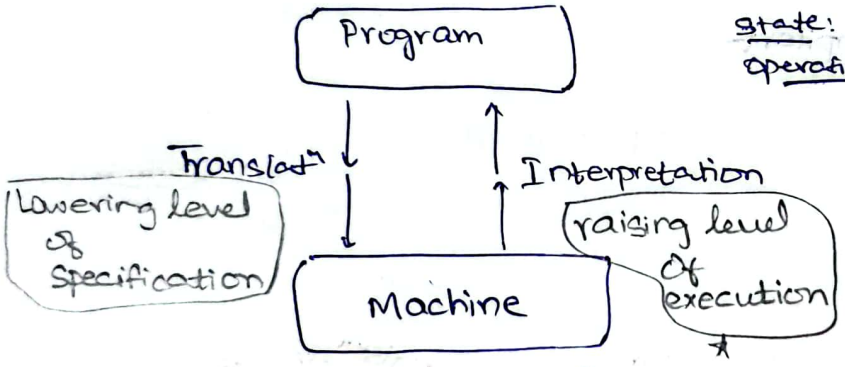
Overview to compilers:-

* **compilation vs interpretation:**

1) Src Program → Translator → Target Program → Execution Machine (Input Data)

Source program → Interpretor (Machine)

2) A gap exists, b/w the program specification and execution.



State: Variables
Operations: Assignments, control flow.

State: registers, memory
Operations: Machine instructions.

* **Why both compilers AND interpreters?**

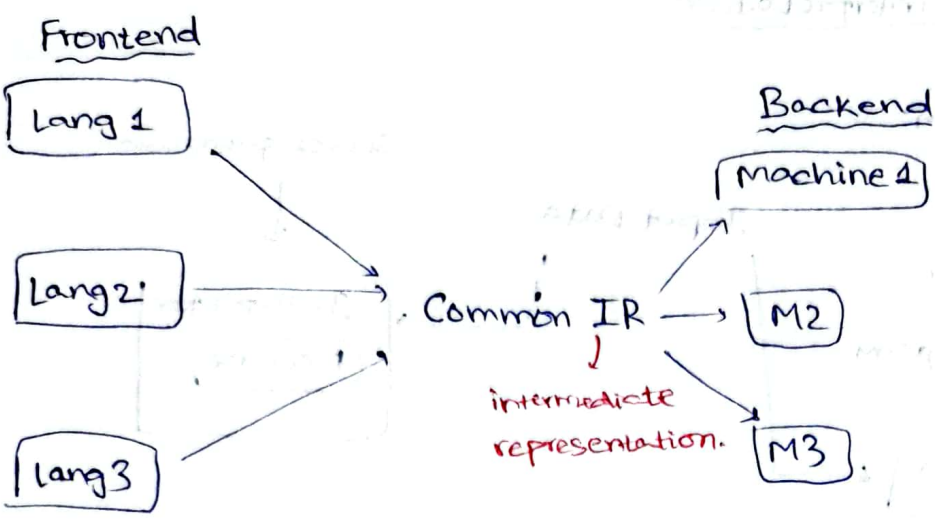
$$t_{compiler} = \underbrace{t_{Analyze} + t_{optimize} + t_{Synthesis}}_{\text{overhead}} + (t_{execute} \times j)$$

$$t_{interpreter} = (t_{Analyze} + t_{book\ keeping} + t_{exec}) \times j$$

Interpreter overhead.

Eg: for a browser, 'j' is large; so use compilers over interpreters.

