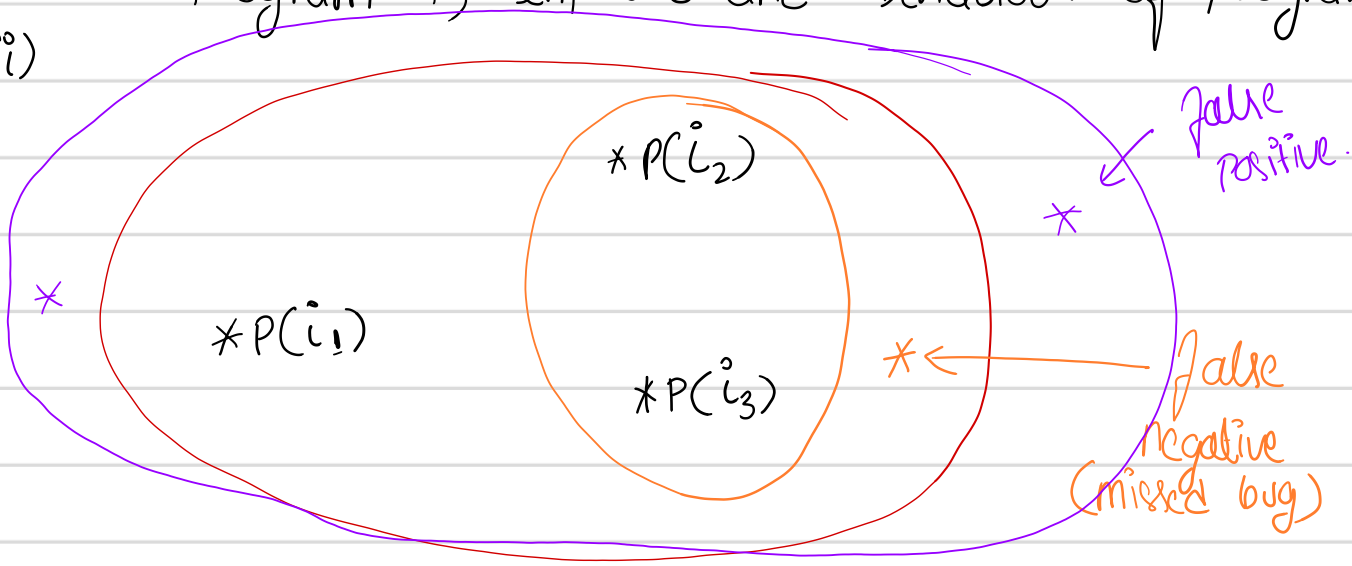
Program Analysis

Let Program P, input i and behaviour of program P(i)



All possible behaviours (This is what we want ideally)
Underapproximation (Example :- testing, dynamic analysis
Overapproximation (most static analysis)

FP => ಇಲ್ಲ ಅಂದ್ರೆ ಇದೆ ಅಂತ
FN => ಬೇಕು ಅಂದ್ರೆ ಇಲ್ಲ ಇದೆ

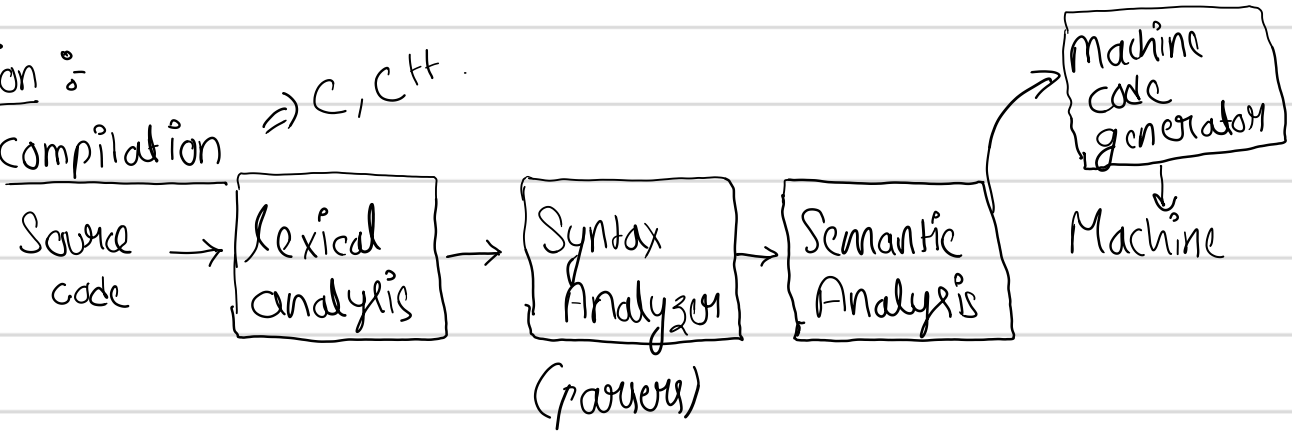
Foundation of PA :-

Programming language = Syntax + Semantic + implementation

- Syntax
  - how do they look like
  - what characters are written (rules)
- Semantic
  - meaning
  - what will happen
- implementation
  - Execution

Implementation :-

a) Compilation ⇒ C, C++



Phases of compiler

b) Interpretation ⇒ Executing a program directly from the source code without first translating it into machine code.

c) Hybrid ⇒ python.

↳ Java, Java Script ↳ python.

Syntax :-

a) Grammar.

Arithmetic expression

Terminal $\Sigma = \{0, 1, 2 \ldots 9, +, -\}$

Nonterminal $N = \{Exp, num, op, digit\}$

$S = Exp$

$P = Exp \rightarrow num \mid Exp\ op\ Exp$

$op \rightarrow + \mid -$

$Num \rightarrow Digit \mid Digit\ Num$

$Digit \rightarrow 0 \mid 1 \mid 2 \ldots \mid 9.$

Q. What is the part of language ??

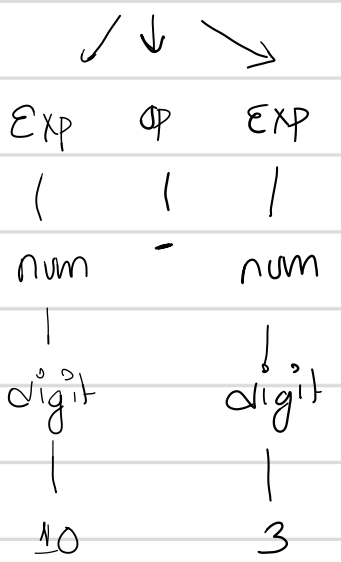
a) 10 - 3 ✓

b) 11 x4 ✗

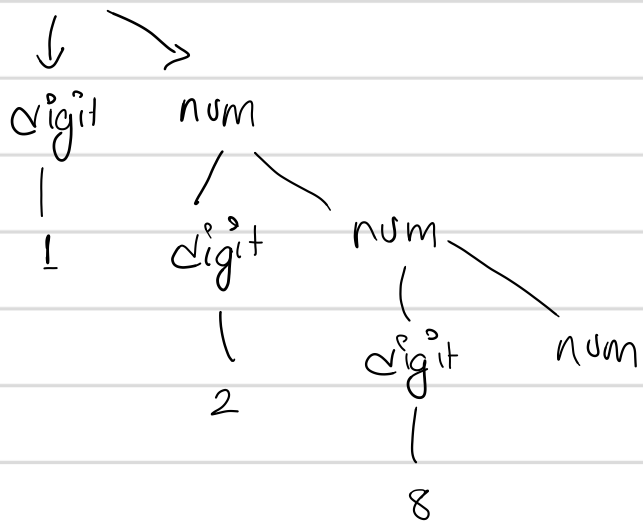
c) 128463 ✓

d) 10 + (4 - 3) ✗

a)



b)



and so on.

ⓗ Abstract Syntax trees :-

Abstract Grammar.

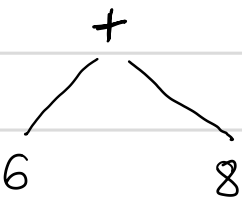
Eg:- $E \rightarrow n \mid op(E, E)$

$op \rightarrow + \mid -$

- terminals are the tokens.

Example :

6 + 8



ⓗ Control flow Graphs

↳ Models the flow of control through the program.

$G = (N, E)$ where, Nodes are basic blocks

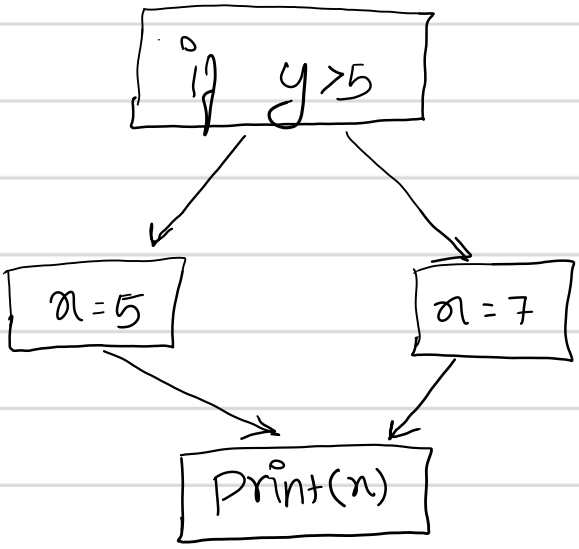
(Sequence of operations executed together)

Graph is Node & edge.

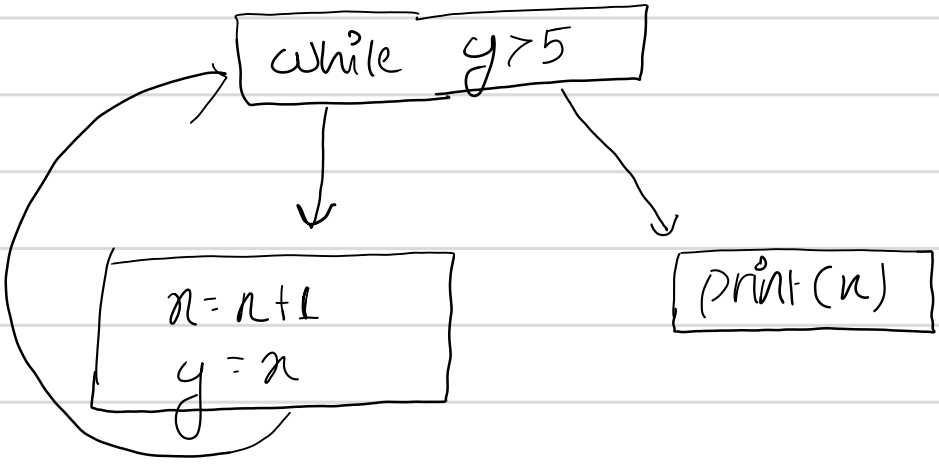
E possible transfer of control

Example:-

①
```
if y>5:
    x = 5
else:
    x = 7
print(x)
```



②
```
while y>5:
    x = x+1
    y = x
print(x)
```



③ Control flow graph focuses on the order of the execution of the program.

Data dependency graph :-

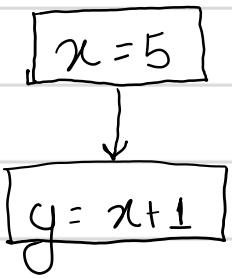
- It models the flow of data from "definitions" to "uses"

$G = (N, E)$; N is operations
E is possible def-use relationship

$e = (n_1, n_2)$ means : at node n_2 some data may be used that is defined at n_1

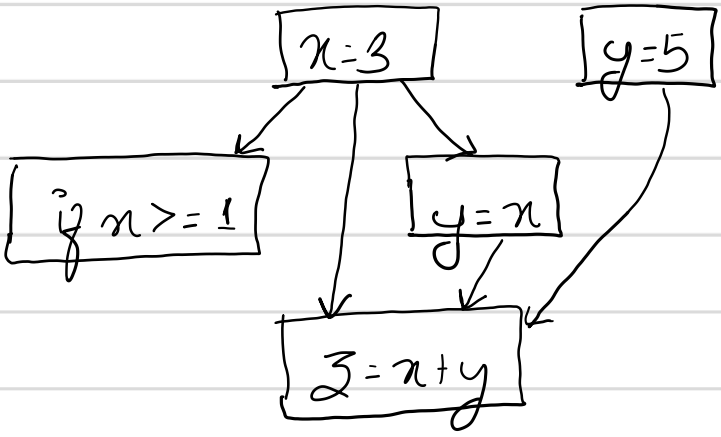
Ex1

```
x = 5
y = x + 1
```



Ex2

```
x = 3
y = 5
if  x >= 1 :
    y = x
z = x + y
```



Ingredients of PA:

① Semantic Properties & Syntatic Properties.

