Compiler

Cross compiler :

source — target (low lang)

implementation (low lang) ↗ generally same

if not then cross compiler.

→ **Semantic Analysis** (Type setting)

(int) i + (int) x — allowed : arithmetic addⁿ

(ptr) ptr + (int) x — allowed : pointer shift

(ptr) ptr + (ptr) x — not allowed.

→ **Content Sensitive Lang**

CF : $A \rightarrow Bcd$

CS : $\alpha A \rightarrow Bcd$, $\beta A \rightarrow Cde$ } α, β are contents so def. on context we can do reductions

α | ~~Bcd~~ A ToS

→ **Curry Harvard Isomorphism Theorem**

Theorem = Types

Theorems ≡ Types
Proofs ≡ Programs for that Types

Type checking is undecidable.

→ SSA

$y = x + z$	~~$y_1 = x + z$~~	$y = 0$	$y_1 = 0$
⋮	⋮	if (. . .)	if (. . .)
$y = 2a$	$y_2 = 2a$	$y = 1$	$y_2 = 1$
⋮	⋮	else $y = 2$	else $y_3 = 2$
$= y$	$= y_2$		$y_5 = \phi(y_2, y_3)$
⋮	⋮	$a = 2y$	
$y = 37$	~~$y_3 = 37$~~		$a = 2y_5$

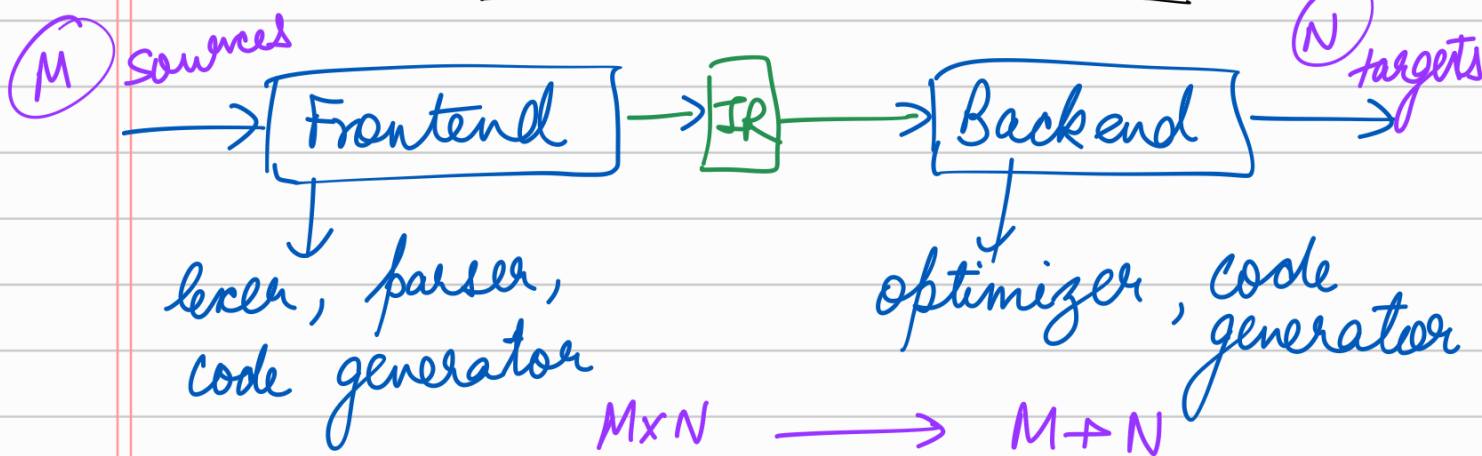
$y_5 = \phi(y_2, y_3)$ → pseudo instr ⇒ dep on branch ill use y_2 or y_3. Keep both