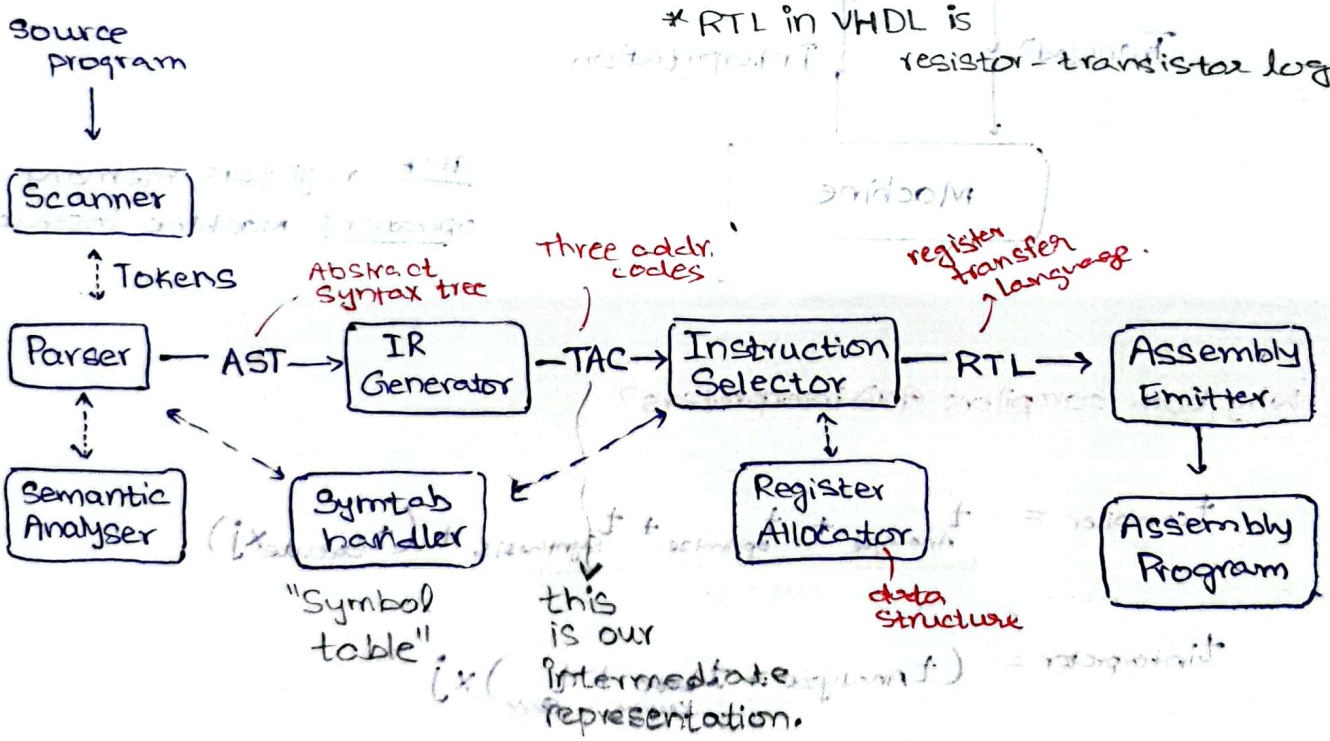
→ Reusability of lang processor modules:-



m x n compilers from m+n modules.

'retargetable'

→ Structure of a simple compiler:-



* RTL in VHDL is resistor-transistor logic.

<slide 55/210 - 101>

Demo of all intermediate States.

parse tree → AST → TAC → RTL → SPIM instructions.

S.no	State	matching string.	Buffer	Next char	Last Final state.	marked position	Action.

SLR(0) < SLR(1) < LALR(1) < CLR(1)

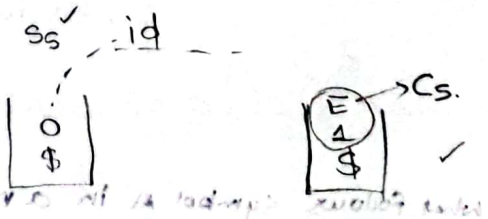
Symbol:	Notation:
Single terminal	a, b, c
Single non-terminal	A, B, C
Single grammar Symbol from (NUT)	X, Y, Z
String of terminals	x, y, z
String of grammar symbols	α, β, γ
null	∈

Shift reduce parsing

Step	Stack	rem. input	action
1	$	id*(id+id)$	shift id.

State	id	+	*	(	)	$	E	T	F
0	S5			S4			C1	C2	C3
1		S6				acc	C5	C4	
2									
3									
4									

Action | Goto



viable prefix:

prefix of a right sentential form that doesnot extend beyond handle.

valid item:

is a grammar production with (•).

$E \rightarrow \bullet A$ (Kernel)

$A \rightarrow \bullet B$
$B \rightarrow \bullet x$ (closure items) } Set of Items form a state in DFA

LR(K)
- left to right input read
- right sentential in reverse
- K lookaheads in items

SLR(0) : LR(0) items & no lookahead in input (shift?)