⇒ Semantic property can be completely defined w.r.t the set of executions of a program, while a Syntatic property can be decided directly based on the program text.

which properties.

1. A program may print text to the console
2. A program may call print
3. A program prints to console.
4. A program consists of one line of code.

→ 1, 2 & 4 are Syntatic properties
→ 3 is Semantic property.

```
if x == 1:
    print("Hello")
```

Suppose this is the code and here are the result.

Rice's Theorem:

- It states that all non-trivial Semantic properties of program are undecidable.
- A property is non-trivial if it is neither true for every partial computable function, nor false for every partial Computable function.

Proof:
Halting Problem is a problem of determining from

1) a description of an arbitrary computer program
2) an input

whether the program will finish running or continue to run forever. A general algorithm to solve problem for all possible input pairs cannot exists.

⊗ we can make approximation

Non-trivial Properties

- A program exists
- A program prints "Hello"
- A program finishes in less than 5 seconds
- A program dies with "Segmentation Fault"
- A program prints user password to the console.

Functionality bug ↖

Security bug ↖

Nontrivial Properties

- A functional property about programs is nontrivial if some programs have the property and some do not
  - Program never goes into an infinite loop — nontrivial
  - Program accepts a finite number of strings — nontrivial
  - Program accepts 0 or more inputs — trivial

Static Analysis:-

```
def fun():
    x = 0
    return 42/x
```

Bug: yes.

```
def fun():
    return 42/x
```

Bug: no

C++ → clang Tidy

Python ⇒ pyright

Rice Theorem → no 100% Sure.

Style checking:

C++ → cpplint.

Python → pylint.

```
int f (int x)
{
    return 42/x;
}
```

C++ ⇒

```
int f(int x){
    return 42/x;
}
```

⇒ Extra Space before (function call. &
Should be at end of previous line

Dynamic Analysis is testing [Address Sanitizer]

Consider the following Program.

→ Pyright
→ Mypy
→ Bandit

```
def fun(x):
    list = [1, 2, 3, 4]
    return list[x]

print(fun(5))
```

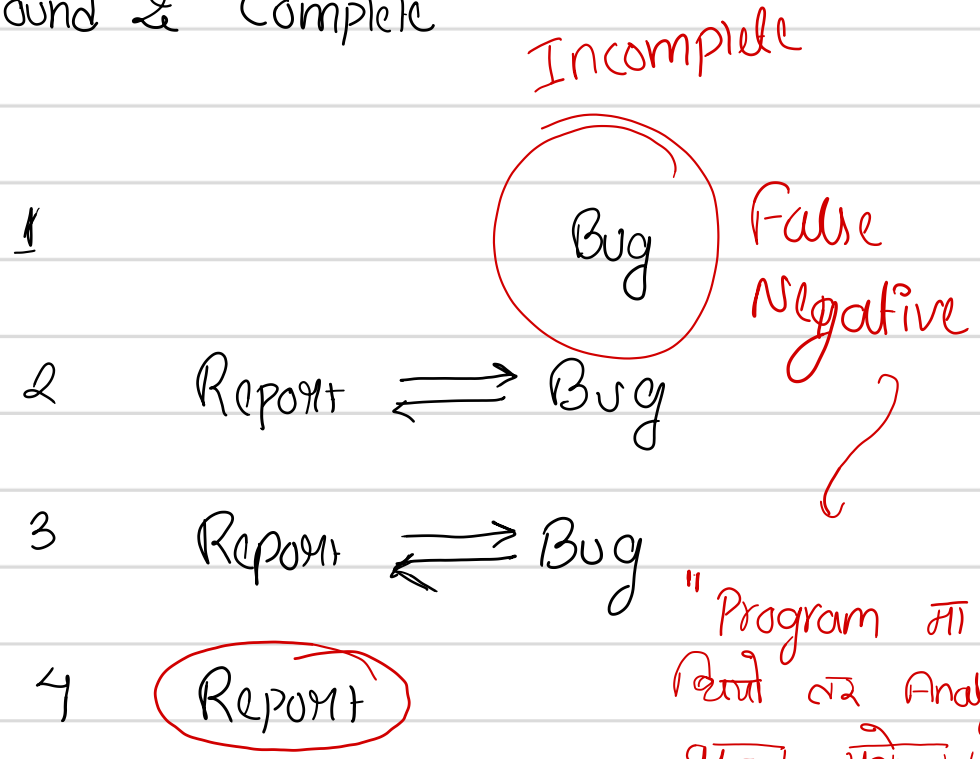
```
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 7, in <module>
  File "<main.py>", line 5, in fun
IndexError: list index out of range

=== Code Exited With Errors ===
```

Index error:

List index out of range

Quality of Analysis :

Sound & Complete



Q. Analyze the following code samples using Pylint, pyright and prepare the report

```
# script1.py

import os

def divide(a: int, b: int) -> float:
    return a / b

result = divide(10, 0)
```

1. no ⟶ Bug

2. yes ⟶ Bug

3. yes. ⟶ Bug

4. yes ⟶ no Bug.

```python
# script2.py

def greeting(name: str) -> int:
    return f"Hello, {name}!"

message = greeting("World")
print(message)
```

```python
# script3.py

def add_numbers(a, b):
    result = a + b
    return result

x = add_numbers(3, 4)
print(x)
```

```python
# script4.py

def check_value(value: int):
    if value > 10:
        return "Greater than 10"
    return "Less or equal to 10"
    print("This is unreachable")

print(check_value(5))
```

```python
# script5.py

def append_to_list(item, my_list=[]):
    my_list.append(item)
    return my_list

print(append_to_list(1))
print(append_to_list(2))
```

Ⓐ Precision = $\frac{TP}{TP+FP}$ (100% precision is Soundness)

Recall = $\frac{TP}{TP+FN}$ (100% Recall is completeness)

FN ⟷ Complete
FP ⟷ Sound

Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$

Lattice :-

Total order is a binary relation $\leq$ (strict total order is $<$)
linear order set (loset) is a set equipped with a total order.

which of them are losets?

{ -1, 5, 2, 0, 42 } ← Yes
{ 3, 5, -9, 5, 12 } ← No
{ 3, 5, "Hello", 12, 5.0 } ← No
{ x, y, z } ← maybe

Partially ordered Set

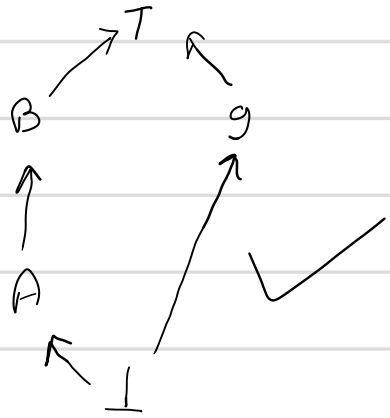
→ Partially order is total order between some elements

→ Partially ordered set (poset) is a set equipped with a partial order.

{ 1, "apple", 2, -7, "orange" } ← yes

Lattice :- It is a poset where two elements (x, y) have least upper bound (join operator $x \sqcup y$) and greatest lower bound (meet operator $x \sqcap y$)

