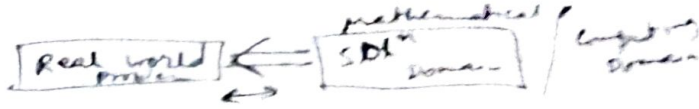
C → imperative
C++, Java → OOP



High level lang ← Machine Lang

Abstraction

Data, control

of simplifies behaviours and attributes of data

control abs. simplifies the properties of modification of the execution path of a prog based on a situation.

data → basic abs. → collects the most localized info. contained in the abs
↓
- number representations
- variables
- data types

x = 12.5 ; IEEE single, double pt precision

(atomic or premitive) (security) data hiding

→ structured → collects intermediate info. about the structure of a prog.

int x[ ][ ];

- data structure is the principle method for collecting relative data values into a single unit.

one element ← a Collection of items

one element

① Hierarchy
② Group of individual components
③ Access

Data structure → Array
→ File (Text)
(Records)

→ Unit abs. → collects large scale info of a prog.

ADT
① Hierarchy
② arb. of individual components
③ Access

(Abstract data type)

seperation of interface from implementation.

① info. hiding ② reusability ③ Interoperability

ADT → Components (module in python / package in java)
↘ class (small scale)

Unit ← ADT
library, API
(Application Programming Interface)
(Java doc)

java. lang.
system.out.println("...") } interoperability.

Control → looking at the prog. logic.

basic — All arithmetic, assignment operators.
syntactic sugar

SUM = A+B;
LOAD A
MOV ...

Syntactic sugar
x += 10;

Syntactic sugar refers to any mechanism that allows a programmer to replace a complex notation with a simpler shorthand notation

→ structured — (if for, ~~for~~)
syntactic sugar
fn/procedure
Higher order.

for (int i : data)
S.O.P (i);

Syntactic sugar for
for (int i=0; i<...; i++)
S.O.P (data[i])

Iterator<String> itr
= enamtleList.iterator();
while (itr.hasNext())
S.O.P. (itr.next());

for (string S : scrambleList) } syntactic sugar.
S.o.p (S);

procedure gcd (int: x,y out: z) ① defn./declaration of the procedure
② invocation.

int f (int x) ← immutable
{ return x+1; }
y = f(x) ↑ 5 x

variables in math are immutable
in prog lang. mutable

f: X × Y → Z
y = x2
y = f(x)
x = x+1

Z(x) = g(f(x)) higher order fn.

Scheme: map (len, ["Arjun", "Basu", "priti", "Tamaghna"])
o/p → [5, 4, 5, 8] reduce (lambda Sum x: Sum+len(x), ["...", "...", ...]) 0
higher order fn. o/p → 22

Control — Structured — Higher order
└ Unit — Package & Library.

Parallel programming Constructs
└ class / Unit abstraction w.r.t Java
In case of Ada it is provided as a task
Structure abstraction

Computing paradigms

implement browsers

1. Imperative. characteristics
   (i) sequential execution of instructions.
   (ii) use of variables representing mem. locations.
   (iii) use of assignment to change the values of variables. (mutable)

(from von neumann arch.)

von neumann: Computation can be described as a seq. of ins. operating on a single piece of data.

restricts ① parallel programming.
② Non-deterministic computation.
(outcome independent of order of execution)

2. Object oriented : logical extension of imperative
3. Functional } math. background.
4. Logic
   └ automatic theorem proving

prolog. (logic prog.) (1 gen.) — Functional Java 8 (2-3 gen.) lambda calculus

(similarity, diff functional, object oriented)

prog lang principles and practice
London & Lambers 1, 2, part of 3, 4

10/1/2020

Language Definition.

Syntax → Context free grammar
Semantics ↑

<if_statement> ::= if (expression) <statement> [else <statement>]

Semantics

1. operational semantics : Reduction machine.
2. Denotational semantics : +: Integer × Integer → Integer
   real: entier → integer
   λx. x identity
3. Axiomatic semantics : Assertion

x/y 0/0.

predicate transformers → predicates
↳ raise new predicates

3 properties of all 3 semantics have to check

1. Complete : Every correct terminating program must have an associated semantics given by the rules
2. Consistent : The same program cannot be given two different conflicting semantics.
3. Independent : no rule should be derivable from other rules
   Minimal.

1. Efficiency :
   - target code efficiency
   - programmer efficiency (debugging)

static typing int i = 0,
Dynamic typing i = 0
(concise, small code)

root find out

java

```
if (x > 0)
{  num. soln = 2 ;
   sol1 = math.sqrt (x) ;
   sol2 = - sol1 ;
}
```

python

```
if x > 0 :
   Sol 1 = sqrt(x)
   sol 2 = - sol 1
   no of sol = 2
```

2. Reliability : closely matches with maintainability of the s/w.
3. Regularity : Regularity refers to 3 things
   i. fewer unusual restrictions on the use of particular constructs.
   ii. fewer strange interactions betwin constructs.
   iii. fewer surprises in general ~~that~~ in the way language features behave.

Regularity — Generality, orthogonality, Uniformity.

Generality — procedures & functions. operators constants.