SLR(1) : LR(0) items & 1 lookahead in input.

CLR(1) : LR(1) items & 1 lookahead in input (canonical)

FIRST() & FOLLOW() sets (terminals only)

contains terminals, that may begin a string derivable from (•)

* if β derives ϵ; $\epsilon \in First(\beta)$

$A \rightarrow X_1 X_2 \cdots X_k$
then $First(A) \supseteq X_i$
provided $\forall j < i, \epsilon \in First(X_j)$

Least fixed point soln of following constraints.

---

follow: → what follows symbol A in a right sentential form.

$A \rightarrow \alpha B \beta$

1) if A is start symbol (end symbol)
$Follow(A) \supseteq \{\$\}$

2) $Follow(B) \supseteq First(\beta) - \{\epsilon\}$

3) if $\epsilon \in First(\beta)$ or $\beta \equiv \epsilon$
$Follow(B) \supseteq Follow(A)$

* SLR(1) parsing needs follow sets for symbols; to determine when to reduce... or reject.

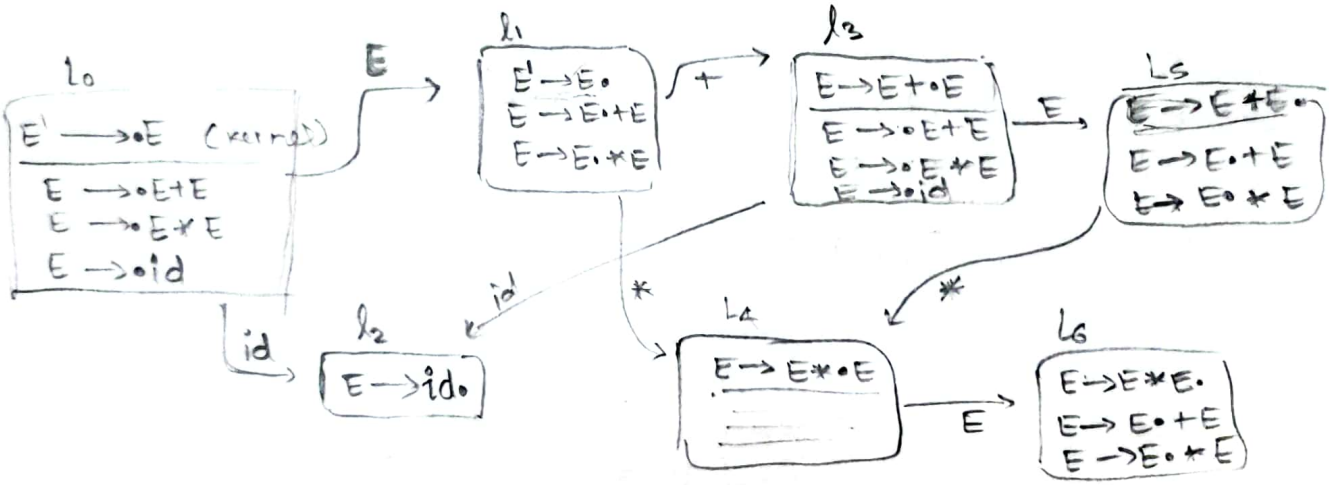
$E' \to E$
$E \to E+E$
$E \to E*E$
$E \to id$

Follow(E') ⊇ {$}
Fol(E) ⊇ Fol(E')
Fol(E) ⊇ {+, *}
Fol(E) = {$, +, *}

$E' \to$

$A \to \alpha$

E'
id + id



	id	+	*	$	E
0	S2				1
1		S3	S4	acc	
2		r3	r3	r3	
3	S2				5
4	S2				6
5		~~S3~~/r1	S4/r1	r1	
6		~~S3~~/r2	~~S4~~/r2	r2	

reduce;
when

- complete item
- no transition
- next inp is in Follow set of LHS.

* $S \to L \cdot = R$, $R \to L \cdot$ —=→ ----

& Follow(R) ⊇ {=}

then, shift-reduce conflict!

LR(1):-

items are

$$A \longrightarrow \alpha \cdot \beta, \underbrace{a}_{\text{lookahead}}$$

* if S is start symbol, then I0 has $S' \to \cdot S, \$$

* lookahead don't change on shift.