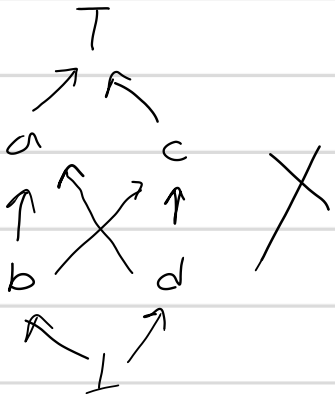
$\{A, \top, g, B, \bot\}$



$A \sqcup g = \top$

$B \sqcap g = \bot$

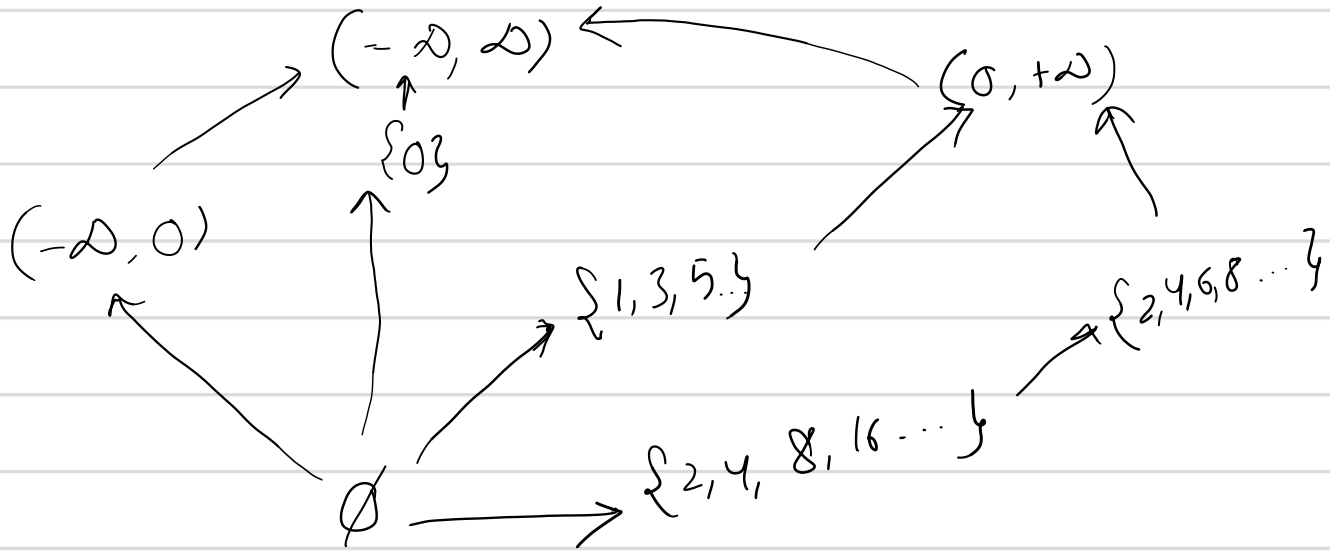
$\{\top, \bot, a, b, c, d\}$



$b \sqcup d = \{a, c\}$

$a \sqcap c = \{b, d\}$

Partial order is $\in$

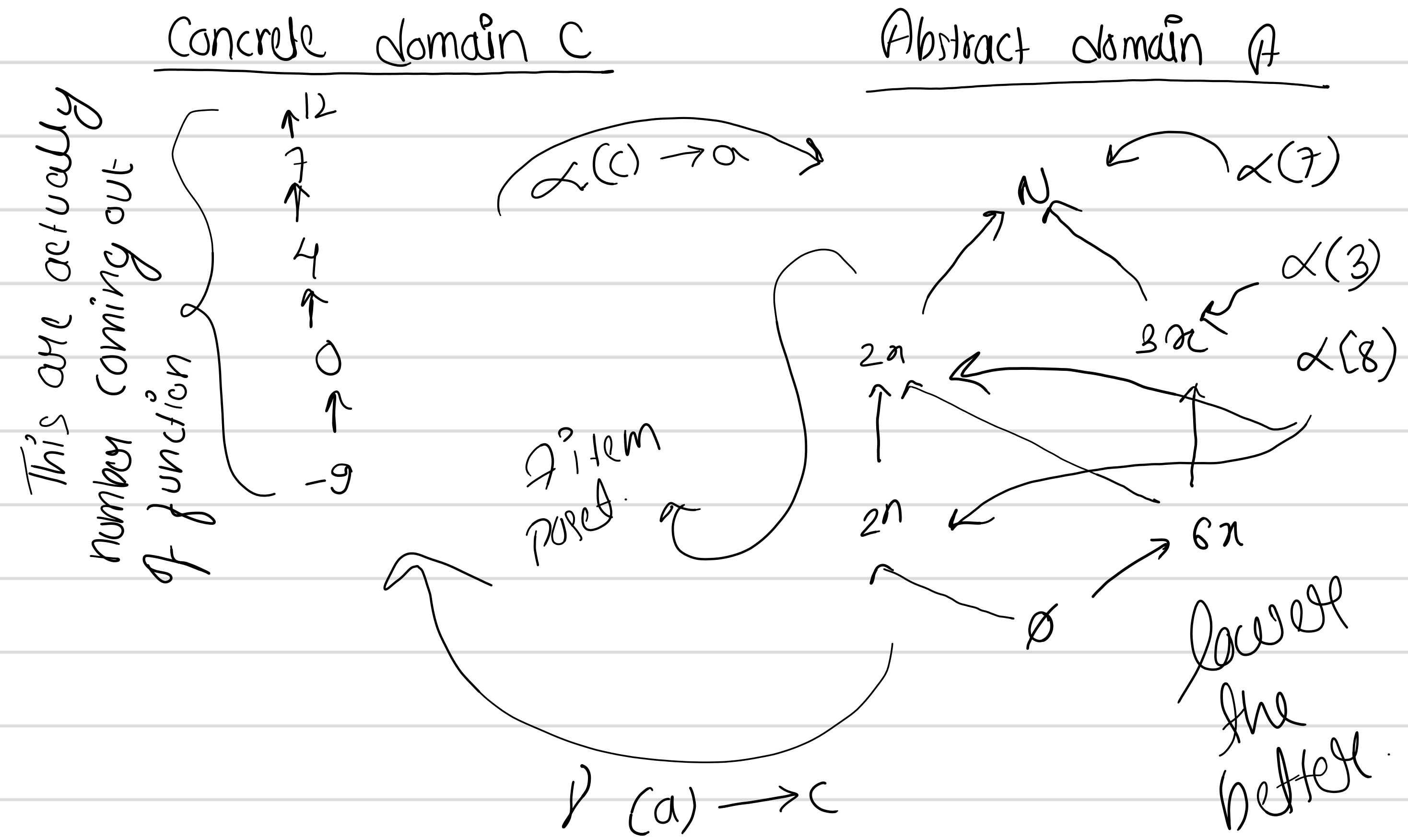


Abstract Interpretation :-

There is a compromise to be made between the precision of the analysis and it's decidability (computability) or tractability (computation cost)

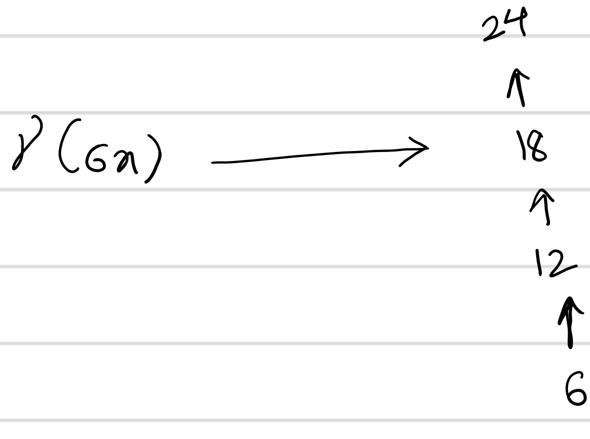
(4) Sa is Over & under approximation.
" too much — too little.

Abstraction & concretization :-



α(c) → a [Abstraction function]
γ(a) → c [Concetrization function]

So, Lower we put on the lattice, the more precise will be the analysis.



Alpha mapping & Gamma mapping.

#1. Ordered Sets (Domains)

↳ Concrete Domains (C) → Precise, detail

↳ Abstract Domains (A) → approximation

→ sets of concrete states.

2. Order Relations

Both C and A are poset.

$\leq_C$ for C.

$\leq_A$ for A.

3. Galois connection

A Galois connection between C and A is defined by two function

1. Abstraction function (α)
2. concretization function (γ)

$\alpha : C \longrightarrow A$

$\gamma : A \longrightarrow C$

This function must satisfy following condition for all $c \in C$ and $a \in A$:

$$\alpha(c) \leq_A a$$

$$\text{iff} \quad c \leq_C \gamma(a)$$

This means α and γ form a pair of adjoint function, ensuring that the abstraction & concretization are consistent w.r.t the ordering in each domain.

Rough.

$[a_1, b_1] \leq_A [a_2, b_2]$

$\Rightarrow [10, 20] \leq_A [5, 30]$

Adjoint new condition

$$\alpha(c) \leq_A a \quad \text{iff} \quad c \leq_C \gamma(a)$$

let, $c = 5$ be a concrete value.

- $\alpha(5) = [5, 5]$

let, $a = [4, 6]$ be an abstract interval.

Tovarify :-

① $\alpha(5) \leq_A [4,6]$ iff ② $5 \leq_C \gamma([4,6])$

check ① $[5,5] \leq_A [4,6]$

this is as $4 \leq 5 \leq 6$

check ② $5 \leq_C \gamma([4,6])$

$5 \leq_C \{z \in \mathbb{Z} \mid 4 \leq z \leq 6\}$

$5 \leq_C \{4,5,6\}$

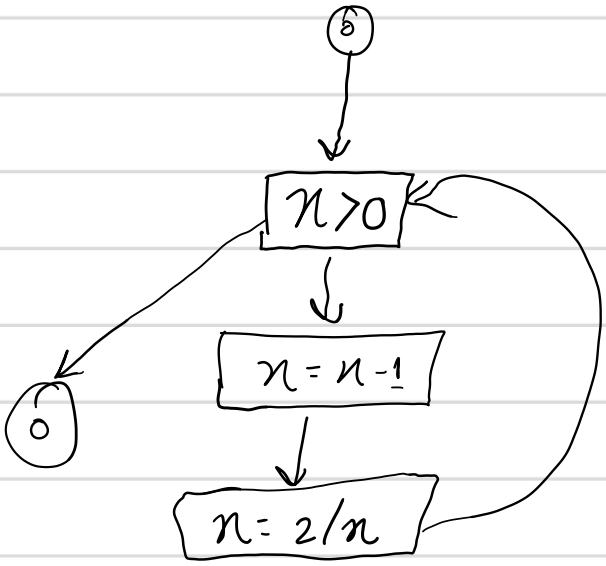
this is also true.

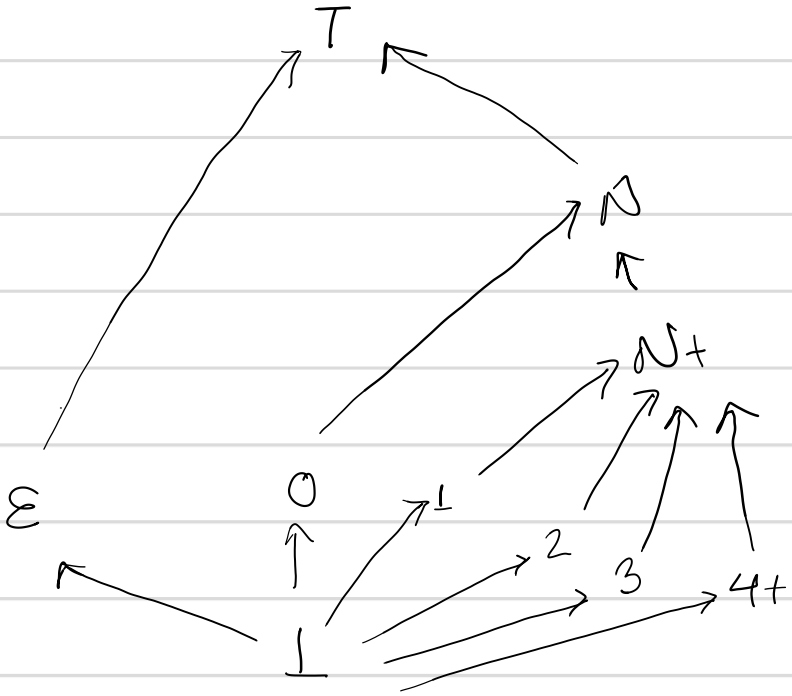
This satisfies Galois connection.

Fixed point calculation :-

FP computation is a repeated Symbolic execution of the Program using abstract semantics until approximation reaches to an equilibrium.

```
def calc(x):
    while x > 0:
        x = x - 1
        x = 2/x
    return x
```





Operational Semantics [meaning to Programs]

- Operational Semantics of imperative langages.

For Operational semantics.

consider a C code :-

```
int i=5;
fun( i++, --i);
```

Show in chatgpt.

What are the arguments passed to fun?

1. 5, 5 (left to right)
2. 4, 4 (right to left)

Both options are possible in C!

→ unspecified Semantics.

→ compiler decides.

want : all behaviour should be clearly specified.

Specifying the Semantics of programs

- Static Semantics eg. type systems.
- Dynamic Semantics
  - ↳ Operational semantics
- is useful for language design.
- is useful for language implementation
- Programming.
- Program analysis.

a) Transition System

- It describes the State of Systems
- Transition from one State to other

- Set called config is set of configurations/ states
- binary relation $(\rightarrow) \subseteq \text{config} \times \text{config}$

"Transition relation"

$a \rightarrow a'$ (transition)

↳ step of computation

↳ should be deterministic.

Notation $\rightarrow^*$ (reflexive, transitive closure of $\overset{*}{\rightarrow}$)

$\forall a.\ a \overset{*}{\rightarrow} a$

$\forall a, a', a''$

$c \overset{*}{\rightarrow} c' \wedge \overset{*}{\rightarrow} c''$

$c \overset{*}{\rightarrow} c''$

b) Rule induction

↳ define a set ("inductive set") with

① a finite set of basic elements ("axioms")

② a finite set of rules that specify how to generate more elements

$$\frac{h_1, \ldots, h_n}{c}$$

$h_1, \ldots, h_n$ ← hypothesis

c ← conclusion

$$\frac{\text{(empty)}}{t}$$

t ← conclusion

Example: Set of natural numbers.

Axiom: $\frac{}{0}$

Rule: $\frac{n}{n+1}$

Example: Arithmetic expression in a P.L.

$+(3,4)$

↳ Set = pairs of AST & values after evaluations.

Notation : $E \Downarrow n$; expression E evaluates to number n.

Axioms : $1 \Downarrow 1$, $2 \Downarrow 2$, etc

axiom scheme $n \Downarrow n$

Rules : $$\frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{+(E_1, E_2) \Downarrow n}$$ if $n = n_1 + n_2$ etc.

rule scheme :

$$\frac{E_1 \Downarrow n_1 \quad E_2 \Downarrow n_2}{OP(E_1, E_2) \Downarrow n}$$ if $n = n_1 \, OP \, n_2$

c) Proof tree

↳ Shows that an element is in an inductive set.

Ex

2 is a natural number

$$\frac{\frac{\overline{0}}{1}}{2}$$ ⇐ It is a axiom

Ex 2 Show that:

$-(+(3,4),1) \Downarrow 6$

$$\dfrac{\dfrac{3 \Downarrow 3 \quad 4 \Downarrow 4}{+(3,4) \Downarrow 7} \quad 1 \Downarrow 1}{-(+(3,4),1) \Downarrow 6}$$

3 axioms & 2 rules

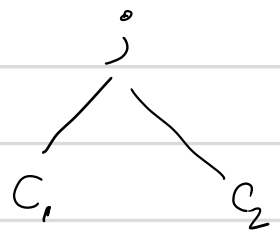
Abstract Syntax of SIMP (Simple Imperative Programming Language)

- features of SIMP
  - ↳ assignment, integer variables
  - ↳ Sequencing
  - ↳ conditionals
  - ↳ loops

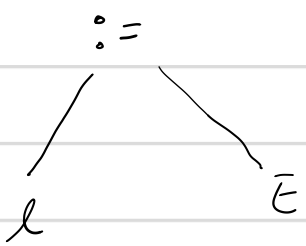
abstract Syntax

$P ::= C \mid E \mid B$

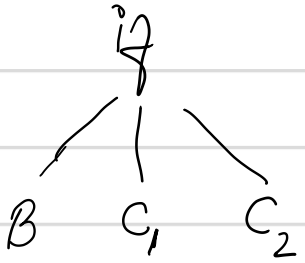
a) commands



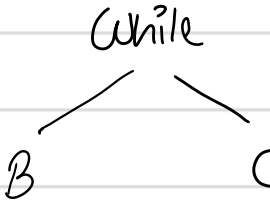
(Sequence)



(assignment of an Expression E to some label l)



(two-way selector)



(while loop)

Skip

(do nothing)

Textual notation :- C ::= C;C | l := E | if B then C_1 else C_2 | while B do C | skip

consists of

b) Integer expressions.

