouble

Scanned by CamScanner

menu.stream().collect(maxBy(Comparator.comparingInt(x -> x.getCalories)));

the field on which we'll be going to compare.

kinda map

Similarly can use minBy.

↓

But empty string returns NULL

So Optional<Dish> R' =

If you don't want Opt Dish, but want Dish obj,

Dish D =))).get();

SUMMARIZING

we want total summation of calorie values

Optional<Integer> C = menu.stream().map(d -> d.getCalories()).reduce((a,b) -> a+b);

Using collect,

int R' = menu.stream().collect(summingInt(d -> d.getCalories()))

OR

Dish::getCalories

averagingInt
summingDouble
summingLong
averagingDouble

Data type for Summarizing Int

⇓

IntSummaryStats result = menu.stream().collect(summarizingInt(Dish::getCalories));

Special method that gives count, sum, min, avg, max

S.o.p(result)

o/p is COUNT = ... min = ... max = ...
sum = ...
avg = ...

Scanned by CamScanner

S.o.p (result.getMin());
S.o.p (result.getMax());

We can also straightaway write,

Int min = ------ .collect(summarizingInt(Dish::getCal)
.getMin()

By this we can get summary of commonly used stats.

JOINING

want ~~[illegible]~~ string that'll mention all the veg dishes in the menu

Using reduce

String joinedResult2 = menu.stream().map(Dish::getName).reduce((a,b) -> a+", "+b
.get();

S.o.p. (joinedResult2);

Using collect

String joinedResult1 = menu.stream().map(Dish::getName)
.collect(joining(", ", "[", "]"));

S.o.p. (joinedResult1);

delimiter, first of string, last of string → overloaded version → can ignore

So o/p =

[salmon, fish, chicken, ...]

---

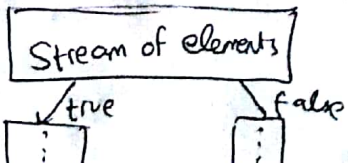
(3) PARTITION (can categorize into two) → we get two sublists (unlike (1))

Map<Boolean, List<T>>

key, value

always Boolean (∵ only 2 possible sublists)

It returns a stream of elem into 2 divisions



Scanned by CamScanner

```
Map<Boolean, List<Dish>> mapResult
    = menu.stream().collect(partitioningBy(d -> d.isVegetarian()));
```

o/p: (false = [chicken burger, chicken sandwich], true = [veg burger, ...])

in fact can use any Predicate here

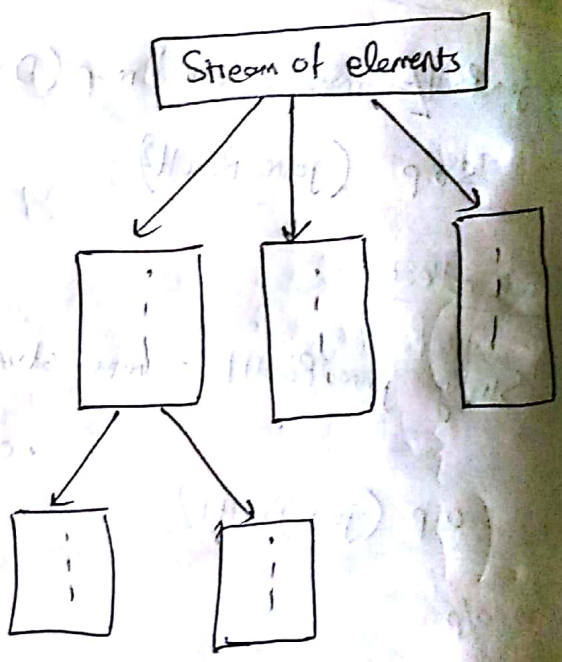
---

④ GROUPING (more generic than Partitioning)

↳ not only you can have 2 sublists, but you can have multiple sublists if you want.

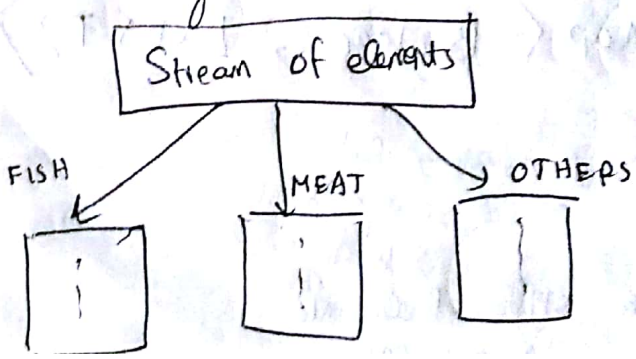
- → can apply Transformation
- → can use Multi-level grouping

Here when we are dividing into more than 2 types, key may not be Boolean (unlike partitioning)




```
Map<Boolean, List<Dish>> result2 = menu.stream().collect(grouping By
                                    (d -> d.isVegetarian()));
```

Say we wanna group into:



Scanned by CamScanner

menu.stream().collect(groupingBy(d -> d.getType()));

Map<Dish.Type, List<Dish>> =

which was already defined in Dish class. Only bcos of this, we are able to categorize into these 3 types.

But lets say, we wana divide according to calorie values

This was not defined in Dish class.

→ LOW
→ NORMAL
→ HIGH

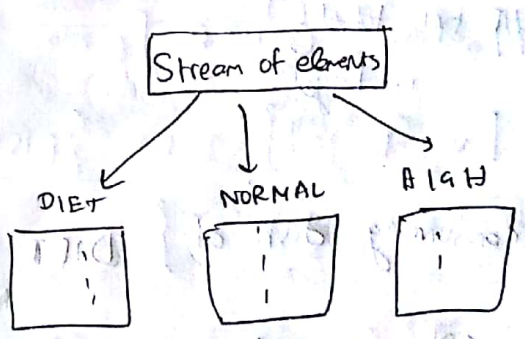
Say even if we hv ENUM type:

CalorieLevel
{ DIET
NORMAL
HIGH }

→ But we hv not defined which dish falls in which category → So hv to define directly

→ menu.stream().collect(groupingBy(d -> {if (d.getCalorie() > 400)
return(CL.HIGH);
else if (d.g....<400 && ....>=150)
return(CL.NORMAL);
else
return(CL.DIET);
}));

CL ≡ CalorieLevel ≡ Enum Type



Map<CL, List<Dish>> R =

Say we want how many dish items we have that are Low calorie, that are high - - - - -

Scanned by CamScanner

→ i.e. You are now summarizing, but not on the entire stream, but each subcategory i.e. each substream.

So,

```
menu.stream().collect(groupingBy(d -> { ....
                                      }), counting()));
```

```
Map<CL, Long> M =
```

↓ if it was summarizingInt() then IntSummaryStats

summingInt()
summarizingInt()

this is an overloaded version.

If we want a set of Dish elements,

```
Map<CL, Set<Dish>> M
    = menu.stream().....collect(grouping By
                                (d -> { ...
                                      }), toSet()));
```

upto grouping By its Lazy
after that toSet() is Eager.

duplicate items will hence be removed

---

we want max calorie consuming dish of DIET, ~~LOW~~ NORMAL, HIGH

```
Map<CL, OptionalDish
    <Dish>> R = menu.stream().collect(grouping BY(d -> {
                                                  }),
                                      maxBy(Comparator.ComparingInt
                                            (d -> d.getCalorie()))));
```

this is a Map can't use .get()

returns Optional<Int>

Scanned by CamScanner

But here, there's no point in doing Optional<Dish>

↓

we do this when we aren't sure stream is not empty

↓

But here we are going for multilevel categorizing, so we know for sure stream is not empty

← we can go for a wrapper CollectingAndThen

⋮

→ Map<CL,Dish> = menu.stream().collect(groupingBy(d → { ? }),

collectingAndThen(maxBy(...),

Optional::get))));

↙ wraps the (here) Optional::get together & gives you a valid result

can use for only those cases when you are sure, stream is never empty.