C shows partial generality w.r.t. (procedures & fn.) → this feature

(==) equality compare two primitive data types

Hash ~~code~~ cell Comple generality w.r.t operators, (operator overloading)

~~C~~ Stat. constants : c = 5; c = a+b; c = f(x)

Ada supports .... constants.

y = f(n)
y' = f'()

Orthogonal design allows language constructs ~~to~~ to be defined in any meaningful way with no unexpected restriction.

orthogonality — function return types.
- C, C++ ~~Partial~~ partially orthogonal
- python, ~~Ada~~ ada complete

placement of variable defns : ANSI C does not support

Primitive type vs. Ref. type :
(int, float..) (objects, structures, ..)

int i2 i1+i2 99
Integer i1

python is completely orthogonal but Java is not.

14/1/2025

1. Efficiency
2. Regularity — Generality, orthogonality, Uniformity

It refers to the consistency of appearance & behaviour of language constructs.

Uniformity → assignment $a = 5$;

"*" → multiplication, pointer.

in pascal function f : boolean;
begin
f := true;
end

|| w.r.t assignment operator does not show uniformity.

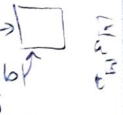
Irregularity fn. defn. can't be nested in java

Algol 68 shows perfect regularity.

int, float — value semantics

Integer object — pointer, reference

3. Security — A language with an eye towards security ∧ it discourages programming errors and allows errors to be discovered & reported.



Java
int x = 5
x = "Hi"; compiler error

Python
x = 5
x = "Hi"
x = x/y (runtime error)

Security — type, type safety (expressiveness is little bit compromised)

1. static typing vs. dynamic typing.
2. strong typing v.s. weak "

in javascript
x = 5
y = "6" ← x = x + y
x = x + y → x = 5, y = "Hi"

implicit type conversion.

Weak typing (hard to det

vs Java strong typing (w.r.t security

3. Semantic safe
x = x/y; ← runtime error available to throw error.

Security → type, type safety → semantic safety

memory leak — orphan node
Garbage collection. type
mem. chunk
Java does it automatically

tamperig efficiency of the code

4. Extensibility: The ability of a lang. to add new features to it

new syntax, new behaviour (function), create new data types.
packages & modules.
replicated: should not be used, in future releases, will be removed

Semantics LISP (more extensibility)

```
(do ()                    Calculate GCD
  ((= 0 b))
  (let ((temp b))
    (set f b(mod a b))
    (set f a temp)))
```

while loop

```
(defmacro while (condn & rest body)
  `(do ()
       ((not, condn))
     ,@body))
```

```
( while ( )
    (( > 0 b))
    :
    ))
```

~~Flon's Axiom~~ Flon's Axiom

There does not now nor will there ever exist a programming language in which it is least bit hard to write bad programs.

1. Nested blocks, procedures having too many parameters
2. Low cohession within a module & high coupling betwn modules (bad)
3. Use of pointers

5) Turing Tar-Pit — All computing languages or computers can compute anything in theory but nothing of practical interest is easy.

Python. (bridge betwn shell scripting & prog. lang.)

1. Simplicity - small set of primitive op^n ~~& data types~~.
   ~~Regularity~~, Extensibility (syntax & behaviour, modules)
   diff API modules for diff applns.
2. Interractivity and portability → (byte code ...) (write once, run anywhere)
   ↑ directly from command prompt. (prototyping applns.) (virtual m/c)
3. Dynamic typing vs. finger typing: ~~easy~~ ~~more~~ ~~costs less~~
   Say more with less — python 3.6. stacking typing introduced

balance betwn two

What of a Computation → declarative programming
How " " → imperative paradigm

memory location x
assignment x

```
int gcd ( int x, int y) {
    if (y == 0) then return x;
    return gcd(y, x % y);
}
```

need

$$gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, x \bmod y) & \text{otherwise} \end{cases}$$

$y = f(x)$ $\quad f : X \to Y$ (domain, range)

3 criteria of a block of code for referential transparency

any fn will be referentially transparent if
(i) if it's output depends only on (i) its arguments and ~~only on~~
(ii) any non local variables (iii) the o/p should be independent of the order of evaluation of its arguments.

$gcd(10, 5) = gcd(5, 10)$

no different than a constant ref transparent fn.

final int x = 5 ← gcd(5, 16)

f() does not have any argument always same value

↑ 1st ~~cat~~ class data value. value semantics.

Composition — Inputs a fn & o/p a data value / return a fn

$f : X \to Y$ $\quad g \circ f : X \to Z$ · Composition

$g : Y \to Z$ $\quad g(f(x))$ Higher order fn. pass a behaviour or return it

1. ~~Data~~ incoming and outgoing values.
2. No assignments
3. No loops. tail recursion
4. fns are treated as first class data value
5. Currying. gcd(5,10)

every time we apply one parameter

17/1/2020

LAMBDA EXPRESSION

$\lambda x.x+1$

λ _ . _ (PARAMETER, BODY OF THE FUNCN)

button.addActionListener(new ActionListener(p.v. actionperformed (ActionEvent event) { s.o.p("button is clicked"); }));

button.addActionListener(event -> s.o.p("button is clicked"));

Lambda expression in java.

It is an ~~An~~ anonymous fn / fn. without a name, that is used to pass around behaviour as if it were data.