preserves loops

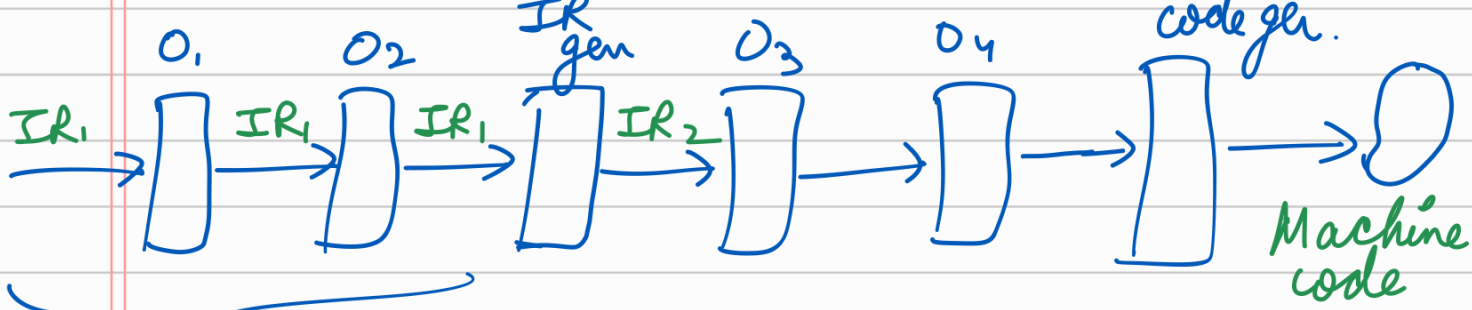
HIR — High level IR
3addr code ← MIR — Med " "
LIR — Low " "
↳ virtual registers

} Lower to Machine code

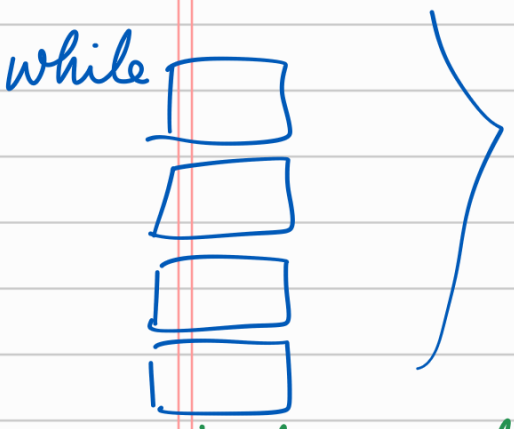
Lec 20/01/25



M × N ⟶ M + N



can rearr. these in any order. take IR_1 and give IR_1



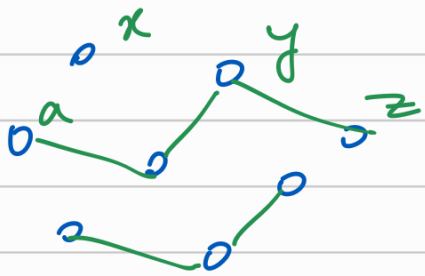
loop unrolling: so that optimisⁿ can occur within loop body. (above & below code can't be brought inside loop for optimⁿ)

loop Unrolling factor — not too big, otherwise L1 Ins cache miss ↑. So entire loop should be in cache. Similarly register pressure should also be low.

RTL → indep of which machine we are using but very similar to machine code

Peephole — very local, small 3-4 line code optimisatⁿ, machine specific

Register Allocⁿ — graph colⁿ → edges b/w variables which can't be allocated on same reg.