$E ::= !l \mid n \mid E \ op \ E$

$op ::= + \mid - \mid * \mid /$

n is integers

$l \in C = \{ l_0, l_1, l_2 \ldots \}$ memory locations

!l value store at location l.

c) Boolean Expression

$B ::= True \mid False \mid E \ bop \ E \mid \neg B \mid B \wedge B$

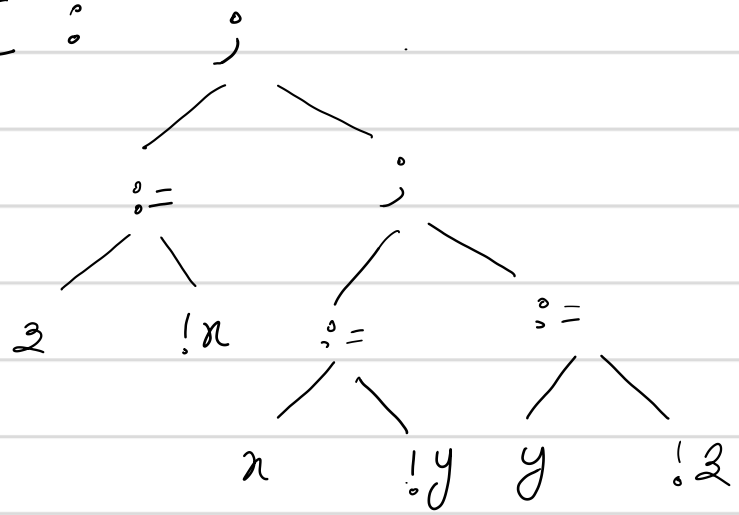
$bop ::= > \mid < \mid =$

Example :-

$z := !x ; (x := !y ; y := !z)$

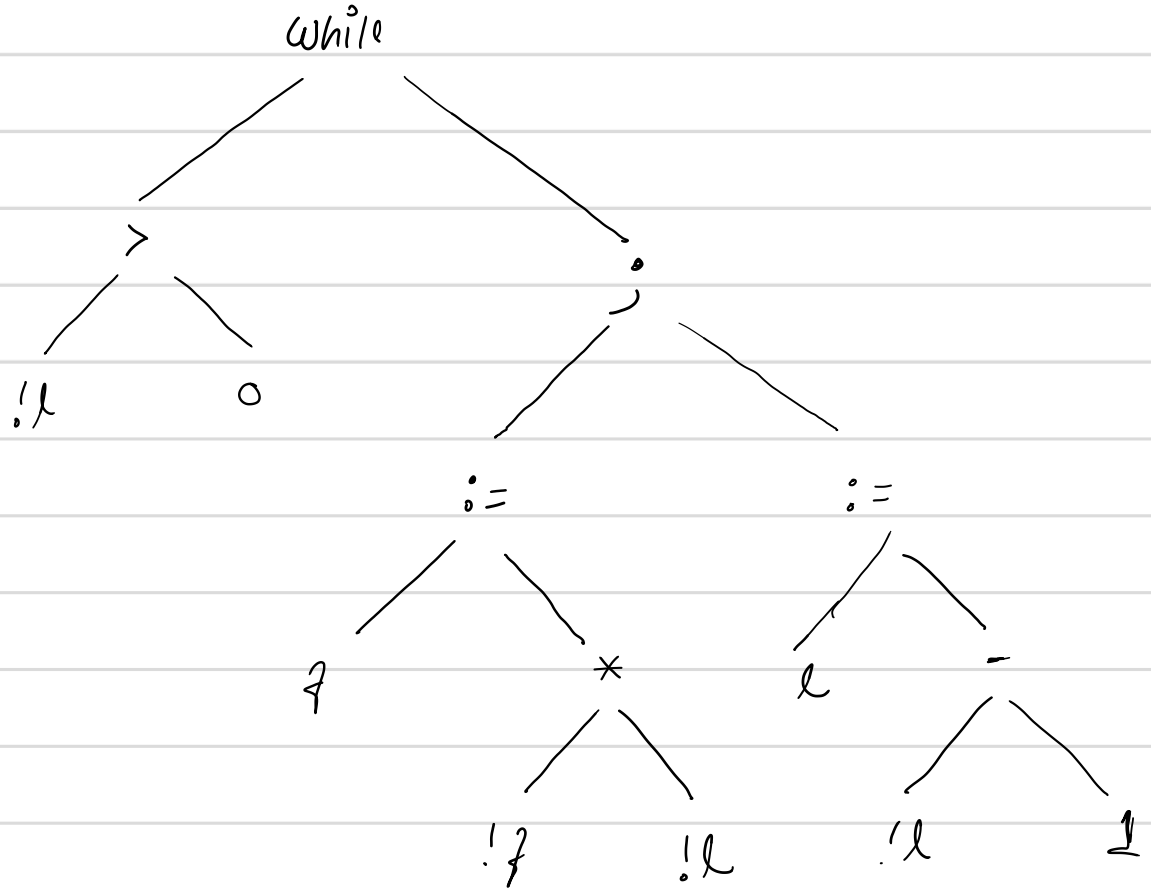
Swapping values of x and y using z.

AST :



Ex2.

```
while ( !l > 0) do (
    f := !f * !l;
    l := !l - 1 )
```



An abstract machine for SIMP

A hypothetical machine with 4 elements

(i) control stack c: store instruction to execute

(ii) auxiliary stack r: store intermediate results

(iii) processor : performs arithmetic operation, comparision, boolean operations

(iv) memory/state m: partial functions mapping location to values (integers)

Notation : $m[l \mapsto n]$ update function m with new mapping $l \mapsto n$

$$m[l \mapsto n](l) = n$$
$$m[l \mapsto n](l') = m(l') \quad \forall l' \neq l$$

Abstract machine

↳ Configuration : $\langle c, r, m \rangle$

$c ::= nil \mid i \circ c$ ← instruction i push on top of stack

↑ empty stack

$i ::= p \mid op \mid 7 \mid n \mid bop \mid := \mid if \mid while$

$r ::= nil \mid por \mid lor$

Model execution of programs as sequence of transitions from initial state to final state.

$\langle c \circ nil, nil, m \rangle$

↓

Execute program c in a given memory state m

$\langle nil, nil, m \rangle$

↓

Stop when all stacks are empty.

Transition rules :- $\rightarrow$

$$\langle c, r, m \rangle \rightarrow \langle c', r', m' \rangle$$

a) Evaluating Expressions.

↳ $\langle n \circ c, r, m \rangle \longrightarrow \langle c, n \circ r, m \rangle$

↳ popping n from c & push it onto r.

↳ $\langle !l \circ c, r, m \rangle \longrightarrow \langle c, n \circ r, m \rangle$ if $m(l) = n$

read the memory at l and push it onto r.

↳ $\langle (E_1 \, op \, E_2) \circ c, r, m \rangle \rightarrow \langle E_1 \circ E_2 \circ op \circ c, r, m \rangle$

$\langle op, n_2 \circ n_1 \circ r, m \rangle \rightarrow \langle c, n \circ r, m \rangle$ if $n_1 \, op \, n_2 = n$

why this is opposite.

Semantics of SIMP Expressions.

Value of E in state m is v if there is a seqn of transitions

$\langle E \circ c, r, m \rangle \xrightarrow{*} \langle c, v \circ r, m' \rangle$

$\Leftarrow m' = m.$ for SIMP

⑧ in case of ass $[m' \neq m]$

Example: What is the value of

$E = !a + !b$ in state

$m = \{a \mapsto 3, b \mapsto 1\}$

Soln.

$\langle !a + !b \circ nil, nil, m \rangle$

$\rightarrow \langle !a \circ !b \circ + \circ nil, nil, m \rangle$

$\rightarrow \langle !b \circ + \circ nil, 3 \circ nil, m \rangle$

$\rightarrow \langle + \circ nil, 1 \circ 3 \circ nil, m \rangle$

$\rightarrow \langle nil, 4 \circ nil, m \rangle$

↑ Ans

b) Executing Commands

(#) $\langle skip \circ c, r, m \rangle \longrightarrow \langle c, r, m \rangle$

(#) $\langle (l := E) \circ c, r, m \rangle \rightarrow \langle E \circ := \circ c, l \circ r, m \rangle$

push l on auxiliary stack.

$\langle := \circ c, n \circ l \circ r, m \rangle \rightarrow \langle c, r, m[l \mapsto n] \rangle$

write n to location l.

(#) $\langle (c_1 ; c_2) \circ c, r, m \rangle \rightarrow \langle c_1 \circ c_2 \circ c, r, m \rangle$

(#) $\langle (\text{if } B \text{ then } c_1 \text{ else } c_2) \circ c, r, m \rangle \rightarrow$

$\langle B \circ \text{if} \circ c, c_1 \circ c_2 \circ r, m \rangle$

↑ Store commands for later.

$\langle \text{if} \circ c, True \circ c_1 \circ c_2 \circ r, m \rangle$

$\rightarrow \langle c_1 \circ c, s, m \rangle$

$\langle$ if $\circ$ c, false $\circ c_1, \circ c_2 \circ r$, $m \rangle$
$\rightarrow \langle c_2 \circ c, r, m \rangle$

(#) $\langle$ (while B do c) $\circ c, r, m \rangle$
$\rightarrow \langle B \circ$ while $\circ C, B \circ C \circ r, m \rangle$

$\langle$ while $\circ c$, True $\circ B \circ C \circ r, m \rangle$
$\rightarrow \langle C \circ$ (while B do c) $\circ c, r, m \rangle$

$\langle$ while $\circ c$, false $\circ B \circ C \circ r, m \rangle$
$\rightarrow \langle c, r, m \rangle$

Semantics of SIMP commands

Program c execute state m terminates successfully & produces state m' if there is a sequence of transitions.

$$\langle c \text{ nil}, \text{nil}, m \rangle \xrightarrow{*} \langle \text{nil}, \text{nil}, m' \rangle$$

Example: C = while $\underbrace{!l > 0}_{B}$ do $\underbrace{\underbrace{l := !l - 1}_{C_2}; \underbrace{f := !f * !l}_{C_1}}_{C'}$;

$m = \{l \mapsto 4, f \mapsto 1\}$

Soln.