Runnable r1 = () -> {s.o.p("This is a thread"); s.o.p(- - - -);};

Binary Operator is an interface that takes two variables and ~~op~~ carry operations, return same type

Binary Operator<Long> addObject = (x, y) -> x+y;

(x, y) -> {return x+y;};

Functional Interface.

(an interface with ~~no~~ exactly one abstract method that is used as the type of the lambda expression)

Binary Operator addObject = (x, y) -> {return x+y;}

(Long x, Long y) -> {return x+y;}

Generic Object ×

Runnable r1 = (+ -)

Runnable r2 = ( ) -> s.o.p("This is thread 2");

p s. void process(Runnable r) { r.run(); }

process(r1);

Public interface Adder { p. int add(int, int); }

public interface SmartAdder extends Adder { p. int addSmart(int, int); }

SmartAdder is no more a functional interface. (two abstract methods)

<T, T>

ActionEvent

Public interface

p. boolean

} List<Int

String

for(int i = 3

}

int

a no. even

IntPredicate

s.o.p. (pred(100));

T → To Long Fun

() → ne

Extract data

Compare two

boolean extra

Combine two

create new

Lambda ex

Closure p

But lambd

Lambda Ex

Callable<Integer> c =

Runnable

<T,T> → Binary Operator <T> → <T>

ActionEvent → ActionListener,

<T> → Predicate <T> → boolean

T → Function <T,R> → R

Public interface Predicate <T> {
    P. boolean test (T);
}

List<Integer> arrList = new ArrayList<>();
String arr[] = { "abc", . . . };
for (int i = 300; i < 400; i++) {
    arrList.add(i);
}

type inference

array less memory than ArrayList due to storing primitive

int → IntPredicate → boolean.

Long BinaryOperator
Double - -

a no. even or not.

IntPredicate prd1 = x → x%2 == 0;

S.o.p. (prd1(100)); True

fast as int

Predicate prd2 = y → y%2 != 0;

S.o.p (prd2(100)); False

y → 100

Object → Integer (wrapper class)

T → ToLongFunction → Long.

() → new Integer();

java.util.function

Extract data from an object.

Unary operator

Function f = (String s) → s.length();

Compare two objects. (String s1, String s2) → s1.length().CompareTo (s2.length());

boolean expr.
Combine two objects.
create new object

(final) String username = ----
button addActionListener (
new ActionListener (
P. V. actionPerformed (ActionEvent e)
{ S.O.P ("Hello" + username); }));

Lambda expr maintains closure property.

But lambdas close over values rather than variables

21/1/2020

Lambda Expressions

Callable<Integer> c = () → 42;

PrivilegedAction<Integer> P = () → 1;

T → Function → R.

= () → { try {

Buffered Reader processor

} catch( ) { }
} ;

```
Public Interface BufferReaderProcessor {
    String readFromBuffer()
        throws IOException;
}
```

java.io.closable (abstract method close)
not a functional interface.

@FunctionalInterface to check whether fn. interface

Predicate

```
interface Predicate<P> {
    boolean test(<T>);
}
```

Predicate<Integer>

```
interface IntPredicate {
    boolean test(int);
}
```

```
p.v. overloaded(Predicate<Integer> p) {
    S.O.P("Integer predicate");
}
p.v. overloaded(IntPredicate p) {
    S.O.P("int predicate");
}
```

overloaded((x) -> true);
overloaded((int x) -> true);

3 overloading rules to call abstract type

1) If there is a single specific type, javac infers the type from the corresponding argument of the functional interface.
2) If there are several specific target types, the most specific type is inferred
3) If there are several specific types and no most specific type, the programmer should manually provide the type.

Inherit interface
No scope for defining any field.

Java streams API

Stream()
should be abstract.

ArrList.stream() now apply fn. proj

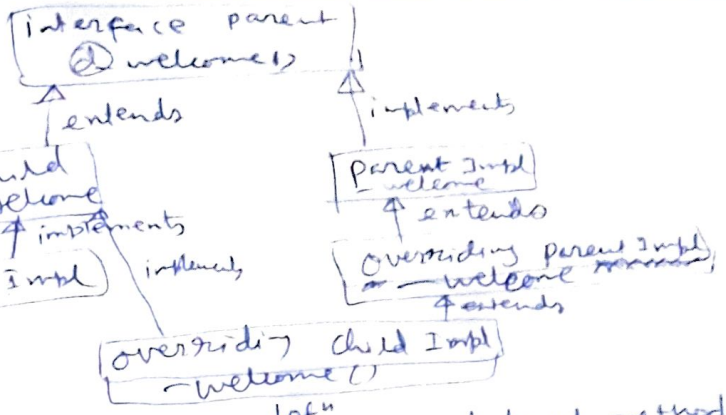
Default Stream()
if no other method defn, then use this defn.
method defn within interfaces

```
@FunctionalInterface
interface OurInterface {
    p.v. ourMethod();
}
default p.v. OurDefaultMethod(...) {
    ...
}
```

```
ParentImpl PI = new ();
PI.welcome();
```

Parent Impl
implements
interface Parent
welcome()

```
interface Parent {
    default v. welcome() {
        S.O.P("Parent interface");
    }
}
```



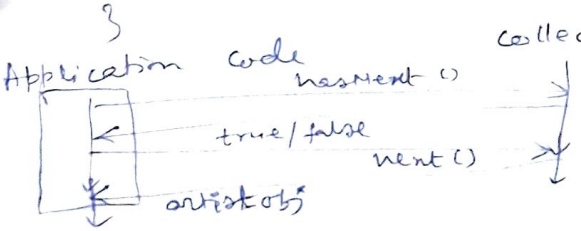
rules
① if there is a method or an abstract method within a superclass class interface version of the default method ignore
② if two interfaces, one extending the other, compete for a default method. Subinterface wins.
③ In case of any ambiguity, subclass must either implement the method or declare this as abstract.