$x = y + z$

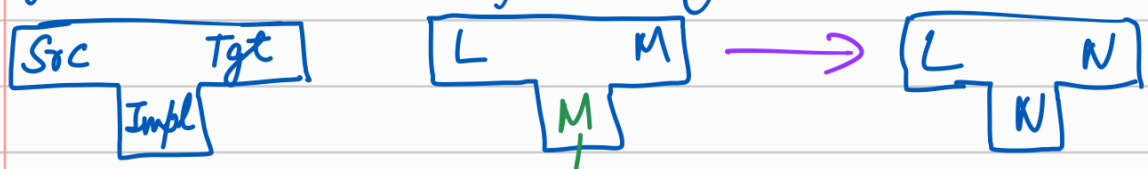
$a = 72$

col → NP complete

Lec 21/01/25

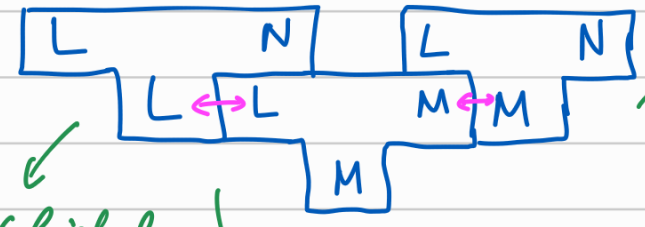
Bootstrapping

Given compiler for lang L for machine M



if nothing ment:
then default impl = Tgt

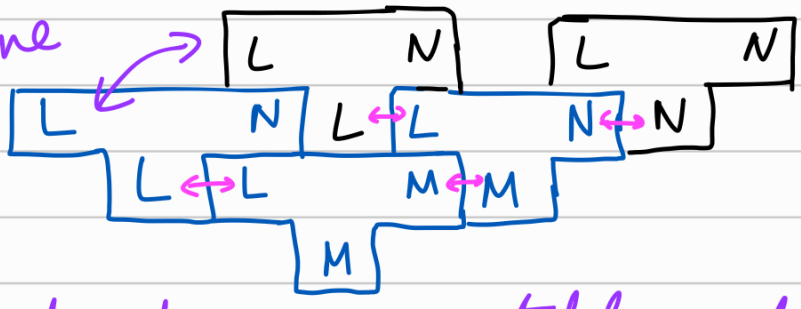
Produce a compiler for L for machine N



output L → N written in M

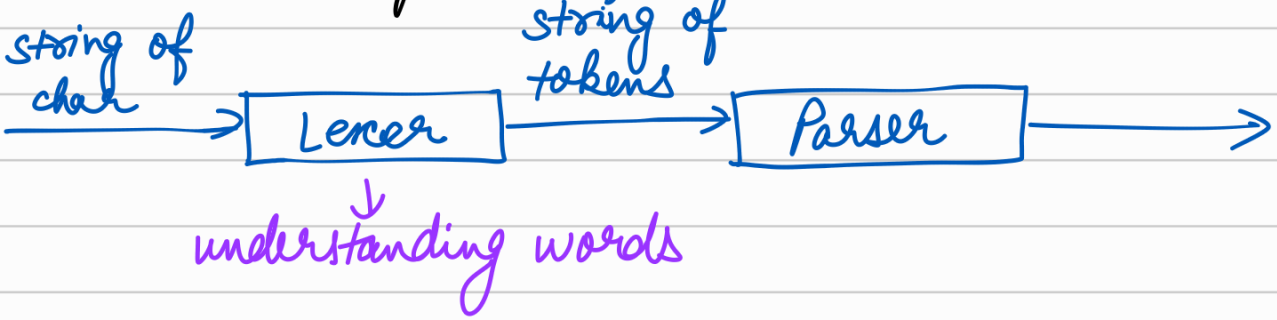
we write compiler in L (high lang)

L we had



we produced an executable compiler L → N in M. we use it on our first written compiler in L (high lang) & it becomes an L → N compiler in N.

