Unit 5

PPL

→ imperative lang design → Von Neuman Architecture.

→ functional lang → mathematical func.

Features of FPL

* computation → obt by applying Maths func — to parameters
* pure FPL → ✗ variable.
* prev def func — built → complex func producing (final result for initial data

Advantages

* not concerned with variables
* less labour
* simple syntax & semantics.
* concurrent prog execution → easy to understand
  easy to design.
  easy to use

Disadvantages

* less efficient.
* slow.

Mathematical functions

* mapping of domain set with another range set
* λ expression specifies parameters & mapping.
* Eg: $\lambda(x) = x * x * x$.
* For function cube $(x) = x * x * x$.

Lambda Expressions

* nameless fun its describes.
* applied to → parameters.

Functional forms (2 types)

* higher order function.
* takes functions as parameters.
* Eg: functional composition takes 2 funct as parameters

$h(x) = F(g(x))$

$f(x) = x + 2$. & $g(x) = 3 * 2$

Eg $g(x) = x * x$.
$h(x) = 2 * x$.
$i(x) = x/2$.

then.

$[g, h, i](4) = g(x) = 4 \times 4 = 16$
$h(x) = 2 \times 4 = 8$
$i(x) = 4/2 = 2$

$\therefore [16, 8, 2]$.

Apply to all function

↓

single fun as parameter

Eg: $h(x) = x * x$

$\alpha(h, (2, 3, 4))$.

$(2*2, 3*3, 4*4)$.

$= (4, 9, 16)$.

Fundamentals of FPL

* Only of design of FPL is mini Maths func to greatest extent as possible
* variables r not necessary.
* evaluation of function → same result from same parameters

FLP's on LISP : INTRO & FEATURES

* 1st language for — FPL.
* oldest & widely used. ATOMS.
* 2 types of data obj : LISP & LISTS.
* Eg (AB (CD/E).
* Org LISP → Typeless lang.
* LISP lists r stored internally as Singly Linked List
* λ notations r available.
* first lisp interpreter appear as demo.
* automatic garbage collection.
* Base on expressions
* Advanced OOPS features.
* Strong support for recursion.

APPLICATIONS OF LISP

* AI, games, Pattern recog, apps of edu, Real time embedded syst

Expressions in LISP

* Operands & Operators are used
* operator operand1 operand 2.
* Eg1:- (+ 10 20)
  30.

Eg 2:- (* (+ 2 3) (+ 5 6))
  55.

Operations performed on ATOM & LISTS in LISP

a) Atoms:-
* All exp in LISP is made up of lists.
* lists: linear arr of obj sep by blanks & surrounded by brackets
* obj are made up of ATOMS.
* space separated elements of list r cld ATOMS
* Ex:- name
  123
  Abc.
  a
  cd.

b) LISTS.
* they are parenthesized collection of sublists or atoms.
* Eg.
  (ab (cd) e)
  (10 20 30).

LISP Functions

① car :- (car '(1234)) = 1.
② cdr :- (cdr '(1234)) = 234.
③ cons :- (cons 10 '(20 30 40)) = 10 20 30 40

LISP MANIPULATIVE FUNCTIONS

① Length :- (length '(10 20 30)) = 3

② Append :- (append '(10 20 30) '(40 50))
= (10 20 30 40 50)

③ Map :- (map+ '(1 2 3) (4 5 6)) = (5 7 9).

④ Associative lists :- (assoc 'age '((name aa) (age 30) (sex m)))
= AGE 30

conditional Statement in LISP

(if (> 10 20) 10).

∴ As 10 > 20 is false, NIL will be returned.

Quoting

> pi = 3.14159.

> (quote pi) or > ('pi) = pi.

defining function in LISP

syntax defun (proc_name (arg1 arg2)
body of procedure).

ex > (defun mysum (a b)
(+ a b)).

now we can use it.

> (mysum 2 3)

5.

Imperative lang support FP?

* JS : function name (formal-param) {
  body
  }.
* C# : support lambda expression.
  i => (i % 2) == 0.
* Py : lambda a,b : a+b

Diff b/w Imperative lang & FL.