Homework define a Java class that operates on a collection (List of integer) define a method _eval(↓ . . .) that can print all numbers from the list, print even numbers from the list, print the numbers more than 5 from the list
(method defn. should not change)

List of artists class Artist { _name ; _city ; _typeofArtist ; _wages; }

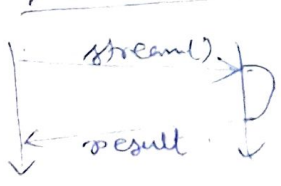
How many artists are from kolkata.

```
int cnt = 0;
                                List<Artist> artists = . . .
Iterator<Artist> itr = artists.iterator();
while (itr.hasNext()) {
    Artist a = itr.item()
    if (a.getCity().equals("kolkata"))
        count = count + 1;
}
```




```
int count = artists.stream().
    filter((a) →
    a.getCity().equals("kol
    .count();
```

Stream interface : filter, count



stream().filter().count()

parallelStream()

or stream().parallel().filter(

```
int count = artists.parallel().filter((a) → a.getCity().equals("kol"))
                .limit(3).collect(toList());
```

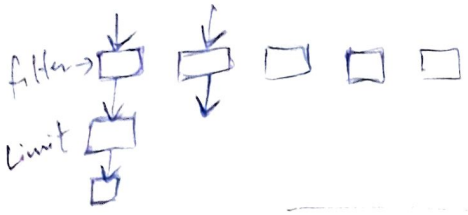
List<Artist> Arr =

Streams can be defined as a sequence of elements from a source that supports data processing operation

Data processing opⁿ

1) Database like opeⁿ. 2) functional programming opⁿ

filter((a) -> a.getCity().equals("kol")) Limit(3)



Composable || declarative parallelize

.equals("kol"), map((b) -> b.getName())

filter map limit — any order — returns stream — lazy operations.

collect, count — returns integer — Eager operation.

Until eager method is there, Lazy operations will not execute

Stream Arr = artists.stream().filter((a) -> {S.O.p(a.getName()); return a.getCity().equals("kolkata")});

No execution

PIPELINE → LAZY OPERATIONS

SHORT_CIRCUITING

INTERNAL ITERATION

Stream & collection

Similarity ① Both provides interfaces to data structures representing a sequenced set of values of a specific type.

difference ① When things are computed

② Stream is consumer driven and collection is supplier driven

③ Streams are traversable only once.

Arr.count(); Arr.count(); x (should not work)

Stream<Artist> Arr = ...;

exactly one eager operation (eager oper" not compulsory)

(A) A stream can be viewed as set of values spread out in time other in space.

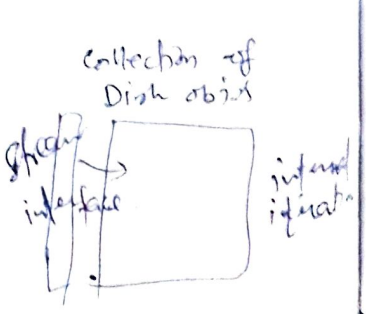
(B) Internal iteration & External iteration (Collection) (Stream)

24/1/2020

class Dish

```
String name;
boolean vegetarian;
int calories;
Type type;
public enum Type
    {meat, FISH, OTHER}.
```

menu.stream()



menu.stream().filter((d) → d.isVegeterian()).collect(toList());

Stream filter (predicate <T>)

Stream<Dish> filter(predicate<Dish>)

menu.stream().filter(d → { S.o.p("Dish" + d.getName()); return d.isvegeterian(); });

Stream<Dish> st

st.collect

List<Dish> vegDish = st.collect(toList()); (if commented prev. line will not run)

menu.stream().filter((d) → d.isvegeterian()).filter(d1 → d1.get.. >_)
.collect(toList()); Imp fusion

Stream.of("56 Arpan", "40 Abhas", "08 Sandipan", "10 Adibar", "Sid");
.filter(S → Character.isNumeric(S.charAt(0)))
.skip(1).collect(....)
(skips 1st element)
.limit(1)

st = menu.stream().filter((d) → d.isvegeterian()).map(d1 → d1.getCalories())

List<Integer> list1

.map(S1 → Integer.parseInt(S1.substring(0,2))).collect(...)

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

Stream<Integer> map(Function<String, Integer>)

List<String> list1 ...

.map(S1 → S1.toUppercase())
.collect(toList());

.distinct()

to list of duplicate words.

Distinct characters.