→ You can club the different grouping together

↓

Multi-Level Grouping

Say we want names of Veg dishes

Map<Boolean, List<String>> result = menu.stream().collect

(groupingBy(d → d.isVegetarian(),

mapping(d → d.getName(), toList())));

↓

maps the dish obj to name & gives you back the List

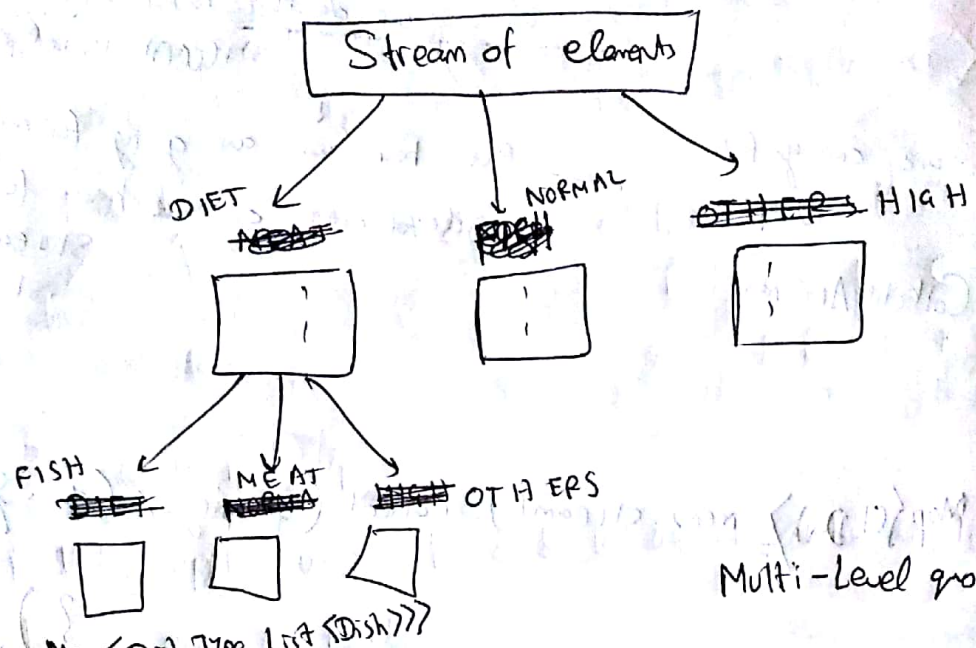
Sum of each categories

(groupingBy(d → d.getType(),

summingInt(d → d.getCalories())));

Scanned by CamScanner

Say we want :



Multi-Level grouping

Map<Category, Map<Dish.Type, List<Dish>>>

dishesByCal = ... collect (grouping By (dish → {

if (dish.getCalories() <= 4

return DIET.

}, grouping By (d2 → d2.getType()))); ☆

as a 2nd argument of collect

So o/p :

{ NORMAL = {FISH = [salmon], OTHER = [pizza, ...],

MEAT = [veg burger]}, DIET = {FISH = [prawns],

.... } .... }

Say we want first level grouping By (Fish, Meat other)

then 2nd level by (Diet, Normal, High)

So we will interchange the two grouping By in ☆

Here, return type would be :

Scanned by CamScanner

Map< ~~Category~~, Map< ~~Dish~~ Category, List<Dish>>)
Dish.Type

We can also do reducing inside collect

menu.stream().collect(reducing(0, (a,b) → a+b))

So we can use reduce on the multi-level grouping

So → collect is a stronger construct
→ multiple ways of doing one thing

No transformation in reduce. But in collect you can transform elem to some other values before folding

But it comes with a cost

⇓

For parallel Stream reduce is much more suitable, bcos its not mutable

But collect is not readily parallelizable, bcos mutable op is used.

But reduce cant give multilevel Categorization that collect can do

---

22/02/19.

Scanned by CamScanner

CURRYING. If your method has n no. of arg, call the method with less than n no. of arg, to get partial result. (used in conversion of metrics etc.)

$a + bx$ → decide whether ur func^n converts Fahrenheit to C

baseline conversion factor

or Miles to K.M.

But we'll not i/p x

Partially apply func^n to a & b, which when applied totally gives a number.

Eg: $f(a, b, x)$

$= \underbrace{(g(a, b))}_{\text{partially apply}}(x)$ → we'll apply it totally when parameter is maintainable

(it'll not give me a no. it gives me ans the func^n)

But, In actual func^{nal} prog. you apply one arg at a time

i.e first a, then b, then x ---

Curried Converter :

```
static ____ curriedConverter (double a, double b){
                    return (a + b*x);
}
```

This method doesn't know about x

return type of this method has to be a functional interface → which we can assign to a lambda expression.

Eg: Function <T, R> ⇒ <R> apply(<T>)

Eg: a = curried Converter (5,6);

a. _____ (5); If a is of type Function, then apply.

functional interface — method of that func^nal interface

(★) If x is of type Float, then only it'll work
If String, char it'll not work
If we use Function <T,R>, it supports different datatype for i/p & o/p, so can't be used (we must ensure i/p & o/p types are same ((a + bx) → float, x → float))

So we use UnaryOperator / BinaryOperator

→ So, static UnaryOperator CurriedConverter (double a, double b){
return (a + b * x);
}

But we just want a value, we don't want to use method of the object (we don't hv anything to do with objects), so boxing is a overhead. So, we do:

✓ static DoubleUnaryOperator curriedConverter (double a, double b){
return (a + b * x);
}

Eg: Centigrade to Fahrenheit
DoubleUnaryOperator CtoF = curriedConverter (32, 9/5);

KM to Mile
" KmToMile = " (0, 0.6214);

So using same method, we form numerous converters.
Now, If we want to get the exact value,

double F = CtoF.applyAsDouble (12);

Scanned by CamScanner

Def^n of Currying: Currying is a technique where a function of 2 arguments (x and y, say) is seen as a function, g, of 1 argument, that returns a function, also of 1 argument.

$$\boxed{f(x,y) = (g(x))(y)}$$

Compose, and Then

But whether to use y first, or use x first.

Eg:

```
Function<Integer, Integer> add3 = (a) -> a + 3;
Function<Integer, Integer> times2 = (a) -> a * 2;
// Compose add with times
Function<Integer, Integer> composedA = add3.compose(times2);
// first applies arg to add then times 2
Function<Integer, Integer> composedC = add3.andThen(times2);
                                (ans = 18) => 9x2
// compose times with add
Function " " composedB = times2.compose(add3)

S.o.p("Times then add : %d\n", composedA.apply(6));
                                    //(6*2)+3
S.o.p("Add then Times with andThen %d\n", composedC.apply(6));
S.o.p("Add then times: %d\n", composedB.apply(6));
                                    //(6+3)*2
```

First argument will be evaluated, then it'll apply the method.

Scanned by CamScanner

→ Theoritical def^n of currying says that you apply 1 arg at a time

Eg:

note arg is an integer

but return type is resting
Bcos no flat method defn.
It returns another func^n itself

Function<Integer, Function<Integer, Function<Integer, Integer>>> ☆

curryAdd = (a) → (b) → (c) → a + b * c

Effectively, it is similar to:

(a) → {(b) → {(c) → {a + b * c}}}

one arg — corresponding method — corresponding method — not arg

Bcos we are resting here, so return type should also show resting.

Note: this can't return a func^n

<R> apply <T> {
return T+1;
}

Function<Integer, Integer>

a = (a1) → a1 + 1

goes to def^n of

☆ Eg: a1 → b1 → a + b1

(∵ a full fledged method def^n)

return type : Function<Integer, Function<Integer, Integer>>

→ This is how JAVA does theoritical concept of currying

PARALLEL STREAMS

2 ways of making a stream parallel:

1. menu.parallelStream().filter(...)
2. menu.stream().filter(...).parallel().map()
3. ..... parallel().map()...

(after map then sequential. Then again parallel)

. sequential()...

. parallel()...

⇓

But not necessarily the exact sequence will follow. The last call (terminal o/p) dominates the pipeline in most of the cases.

→ No of threads it'll create = no. of processors you have on your machine

→ But if you chng default value of thread, it'll be changed for all processes on that JVM

Scanned by CamScanner

Eg:

① long startTime = System.nanoTime();
long result1 = Stream.iterate(1L, i → i+1).limit(2500)
.parallel().reduce(0L, Long::sum);
(or a → a+b)

long duration = (System.nanoTime() − startTime) / 1_000_000;

S.o.p ("Sum=" + result1 + " Done in parallel in" + duration + "ms");

o/p:
29 ms
3 ms.

---

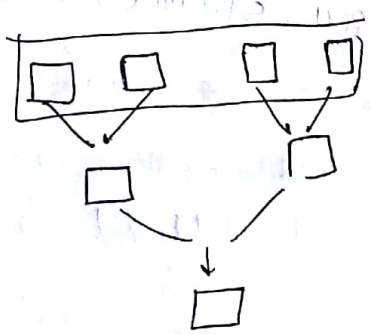
②
long startTime = System.nanoTime();
long result2 = LongStream.rangeClosed(1, 2500)
.parallel().reduce(0L, Long::sum);

long duration1 = " " — —

S.o.p (. — — — — duration1 — — —)

2nd one is much faster bcos iterate is not parallel friendly
→ You can apply parallel o/p iff you hv the entire stream of nos. available to me

Eg:



Iterate have a state & generates no. one by one → so not ∥ friendly

Also note, in ①, there is a cost of boxing
(we are making with primitive value (so we can do a,b a+b))
then we again box it to obj (each time box & unbox)

Scanned by CamScanner

in ②, it is LongStream (so it remains unboxed only)
So no cost of boxing, unboxing

LongStream.rangeClosed(. . .)
⇓
parallel friendly ⇒ we hv the entire range of nos.