$\checkmark \langle C \circ nil, nil, m \rangle$ $\checkmark$

$\checkmark \rightarrow \langle B \circ while \circ nil, B \circ C' \circ nil, m \rangle$

$\rightarrow \langle !l \circ 0 \circ > \circ while \circ nil, B \circ C' \circ nil, m \rangle$

$\rightarrow \langle 4 \circ 0 \circ > \circ while \circ nil, 4 > 0 \circ B \circ C' \circ nil, m \rangle$

$\rightarrow \langle true \circ while \circ nil, true \circ B \circ C' \circ nil, m \rangle$

$\rightarrow \langle C' \circ C \circ nil, nil, m \rangle$

$\rightarrow \langle C_1 \circ C_2 \circ C \circ nil, nil, m \rangle$

$\rightarrow \langle !f * !l \circ := \circ C_2 \circ C \circ nil, f \circ nil, m \rangle$

$\rightarrow \langle !f \circ !l \circ * \circ := \circ C_2 \circ C \circ nil, f \circ nil, m \rangle$

$\rightarrow \langle !l \circ 1 \circ * \circ := \circ C_2 \circ C \circ nil, 1 \circ f \circ nil, m \rangle$

$\rightarrow \langle * \circ := \circ C_2 \circ C \circ nil, 4 \circ 1 \circ f \circ nil, m \rangle$

$\rightarrow \langle := \circ C_2 \circ C \circ nil, 4 \circ f \circ nil, m \rangle$

$\rightarrow \langle C_2 \circ C \circ nil, nil, m[f \mapsto 4] \rangle$

$\rightarrow \langle !l - 1 \circ := \circ C \circ nil, l \circ nil, m[f \mapsto 4] \rangle$

$\rightarrow \langle !l \circ 1 \circ - \circ := \circ C \circ nil, l \circ nil, m[f \mapsto 4] \rangle$

$\rightarrow \langle 1 \circ - \circ := \circ C \circ nil, 4 \circ l \circ nil, m[f \mapsto 4] \rangle$

$\rightarrow \langle - \circ := \circ C \circ nil, l \circ 4 \circ l \circ nil, m[f \mapsto 4] \rangle$

$\rightarrow \langle := \circ C \circ nil, 3 \circ l \circ nil, m[f \mapsto 4] \rangle$

$\rightarrow \langle C \circ nil, nil, m[f \mapsto 4, l \mapsto 3] \rangle$

$\vdots$

$\rightarrow \langle nil, nil, m[f \mapsto 24, l \mapsto 0] \rangle$

1. Evaluation of Expressions:

$$
\begin{array}{rcll}
\langle n \cdot c, r, m\rangle & \to & \langle c, n \cdot r, m\rangle & \\
\langle b \cdot c, r, m\rangle & \to & \langle c, b \cdot r, m\rangle & \\
 & & & \\
\langle \neg B \cdot c, r, m\rangle & \to & \langle B \cdot \neg \cdot c, r, m\rangle & \\
\langle (B_1 \ \wedge \ B_2) \cdot c, r, m\rangle & \to & \langle B_1 \cdot B_2 \cdot \wedge \cdot c, r, m\rangle & \\
\langle \neg \cdot c, b \cdot r, m\rangle & \to & \langle c, b' \cdot r, m\rangle & \text{if } b' = not\ b \\
\langle \wedge \cdot c, b_2 \cdot b_1 \cdot r, m\rangle & \to & \langle c, b \cdot r, m\rangle & \text{if } b_1\ and\ b_2 = b \\
 & & & \\
\langle (E_1\ op\ E_2) \cdot c, r, m\rangle & \to & \langle E_1 \cdot E_2 \cdot op \cdot c, r, m\rangle & \\
\langle (E_1\ bop\ E_2) \cdot c, r, m\rangle & \to & \langle E_1 \cdot E_2 \cdot bop \cdot c, r, m\rangle & \\
\langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \to & \langle c, n \cdot r, m\rangle & \text{if } n_1\ op\ n_2 = n \\
\langle bop \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \to & \langle c, b \cdot r, m\rangle & \text{if } n_1\ bop\ n_2 = b \\
 & & & \\
\langle !l \cdot c, r, m\rangle & \to & \langle c, n \cdot r, m\rangle & \text{if } m(l) = n
\end{array}
$$

2. Evaluation of Commands:

$$
\begin{array}{rcl}
\langle skip \cdot c, r, m\rangle & \to & \langle c, r, m\rangle \\
 & & \\
\langle (l := E) \cdot c, r, m\rangle & \to & \langle E \cdot := \cdot c, l \cdot r, m\rangle \\
\langle := \cdot c, n \cdot l \cdot r, m\rangle & \to & \langle c, r, m[l \mapsto n]\rangle \\
 & & \\
\langle (C_1; C_2) \cdot c, r, m\rangle & \to & \langle C_1 \cdot C_2 \cdot c, r, m\rangle \\
 & & \\
\langle (if\ B\ then\ C_1\ else\ C_2) \cdot c, r, m\rangle & \to & \langle B \cdot if \cdot c, C_1 \cdot C_2 \cdot r, m\rangle \\
\langle if \cdot c, True \cdot C_1 \cdot C_2 \cdot r, m\rangle & \to & \langle C_1 \cdot c, r, m\rangle \\
\langle if \cdot c, False \cdot C_1 \cdot C_2 \cdot r, m\rangle & \to & \langle C_2 \cdot c, r, m\rangle \\
 & & \\
\langle (while\ B\ do\ C) \cdot c, r, m\rangle & \to & \langle B \cdot while \cdot c, B \cdot C \cdot r, m\rangle \\
\langle while \cdot c, True \cdot B \cdot C \cdot r, m\rangle & \to & \langle C \cdot (while\ B\ do\ C) \cdot c, r, m\rangle \\
\langle while \cdot c, False \cdot B \cdot C \cdot r, m\rangle & \to & \langle c, r, m\rangle
\end{array}
$$

. 4.1 Transition rules of the abstract machine for SIMP

Structural O.S. → Small-Step Semantics.
→ Big-Step Semantics..

Small - Step Semantics / Reduction Semantics

- Transition system with
  - configuration $\langle P, S \rangle$ — P is program, S is store — (function from locations to integers)
- Transition relation between configurations. It is defined via axioms and rules.

Axioms & rules for expressions

Axiom

$$\langle !l, S \rangle \longrightarrow \langle n, S \rangle \text{ if } S(l) = n \qquad \text{(var)}$$

$$\langle n_1 \text{ op } n_2, S \rangle \longrightarrow \langle n, S \rangle \text{ if } n = (n_1 \text{ op } n_2) \qquad \text{(op)}$$

$$\frac{\langle E_1, S \rangle \longrightarrow \langle E_1', S' \rangle}{\langle E_1 \text{ op } E_2, S \rangle \longrightarrow \langle E_1' \text{ op } E_2, S' \rangle} \qquad \text{(op}_L\text{)}$$

$$\frac{\langle E_2, S \rangle \longrightarrow \langle E_2', S' \rangle}{\langle n_1 \text{ op } E_2, S \rangle \longrightarrow \langle n_1 \text{ op } E_2', S' \rangle} \qquad \text{(op}_R\text{)}$$

Axioms for commands

(#) $\dfrac{}{\langle l := n, s\rangle \rightarrow \langle skip, s[l \mapsto n]\rangle}$ (:=)

(#) $\dfrac{\langle E, s\rangle \rightarrow \langle E', s'\rangle}{\langle l := E, s\rangle \rightarrow \langle l := E', s'\rangle}$ $(:=)_R$

(#) $\dfrac{}{\langle skip ; C, s\rangle \rightarrow \langle C, s\rangle}$ (skip)

(#) $\dfrac{\langle C_1, s\rangle \rightarrow \langle C_1', s'\rangle}{\langle C_1 ; C_2, s\rangle \rightarrow \langle C_1' ; C_2, s'\rangle}$ (Sequence)

(#) $\dfrac{}{\langle \text{if True then } C_1 \text{ else } C_2, s\rangle \rightarrow \langle C_1, s\rangle}$ (if_T)

(#) $\dfrac{}{\langle \text{if False then } C_1 \text{ else } C_2, s\rangle \rightarrow \langle C_2, s\rangle}$ (if_F)

(#) $\dfrac{\langle B, s\rangle \rightarrow \langle B', s'\rangle}{\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s\rangle \rightarrow \langle \text{if } B' \text{ then } C_1 \text{ else } C_2, s'\rangle}$ (if rule)

(#) $\dfrac{}{\langle \text{while } B \text{ do } C, s\rangle \rightarrow \langle \text{if } B \text{ then } (C ; \text{while } B \text{ do } C) \text{ else } skip, s\rangle}$ (while)

Ex:-

P: $z := !x ; (x := !y ; y := !z)$

$S = \{ z \mapsto 0, x \mapsto 1, y \mapsto 2 \}$

Excerpt of proof tree :-

$\langle z := !x ; (x := !y ; y := !z), S \rangle$

$\rightarrow \langle z := 1 ; (x := !y ; y := !z), S \rangle$

$\rightarrow \langle skip ; (x := !y ; y := !z), S[z \mapsto 1] \rangle$

$\rightarrow \langle x := !y ; y := !z, S[z \mapsto 1] \rangle$ ✓

$\rightarrow \langle skip ; y := !z, S[z \mapsto 1, x \mapsto 2] \rangle$

$\rightarrow \langle y := !z, S[z \mapsto 1, x \mapsto 2] \rangle$

$\rightarrow \langle skip, S[z \mapsto 1, x \mapsto 2, y \mapsto 1] \rangle$

Big-Step Semantics

$\langle P, S \rangle \Downarrow \langle P', S' \rangle$ } evaluation relations

evaluates to

$$\frac{}{\langle !l, S \rangle \Downarrow \langle n, S \rangle} \; \text{if } S(l) = n \quad \text{(var)}$$

axioms

$$\frac{\langle E, S\rangle \Downarrow \langle n, S'\rangle}{\langle l :=^{o} E, S\rangle \Downarrow \langle skip, S'[l \to n]\rangle} :=$$

Ex:

$$(z :=^{o} !x ; x :=^{o} !y) ; y :=^{o} !z$$

$S(z) = 0, \quad S(x) = 1, \quad S(y) = 2.$

Prove :- $\langle P, S\rangle \Downarrow \langle skip, S'\rangle$ using ~~Big~~ Evaluation Relation.

$$\frac{\overline{\langle !x, S\rangle \Downarrow \langle 1, S\rangle}\ (var)}{\langle z :=^{o} !x, S\rangle \Downarrow \langle skip, S[z \mapsto 1]\rangle}\ (:=^{o})$$

again,

$$\frac{\overline{\langle !y, S[y \mapsto 1]\rangle \Downarrow \langle 2, [x \to 1]\rangle}\ (var)}{\langle x :=^{o} !y, S[z \mapsto 1]\rangle \Downarrow \langle skip, S[z \mapsto 1, x \mapsto 2]\rangle}\ (:=)$$

and so on book page no 83.

Exercise 1: Operational Semantics :-

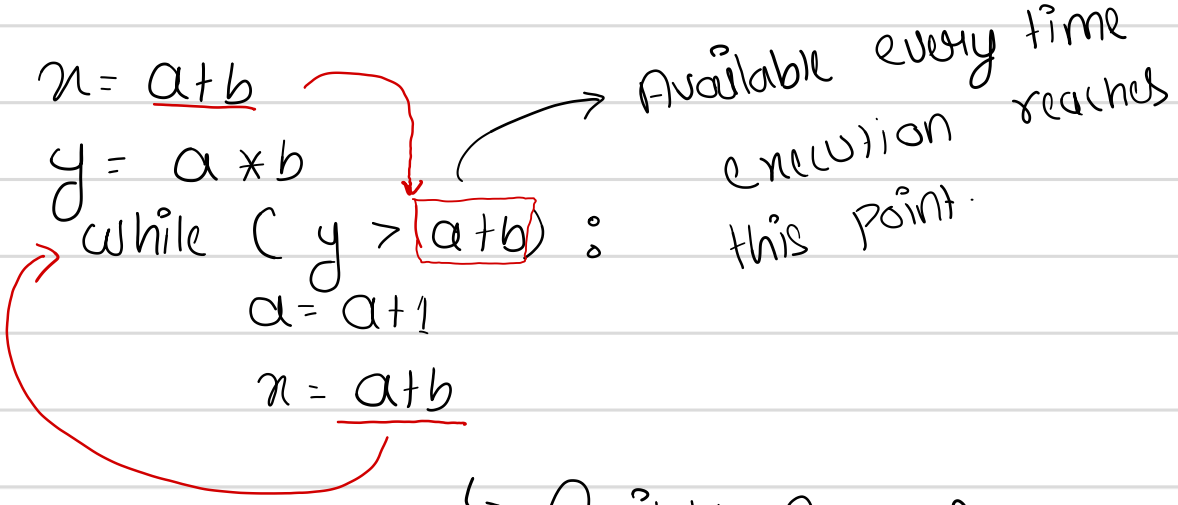
Data flow Analysis ⇒ Static Analysis हो।

Available Expression analysis

↳ For each program point (code, statement), compute which expressions must have already been computed, & not later modified.