Stream<String[]> st =

.map(S1 → S1.split(""))

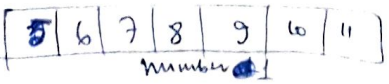
List<String> list1 =

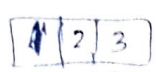
.flatMap(x → Arrays.stream(x)).distinct().collect(toList());

Concatenates diff. streams of objects to one stream

Stream.of(Arrays.asList(1,2), Arrays.asList(3,4))
.flatMap(numbers → numbers.stream()).collect(toList())

Stream<R> flatMap(Function<T, stream<R>>)





(5,1)
(5,3)
(11,1)
(11,3)

List<int[]> pairs = ~~Arrays.asList~~
number1.stream()

flatMap(i → number2.stream().filter(j → (i+j)%2 == 0)
.map(j → new int[]{i,j})
.collect(toList()))

Return List of square of numbers
- map (i -> i * i)

Book
// Java 8 in action
Java 8 Lambdas

28/1/2020

filter, map, flatmap, skip, limit, distinct.

```
stream<Dish> filter(Predicate<Dish>)
                    ↑
interface Predicate<Dish>
boolean test(Dish d) {
    s.o.p(---);
    return ---;
}
```

stream.map(Function<Dish, String>) → Stream<String>

```
interface Function<---> {
    apply(Dish d) {
        s.o.p(---);
        return d.getName();
    }
}
```

---

allMatch, anyMatch, noneMatch → returns boolean. (eager operation) predicate as input.

findAny, findFirst → eager open.

menu.stream().allMatch(dl -> dl.isVegetarian());

boolean

Optional<T> (T → Dish)
.ifPresent(s.o.p(---));
.orElse(c); (c → Dish)

```
interface Consumer<T>
    <method name>
    void accept(Dish ds) {
        s.o.p(--);
    }
```

```
menu.stream()
.filter(dl -> dl.isVeg())
.findFirst().ifPresent
(ds -> Sys.o.p(--));
```

---

reduce (concept of fold in functional programming)

```
Integer[] arrays1 = {1, 2, 44, 32, 56};
List<Integer> aList1 = new ArrayList<Integer>(Arrays.asList(arrays1));
int sum = aList1.stream().reduce(0, (a, b) -> a+b);
```

only associative operations can be done by reduce.

accumulator pattern.

Optional<Integer> totalCalories
---.reduce((a1, b1) -> a1+b1)

Optional Integer object.

---.totalCalories.get()

filter	stateless	allMatch, anyMatch, noneMatch	stateless
map	"	findAny, findFirst	
flatmap	"		
skip, limit	stateful (bounded)	reduce	stateful (bounded)
distinct	stateful (unbounded)		

arrays.stream().reduce(0, (a,b) → a+b);

11/2/2020

① dynamically { Stream.of("...", "...", ...); Stream.empty() empty stream.

int[] arr = {1,2,...} → Arrays.stream(arr) from array

Bounded streams

Stream interface

1. abstract method.
2. default methods. (backward compatibility)
3. static method

② Similar for collection.

③ from file → Files.lines(...)

④ iterate ⑤ generate → unbounded stream create

Stream.iterate(0, n → n+1)

interface UnaryOperator<Integer>{ Integer fn(Integer n){ return n+1; } }

iterate takes unary operator

Stream.iterate(0, n → n+1).limit(10).skip(2).forEach(a → s.o.p(a));

fibonacci

Stream.iterate(new int[]{0,1}, t → { t[1], t[0]+t[1] }).limit(20).forEach(a → s.o.p(a[0]));

t → new int[]

Generate stores a state because it uses supplier.

Stream.generate(() → Maths.random()) (random nos)

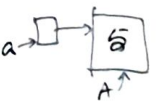
Stream.generate(Math::random).limit(5).forEach(a → s.o.p(a));

H.W Use generate to print fibonacci nos.

<T> → UnaryOperator<T> → <T>

int → int UnaryOperator → int.

Integer a;



IntStream, LongStream, DoubleStream → stream of int... values.

mapToInt takes an object, gives back stream of int.

IntStream.rangeClosed(0,99).reduce(0, (a,b) → a+b); [0,99]

.range(-1,100)... (-1, 100)

IntStream.rangeClosed(0,99).filter(b → b%2==0).forEach(a → s.o.p(a));

IntStream.rangeClosed - - - -

- .max()
- .count()
- .sum()

} know on numeric streams.

Sum of calories

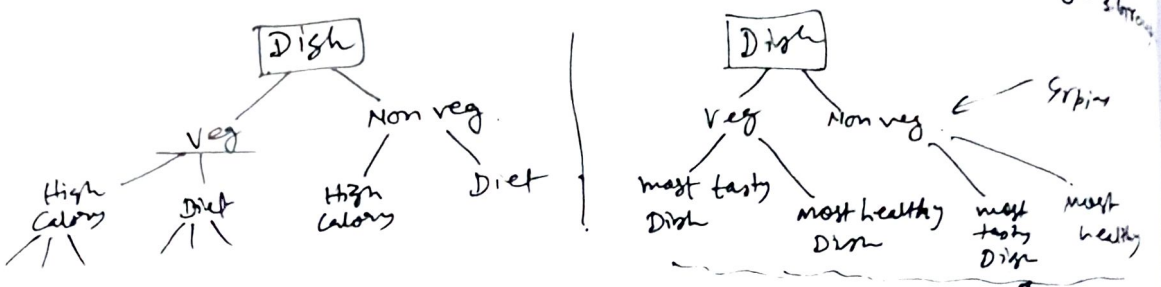
menu.stream().mapToInt(d → d.getCalories())

- .sum();
- .boxed() → IntStream must be converted to stream of Integers
- .forEach(...);

collect(toList());

Using collect

1. Reducing and Summerizing 2. Partitioning. 3. Grouping



menu.stream().filter(m → m.isVeg())
.collect(Collectors.counting());

summing() factory method.