Another method that's not parallel friendly: findFirst

① can't get extra advantage if we use this in ∥ stream

② can give different values in parallel stream

→ Methods that apply on ordered collection are not parallel-friendly (eg. hv to execute first part first only)

[If collection is unordered, limit is a ∥ friendly method
ordered, " " not " " "]

06/03/19

LAMBDA-CALCULUS

→ 2 funcn. are extensionally equal if they hv the same i/p - o/p behaviour

→ 2 funcn are intensionally equal if they are given by (essentially) the same formula

2 key words: '.' & 'λ'

Eg: $\lambda x . x$ ≡ int func (int x){
return x;
}

identity func^n

func(5) → returns 5
func(7) → return 7

Scanned by CamScanner

$$f: \underbrace{\mathbb{N} \times \mathbb{N}}_{\text{domain}} \longrightarrow \underbrace{\mathbb{N}}_{\text{range}}$$

A func^n is a mapping

(1) So when we say $f(x)$ & $g(y)$ are equivalent

$\Downarrow$

if you pick up any pt from domain, it'll map to same value of range (for f & g)

EXTENSIONAL BEHAVIOUR (compare 2 func^n by i/p - o/p behaviour)

---

(2) Another way of checking equivalence → INTENSIONAL BEHAVIOUR

define rules → How they calculate

Eg: $f(x) = x^2$, $g(y) = y^2$ } intensionally equal

Here we ignore → type of the variable.

But since they follow same set of rules, complexity is same

So for CS, it is more useful & more intuitive to follow these notion

---

(3) Graphs. → easier to visualize, func^n as graph rather than i/p-o/p behaviour OR rules they follow

So this notion is more closer to Mathematics.

Intuitive Computability

given an x part, can we compute that particular func^n after some finite time

Scanned by CamScanner

An ~~intu~~ intuitive def^n of computability is that there should be a pencil & paper method allowing a trained person to calculate $f(n)$, for any given m/c

① Turing Machine:

funcns that can be represented/solved by Turing m/c is intuitively computable (we are not saying the opposite)

② Godel → proposed the concept of "General Recursive functions"

↳ as the smallest set of funcn:

→ all the constant funcn, the Successor funcn & closed under certain operations (such as Compositions & recursion)

→ a func^n is computable (in intuitive sense) iff it is generally recursive.

Eg:

```
int y=5
f(x) {
return x*y
}
```

→ Not closed as depends on external variable.

③ Lambda Calculus

whatever func^n can be represented by Lambda calculus are intuitively computable.

also proves that Lambda Calculus is Turing complete.

NOTE: all of ① ② ③ are equivalent ⇓ just explored in their own direction.

Scanned by CamScanner

$\lambda x . E \Rightarrow$ func^n creation/abstraction

$(\lambda x . E) E_2 \Rightarrow$ func^n application

(passing E_2 as an argument to this func^n)

Eg: want to apply an argument y to identity func^n

$$(\lambda x . x)\, y \xrightarrow{\text{reduced to}} = y$$

Eg: $(\lambda x . x)(\lambda y . y)$ → this entire expression is placed in x

$$= \lambda y . y$$

Order of evaluation

$\lambda x . E_1 E_2 E_3 E_4$ → which one will be evaluated first?

we can use parenthesis

↓

but this is not a keyword ← can use for clarity

Eg:

$\lambda x . (((E_1 E_2) E_3) E_4) \Rightarrow$ left associative

→ Lambda calculus always takes exactly one argument

$\lambda xy . xy \Rightarrow$ not 2 parameter

⇓

It actually maps to:

$\lambda x . \lambda y . xy$ ← can never take more than one parameter.

Scanned by CamScanner

But then, how can we represent func^n that takes more than 1 parameter. Eg:

```
f(x,y){
    return x*y;
}
```

So we use currying type thing

$$(\lambda x.\lambda y.xy)\,a \longrightarrow (\lambda y.ay)$$

$$\Downarrow$$

$$ab \longleftarrow (\lambda y.ay)\,b$$

We address multiple param by step-by-step composability.

---

have (to write this defn only)

Def^n of Lambda Calculus

The central concept in λ calculus is the "expression". A "name" also called a "variable", is an identifier which, for our purposes, can be any of the letters a, b, c, ...

3 types of constructs:

$\longrightarrow$ $\lambda x.E$ (basic abstraction)

$\longrightarrow$ $y \mid \lambda x.E \mid (\lambda x.E_1)E_2$

$\longrightarrow$ $E_1 E_2$ (composability of 2 exp)

Eg:

```
f(x){
    return x;
}
```

scope of x. x is bound to f

Similarly $\lambda x.xy$ → not bound to f. free variable

x is a bound variable in this exp.

Scanned by CamScanner

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

bound

free

y is a bound variable

For a lambda calc exp., we hv a variable bound somewhere & free somewhere else

If you don't hv any free variable in Lambda Calc. Exp. then <u>closed</u>.

Eg: $\lambda x . x$

So, $y \mid \lambda x . E \mid \underbrace{(\lambda x . E_1)}_{\text{rator (operator)}} \underbrace{E_2}_{\text{rand (operand)}}$

$\lim_{n \to \infty} \frac{1}{n}$ → free variable

concept of bound & free variables is taken from Mathematics.

<u>Defn of free & bound variable.</u>

① $\text{Free}(x) = \{x\}$

② $\text{Free}(E_1 E_2) = \text{Free}(E_1) \cup \text{Free}(E_2)$

Eg: $(\lambda x . xy)(\lambda y . y)$

y is free in this expression

y is bound here.

y is a free var in $E_1 E_2$ does not mean y is free in both E_1 & E_2

③ Free $(\lambda x . E) = \text{Free}(E) - \{y\}$

Eg: Free $(\lambda x . x (\lambda y . xyz)) = \{z\}$

NOTE: When we apply variable it is applied to the rightmost side

Eg: $\lambda x . x \lambda y . z x$

bound

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

free here

$\lambda y . yy$ so it gets bounded here

Not correct as it was an external variable

In order to solve this ambiguity, rename the variables

$(\lambda x . xy)(\lambda z . z)$

$= \lambda z . zy$

Now y remains free, Not bound anymore

① Renaming, keeping the meaning fixed : α - reduction

② applying a particular expression (func^n application) : β - reduction

When a func^n is not at all β - reducible, we call it to be in its NORMAL FORM

Scanned by CamScanner

Eg: $(\lambda x.x)(\lambda z.z)$

$= \underline{\lambda z.z}$ a normal form

$(\lambda x.x)y$

$= \underline{y}$: a normal form

→ So can't apply β reduction as many times. We'll apply until a normal form comes.

→ α-reduction only if there is a name clash problem

All properties of equivalence satisfy α-reduction.

So we say equivalence to each other

$E_1 \to E_1' \qquad E_2 \to E_2'$

$E_1 E_2 \to E_1' E_2'$ (congruence)

$(\lambda f.f\ (\lambda x.x))(\lambda x.x)$ Evaluate this.

$\xrightarrow[((x/z)\lambda x.x)]{} (\lambda f.f\ (\lambda x.x))(\lambda z.z)$

$\xrightarrow[\alpha]{} (\lambda f.\underline{f}\ (\lambda x.x))(\lambda z.z)$

$\xrightarrow[\beta]{} (\lambda z.z)\underbrace{(\lambda x.x)}_{\text{treated as an argument}}$

$\xrightarrow[\beta]{} \lambda x.x$

For step-jump, write $(\xrightarrow{*}\beta)$

Scanned by CamScanner

Eg: $(\lambda x.xx)(\lambda x.xx)$

$$\xrightarrow[\alpha]{} (\lambda x.xx)(\lambda y.yy)$$

$$\xrightarrow[\beta]{} (\lambda y.yy)(\lambda y.yy)$$

$$\xrightarrow[\alpha]{} (\lambda y.yy)(\lambda z.zz)$$

$$\xrightarrow[\beta]{} (\lambda z.zz)(\lambda z.zz)$$

$\vdots$

So, not all lambda expressions can be reduced to a normal form. Here always β-reductions are possible.