→ # Lexical Analysis



Token: class of strings that belong to the same eq. classes. e.g. identifiers, datatypes

Lexeme: The objects that belong to a token

token = number "5" lexeme

token.type | int | a = 5 "int" lexeme

token = keyword

token = identifier "a" lexeme

token = identifier, type = int, lexeme = a

token = const, type = int, lexeme = 5

token = keyword, lexeme = for

- Driven by syntax analyser, not by lexer

regex for every token. lexeme is a member of a token. But parallel matching into many token classes. Who can accept this tell me ——

Priority

Keyword > Identifier

Maximal Munch

· Eat as big a token as possible

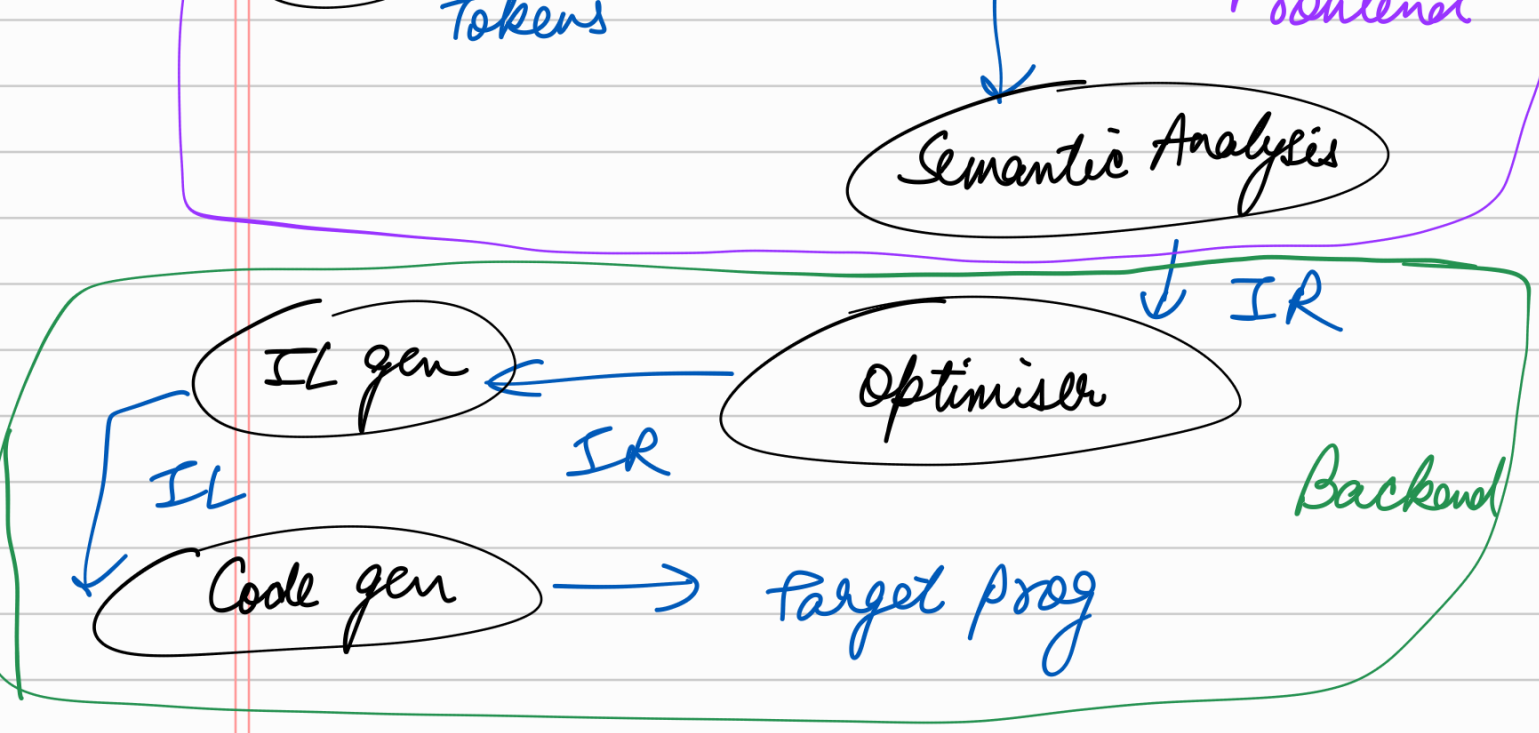
* : that token needs to be pushed back into the input buffer as it is a part of someother token.