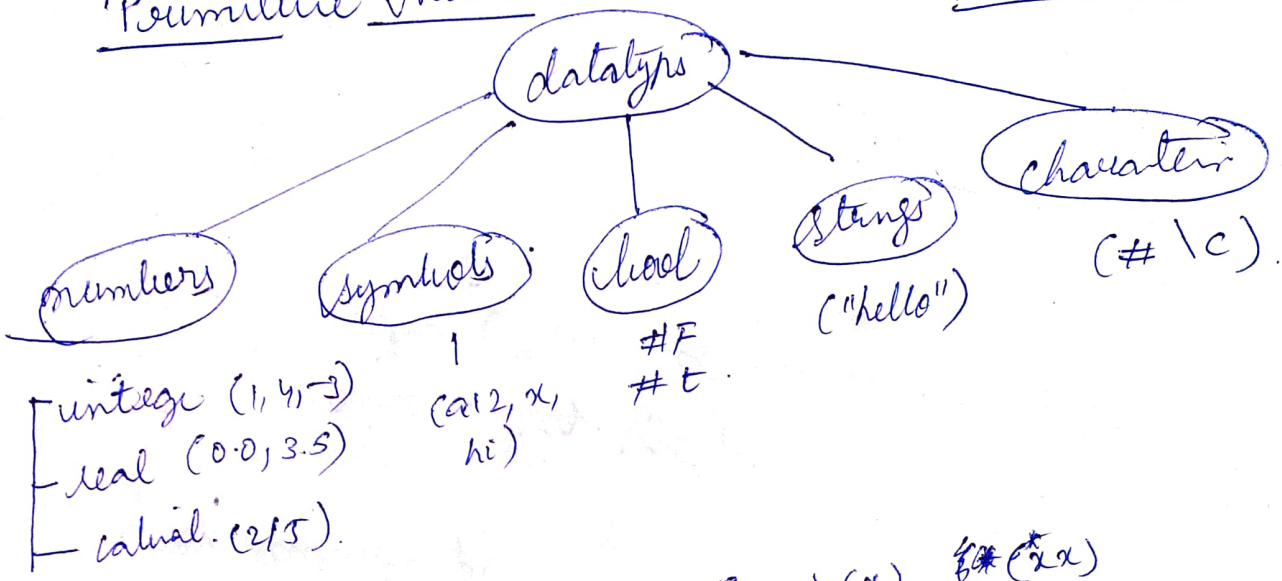
IL	FL
* need not be concerned with variables	* must use variables
* less efficient	* more efficient.
* less labour	* more labour.
* simple syntactic structure	* complex syntactic str
* simple syntax	* complex syntax
* dynamically divided into comment parts	* statically divided
* Order of execution is imp	* Order of execution X imp
* control flow manipulated by control statement	* CF manipulate by functional calls & Recursion
* Eg: C, Java, Py, JS, Ruby etc.	* Eg: LISP, Haskell, ML

Scheme

* Used for technical computer concepts
* LISP is for practical day

Primitive fumt.
- CAR
- CONS
- CDR
- QUOTE
- ARITHETIC

Primitive Numeric Functions



* namelless fumt – LAMBA. Eg λ(x) (* 2 x)
* output fum – (DISPLAY expression) & (NEWLINE)

Predicate Numeric Functions.

* Returns boolean value
  Eg:- =, <>, >, <, >=, <=, EVEN?, ODD?, ZERO.
* control flow.
  → if.
  → grouping conditions

ML

* Meta lang
* Syntax is close to pascal.
* Static scoped FL.
* Strong & applicative approach
* Similar to scheme
* Includes Exception Handling
* Implements abstract Data types

* error parity available
* control flow available

do prgm execution.

* function :-
  Fun Sub (p, q) :
  .
  .
  unit p-q.

- Value declaration :

Syntax :- val identifier = expression.

ex :- val velocity = acc * t ;.

- Expressions .

① Arithmetic :- *, /, div, mod → multiplicative operator
+, − → additive "

② Boolean :- andalso, orelse, not
T F complint.

③ String expansion :- "Hi" ^ "Bye" = "Hi Bye".

④ control flow
- if then - if (exp) then true . else false
- while - while exp1 do exp 2.

- List Expressions

* ML supports lists & list operations
* lists are rep in square brackets.
* & their elements are seperated by commas
* [1,2] @ [3,4] = [1, 2, 3, 4]

Haskel

* purely FPL.
* has lazy evaluation technique
* supports static type checking.
* makes use of functions to define offside rule.
* Uses List comprehension technique
* supports pattern matching
* active & growing community exists.

Haskel VS ML

HASKEL

* Haskel is pure, without side effects or exceptions
* Haskel has lazy evalution
* Haskel functions are curried
* minimal syntax

ML

* ML has mutable data structures such as arrays.
* ML has strict evalution.
* ML uses tuple args for multi args.
* heavier syntax.