Q Can every lambda exp be reduced to a normal form $\longrightarrow$ NO

Q

Is there more than one way to reduce a particular lambda expression?

$$(\lambda y.5)((\underbrace{(\lambda x.xx)}_{E_1}\underbrace{(\lambda y.xx)}_{E_2}))$$

One way: ①

ans: 5

① (rightmost part)

inf. loop (call by value)

5 (call by name) (this way can cates more no. of func. evaluation)

another way ②

first we reduce E_1E_2 part & then apply

If we do that, we fall in a loop again

So evaluation can start from rightmost / leftmost.

Scanned by CamScanner

Eg: getResult (getRoll(), getName())

Normally we evalue arg first then apply to method

if we do this ← (call by value)

in prev. eg ⇒ inf. loop.

call by name

first apply, then evaluate arg

To deal with mutable variables, only resort is call by value.

$\lambda f.\lambda y.fy$ 08/03/19

$\lambda f.\lambda y.fy$

bound variable

so, closed ⟸ bound (∵ f is passed as parameter & f is in the scope of this

But $(\lambda f.\lambda y.fy)f$

free variable.

Scanned by CamScanner

$(\lambda f.\lambda y.\text{add}\ f\ y)\ 5$

$\rightarrow \lambda y.\text{add}\ 5\ y$

β reduction

β-abstraction

β-reduction

$(\lambda v.E)\ E_1 \Rightarrow_\beta E[v \rightarrow E_1]$

$(\lambda f.f\ (\lambda x.x))(\lambda x.x) \rightarrow$

$[\lambda x.x/f]f(\lambda x.x)) = [(\lambda x.x)/f]\ f(\lambda y.y)) =$

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

$[\lambda y.y/x]\ x = \lambda y.y$

---

2 types of reduction

Normal order reduction → always reduces the leftmost outermost β-redex (or δ-redex) first

A normal exp is where no more β-reduct. or (δ-redex) possible.

Applicative order reduc.

always reduces the leftmost innermost β-redex (or δ-redex) first.

Scanned by CamScanner

So Eg: $(\lambda y.5)((\lambda x.xx)(\lambda x.xx))$

leftmost outermost. leftmost innermost

Correspondingly, there will be 2 paths to reduce

If there is more than one reduction strategy, does each one lead to the same normal form expression?

→ In a λ-term, there could be more than one instance of $(\lambda x.E)E'$.

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

→ could reduce the inner or outer λ

→ which one should we pick? this bracket is important

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

inner

outer.

$$(\lambda y.[y/x]E = (\lambda y.y)E \qquad [E/y](\lambda x.x)y = (\lambda x.x)E$$

$$\rightarrow E \leftarrow$$

Order of Evaluation (cont.)

The Church-Rosser Theorem says that any order will compute the same result

→ A result is a λ-term that cannot be reduced further

(DIAMOND PROPERTY (confluence))

Church Rosser Theorem I : For any lambda exp. E, E & G,

if $E \Rightarrow^* F$ & $E \Rightarrow^* G$, there is a lambda exp Z such that $F \Rightarrow^* Z$ and $G \Rightarrow^* Z$

$$\begin{tikzcd}
 & E \arrow[dl, Rightarrow, "*"'] \arrow[dr, Rightarrow, "*"] & \\
F \arrow[dr, Rightarrow, "*"'] & & G \arrow[dl, Rightarrow, "*"] \\
 & Z &
\end{tikzcd}$$

<u>Corollary</u>: For any lambda exp E, M and N, if $E \Rightarrow^* M$

A normal order reduction can have either of the following outcomes

① It reaches a unique (up to a-conversion) normal form lambda expression

② It never terminates

<u>Church-Rosser Theorem II</u>: For any lambda expressions E and N, if $E \Rightarrow^* N$ where N is in normal form, there is a normal order reduction from E to N.

Unfortunately, there is no algorithmic way to determine for an arbitrary lambda expression which of these 2 outcomes will occur

---

The <u>Conjecture</u>

<u>Church's Thesis</u>: The effectively computable functions on the <u>positive integers</u> are precisely those functions definable in the pure lambda calculus (and computable by Turing machines)

Scanned by CamScanner

→ The conjecture cannot be proved since the informal notion of "effectively computable func^n" is not defined precisely.

→ - - -

Is there a reduction strategy that will guarantee that a normal form expression will be produced?

→ ?

→ halting problem

→

→

Call by Name

→ Same as normal order reduction

→ do not evaluate the argument prior to call (lazy evaluation)

Eg:

$$(\lambda y.(\lambda x.x)y)\,\underline{((\lambda u.u)(\lambda v.v))} \to_{\beta_n}$$

$$(\lambda x.x) \ldots$$

Call by Value

→ same as applicative order reduction

→ evaluate an argument prior to call

Eg:

$$(\lambda y.(\lambda x.x)y)((\lambda u.u)(\lambda v.v)) \to_{\beta_v}$$

follow β_v in all steps

$$(\lambda y.(\lambda x.x)y)(\lambda v.v) \to_{\beta_v}$$

$$(\lambda x.x)(\lambda v.v) \to_{\beta_v} \lambda v.v$$

Scanned by CamScanner

Call by Name and Call by Value

CBN
→ difficult to implement
→ order of side effects not predictable

CBV
→ easy to implement efficiently
→ might not terminate even if CBN might terminate
→ Eg: $(\lambda x.\lambda z.z)$ - - - -

Def^n: η - reduction

If v is a variable, E is a lambda exp. (denoting a func) & v has no free occurrence in E

$\lambda v.(E\,v) \Rightarrow_\eta E$

$\lambda x.(sqr\ x) \Rightarrow_\eta sqr$

$\lambda x.(add\ 5\ x) \Rightarrow_\eta (add\ 5)$

(*) $\boxed{\lambda x.(5\,x) \text{ is not } 5}$ → constant. Not an expression

Def^n: δ - reduction

We will not use in classes

If the lambda calculus has predefined constants (i.e. if it is not pure), rules associated with those predefined values & functions are called δ rules;

Eg: $(add\ 3\ 5) \Rightarrow_\delta 8$ and $(not_true) \Rightarrow_\delta false$

The η-reduction rule can be used to justify the extensionality of functions; namely if: $f(x) = g(x)$ for all x, then $f = g$. (*)

Lambda Calculus & Programming Languages

→ Pure lambda calc. has only functions.
→ what if we want to compute with Boolean, no, list etc?

Scanned by CamScanner

→ All these can be encoded in pure λ-calc.

→ Need not encode what a value is

→ For each data type we have to describe how it can be used as a function

→ then we write that func^n in λ-calc.

Encoding Booleans in Lambda Calculus

→ we can make a binary choice.

→ true $=_{def}$ $\lambda x.\lambda y.x$ def $\lambda_{then_do}.\lambda_{else_do}.\ then_do$

→ false $=_{def}$ $\lambda x.\lambda y.y$ def $\lambda_{then_do}.\lambda_{else_do}.\ else_do$

→ if E_1 then E_2 else E_3 $=_{def}$ $E_1\ E_2\ E_3$

NOTE: Not equal

→ if_then_else $\equiv_{def}$ $\lambda_{cond}.\lambda_{then_do}.\lambda_{else_do}.\ cond\ (then_do)(else_do)$ ☆

```
if (condⁿ) {
    then_do
}
else {
    else_do
}
```

hv to reduce.

Eg: holiday = true

Wake-upTime = if_then_else (holiday)(nine)(seven)

Scanned by CamScanner

Lambda Calculus: An Introduction

12/03/19

adapted from lectures by Prof. of various universities

contd...

if cond^n is true

$$\circledast \; (\lambda_c . \lambda_{td} . \lambda_{ed} . c \, (td)(ed)) \, (true) \, (nine)(seven)$$

$$\Rightarrow_\beta \lambda_{td} . \lambda_{td} . \lambda_{ed} . true \, (td)(ed) \, (nine)(seven)$$

$$\Rightarrow \lambda_{td} . \lambda_{ed} . (\lambda_x . \lambda_y . x)(td)(ed) \, (nine)(seven)$$

$$\Rightarrow \lambda_{td} . \lambda_{ed} . (\lambda_y . td)(ed) \, (nine)(seven)$$

$$\Rightarrow (\lambda_{td} . \lambda_{ed} . td)(nine)(seven)$$

$$\Rightarrow \underbrace{(\lambda_{ed} . nine)}_{\text{constant func}^n} seven$$

$$\Rightarrow nine$$

it returns nine

Predicates

- $is_zero = (\lambda n . n \, \overbrace{(\lambda x . false)}^{\text{const func}^n \text{ that always gives False}}) (true)$
- $is_zero \, (\underbrace{zero}_{\text{a number}}) = true$, otherwise false

Opposite:

$$Opposite =_{def} \lambda_{cond} . \lambda_{then_do} . \lambda_{else_do} . cond \, (else_do) \, (then_do)$$

$\Downarrow$

opp of if-else

if true $\longrightarrow$ else block

if false $\longrightarrow$ then block.