↳ avoids re-computing

↳ compiler Optimizations.



↳ Available Expressions

Transfer Functions

↳ How the statement affects the analysis state.

↳ available expressions.

Two functions

- gen :- available expressions generated.
- Kill :- " " killed.

① gen function :-

gen : Stmt ⟶ P(Expression)

Ex :-

x = a*b generates a*b

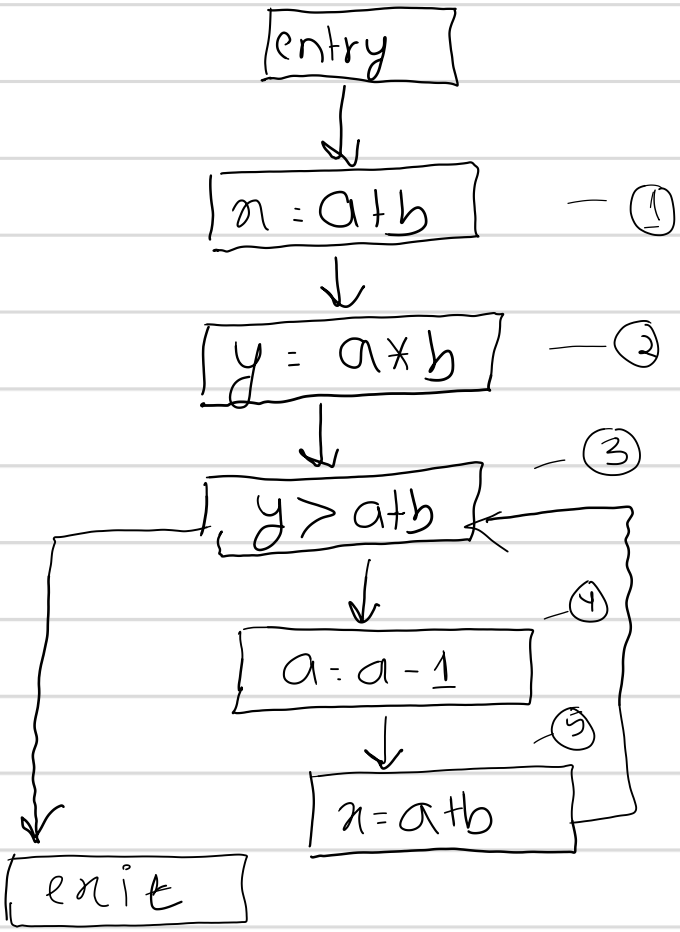
② Kill function

Kill : Stmt ⟶ P(Expression)

Example a = 23 Kill a*b

```
x = a+b
y = a*b
while y > a+b:
    a = a+1
    x = a+b
```

CFG



Non - trivial Expression

a+b
a*b
a - 1

Transfer function for each Stmt:

Stmt	gen(S)	kill(S)
1	{a+b}	∅
2	{a*b}	∅
3	{a+b}	∅
4	∅	{a+b, a*b, a-1}
5	a+b	∅

Propagating Available Expression

↳ Forward analysis.

↳ for each statment S, outgoing available expressions are :-

incoming avail. expression minus (kill(S) plus gen(S).

Data flow equations :-

$AE_{entry}(S)$ → available expression at entry of S.

$AE_{exit}(S)$ → " " " exit of S.

$AE_{entry}(1) = \emptyset$

$AE_{entry}(2) = AE_{exit}(1)$

$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$

$AE_{entry}(4) = AE_{exit}(3)$

$AE_{entry}(5) = AE_{exit}(4)$

$AE_{exit}(1) = AE_{entry}(1) \cup \{a+b\}$
$AE_{exit}(2) = AE_{entry}(2) \cup \{a*b\}$
$AE_{exit}(3) = AE_{entry}(3) \cup \{a+b\}$
$AE_{exit}(4) = AE_{entry}(4) \setminus \{a+b, a*b, a-1\}$
$AE_{exit}(5) = AE_{entry}(5) \cup \{a+b\}$

S	$AE_{entry}(S)$	$AE_{exit}(S)$
1	$\emptyset$	$\{a+b\}$
2	$\{a+b\}$	$\{a+b, a*b\}$
3	$\{a+b\}$	$\{a+b\}$
4	$\{a+b\}$	$\emptyset$
5	$\emptyset$	$\{a+b\}$

Q.

```
m = n-y
if (random()):
    while m>0:
        n = y+1
else:
    n = x-y
z = x-y
```

is x-y an available expression when exiting

this statement ??

Basic principles :-

1. Domain
2. Direction → forward, backward
3. Transfer function
4. Meet operator
5. Boundary condition
6. Initial values.

→ All possible elements the set may have
Available expr → Non trivial & expression.

→ how a statement affects the propagated information

$DF_{exit}(s)$ = Some function of $DF_{entry}(s)$.

4. Meet operator :-

↳ two statement S_1 and S_2 flow to a statement S.

union : $DF_{entry}(s) = DF_{exit}(s) \cup DF_{exit}(S_2)$

Intersect : $DF_{entry}(s) = DF_{exit}(S_1) \cap DF_{exit}(S_2)$

Boundary Condition :-

Information to start with at the first CFG node.

Initial values :- information to start with at Intermediate nodes.

$AE_{exit}(s) = (AE_{entry} \setminus kill(s)) \cup gen(s)$

Defining a Data Flow Analysis

Any data flow analysis:
Defined by six properties

Domain	Non-trivial expressions
Direction	Forward
Transfer function	$AE_{exit}(s) = (AE_{entry} \setminus kill(s)) \cup gen(s)$
Meet operator	Intersection ($\cap$)
Boundary condition	$AE_{entry}(entryNode) = \emptyset$
Initial values	$\emptyset$

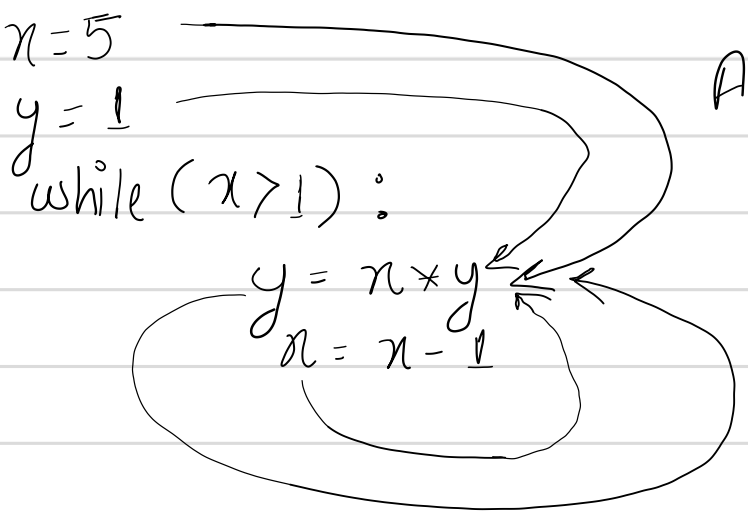
Example: Available expressions 24 - 2

Examples :-

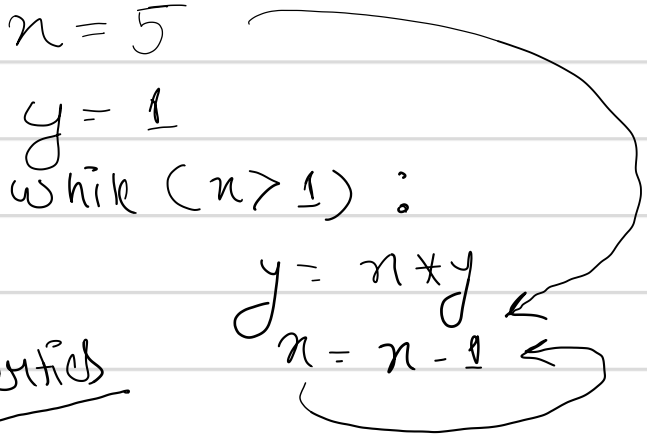
1. Reaching Definitions
2. Very busy Expressions
3. Live variables.

(1)

Goal :- For each program point, compute which assignments may have been made and may not have been overwritten



All definitions reach the entry of this statement.



6 properties

1) Domain :-

Set of pairs (V, S)

variable Stmt

3) Direction :- forward.

3) Meet operator :- Union

4) Transfer function :-

$$RD_{exit} = (RD_{entry}(s) \setminus kill(s)) \cup gen(s)$$

a) gen(s)

= if S is assignment to V : (V, S)

b) kill(s)

= if S is assignment to V : (V, S') for all S' that define V.

5) Boundary condition.

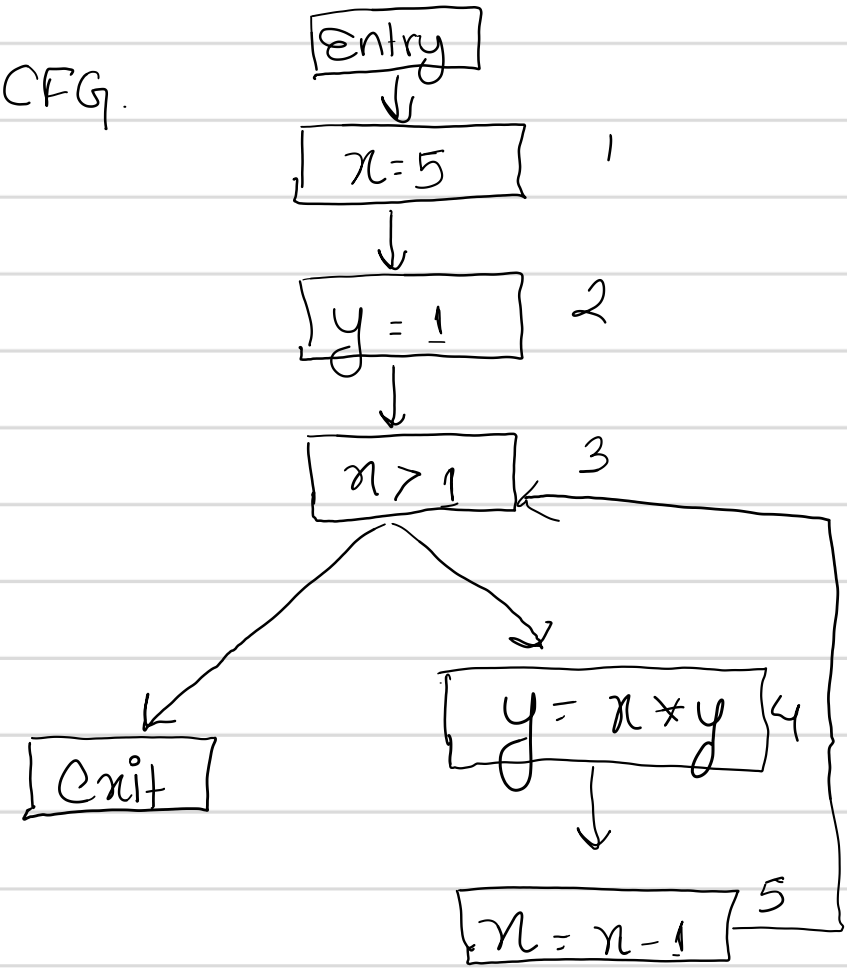
↳ Entry node starts with all variables undefined.

$$RD_{entry}(entryNode) = \{(V, ?) \mid V \in Vars\}$$

6) Initially, all nodes have no reaching definitions

```
x = 5
y = 1
while  x > 1 :
    y = x * y
    x = x - 1
```

CFG.