Prefix Matching: So as to reduce parallel matching to many possibilities — Prefix tree thing.

Lec 27/01/25

Source Prog → Lexer → Parser → Parse Tree (AST)

Front End



Optimisers → take same IR and output the same IR (low → low, m → m, high → high)

IR code gen → one IR to other IR

- every phase interacts with the symbol table.
- AST preserves all features of source code. Exactly.

Lexeme	Token	Type	Mem. align

Store just pointers

xyz
x
yzt

$ = demarker

x	y	z	$	x	$	y	z	t	$

uses cache locality.

Can use hashes to index in the symbol table for faster access.

→ func 3AC foo(a, b) ⟵ param b, param a, call foo, 2

→ 3AC in-mem representation

x = y op z (atmost 3 addr, 1 operator)

```
typedef struct {
    InstoType typ;
    in 1
    in 2
    out
    jmp tgt,
    operator
} 3AC;
```

→ direct array index of ir[] as jmp target.

whole prog → struct 3AC ir[];

→ Java Bytecode is an IR.

→ Stack machine : Push a, Push b, add, store c } c = a+b

→ declarations of all includes
preprocessor
compiler (file by file)
assembler (.s → .o)
linker (library func def^n's)
loader (store it appropriately on the disk)

main.c → compiler → main.s → assembler → main.o
a.c → compiler → a.s → assembler → a.o

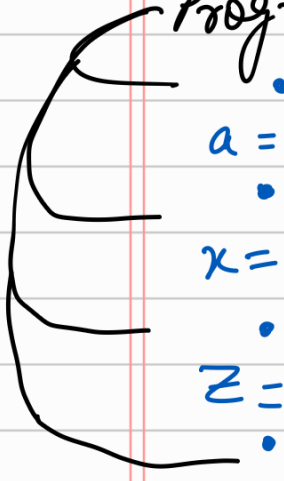
dynamic linker + loader ⇒ a bit slow execution as binary has to do a bit of

linker loader work.

Lec 28/01/25

Liveness of variables

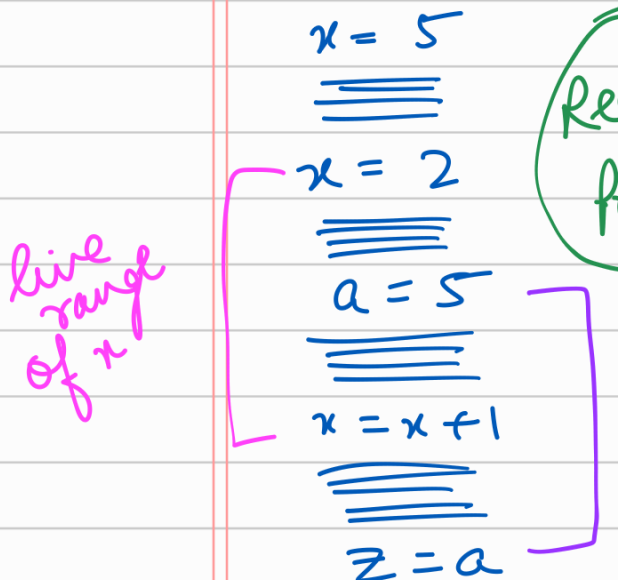
Program Points → points of execution.



- • nothing live
- `a = 2`
- • nothing live (a never used)
- `x = 5`
- • x live
- `z = y + x`
- • nothing live

live range of x

We want live variables in registers in their live ranges.



Register Pressure

Best Performance:

if we have no. of regs = max. size of live ranges intersection

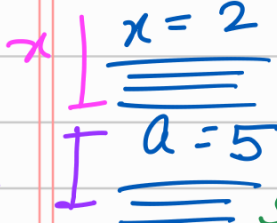
⇓

i can keep all my live variables in regs (fastest) during their live ranges.