.collect(maxBy(caloriesConsumed))

Comparator interface.

Comparator<Dish> caloriesConsumed = Comparator.comparingInt(d → d.getCalories());

.collect(summingInt(d → d.getCalories()));

int sum = menu.stream().filter(m → m.isVeg())
.collect(summingInt(d → d.getCalories()));

average

IntSummaryStatistics s

averagingInt

summerizingInt

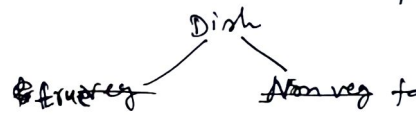
1. Reducing & Summerizing

joining
summing
counting
summerizing

maxBy
minBy

2. partitioning

partitioningBy.



Map<Boolean, List<Dish>> groupOfItems = menu.stream().collect(partitioningBy(d → d.isVeg()));

3. Grouping

groupingBy

Map<Boolean, List<Dish>>
groupOfItem = menu.stream().collect(GroupingBy(d → d.is

H.W joining using reduce.

14/2/2020

collect

① partitioning — true, 2 groups, false

② reducing & summerizing — counting, summing of elements, averaging, maxBy, minBy } int/float data

float avg = students.stream().collect(averagingInt(s → s.getMarksPPL());

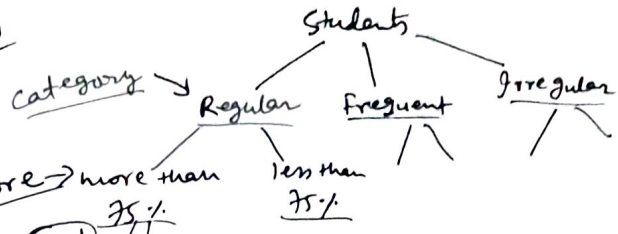
marks PPL
marks Network
marks Compiler

Optional<Student> s
= students.stream().
collect(maxBy(Comparator.comparingInt(S → S.getMarksPPL()));

③ Grouping

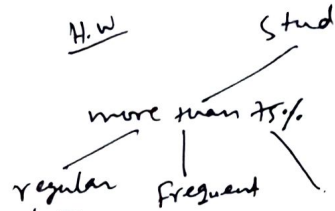
Map<Dish.Type, List<Dish>> m = menu.stream().collect(groupingBy(d -> d.getType()));

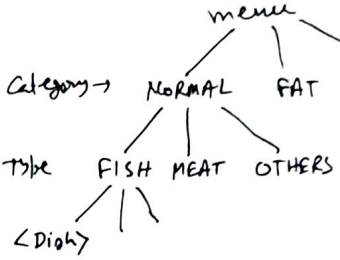
H.W

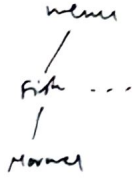


Map<Dish.Type, Integer> m = menu.stream().collect(groupingBy(d -> d.getType(), summingInt(d -> d.getCalo...

H.W







Map<Category, Map<Dish.Type, List<Dish>>> dishesByCalorie = menu.stream().
collect(groupingBy(dish -> { if(dish.getCalorie() <= 400) return ...
else if( ) return
else return Category.FAT; }, groupingBy(d2 -> d2.getType())));

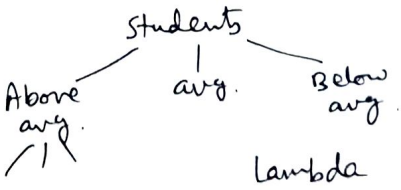
Map<Category, Map<score, List<students>>> ____ =

collect(toList()); ① toList() ② toSet() ③ toCollection(ArrayList::new

menu.stream().collect(groupingBy(d -> d.isVeg(), mapping(d1 -> { if(...) ... ; }, toSet())));

Collecting And Then (maxBy(

18/2/2020



Lambda Calculus.

https://pyvideo.org/europython-2014/many-had-a-little-lambda.html

λ.

Set of (inputs, outputs)

$y = f(x)$ $y = g(x)$

$f(5)$ 25

$g(5)$ 25

(x, y)

$f: X \to Y$

Two operations with fns essential → function creation, function application

$(\lambda x.x^2)5$

E_1 E_2

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

3

λ X. E

E_1 → (ope)rator

E_2 → (ope)rand.

$(\lambda x.x^2)(\lambda y.y)$

$I =_{def} \lambda x.x$ | $\lambda y.(\lambda x.x)$
$\lambda f.f(\lambda x.x)$

int y = $f_1(f_2(x))$

int f1(int y)
:
int f2(int x)
:

$\lambda x.x\ \lambda y.xyz$ parses as $\lambda x.(x(\lambda y.((xy)z)))$

Free(x) = {x}

free (E_1, E_2) = Free(E_1) ∪ Free(E_2)

Free($\lambda x.E$) = Free(E) − {x}.