Scanned by CamScanner

Eg: $(\lambda x.x)one$

$= one\ (\lambda c.\lambda td.\lambda ed\ c\ (ed)\ (td))\ (true)$

$\lambda td.\lambda cd\ (true)\ (ed)\ (td)$

$= \lambda td.\lambda cd.\ ed$

- $is_even = \lambda n.n\ (opposite)\ (true)$

  n no. of times applying opposite func^n to true.

- $is_odd = \lambda n.n\ (opposite)\ false$
- $is_even\ (zero)$

AND, OR, NOT $\longrightarrow$ using Lambda

Encoding Pairs in Lambda Calculus

The function "Pair" encapsulates two values in a given order; it is essentially the dotted pair notation in Lisp (cons)

define $Pair = \lambda a.\lambda b.\lambda f.f\ a\ b$

Head Tail

Two selector func^{n}s "Head" and "Tail" confirm that Pair implements the cons operation.

Scanned by CamScanner

define Head $= \lambda g.g\ (\lambda a.\lambda b.a)$

define Tail $= \lambda g.g\ \underbrace{(\lambda a.\lambda b.b)}_{\text{False}}$

Representing Lists

(is_empty, (1, (2, (3, (4, null)))))

Now the correctness of the def^n is verified by:

Given a (pair p.q) I want to hv head of the list

$$\text{Head}\ (\text{Pair}\ p\ q) \Rightarrow (\lambda g.g\ (\lambda a.\lambda b.a))((\lambda a.\lambda b.\lambda f.f\,a\,b)\ p\ q)$$

$$\Rightarrow_\beta ((\lambda a.\lambda b.\lambda f.f\ a\ b)\ p\ q)\ (\lambda a.\lambda b.a)$$

$$\Rightarrow_\beta ((\lambda b.\lambda f.f\ p\ b)\ q)\ (\lambda a.\lambda b.a)$$

$$\Rightarrow_\beta (\lambda f.f\ p\ q)\ (\lambda a.\lambda b.a)$$

$$\Rightarrow_\beta \quad \vdots$$

Lists

- A list may contain
  - → Nothing (empty)
  - → One thing
  - → Multiple things

Scanned by CamScanner

→ List contains 2 values :

- make_pair = $\lambda_{left}.\lambda_{right}.\lambda_f\, f\,(left)\,(right)$
- get_left = $\lambda_{pair}.\ pair\,(true)$
- get_right = $\lambda_{pair}.\ pair\,(false)$
- get_right (make_pair (three) (two)))

$$\left(\lambda_p.\ p\,(false)\right)\left(\lambda_l.\ \lambda_r.\ \lambda f.\ f\,(left)\,(right)\right)(three)\,(two))$$

$$\Rightarrow_{*\beta} \left(\lambda_p.\ p\,(false)\right)\left(\lambda_f.\ f\,(three)\,(two)\right)$$

$$\Rightarrow_{*\beta} \left(\lambda f.\ f\,(three)\,(two)\right)(false)$$

$$\Rightarrow_{*\beta} (false)\,(tree)\,(two)$$

$$\Rightarrow two$$

→ A list will have the form ~~#~~ (empty, (head, tail))

→ null = make_pair (true) (true)

→ is_empty = get_left

→ is_empty (null) would return true

Scanned by CamScanner

Lists

prepend = λ item . λ l . make_pair (false) (make_pair (item) (l))

not append

(∵ is_empty, (1, (2, (3, (4, null))))))

head/left is always a single element)

- non-empty lists

→ [3, 2, 1] ⇒ (empty=false, (3, (2, (1, null))))

→ # (false, (1, null))

appending one to it

• single_item_list = prepend (one) (null)

initially list is null

(☆) → # (false, (3, (2, (1, null))))

→ multi_item_list

= prepend (three) (prepend (two) (single_item_list))

→ head = λl. get_left (get_right (l))

→ tail = λl. get_right (get_right (l))

Q What is the lambda exp for extracting elem 2 from the list (☆) ?

A.

Lists

→ hours_day_1 = prepend (2) (null)

→ hours_per_day = prepend (three) (prepend (two) (hours_day_1))

→ head (tail (hours_per_day))

Scanned by CamScanner

Polymorphism :

→ Functions that allow arguments of many types, such as this identity function, are known as polymorphic operations

→ $((\lambda x.x)\ E) = E$

→ define Twice = $\lambda f.\lambda x.f(f\,x)$

→ If D is any domain, the syntax (or signature) for Twice can be described as :

→ Twice : $(D \to D) \to D \to D$

→ Given the square func^n, sqr : $N \to N$ where N stands for the natural numbers, it follows that

- (Twice sqr) : $N \to N$
- Is twice a higher order function ?

Twice $(\lambda n.(\text{add } n\ 1))\ 5$

- The mechanism that allows functions to be defined to work on a no. of types of data is also known as parametric polymorphism

Scanned by CamScanner

- What can we do with a natural no.?

→ we can iterate a number of times.

one $\equiv \lambda f.\lambda x.\, f\, x$

→ A natural no. is a func^n that given an operation f & a starting value s, applies f a no. of times to s:

variation of identity

$1 =_{def} \lambda f.\lambda s.\, \underline{f s}$

$2 =_{def} \lambda f.\lambda s.\, \underbrace{f(f s)}_{2 \text{ times}}$ & so on

So apply no; $n = \lambda x.\, \lambda f.\lambda x.\, \underbrace{f\, x}_{f(f(f \cdots)) \; n \text{ times}}$

$\Rightarrow \quad 0 = \underbrace{\lambda f.\lambda x.\, x}_{\text{false}}$

From is_zero

$(\lambda n.\, n\, (\lambda x.\, \text{false})\, \text{true})\, (\text{zero})$ — numbers

$\Rightarrow_{*\beta} (\lambda n.\, n\, (\lambda z.\, \text{false})\, T)\, (\lambda f.\lambda x.\, x)$

$\Rightarrow_{*\beta} (\lambda f.\lambda x.\, x)\, (\lambda z.\, \text{false})\, (\text{True})$

$\Rightarrow (\lambda x.\, x)\, \text{true} \Rightarrow \text{true}$ (for all other cases, false)

Scanned by CamScanner

Computing with Natural No.

$$1 \equiv \lambda sz.s(z)$$

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

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

15/03/19

True : $\lambda xy.x$

False : $\lambda xy.y$

if-then-else : $\lambda cxy.cxy$

Pair : $\lambda pxy.\ldots$

Natural no.

-ve. no. is basically a pair → first represents sign
2nd ----- nat. no.

$$1 =_{def} \lambda f.\lambda s.fs$$

$$2 =_{def} \lambda f.\lambda s.f(fs)$$

Successor Function :

$$succ \equiv f(number) \equiv \lambda fx.f(fx)$$

So, $1 \equiv \lambda fx.f(x)$

and thus any nat. no. can be represented as: $n \equiv \lambda nfx.\overbrace{nf(x)}^{n \text{ times } f}$

Scanned by CamScanner

how many times going to apply f on the argument

$\therefore \lambda_{nfx}.nf(x)$ zero — i.e applying f zero times on x.

$\Rightarrow \lambda_{nfx}.nf(x)(\lambda_{sz}.z)$

$\Rightarrow \lambda_{fx}.(\lambda_{sz}.z)\, f(x)$

$\Rightarrow \lambda_{fx}.(\lambda z.z)\, x$

$\Rightarrow \lambda_{fx}.x \equiv 0$

$\therefore$ Any no. n can be represented by $\lambda_{nfx}.nf(x)$

$\therefore$ Successor of any number

$\equiv \lambda_{nfx}.f(nf(x))$

If I wanna add m & n, I will apply succ. m on n.