What if we have just 1 reg in above ex:

(Splitting live ranges)

↳ by spilling into memory



store (x) onto Stack (memory)

store(a) load (x)

x x = x+1 Store (x) load (a)

a z = a

Split by spill

→ Next use Algo for reg alloc

Keep in reg the variable to be used just next.

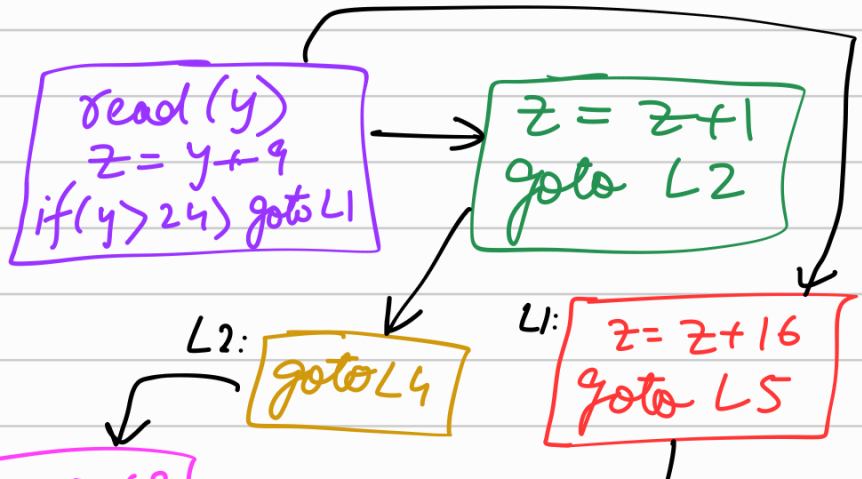
→ Basic Blocks

↳ on the IR (not source code)

Control Flow Graphs

- nodes : basic-blocks → single point of entry & exit
  - seq of st line code ends with if else.
- edges : if we can go from one bb to other then add edge b/w these

```
read (y)
z = y+9
if (y > 24) goto L1
  z = z+1
goto L2
L1: z = z+16
    goto L5
L2: goto L4
L4: a = z+2
    goto L5
L5: return
```



L4: a = z+2
goto L5

L5: return

if we put L5: return in L4: bb then we have a problem.
how will L1: jump to L5 since it always jumps on start of a bb NOT in the middle of a bb

---

(✓ in purple)

Leader — entry pt of a prog
— tgt of a jmp
— fall through of condn. jump

control can enter a bb only through leaders SO we mark leaders and they demarcate basic blocks.

```
read (y)                    ✓
z = y+9
if (y>24) goto L1
z = z+1                     ✓
goto L2
L1: z = z+16                ✓
    goto L5
L2: goto L4                 ✓
L4: a = z+2                 ✓
    goto L5
L5: return.                 ✓
```

---

dead code v/s unreachable code

control-flow : how control (instructions) flow during execution

data flow : how data flows between variables. when where used.