Free $(\lambda x.x(\lambda y.xyz)) = \{z\}$
no free variable → closed.

{let x ← E in x + (let x ← E' in x) + x

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

substitution, α-reduction
β-reduction

$(\lambda x.x*x)\ 5$

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

β-redex

$(\lambda x.\lambda y.\text{add}\ x\ y)\ 5\ 3$

$\Rightarrow_\beta (\lambda y.\text{add}\ 5\ y)\ 3 \Rightarrow_\beta \text{add}\ 5\ 3$

currying

function <Integer, Integer> add3
= (a) → a+3;

function <Integer, function <Integer, function <Integer, Integer>>>
curryAdd = (a) → (b) → (c) → a+b*c;

$\lambda a.\lambda b.\lambda c.a+b*c$ $\lambda abc.a+b*c$.

HW Use a functional interface to design a converter. Fahrenhite to celcius, pound to INR, $ to INR, meter to milometer. curried converter

tail recursion

25/2/2020

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

after renaming. $[y\ (\lambda v.v)/x]\ \lambda z.(\lambda u.u)\ z\ x$

β-reduction

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

y = f(5);

$(\lambda x.x)(5) \Rightarrow_\beta 5$

var identity = x => x;
alert(identity("Hello"));

$((\lambda x.(xy))(\lambda z.z))$
$\Rightarrow^\beta (\lambda z.z)\ y$
$\Rightarrow^\beta y.$

$((\lambda x\ ((\lambda y.(xy))x))(\lambda z.w))$
$\Rightarrow^\beta ((\lambda y.(\lambda z.w)y)(\lambda z.w))$
$\Rightarrow^\beta (\lambda z.w)(\lambda z.w)$
$\Rightarrow^\beta w$

β-redex

Multiple arguments:

$(\lambda x.\ \lambda y.\ add\ x\ y)\ E_1\ E_2 \xrightarrow{\beta} (\lambda x\ add\ x\ E_2) \xrightarrow{\beta} add\ E_1 E_2$

```
var add = x => y => x + y
alert (add(5)(1));
        5+1
```

```
add (x) {
    add'(y)
    return x+y;
}
```

Evaluation and the static scope → free variable.

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

(renaming is a must)

$\xrightarrow{\beta} \lambda y'.\ y'\ (y\ (\lambda x'.\ x'))$

$\lambda x y.\ x\ y$
$\lambda x y.\ add\ x\ y$
$\lambda x.\lambda y.\ add\ x\ y$

Reduction strategies

normal form (no more β redun)
(or δ rule)

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

$(\lambda x'.\ x'x')\ (\lambda x.xx) \rightarrow (\lambda x'.\ xx')\ (\lambda x.xx) \rightarrow (\lambda x'.\ x'x')\ (\lambda x.xx)$
(non terminating evaluation)
$\rightarrow \cdots$

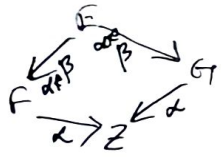
path 1: $(\lambda y.\ 5)\ ((\lambda x.\ xx)(\lambda x.\ xx)) \Rightarrow_\beta 5$ (normal order reduction)
(leftmost outermost β redu

2: $\cdots \rightarrow \cdots$ (applicative order)
(leftmost innermost β redu

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

Church-Rosser theorem I

Diamond property (confluence)



if two evaluation are β finished.
(terminated)

— II $E \overset{*}{\Rightarrow} N$ (N normal form)
normal order reduction from E to N.

Church's Thesis: lambda expression $\leftrightarrow$ turing m/c.

$(\lambda x.\lambda y.\ x+y)\ (\lambda z.\ z+1)\ (5)$

```
f1(x,y){
    return x+y
}
```

```
f2(z) {
    return z+1;
}
```

$(\lambda z.\ z+1)\ (5) + y.$

call by value → applicative order.

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

call by name → normal order / does not evaluate the argument prior to call.

δ-reduction

add (3 5)

⇒δ 8

η-reduction

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

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

$\lambda x.(5x)$ is not 5

1. Literals.
2. Data types. ⇒ collection
3. Conditional Constructs.
4. iterative Constructs.

Encoding Booleans in Lambda Calculus.

true = $\lambda x.\lambda y.\ x$ (then do)

false = $\lambda x.\lambda y.\ y$ (else do)

if_then_else = def $\lambda cond.\lambda then\_do.\lambda else\_do.(cond)(then\_do)(else\_do)$

weekday = true/false.

SleepDuration = if_then_else (weekday) (six) (eleven)

$\lambda cond.\lambda then\_do.\lambda else\_do.\ cond\ (then\_do)(else\_do)$

~~$\lambda(\lambda then\_do.\lambda else\_do.then\_do)\ \lambda then\_do.\lambda else\_do.$~~

~~$(\lambda then\_do.\lambda else\_do.then\_do)$~~ (6) ~~(11)~~

→ ~~$\lambda then\_do.$~~ $(\lambda td.\lambda ed.(\lambda x.\lambda y.x)$ ~~(6)(11)~~ $(td)(ed)$

→ (6) (11)

① Functional languages (20-40 marks)
(streams, collection, lambda expression). filter using any reduce.