Applications of functional prog

* APL is used for throw away prog.
* LISP is used for AI, ML, NLP....
* Use of ML & haskel are confined to research labs

Logical Programming Languages

Intro

* called declarative programming
* express prog in form of symbolic logic
* use a logical reasoning process to produce res
* proposition: A logical statement that may or may not be True
* symbolic logic: [express proposition
  express relation b/w 2 propositions
* Object representation: [constants - same obj
  variables - diff obj
* Compound terms: Atomic proposition consist of comp terms
* consists facts & rules
* can be viewed as control deduction
* Algo = Logic + control
* concept of logic prog is linked up built Prolog
* Prolog is an acronym for Programming Logic

Applications of Logical Prog Lang

* Prolog is used in AI & expert system to solve complex problems, RDBMS, NLP
* It is used on pattern matching algo
* Used to develop systems based on OOPs concepts
* Prolog is used to build decision support systems

Features

Relations, Unification, Backtracking, Recursion, Cuts

* Basic elements of Prolog.

Rules. Facts. Goals. Terms. Queries. Relations.

(1) Relations

* Prolog uses Relations instead of functions
* Relations are more flexible.
* ~~e.g. (a,b,c)~~
* As w.r.t [H/T] means Head covers first & the Tail. Hence prolog will treat [a,b,c] as [a/[b,c]].

(2) Queries

* Simple queries are created to check whether tuple belongs to a relation.

(3) Terms.

* Prolog term is constant, a variable, a structure.
* constant is either an atom or integer.
* variable is any string of letters, digits.

(4) Goals

* Starts with predicate.
* for goal to be valid, predicate must have appeared in atleast one fact or rule.
* All arguments r constants.
* The purpose of submitting a goal is to find whether the statement represented by the goal is true according to the knowledge database.

⑤ Fact

* If a goal is hypothesis, the facts are axioms.
* Start - predicate
* end - full stop
* Args can be constants, no's, variable's, lists
* Syntax : Pred(arg1, arg2, arg3)...argn)...

⑥ Rule

* If a goal is hypothesis, the rules are theorems.
* Rule is intention to fact.
* Consists 2 parts
* Syntax : head :- body.

Scripting Programming Languages

Pragmatics

* Embedded within HTML, adds functionality
* Uses interpreter
* They run inside another prog
* Designed to make coding fast & simple
* Does not create .exe file
* Scripts are just view of code
* Easy to learn & write
* Flexible dynamic typing
* High level data types

✱ Advantages of Scripting languages

* Easy to learn & use.
* Requires min prog knowledge.
* & performs complex tasks in few steps.
* Simple creation & edit in variety of text editors.
* Allows dynamic activities to web pages.
* Editing & running mode is fast.

✱ Common Characteristics of Scripting language.

* These lang are kept in source form & not in compiled form
* Dynamically typed lang.
* not concerned for underlying machines.
* These are extensible.
* They are application specific
* Have interactive shell.
* Support event handling.
* emphasize flexibility.
* emphasize rapid development.

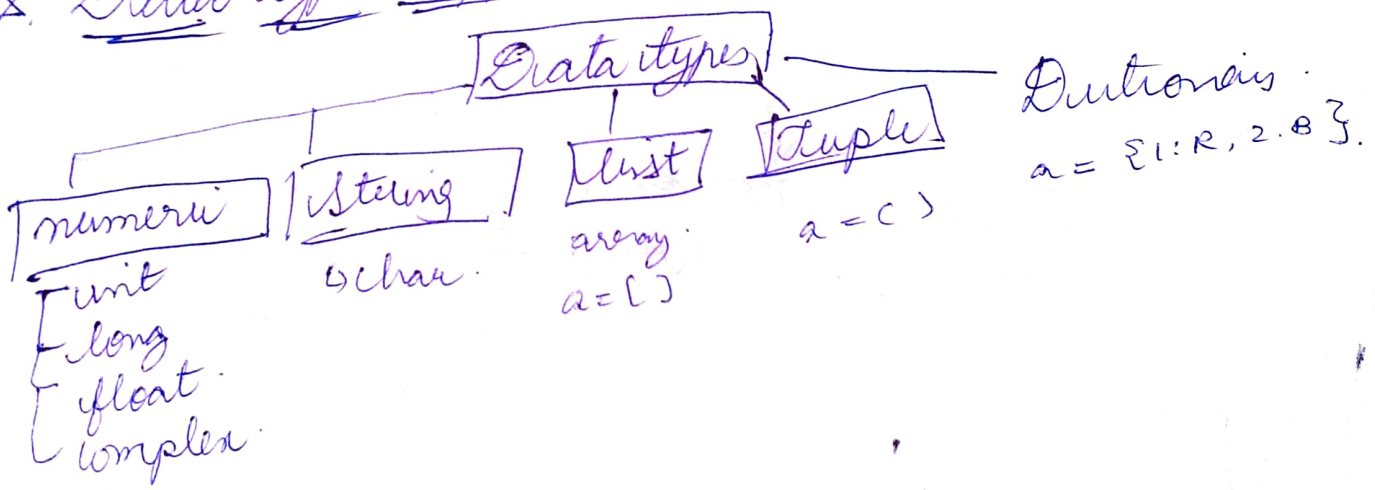
✱ Key concepts of Scripting lang.