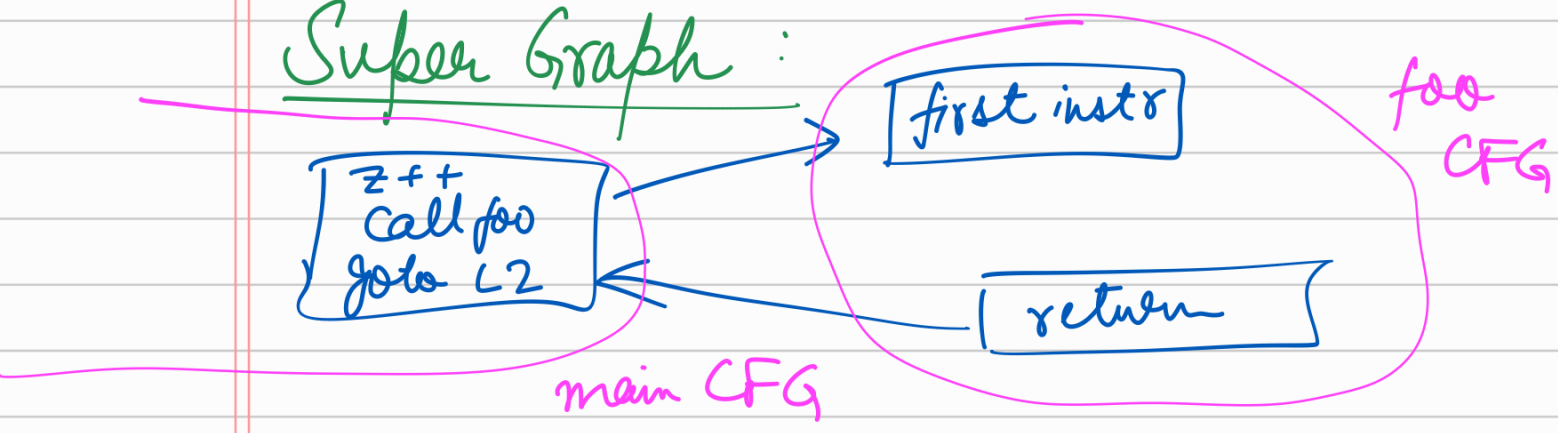
Types of analyses

1. **Local** : restricted to a basic block.
2. **Global** : restricted to a function / procedure. Each procedure gets represented to a single CFG. so we see the whole CFG (one procedure) on the Global scale.
3. **Interprocedural** : See across diff CFGs i.e. across functions (optimisations)

```
z = z+1
call foo(y)
goto L2.
```

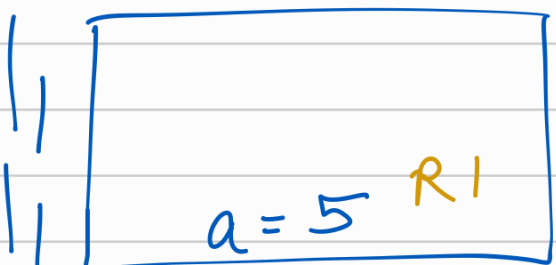
→ Single bb, because execution goes at call & comes back here so it is a linear execution (calls & returns need to be matched)

Super Graph :

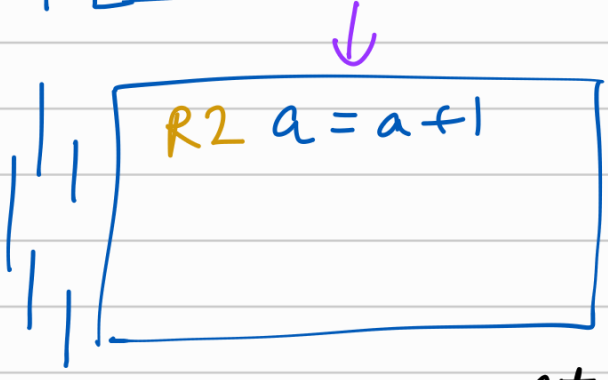


* we will analyse live ranges locally only. (only in each bb).

→ ## Live Ranges in BBs



when we exit a BB we spill all registers into memory & now entering



the other BB all regs are free and mem has all correct values to load in next BB

		at this point
1	a = 2	~~a is live 5~~ x is dead 5
2	a = 5	x is live 4 y is dead 4
3	y = z+1	z is live 3 q is dead 3
4	y = x+1	a is dead 2
5	x = a	

$t = x+1$
$x = t$

$x = x+1$

first dead then live so that every assignment of x gets a use in the future. So x is live.

x is dead
x is live

Live = use in the future

Faint Variable Analysis:

$x = x+1$ in loop. (no where else)

if a variable is used only in that statement and no where else then faint variable.

Not faint ⇒ has to be used in a diff statement.