$m+n \equiv mSn$

Addition:

def $\lambda m.\lambda n.\lambda f.\lambda x.m(f)(n(f)(x))$

$2S_3$ is def $(\lambda sz.s(s(z)))(\lambda wyx.y(wyx))(\lambda uv.u(u(u(v))))$

Addition / Multiplication

(int X int) → int (not pure lambda calculus

(int → int) → int (pure form, using curlying)

$\lambda x \lambda y.x+y$

Evaluate $((\lambda x.\lambda y.yx)3)$

Scanned by CamScanner

Multiplication

def $\lambda m.\lambda n.\lambda f.\lambda x.\ \underbrace{m\ (n\ (f))}_{m \text{ times applying } n(f)}\ (x)$ on x

Power

$$(m\ n)\ f(x)$$

$\underbrace{\quad}_{m^n \text{ times } f(x)}$

→ check not

Predecessor

→ the general strategy will be to create a pair $(n, n-1)$ and then pick the second element of the pair as the result.

→ make_pair = $\lambda_{left}.\ \lambda_{right}.\ \lambda_f.\ f(left)(right)$

→ make_pair (true) = left

→ make_pair (false) = right

2 powerful combinators in Lambda calc are ϕ and Y.

- ϕ combinator generates from the pair $(n, n-1)$ (which is the argument p in the func^n) the pair

$$(\underbrace{n+1}_{\text{successor of left element}},\ \underbrace{n}_{\text{left element}})$$

Scanned by CamScanner

- $\phi = \lambda p.\lambda f.f\,(\text{succ}\,(p\ \text{true}))\,(p\ \text{true})\ (4,3)$

  $\Rightarrow \lambda f.f\,(5)\,(4)$

So,
$(4,3)$
$\downarrow$
$(5,4)$
$\downarrow$
$(6,5)$.

→ (one, zero) → (two, one)
→ (three, two) → ....

→ If we start with (0,0), applying ϕ combinator we get

(one, zero)

successor of zero (left) — zero (left)

→ The predecessor of a no. n is obtained by applying n times the func^n to pair (f zero zero) & then selecting the second member of the new pair

→ $P = \lambda n.\ n\,\phi\,(\lambda f.f\ \text{zero}\ \text{zero})\ (\text{false})$

applying ϕ combinator n-times — left — right

~~Predecessor~~

Subtraction & Comparison

→ $\boxed{\text{Subtraction}}$: $m - n$ → applying n times the predecessor on m

$\lambda m.\lambda n.\ n\,\text{PRED}\ m$

Eg: $\lambda m.\ \text{two}\ \text{Pred}\ m$ → Pred m

$\Rightarrow \lambda m\ (\lambda f x.f\,(f(x)))\ \text{Pred}\ m$

Scanned by CamScanner

$\phi = \lambda p.\lambda f.f\,(\overbrace{succ\,(p\ true)}^{left\ elem})\,(\overbrace{p\ true}^{left\ elem})$ (4, 3)

$\Rightarrow \lambda f.f\,(5)\,(4)$

So,
(4, 3)
↓
(5, 4)
↓
(6, 5).

→ (one, zero) → (two, one)
→ (three, two) → ....

→ if we start with (0, 0), applying ϕ combinator we get

(one, zero)

one: successor of zero (left)
zero: zero (left)

→ The predecessor of a no. n is obtained by applying n times the func^n to pair (f zero zero) & then selecting the second member of the new pair

→ $P = \lambda n.\underline{n\,\phi}\,(\lambda f.f\ zero\ zero)\,(false)$

$n\,\phi$: applying ϕ combinator n-times
zero: left
zero: right

~~Predecessor~~

Subtraction & Comparison

→ $\boxed{\text{Subtraction}}$: m − n → applying n times the predecessor on m

$\lambda m.\lambda n.n\,PRED\,m$

Eg: $\lambda m.\,two\,Pred\,m$ → Pred m.

→ $(\lambda x.f(f(x)))\,Pred\,m$

Scanned by CamScanner

$\Rightarrow$ ~~$\lambda m.$ Pred~~ $(\lambda m.\ \text{Pred}\ (\text{Pred}\ (m)))$

$\Rightarrow m-2$ applying Predecessor 2 times on m

Comparison → Subtraction & → is zero.

Eg: is zero m $= \begin{cases} \text{true} & (\text{if } m \text{ is zero}) \\ \text{false} & (\text{if } m \text{ is not zero}) \end{cases}$

→ opposite than expected. Rather chng it to lesser Equal.

$(m \geq n)$

→ greater Or Equal $= \lambda n.\ \lambda m.$ isZero (subtract n m)

$(m \leq n)$

→ less Or Equal $= \lambda n.\ \lambda m.$ isZero (subtract m n)

$(m = n)$

→ are Equal $= \lambda n.\ \lambda m.$ AND (greater Or Equal n m) (lessOr Equal n m)

Pred of 0 is 0.

So 5 − 6 → apply predi. func^n on 5, 6 times

5 ← less, 6 ← big.

⇓

reaches 0 & stays there.

→ So if 1st is less & 2nd is greater, it always gives 0.

AND func^n

X. $(\lambda xy.\ xy\ T)\ T\ T = T$

(Chek. Not sure) $(\lambda xy.\ xy\ T)\ T\ F = F$ X.

T	T	T
T	F	F
F	T	F
F	F	F

So, AND func^n is $(\lambda xy.\ xy\ F)$

if x is false takes F

if x is true takes y

$(\lambda xy.xy\ F)\ (T\ T)$ — T
$(T\ F)$ — F
$(F\ T)$ — F
$(F\ F)$ — F

Similarly can get OR, XOR func^n

but Lambda Calc. should be anonymous.
By doing self, we use the name of the func^n.

Division

→ a/b ⟹ Recursive construct

→ if a >= b, then 1 + (a − b)/b else 0

∴ divide = λa. λb. if_then_else (greaterOrEqual a b)
(succ (self (subtract a b) b))
(zero)

+1

call a method itself with 2 param: a − b & b

≡ divide = λa. λb. if_then_else (greater b a) (zero)
(succ (self (subtract a b) b))

Eg: divide seven three

interchng rather defn. of greaterOr= and lessOr=

⇒ if_then_else (greaterOrEqual seven three)
(succ (self (subtract seven three) three))
(zero)

⇒ (succ (self (subtract seven three) three))

⇒ (succ (if_then_else (greaterOrEqual four three)
(succ (self (subtract four three) three))
(zero)))

⇒ (succ ((succ (self (subtract four three) three)))

Scanned by CamScanner

$\Rightarrow$ (succ (succ ...

$\Rightarrow$ (succ (succ (zero)))

So we need some construct that allows to call a method itself w/o requiring/ the method name using

Y Combinator

→ Y Combinator can be defined as:

$$Y = \lambda t.\ (\lambda x.\ t(x\ x))\ (\lambda x.\ t(x\ x))$$

$$Y t = t\ (Y t) = t(t\ (Y\ t)) = \ldots$$

define factorial = λn. if (= n 1) 1
(* n (factorial (- n 1)))

$$\cancel{YI = (\lambda t.\ (\lambda x.\ t(x\ x))\ (\lambda x.\ t(x\ x)))\ I}$$

Not a pure lambda calc. exp
(∵ use the name Func)

→ Fact = λn. (isZero n) one (mult n (Fact (pred n)))

Lambda are anonymous.

define factorial = I factorial

define I = λf. λn. if (= n 1) 1
(* n (f (- n 1)))

$$(Y\ I)\ 1 = I\ (Y\ I)\ 1$$
$$= \text{if}\ (=\ 1\ 1)\ 1\ (*\ 1\ (Y\ I\ (-\ 1\ 1)))$$
$$= 1$$

β-reduction

(calculating arithmetic)

Scanned by CamScanner

Fact: $= Y\,(\lambda f.\lambda n.\,(\text{isZero } n)\text{ one } \underline{(\text{mult } n\,(f\,(\text{pred } n))}$ (*)

(*) $YT = (\lambda x.\,T(x\,x))\,(\underline{\lambda z.\,T(z\,z)})$

$= T\,(\underline{\lambda z.\,T(z\,z)}\ \lambda z.\,T(z\,z))$

YT

$= T\,(YT)$

again reducing YT

$\Downarrow$

$T\,(T\,(YT))$

Q: $3!$

$YF\ 3$

$= Y\cdot(\lambda f.\lambda n.\,(\text{isZero } n)\text{ one } (\text{mult } n\,(f\,(\text{pred } n))))$ three

F

the lambda in Y is the leftmost lambda

Hv to replace leftmost λ in Y with whole of this

$= YF$ three

applying F on Y

$= F\,(YF)\ 3$

1st argument

Now, hv F in the leftmost side

So place 1st arg = YF in f

and 2nd arg = 3 in n.