* Simple scoping rules.
* Direct access to other program.
* Flexible dynamic typing.
* High level pattern matching
* Have & In built high level data type.
* Direct access to other program.

Case study - Python

Features of py
* High level prog lang.
* easy & simple
* portable.
* OOPS
* rich set of functionality
* Inbuilt in data types

Data types supported in Python



Storage & Control Statements

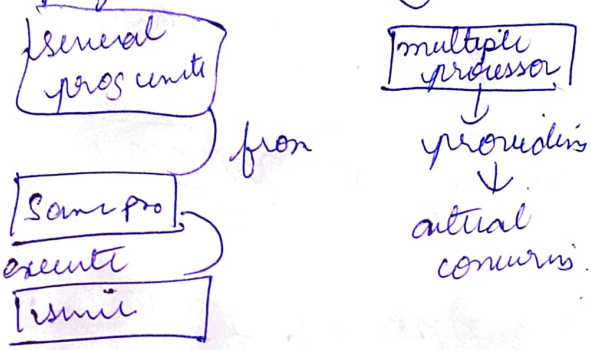
① Selection
- if
- if else
- if-elif-else
- while loop
- for loop

Unit - 4

Subprog level concurrency

Levels
- Instruction level — 2 OR MORE Machine instr.
- Statement — 2 OR MORE. High level language
- Unit — 2 OR MORE subprog units

2 concurrency - physical & logical



✱ Semaphores

* Shared variable
* Used to :- synchronise the process of interactions process
* 2 operations :- [wait P() ↓
  signal V() ↑.

* Working
  1. initialise non-ve value.
  2. wait — a ↓ Sema value by 1.
     — if value become -ve, process is blocked.
  3. signal — ↑ sema value by 1.
     — if value become +ve, process unblock.

(4) Ex :- * Let P_1 & P_2 in a semaphore is initialized to 1,

* If P_1 enters critical section, value of sema becomes 0.
* Now P_2, wants to enter critical section, then it will wait until $S > 0$.
* This can only happen when P_1 finishes critical section & calls V operation on semaphores.
* This way mutual exclusion is achieved

* Types of Semaphores

Binary | Counting

Binary :- * Used when there is only one shared resource.

* 2 states [Acquire / Take.
* no ownership - can be released by any task.
* only 1 thread can access the resource at a time

Counting * To handle more than one shared resource

* initialized with count (N)
* allocates resource as long as count becomes 0.

** Binary Semaphore provides soln to compilation sync prob with example Pseudo code

* Consider an implementation of counting sema S
* The impln needs to keep a track of value of S, i.e S.val
* If a process needs to wait on S, the impln has to make the process wait.
* Bcaz: Binary semaphores are the only sync construct allowed in implementation
* Consider P(S) & V(S) to denote wait & signal.
* Pseudo code:

record S {
integer value initially K;
BinarySemaphore wait initially 0.
}.

P(S) {
if S.val = 0
then {P(S.wait)}.
else { S.val := S.val - 1 }.
}

V(S) {
if "process are waiting on S.wait"
then {V(S.wait)}.
else {S.val := S.val + 1}.
}

Monitors

* Monitor type is an extract data type.
* Its construct allows one process to activate at 1 time.
* Only one process can be active in a monitor at a time.
* Other process which needs to access shared variables in a monitor have to line up in a queue.
* They are provided access when previous process release shared variable.

Advantages of Monitors over Semaphores

	M	S
① Mutual Exclusion	automatic	manual.
② Shared variables	global to all processes	hidden
③ Synchronization prov by	conditional variables	no conditional variables
④ Access to shared through V	provided	not protected

⑤

Message Passing Priority in ADA

Pragma Priority.

pragma Priority (static expression);

The priority of a task applies to it only when it is in task ready queue

Java Threads

Thread - light weight program.