S	gen(S)	Kill(S)
1.	{(x, 1)}	{(x, 1), (x, 5), (x, ?)}
2.	{(y, 2)}	{(y, 2), (y, 4), (y, ?)}
3.	∅	∅
4.	{(y, 4)}	{(y, 2), (y, 4), (y, ?)}
5	{(x, 5)}	{(x, 1), (x, 5), (x, ?)}

Data flow equations

$RDentry(1) = \{(x, ?), (y, ?)\}$

$RDentry(2) = RDexit(1)$

$RDentry(3) = RDexit(2) \cup RDexit(5)$

$RDentry(4) = RDexit(3)$

$RDentry(5) = RDexit(4)$

$$RDexit(1) = (RDentry(1) \setminus \{(x,1), (x,5), (x,?)\}) \cup \{x,1\}$$

$$RDexit(2) = (RDentry(2) \setminus \{(y,2), (y,4), (y,?)\}) \cup \{(y,2)\}$$

$$RDexit(3) = RDentry(3)$$

$$RDexit(4) = (RDentry(4) \setminus \{(y,2), (y,4), (y,?)\}) \cup \{(y,4)\}$$

$$RDexit(5) = (RDentry(5) \setminus \{(x,1), (x,5), (x,?)\} \cup \{(x,5)\}$$

Solution :-

S	RDentry (S)	RDexit (S)
1	$\{(x,?), (y,?)\}$	$\{(x,1), (y,?)\}$
2	$\{(x,1), (y,?)\}$	$\{(x,1), (y,2)\}$
3	$\{(x,1), (y,2), (y,4), (x,5)\}$	$\{(x,1), (y,2), (y,4), (x,5)\}$
4	〃	$\{(x,1), (y,4), (x,5)\}$
5	$\{(x,1), (y,4), (x,5)\}$	$\{(y,4), (x,5)\}$.

Q. Very Busy Expression Analysis - S.K

Q. Live Variable 〃 - AK

Intra - Procedural Analysis.

↳ Reasons about a function in isolation.

Inter - Procedural Analysis:

↳ Reasons about multiple functions.

↳ Calls and returns.

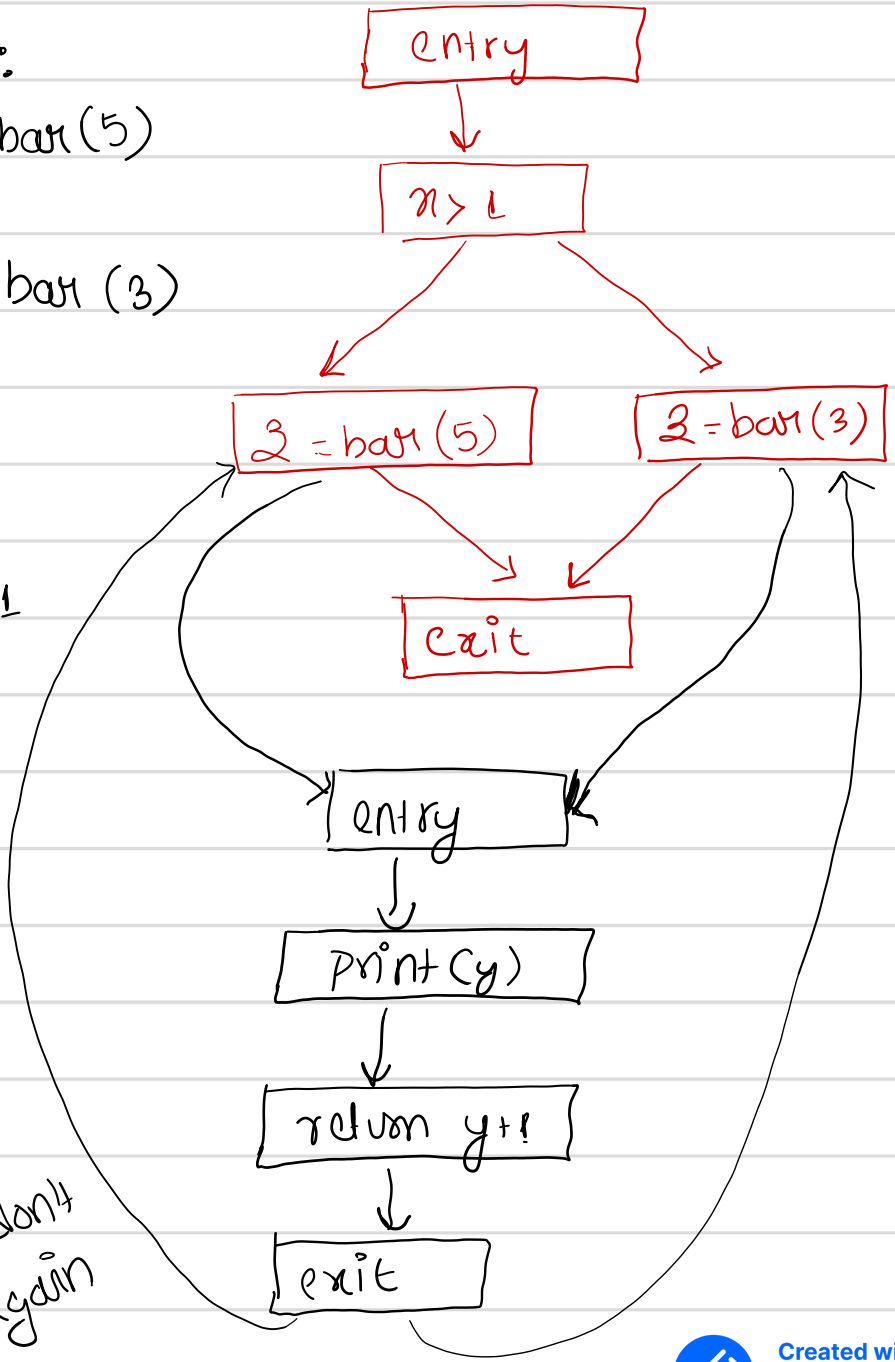
↳ one CFG per function.

↳ connect call sites to entry node of callee

↳ connect exit node back to call site.

Example:-

```
def fun(x):
    if x > 1:
        z = bar(5)
    else:
        z = bar(3)

def bar(y):
    print(y)
    return y+1
```



Analysis considers only possible inter-procedural flows:

After return, don't enter again

when, returning, go back to call site.

Sensitivites :-

① Flow - Sensitive :- Takes into account the order of Statements.

② Path - sensitive :- Takes into account the predicates at conditional branches.

③ Context - Sensitive : Takes into account the specific call site that leads into another account.
(inter - procedural analysis only).

Example :-

```
if .... :
    x = 3
    x = 5
```

value of x ?

Flow Sensitive = 5

Flow - insensitive = 3 or 5

Path Sensitivite

```
x = 0
if a > 0 :
    x = 1 ←
else
    x = 2
if  a > 0 :
    x = x + 3 ←
```

Can x have value 5:

Path Sensitive : NO (4)

Path-insensitive :- yes.

Context Sensitivity :-

```
n = 1
def g(x) :
    if x :
        g(3)
    else :
        n = 3
        g(5)

def g(y) :
```

→ Context-insensitive :- yes. (conflates all call sites of g)

--> Context-Sensitive :- NO.

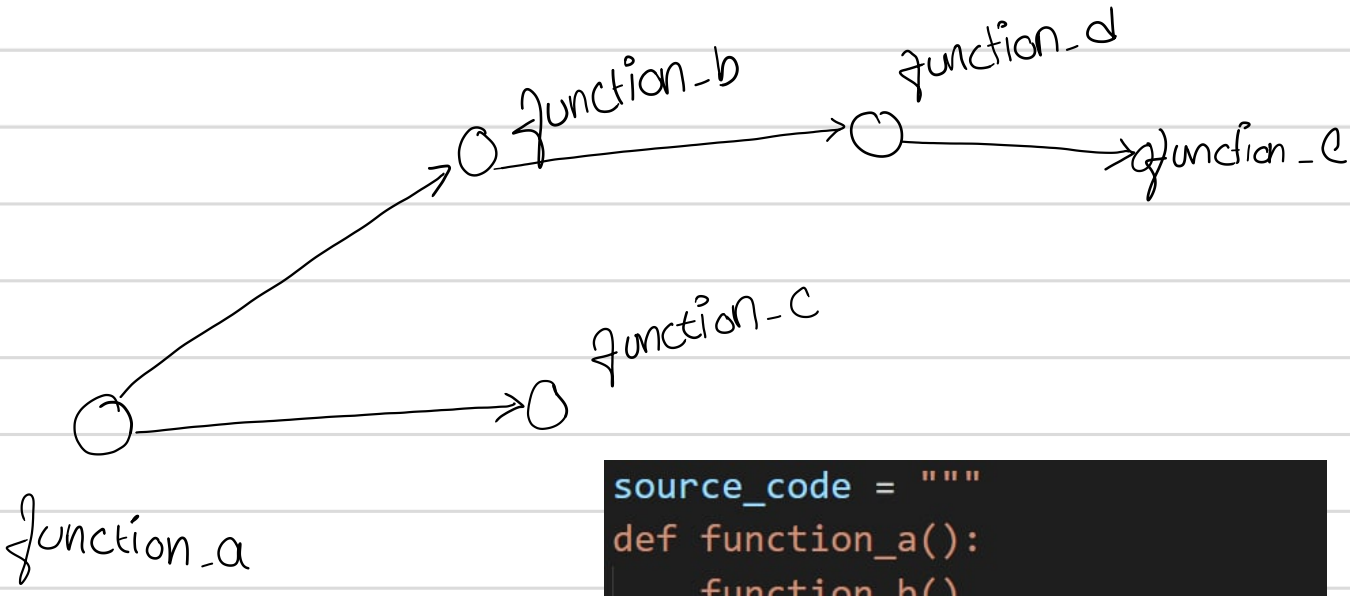
← can n be equal to y?

Call graph Analysis:-

↳ call graph :- Abstraction of all method calls in a program.

- Nodes : Methods
- Edges : calls
- Flow - insensitive.

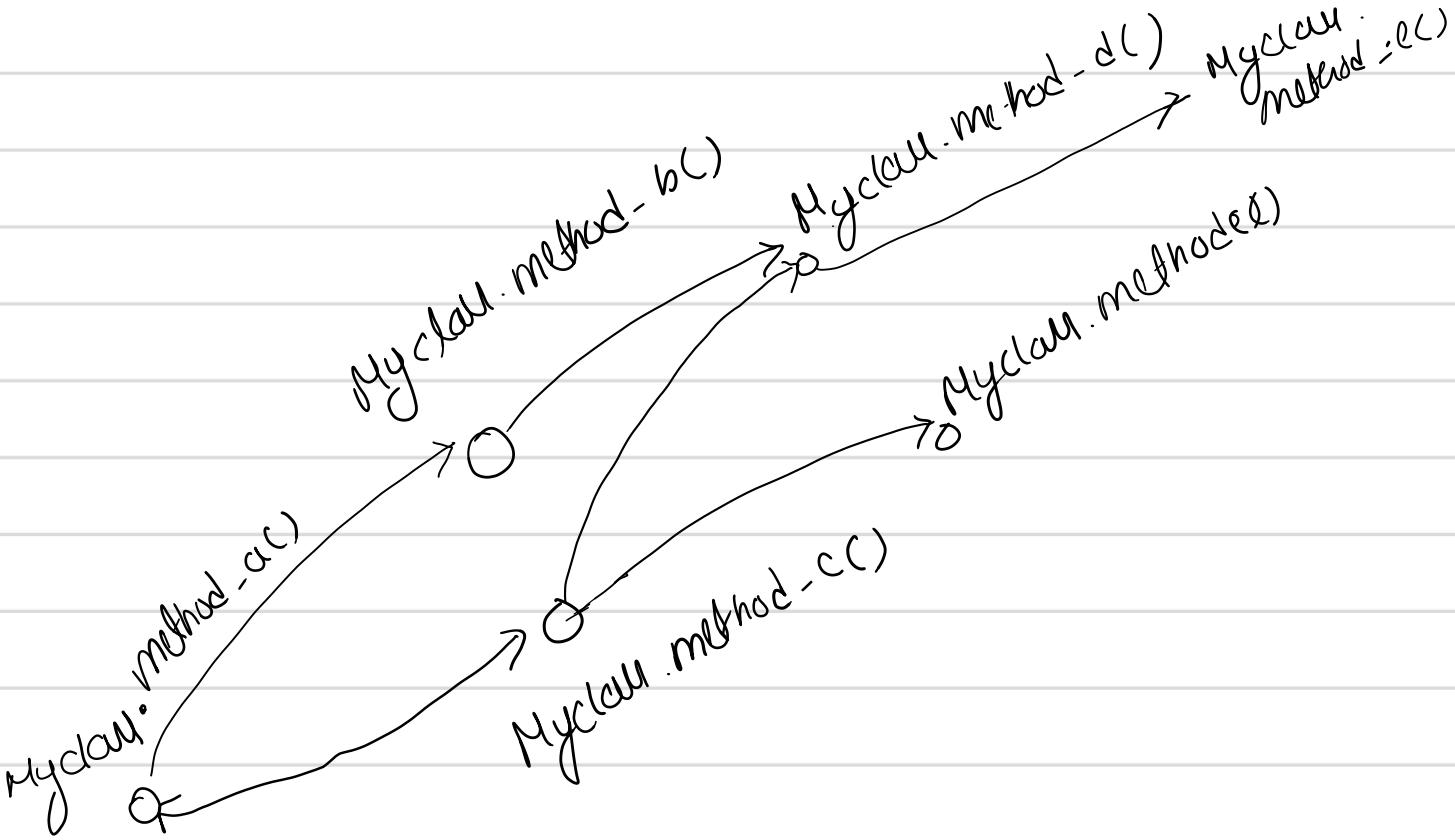
(1)