$= \text{mult } 3\ YF\,(2)$ (*)

1st arg(f) 2nd arg (n)

$= \text{mult } 3\ F\,(YF)\ 2$

$= \text{mult } 3\ \text{mult } 2\ (YF)\ 1$

$= \text{mult } 3\ \text{mult } 2\ \text{mult } 1\ (1)$

Scanned by CamScanner

$= \text{mult } 3 \; (\text{mult } 2 \; (\text{mult } 1 \; (1)))$ → 2 param. of mult.

1

2

6

Division again!

$$D := \lambda f.\lambda a.\lambda b.\text{ if_then_else } (\text{greaterOrEqual } a\ b)$$
$$(\text{succ } (f\ (\text{subtract } a\ b)\ b))\ (\text{zero})$$

Eg: $Y D$ five two

$\Rightarrow \ldots$

$\Rightarrow$ succ (YD four two)

$\Rightarrow$ succ (D(YD) three two)

$\Rightarrow \ldots$

$\Rightarrow$ succ (succ (YD one two))

$\ldots$

$\Rightarrow$ succ (succ (zero))

$$Y \equiv (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

This function applied to a func^n R yields:

$$YR = (\lambda x.R(xx))(\lambda x.R(xx))$$

which further reduced yields:

$$R((\lambda x.R(xx))(\lambda x.R(xx)))$$

This means that $YR = R(YR)$ i.e. the func^n R is evaluated using the recursive call YR as the 1st arg

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$$

To compute sum of nat. nos. from 0 to n.

$$R \equiv (\lambda r\, n.\, Zn\, 0\, (n\, S\, (r\, (Pn))))$$

Zn 0 : if n==0 then the result of the sum is 0 else the successor func^n is applied n times through the recursive call (R)

Fiboracci Series

$$F(n) = F(n-1) + f(n-2)$$

19/03/19

Scanned by CamScanner

Logic & Logic Programs Contd.

Universal quantifier.

Existensial quantifier.

---

natural (0)

natural (2)

For all x, natural (x) natural (successor (x))

→ natural (-1)

→ A variable not bound by a quantifier is said to be free

→ Arguments to predicates & functions can only be terms : combination of variables, constants & ...

Quantifier examples

① $\forall x$ (speaks (x, Russian))

True if everyone on the planet speaks Russian ;

false otherwise

Scanned by CamScanner

② $\exists x$ (speaks $(x$, Russian$))$

→ True if at least one person on the planet speaks Russian

false otherwise

③

④

Eg. 2

→ Every man is mortal : $(\forall x)(MAN(x) \rightarrow MORTAL(x))$

John is a man : $MAN(John)$

" " mortal : $MORTAL(John)$

→

Eg 3

A horse is a mammal

A human is a mamal

Mml have 4 legs & no arms or 2 legs & 2 arms

A horse has no arms

A human has arms

A human has no legs. - - -.

Ans.

↓

mammal (horse)

mammal (human)

for all x, mammal(x) →

legs$(x, 4)$ and arms$(x, 0)$ or legs$(x, 2)$ and arms$(x, 2)$

arms (horse, 0)

not arms (human 0)

legs ~~arms~~ (human, 0)

Scanned by CamScanner

→ ⊤

Logic prog^m : A collection of statements is assumed to be axioms, and from them a desired fact is derived by the appl^n of inference rules in some automated way

Logic progm. lang : a notational system for writing logical statements together with specified algo for implementing inference rules

Logic prog : set of logical stmnt that are taken to be axioms

---

→ The stmt to be derived can be viewed as the i/p that

→ also called queries / goals.

→ there exists y, legs (Human, 2) ?

→ Yes

Logic progm systems are sometimes referred to as deductive databases.

Control problem : specific path or seq. of steps used to derive ...

→ Logical prog^m paradigm (Kowalski):

algorithm = logic + control

→ Compare this with imperative prog^m (with)

algorithms = data str + programs.

Scanned by CamScanner

→ Since logic progms do not express the control in theory, operations can be carried out in any order / simultaneously

→ Autorated deduction systems have difficulty handling all of first-order predicate calculus.

→ Too many ways of expressing the same stmt

→ Too many inference rules

Horn Clauses

Horn Clause: a stmt of the form:

$$a_1 \text{ and } a_2 \text{ and } a_3 \ldots \text{and } a_n \rightarrow b$$

→ a_i is are allowed to be simple stmnt

→ No or connectives & no quantifiers allowed

Mathematical Logic

↓

First Order predicate Logic

↓

Horn clauses

This statement says that a_1 through a_n imply b, or that b is true if all the a_i are true.

Eg 4:

First order predicate calculus:

→ Translate these into Horn clauses by dropping the quantifiers

Eg: 5 logical description for the Euclidian algo for greatest common divisor of two positive integers u & v

→ The gcd of u & 0 is u gcd (u, 0, u)

→ gcd of u & v, if v is not 0, is the same as the gcd of v & the remainder of dividing v into u

Horn clause

gcd (u, 0, u)

not zero(v) and gcd (v, u mod v, w) → gcd (u, v, w)

Eg 6: x is a grandparent of y if x is the parent of someone who is the parent of y

→ Predicate calculus

→ Horn clause:

Eg 7: For all x, if x is a mamal then x has 2 or 4 legs.

Scanned by CamScanner

Procedural interpretation: Horn clauses can be reversed to view them as a procedure:

$$b \leftarrow a_1 \text{ and } a_2 \text{ and } a_3 \ldots \text{ and } a_n$$

Variable scope:

⋮

→ Queries or goal statements: the exact opp. of a fact

Resolution & Unification

Eg: given $b \leftarrow a$ and $c \leftarrow a$

Resolution gives $c \leftarrow a$

Eg: given $b \leftarrow a$ & $c \leftarrow a$

Combine: $\cancel{b}, c \leftarrow a, \cancel{b}$

Eg 10: gcd with resolution & unification

No chng in ans irrespective of order of these clauses. Just some slight extra computation

$$\text{gcd}(u, 0, u)$$

$$\text{gcd}(u, v, w) \leftarrow \text{not zero}(v), \text{gcd}(v, u \bmod v, w)$$

Goal:

$$\leftarrow \text{gcd}(15, 10, x)$$

→ Resolution fails with first clause (10 does not match 0) so use the second clause & unify:

$$\text{gcd}(15, 10, x) \leftarrow \text{not zero}(10), \text{gcd}(10, 15 \bmod 10, x)$$

$$\cdot\ \text{gcd}(15, 10, x)$$

Scanned by CamScanner

↑

---- gcd(5, 0, x) (1st clause)

↙

gcd(5, 0, 5)

So, ultimately:

gcd(5, 0, 5) ← gcd(5, 0, x)

---

① ancestor(x, y) ← parent(x, z), ancestor(z, y)

② ancestor(x, x)

③ parent(amy, bob)

If we provide the query:

ancestor(x, bob)

↑

Solving:

ancestor(x, bob) ← parent(x, z), ancestor(z, bob),

ancestor(x, bob)

What will be the o/p on changing order?

③ ____

② ____

① ____

The Language Prolog:

Prolog notation is almost identical to Horn clauses.

Scanned by CamScanner

→ can specify head & tail of list using a vertical bar

Eg: $[H|T] = [1, 2, 3]$ means

$H = 1, \; T = [2, 3]$

Eg: $[x, y|z| = [1, 2, 3]$ means

$x = 1, \; y = 2$ and $Z = [3]$

Execution in Prolog

Eg: // : clauses entered into database

ancestor (x, y) : parent (x, z), ancestor (z, y)

ancestor (x, x)

parent (amy, bob)

Queries:

?- ancestor (amy, bob)

Recursion

numeral (0)

numeral (succ (x)) :- numeral (x)

→ So if we pose queries like

numeral (succ (succ (succ (0))))

Scanned by CamScanner

Tail Recursion :

→ Fact (acc, n) { return n == 1 ? acc : Fact(acc*n, n-1) }

→ factorial (0, F, F).

→ factorial (N, A, F) :- N > 0, A1 is N*A, N1 is N - 1, factorial (N1, A1, F).

→ ? - factorial (5, 1, F).

---

①	Logic Prog	10
②	Lambda Calculus	10
③	Collectors	10

29/3/19

Arithmetic

→ Prolog has built-in arithmetic operations

→ Prolog cannot tell whether a term is arithmetic or strictly data

Scanned by CamScanner

→ Greatest common divisor algorithm

→ In generic Horn clauses :

→ In Prolog

gcd (U, 0, U) ⊙ → to end statement in Prolog

gcd (U, V, W)

there are no other solutions.

gcd(15,0,15) = true
false
gcd(15,10,x)
X = 5
false

Resolution & Unification (cont'd)

→ Eg. 10 (cont'd.)