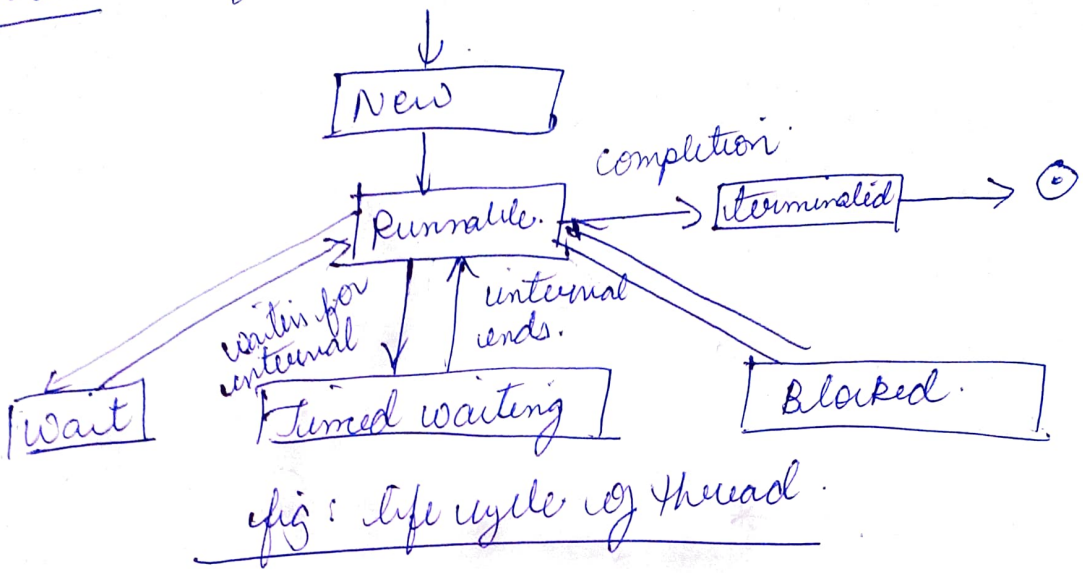


fig: life cycle of thread

Thread creation

Thread class

Runnable interface

```
class MyThread extends Thread
{
    public void run()
    {
        Sysout ("Thread");
    }
}

class ThreadProg
{
    public static void main (String args[])
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

- we have to implement Runnable

obj = new

Methods to control threads

- Sleep
  - block thread
- Join
  - delay execution

Thread Priority - default NORM-PRIORITY

- set priority
- get Priority
  - MIN-PRIORITY
  - MAX-PRIORITY

Thread_Name.SetPriority(priority_val)

Thread_Name.getPriority()

Sync in Thread Execution

- Using Sync Methods
- Using Sync blocks

Co-operative sync achieved by wait(), notify(), notifyAll()

C# thread

System.Threading.Thread

JAVA	C#
To start a thread we write t.start()	t.Start().
To kill the thread we write t = null;	t.Abort()
To interrupt thread. t.stop().	t.Interrupt()

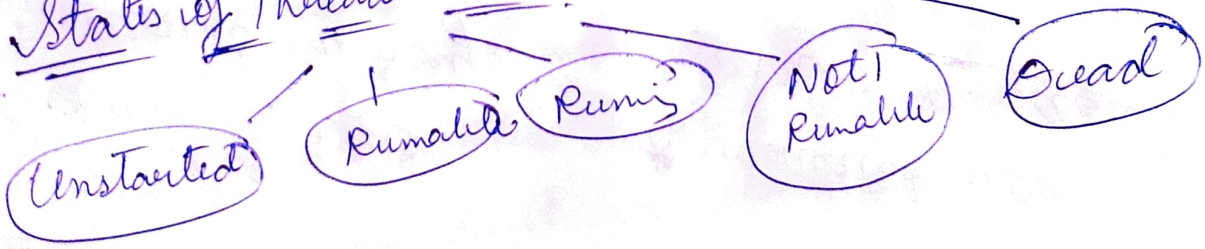
Creation :

Make a class extend TC(R) RI

Thread t = new Thread(class).

Thread t = new Thread(new ThreadStart(method)).

States of Thread in C#



✡ Design Issues of Exceptional Handling

* How and where are exception handlers specified & what is their scope?
* How is an exception communicated bound to an exception handler?
* Can info about the exception be passed to the handler?
* Where does execution continue, if at all, after an exception handler completes its execution?
* Is some form of finalization provided?

✡ Exception Handling Exception in C++.

* When run time errors occur in our prog, then exceptions r raised by handing control to special functions called Handlers.
* This provides built-in error handling mechanism which is known as exceptions handling.
* 3 keyword, try catch, throw.

```
try
{
throw exception;
}
catch (argument)
{
}
```

* try block - error detection.
* error occur - throws.
* using catch, error is caught.
* catch should be immediate follow by try block.

Ex: divide by zero

```
int main()
{
  double i,j;
  void divide(double, double);
  cout << "Enter numerator": " j
  cin >> i;
  cout << "Enter denominator : ";
  cin >>j;
  divide (i,j)
}

void divide (double a, double b)
{
  try {
      if (b==0)
          throw b // div by 0.
      cout << "Result : " << a/b << endl.
    }
  catch (double b)
    {
      cout << "can't divide by zero \n";
    }
}
```

Exception Handling in Java.