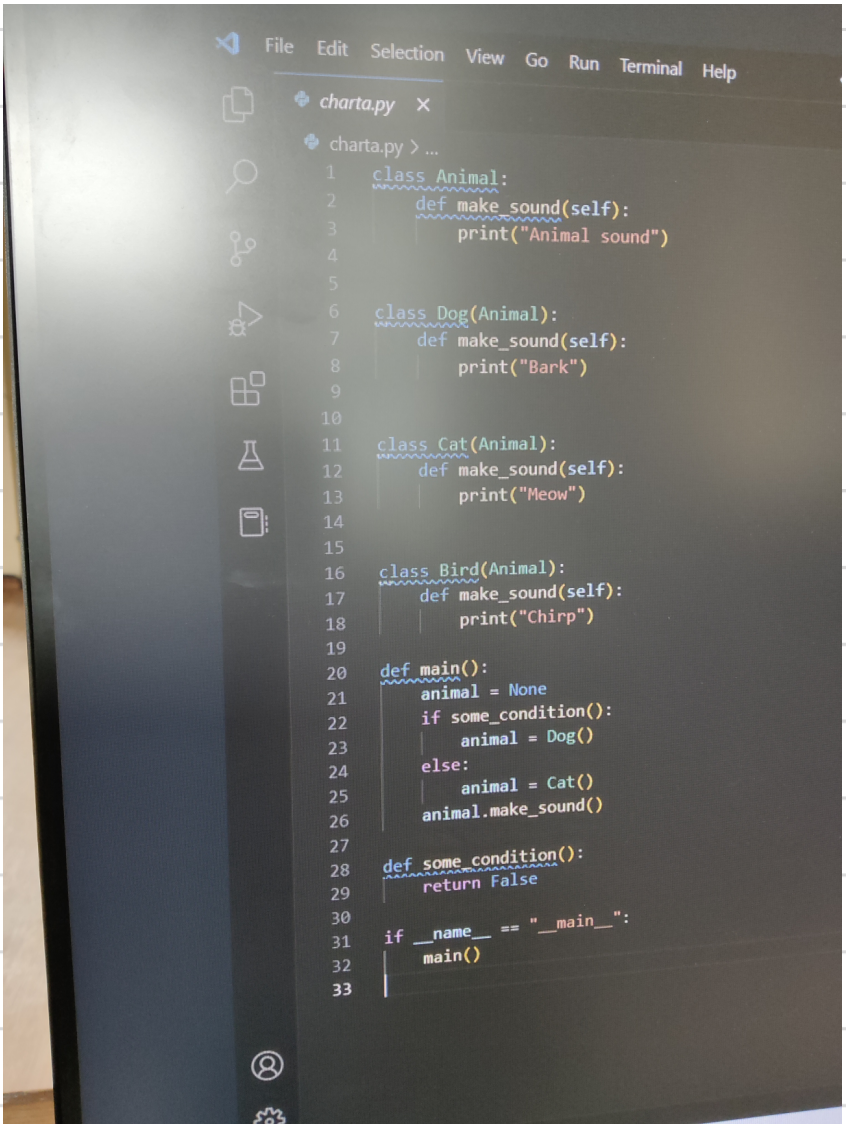

```
source_code = """
def function_a():
    function_b()
    function_c()

def function_b():
    function_d()

def function_c():
    pass

def function_d():
    function_e()

def function_e():
    pass

if __name__ == "__main__":
    function_a()
"""
```

(2)

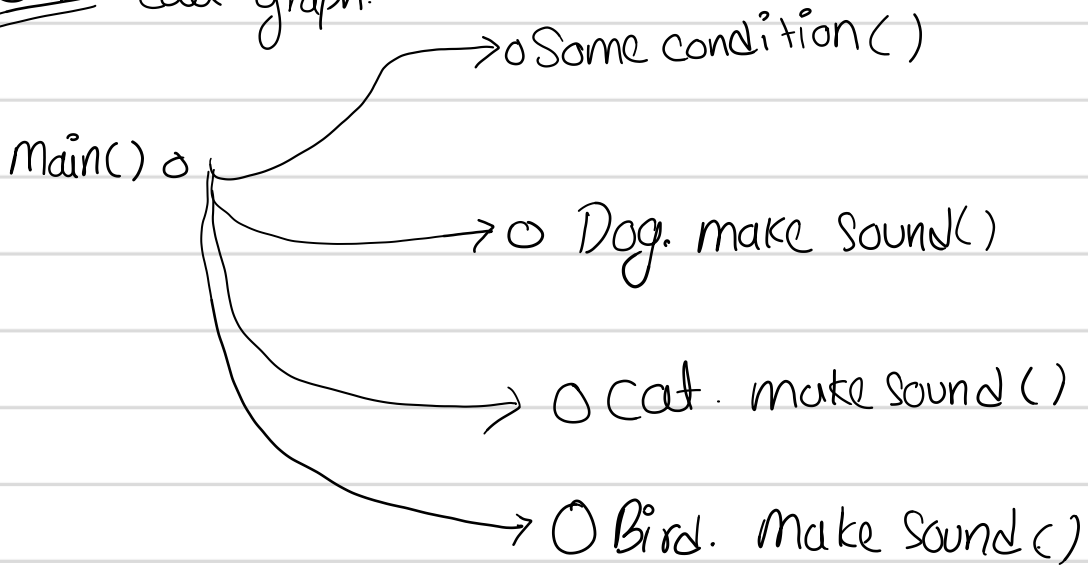
```
class MyClass:
    def method_a(self):
        self.method_b()
        self.method_c()

    def method_b(self):
        self.method_d()

    def method_c(self):
        self.method_e()
        self.method_d()
        MyClass.method_a()

    def method_d(self):
        self.method_e()

    def method_e(self):
        pass

if __name__ == "__main__":
    obj = MyClass()
    obj.method_a()
"""
```



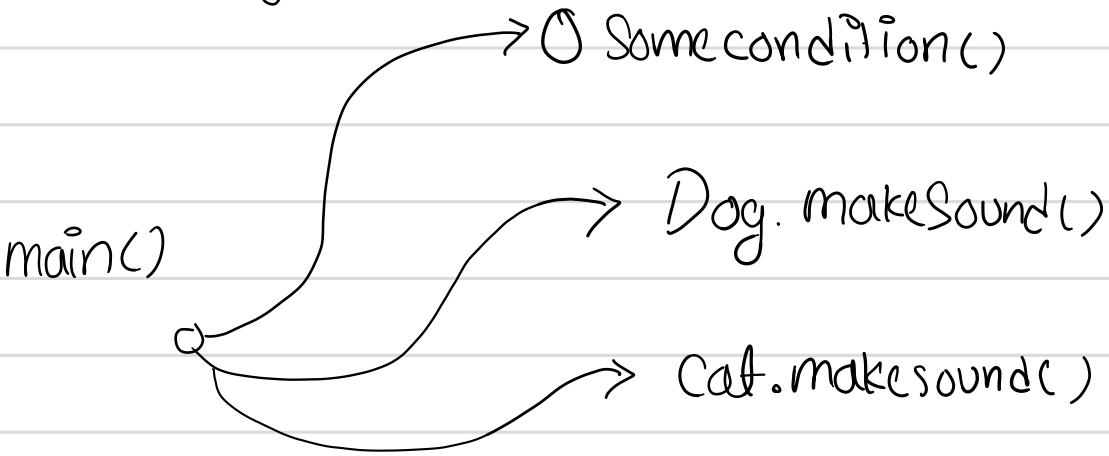
Class hierarchy Anysis (CHA)



CHA call graph.



RTA call graph



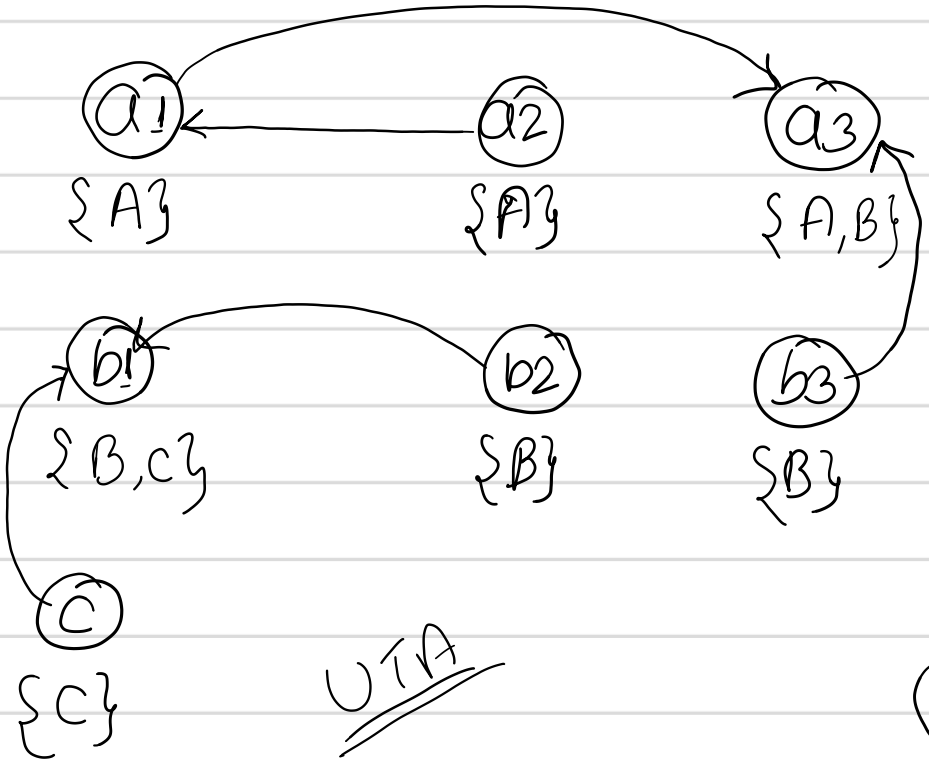
VTA and DTA. ⇒ Declared - type analysis.

Variable Type Analysis

a1 → A()
a2 → A()
a3 → A()
b1 → B()
b2 → B()
b3 → B()

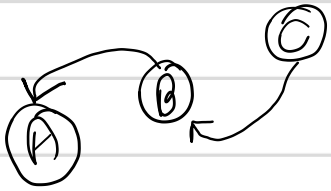
C → C()

a1 = a2
a3 = a1
a3 = b3
b1 = b2
= C



more precise.

UTA



DTA

less precise.

```
class A:
    pass

class B:
    pass

class C:
    pass

a1 = A()
a2 = A()
a3 = A()
b1 = B()
b2 = B()
b3 = B()
c = C()

a1 = a2
a3 = a1
a3 = b3
b1 = b2
b1 = c
```