→ If zero (10) is false, then not zero(10) is true.

→ (Simplify) 15 mod 10 to 5 and cancel...

Recursion.

→ numeral (0)

numeral (succ (x)) :- numeral (x)

→ So if we pose queries like :

numeral (succ (succ (succ (0))))

→

→ factorial (0,1).

factorial (N,F) :- N > 0, N1 is N-1, factorial(N1,F1), F is N * F1

Scanned by CamScanner

Tail Recursion

o.s. → If the continuation is empty & there are no backtrack points, nothing need be placed on the stack;

execution can simply jump to the called procedure w/o storing any record of how to come back.

This is called LAST-CALL OPTIMIZATION

Unification

Unification: process by which variables are instantiated to match during resolution.

Prolog's Unification Algorithm

Eg: Unification (cont'd)

```
?- me = me.
yes

?- me = you.
no

?- me = X.
X = me

?- f(a,X) = f(Y,b).
X = b
Y = a
```

```
?- f(X) = g(X).
no

?- f(X) = f(a,b).
no

?- f(a,g(X)) = f(Y,b).
no

?- f(a,g(X)) = f(Y,g(b))
X = b
Y = a
```

func^n term & constant don't match

Scanned by CamScanner

Unification (cont'd)

→ Unification causes uninstantiated variables to share memory (to become aliases of each other)

Eg: two uninstantiated variables ~~are~~ unified

?- X = Y.

X = _23

Y = _23

pipe appears for 2 constant concatenate (cons)

cons (1, 2, L) → pipe here

L = [1 | 2]

cons (1, [2, 3, 4], L).

L = [1, 2, 3, 4].

cons (X, Y, [1, 2, 3, 4]).

X = 1

Y = [2, 3, 4]

cons (1, [2], L)

L = [1, 2] → pipe not here.

→ Pattern directed invocation:

using a pattern in place of a variable unifies it with a

Unification (cont'd)

→ Append procedure:

if one list _ no pipe

append ([X, Y, Z]) :- X = [], Y = Z.

append (X, Y, Z) :- X = [A | B], Z = [A | W], append (B, Y, W).

→ First clause: appending a list to the empty list gives just that list

Scanned by CamScanner

```
?- append(X, Y, [1,2,3,4,5]).
    X = [],
    Y = [1,2,3,4,5] |
    X = [1],
    Y = [2,3,4,5] |
    X = [1,2,3]
    Y = [4,5]  |. . . . . .
```

→ Append procedure rewritten more concisely:

```
append([], Y, Y).
```

→ Reverse procedure

```
reverse([], []).
reverse([H|T], L) :-
    reverse(T, L1),
    append(L1, [H], L).
```

reverse([1,2,3,4,5], L) L = [5,4,3,2,1];
reverse(L, [1,2,3,4,5]) L = [5,4,3,2,1]

```
accReverse([H|T], A, L) :- accReverse(T, [H|A], L).
accReverse([], A, A).
```

Scanned by CamScanner

Prolog's Search Strategy

→ Prolog applies resolution in a strictly linear fashion

→ Replaces goals from left to right

→ Considers clauses in the database from top down

Figure 9.2 A Prolog search tree showing subgoals and variable instantiations

If you put a ; , Prolog back tracks

02/04/19

Scanned by CamScanner

Max counting:

Predicate max/3 which takes integer as arg ...

mutually exclusive

max (X, Y, X) :- X =< Y

max (X, Y, Y) :- X > Y

Eg: max(3, 2, 3) gives true

max (3, 2, X) gives X = 3

max (1, 3, X) gives X = 3

max (3, 2, 6) gives false

max (3, 2, 2) gives false

But if clauses were:

max (X, Y, Y) :- X =< Y

max (X, Y, X)

since mutex, we try removing it

Then, max (1, 3, X) gives

X = 3

X = 1

(bcos when 1st cond false it backtracks)

---

max (1, 3, 1) gives true

bcos. (1st false but 2nd true)

Scanned by CamScanner

→ There can never be any second solution. So it should not back track.

→ The 2 clauses are mutually exclusive.

So,

1. max (X, Y, Y) :- X =< Y, !.
2. max (X, Y, X) :- X > Y

cut

like a one way switch. When you go from 1st clause to next, you can't come back.

Second clause will be evaluated only if first one does not satisfy.

Now, max (1, 3, X). gives X = 3

max (4, 3, X) gives X = 4

Now it don't backtrack.

If we don't put this clause condⁿ, system gives wrong value for max (5, 2, 2) → you want to check whether this no. is maxⁿ of 5 & 2.

This was,

Green cuts: which doesn't change the meaning of the program

---

1. max (X, Y, Y) :- X =< Y, !.
2. max (X, Y, X).

Here we need red cut

?- max (100, 101, X). X = 101. yes

max (3, 2, X) X = 3. yes

max (2, 3, 2) ✗ (wrong ans)

Scanned by CamScanner

So we modify as :

1. max (X, Y, Z) :- X = < Y, !, Y = Z.
2. max (X, Y, X).

↑

max (2, 3, 2)

2 = < 3, (!), 3 = 2

true false

false

If this was not there it would backtrack to root & print true for the 2nd clause

But now restricted to backtrack

So if we don't put a cut here, we will get 2 solutions ⇓ wrong.

So, this is called redcut.

→ If a cut is reached on backtracking

Figure 4.4 Consequences of the cut for the search tree of Figure 4.2

Rewriting this example :

① ancestor (X, Y) :- !, parent (X, Z), ancestor (Z, Y).

② ancestor (X, X)

③ parent (amy, bob).

Scanned by CamScanner

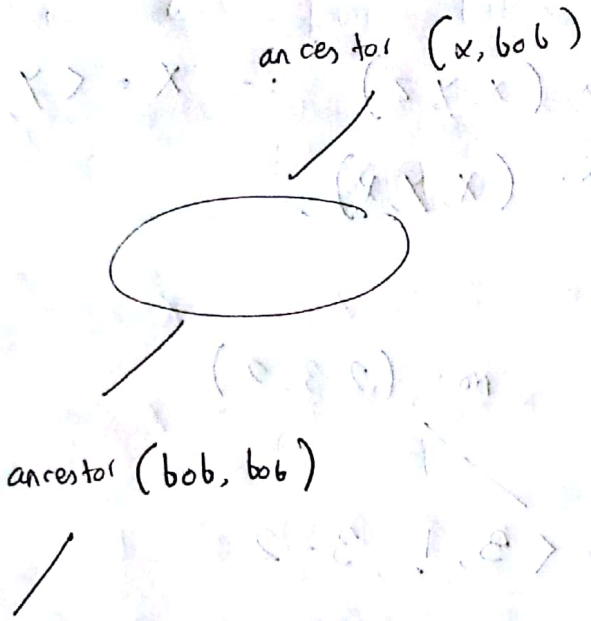


Fig 9.5

Where will you put the cut, so that no sol^n is pruned:

A.

1.
2. - - - , !
3.

→ Both sol^n obtained

→

→ One sol^n to infinite loop shown earlier:

```
num(0).
num(X) :- num(Y), X is Y+1.

writenum(I, J) :- num(X),
                  I =< X,
                  X =< J,
                  write(X), nl,
                  X = J, !,
                  fail.
```

→ X = J will succeed when the upper-bound J is reached

→ The cut will cause backtracking to fail, halting the...

→ Can also use cut to imitate if else constructs

D = if A then B else C

Prolog code:

```
D :- A, !, B.
D :- C
```

Summation of a list

Scanned by CamScanner

Problems with Logic Programming

→ Original goal of logic prog.. was to make prog... a specific activity

---

The Occur-Check Problem in Unification

Eg: is_own_successor :- X = successor(X).

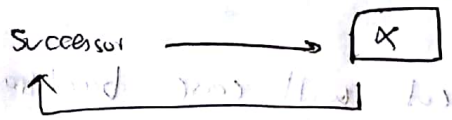


Figure 9.7: Circular str created by unification

Negation as Failure

--- because not (human(x)) but the instantiation of X to bob is released

Horn Clauses Do Not Express All of Logic :

Exploration

Ex: Insertion Sort , Insert , - - -

Quick Sort

Not Exam

FP vs OOP

- In FP (& procedural), programs break down into functions that perform some operations
  - → Functions may take one or more arguments
- → In OOP, prog. break down into classes that give behaviour to — — — —

Basic Set-up

→ Expressions for a small "language" such as for arithmetic

	Evaluate	toString
Integer		
Negation		
Addition		

→ It is "more natural" to have the cases for eval together rather than the operations for Negate together

Scanned by CamScanner