```
class Demo
{
public static void main (String[] args).
{
int a, b, c;
a = 10;
b = 0;
try {
    c = a/b
}
catch (ArithmeticExpression e) {
    System.out ("In Divide by 0");
}
System.out ("In The value of a: " + a);
System.out ("In The value of b: " + b);
}
}
```

Output -

Divide by zero.
The value of a : 10.
The value of b : 0.

* Exception Handling in ADA

Exception:

* The frame of an exception handler in ADA is either a subprogram, a package body, a task, or a block.
* Many exception handlers are usually local. the exption raisd do not have parameters.

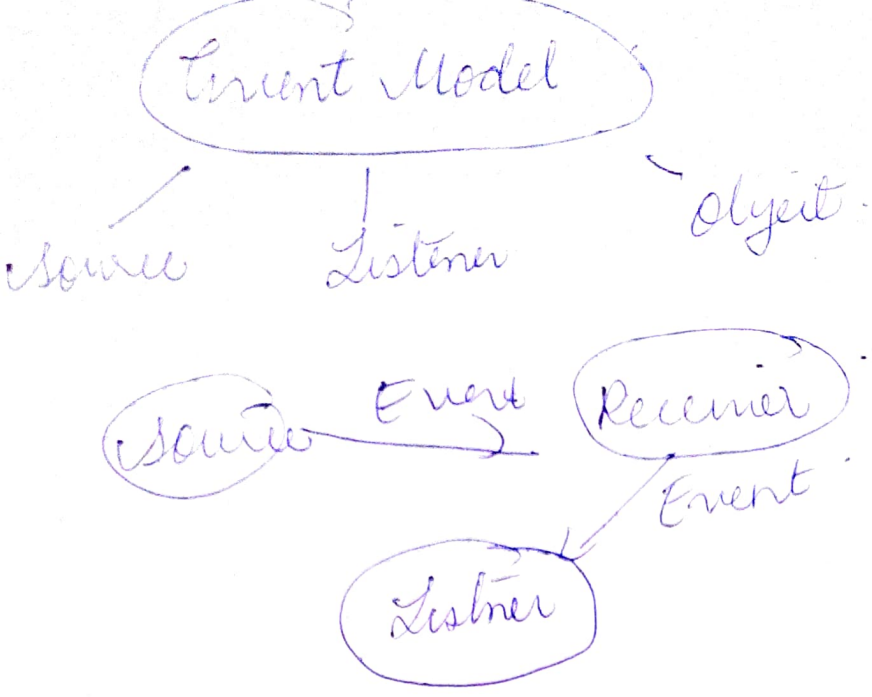
* Syntax.

User-difinid Exception form:

exception_name_list: exception;

Raising Exception form.

raise. [exception name]



Event Handling - 90

Unit 3

Subprogram: Small programs written with a large main program.

- **procedure** — no return — F P — ADA, Pascal
- **functions** — return a val — P F X, , C, C++.

☆ Subprog char

* Each subprog has a single entry point.
* The calling prog is suspended during execution
* control always returns to the caller. when the caller subprog terminates

☆ Design Issues of subprog

* Are local variables static or dynamic?
* can subprog definations appear in other subprog definations
* what parameter passing methods are provided?
* Are parameter types checked?
* If subprogs can be passed as parameters & subprogs can be nested, what is referencing environment of passed subprog?
* Can subprogs be overloaded.
* Can " " generic.
* If lang allows nested subprog, are closures supported?

Referencing Environment

- Local referencing env
- non local RE
- Global RE
- Predefined RE

PPM – 58
PSP – 62
OSP – 63. Q27
QSP – 64 Q28
DIF – Q31
VDOO – Q33, Q34
C – Q35
CR – Q36
ISS – Q39
IDS – 46, 52
Q57.
ZE – 61, 62

Part (1)

Q1
Q2
Q3
Q4
Q5
Q7
Q16
Q20
Q31
Q33
Q36.
*Q46
Q51
Q54.

(2)

Q2 Q70
Q6 Q71
Q8. Q72,
Q18 Q78
Q19. Q84
Q29 Q86
Q30
Q34.
Q38
Q40
Q45
Q54
Q63

Syntax: set of rules that defines what sequences of symbols r considered to be valid (expression).

* source prog → obj prog

Valid exp - x=y+z.

Invalid xy+-*

ex 2+3×4
↑ high.

2+12
↑ high

Semantic: meaning of a expression [program] in programming lang.

ex: int a[10]