② Logic programming (20) ③ lambda calculus (20) ④ mapping section in lambda calc. & prog. lang. (Javascript) (10-20)

⑤ design issues of obj. oriented lang.

⑥ apply topper prog. paradigm / mix of it / combination of it depending on problem description. ~~(10)~~ ⑤+⑥ → 20

28/2/2020

var identity = $x \Rightarrow x$.

alert(identity("Hello"));

polymorphism

$(\lambda x.x)$ (Hello)

β-reduction

true = $\lambda then\_do.\lambda else\_do.\ then\_do$.

false = $\lambda then\_do.\lambda else\_do.\ else\_do$.

H.W

if_then_else, switch case

not. $\boxed{\lambda b.\lambda td.\lambda ed.\ b\ (ed)\ (td)}$ $\lambda c.\lambda td.\lambda ed.\ c(td)(ed)$

truth table for not, and, or.

AND = $\lambda a.\lambda b.\ a\ b\ a$

$\lambda td.\lambda ed.\ td.$

true

if a = true then return b
else return false;

OR = $\lambda a.\lambda b\ a\ a\ b.$

Encoding pairs in Lambda Calculus

Head = $\lambda pair.\ pair\ true.$

Head(a,b) = (ab) true = a.

$\lambda a.\lambda b.\lambda f\ f\ a\ b.$

$\lambda g.g(\lambda a.\lambda b.\ a)$

$(\lambda g.\ g(\lambda a.\lambda b.a))((\lambda a.\lambda b.\lambda f.f\ a\ b)\ b\ 1)$

$= ((\lambda a.\lambda b.\lambda f.f\ a\ b)\ p\ q)\ (\lambda a.\lambda b.a)$

$= (\lambda f.f\ p\ q)\ (\lambda a.\lambda b.a)$

$= (\lambda a.\lambda b.a)\ (b\ q)$

$= b$

pair (one)

pair (one, pair (two, pair (three, ...)))

pair (five, pair (four, pair (three), pair (two, one)))

last element ↑ ↑ last element

pair (false, one) → [1]

pair (false, pair (two, one))

pair (false/true, list)

empty or not

non empty. empty. nil.

prepend

λ,item. λl. make-pair (false) (make-pair (item) (l)).

non-empty lists. [3,2,1] ⇒ empty = false, (3, (2, (1, null)))

make-pair (false) (make-pair (3) (make-pair (2), make-pair (1, null)))

single_item_list = prepend (one) (null).

= make-pair (false) (make-pair (one) (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))

Encoding Natural numbers in Lambda Calculus

<type> f(s) { return s; }. f(f(s)) → 2 f(f(f(s))) → 3.

f(s)

one = $\lambda f.\lambda s.$ f(s) two = $\lambda f.\ \lambda s.\ f(f(s))$

three = $\lambda f.\lambda s.\ f(f(f(s)))$

alert(toNumber(three));

var toNumber = n ⇒ n(i ⇒ i+1)(0);

zero = $\lambda f.\ \lambda s.\ s.$ (false)

AnyNumber = $\lambda n \lambda f.\lambda s.\ n(f)(s)$

successor of a number = $\lambda n\ \lambda f.\lambda s.\ f(n(f)(s))$

nth successor → $\lambda m \lambda n \lambda f \lambda s\ m(f)\ n(f)(s)$

add = $\lambda m \lambda n.\lambda f \lambda s\ m(f)\ n(f)\ s$

m+n = mth successor of n

multiply $\lambda m.\lambda n.\lambda f.\lambda s\ m(n(f))(s)$

Predicate

Here predicates.

is-zero = $\lambda n.\ n(\lambda x.\text{false})$ (true)

opposite = λ cond λ a λ b cond (b) (a)

is-even = $\lambda n.\ n$ (opposite) (true)

$(\lambda n.\ n(\lambda x.\ false))\ (true)\ (\lambda f.\lambda s\ f(s))$

$= (\lambda f.\ \lambda s.\ f(s)\ (\lambda x.\ false))\ (true) = (\lambda s\ (\lambda x.\ false)\ s)$ (true)

$= (\lambda x.\ false)\ (true) \quad = false.$

$(\lambda f.\ \lambda s.\ s)\ (\lambda x.\ false)\ (true) \quad = \underline{true}.$ (zero)

Predecessor

(create a pair $(n, n-1)$) **tail**

pair (zero, zero)

↓ f (successor) itself

pair (one, zero) $\xrightarrow{f}$ pair (two, one) .... pair (five, four)

make_pair ϕ combinator $(n, n-1) \Rightarrow (n+1, n)$

$\phi = \lambda p\ \lambda f\ f\ (succ\ (p\ true))\ (p\ true)$

make pair (one) (two) $= (\lambda_{left}.\ \lambda_{right}.\ \lambda_f.\ f\ (left)\ (right))\ (one)$

$= (\lambda_f \cdot f\ (one)\ (two))\ (true).$

$\phi = \lambda_p \lambda_f \cdot f\ (succ\ (pair\ true))\ (pair\ true)$

$(true)\ (one)\ (two) = \underline{one}$

$P = \lambda n.\ n\ \phi\ (\lambda_f.\ f\ zero\ zero)\ (false)$

predecessor of n.