defines an array.

* Explain various primitive data types with examples:

* Primitive data types are elementary data types which are not built from other types.
* The value of Primitive DT is atomic ie can't be decomposed further.
* The primitive data type reflects the behavior of the hardware.

① Boolean - T or F.
Boolean algebra - and, or, not

② character :- char.
In C, - char choice;

③ integer :- max & min limit.
In C, int, short, long, char.

④ real :- Real no - mantisa exponent model
In C, floating point - float

Content free grammers

* It is simply a grammer
* collection of 4 things
① set of (tokens) or terminals.
② set of non-terminals (variables).
③ set of (rules) called production . :=
④ starting non terminal .

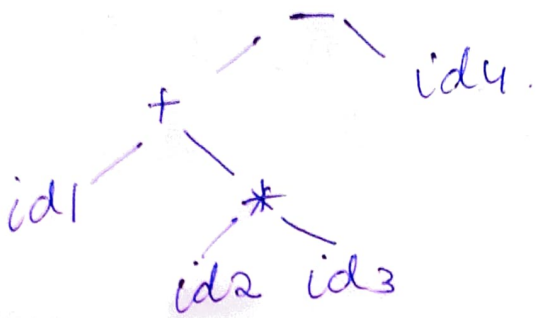
Eg:-

E::= E+T | E-T | T

T::= T*F | T/F | F

F::= (E) | id

parse tree for id1 + id2 * id3 - id4



* CFG is a formal grammer which is used to generate all possible strings in a formal lang.

$G_f = (V, T, P, S)$

=

* Ambiguity

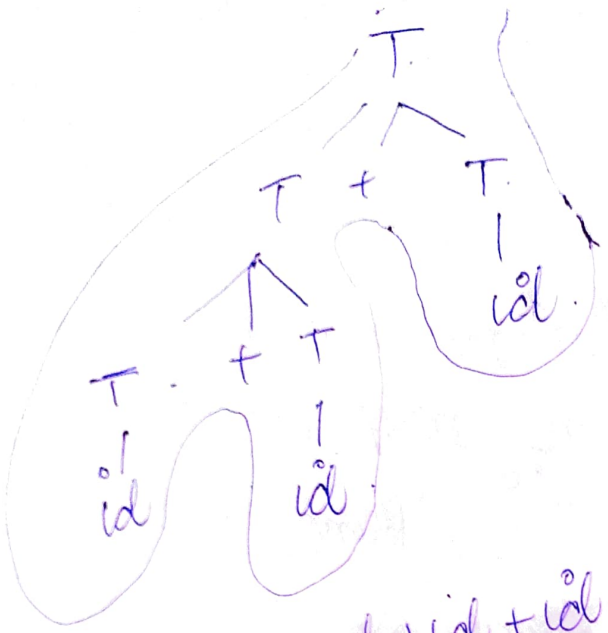
* A grammer is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than 1 parse tree for the given input string.

* If a grammer is not ambiguous, its unambiguous.

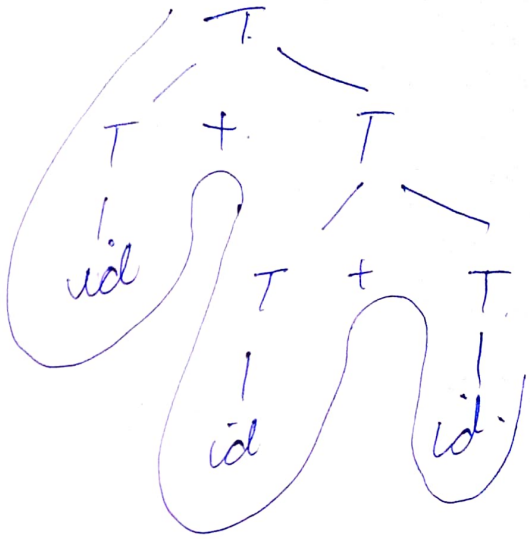
Eg: $T \rightarrow T+T/id$.

$\rightarrow id + id + id$.

LMDT



(OR)



id + id + id

id + id + id

Exists more than 1, derivation tree for a given input str.

Unambiguous – unique derivation tree.

Eg 2 Consider a grammar G is

$S \rightarrow AB/aaB$.

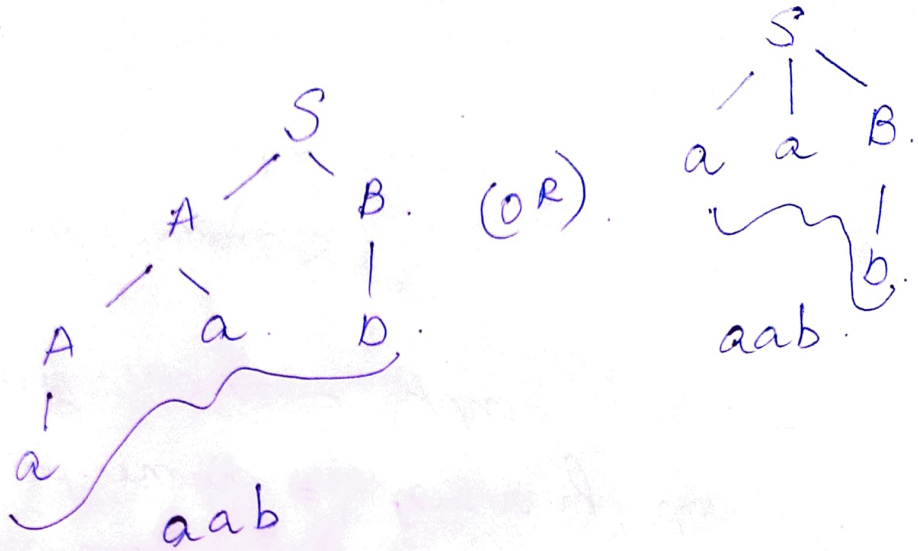
$A \rightarrow a/Aa$.

$B \rightarrow b$.

Determine whether the grammar is ambiguous or not. If G is ambiguous, construct an unambiguous grammar equivalent to G

sol "aab".

Input is string.



(OR)

∴ grammer is ambiguous.

production

$S \to AB/aaB$ } ambiguous.

$A \to a/Aa$ }

$B \to b$

↓

unambiguous.

left recursion.

$S \to AB$.

$A \to Aa/a$.

$B \to b$.

Arrays

(1) Static Array.

* An array whose size is known & whose storage is allocated at compile time.

  Eg int static_array [7];

(2) Fixed Stack Dynamic Array.

* You know the size of your array at compile time, but allow it to be allocated automatically on stack.

```
void foo()
{
  int fixed_stack_dyn_arr [7];
    /* ... */
}
```

(3). Stack dynamic Array

* you dont know the size until runtime

```
void foo(int n)
{
  int stack_dyn_arr [n];
}
```

③ fixed heap dynamic array :- same as ② but explicit heap allocation

```
int * fh_d_a = malloc (7 * sizeof (int));
```

⑤ Heap dynamic array.

```
void foo (int m)
{
int * h_d_a = malloc (m * sizeof (int));
}
```

---

* Nested subprog.
  * Organizing program by nesting of subprogram definitions inside the logically target subprograms that use them
  * It is sup in ADA, Fortran, Python, Ruby.
  * Encapsulation is possible in C & C++

---

* Design issue for OOPS.

When we design a prog lang a no of issues must be considered to support inheritance & binding.

① Exclusivity of objects ?

② Use subclasses subtypes?

③ Type checking & polymorphism.

④ Single & multiple inheritance.

⑤ Object allocation & deallocation.

⑥ Dynamic binding & static binding.

⑦ Nested classes.

⑧ Initialization of objects.

Parameter Passing Techniques.

① call by value

② call by reference.

③ copy restore.

④ call by name

① * caller evaluates the actual parameters
* passes r-values of actual " to formals.

```
void swap (int x, int y).
{
    int t;
    t = x;
    x = y;
    y = t;
}
```

```
void main()
{
int a=1, b=2;
swap(a,b);
printf("a = %d, b = %d", a,b);
}
```

o/p = 2,1.

call by reference (location)

```
ex: void swap(int *x, int *y)
{
int t;
t = *x
*x = *y;
*y = t;
}
```

o/p = 2,1.

```
void main()
{
int a=1, b=2;
swap(&a, &b);
cout << "a = " << a << "b = " << b;
}
```

copy constructor

hybrid

```
int y;
copy restore (int x)
{
  x=2;
  y=0;
}
main()
{
  y=10;
  copy restore (y);
  print(y);
}
```

o/p = 2

call by name.

* macro expansion.

```
int i=100;
void call by name (int x)
{
  yprint (x);
  yprint (x);
}
main().
{
call bynome (i++)
}
```

Recursion in Scheme

* Recursive func in schemes.
* function are diff in Scheme & M2

Predicate Calculus

Predicate calculus provides a method of expressing collection of proposition.

Resolution :- an interface principle that allows unified proposition to be computed for a given proposition

Ex :- $P_1 \subset P_2$

$Q_1 \subset Q_2$

Meaning :- P_2 implement P_1 & Q_2 implements Q_1.

if $P_1 = Q_2$.

$\Rightarrow Q_2 \subset P_2$

$Q_1 \subset Q_2$.

$\therefore Q_1 \subset Q_2 \subset P_2$

$\therefore Q_1 \subset P_2$.

* LISP (List Structure programs)

Significance :

* It is machine independent.
* Uses iterative designing methodology
* It allows updating the program dynamically
* It provides advance loop.
* It prefers prefix notation