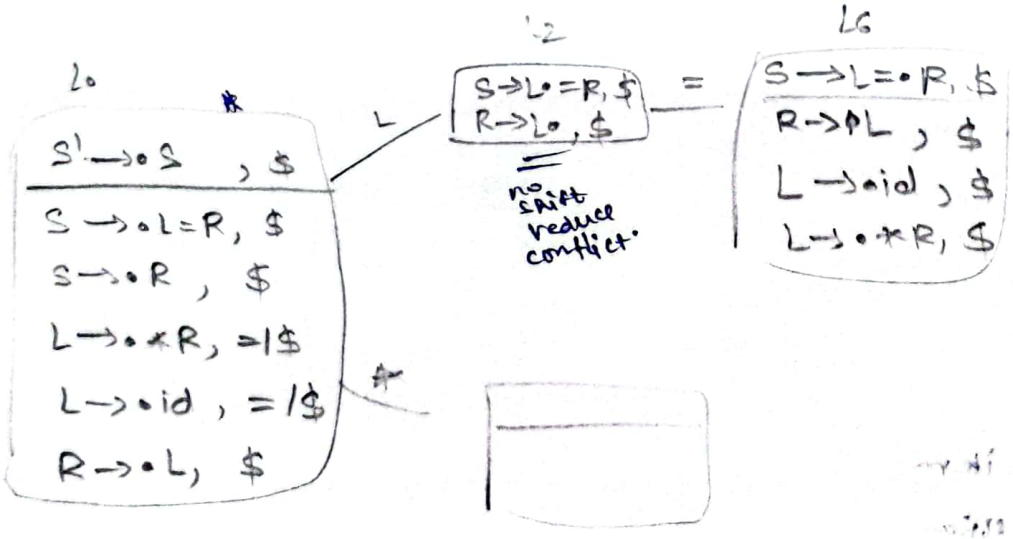
lookahead only change on closure.

* closure of $A \to \alpha \cdot B\beta, a$ contains

$B \to \cdot\gamma$, First(β, a)

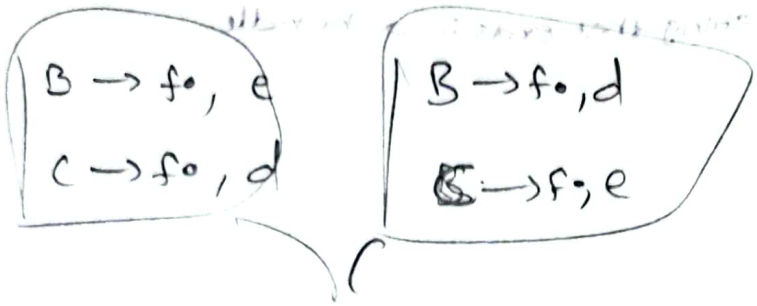
$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

All follows	
S'	{$}
S	{$}
R	{=, $}
L	{=, $}



LALR(1) merge state

introduces reduce-reduce parser.



4 arrays representation

Transition table:

state	0	1	2	3
0	0	1	2	3
1	1	1	0	3
2	1	1	3	3
3	0	1	2	

→ input symbol

index	Next	check
0	0	3
1	1	3
2	2	3
3	3	0
4	1	2
5	0	1
6	3	2
7	NONE	NONE

state	Default	Base
0	3	0
1	2	8
2	0	4
3	NONE	0

write none!! this will hint at empty transition...

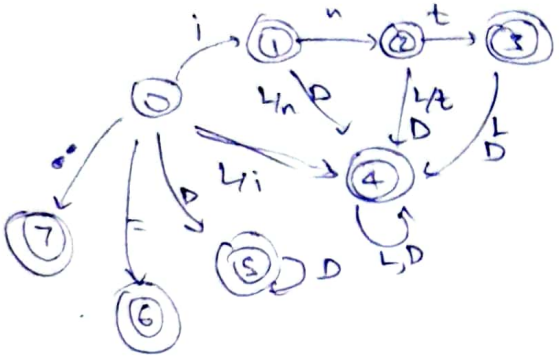
Step 3: identifying handles in right sentential forms:-

Viable prefixes:-

prefix of a RSF, that does not extend beyond the handle

- its either a string with no handle
- (or)
- a string that ends in a handle.

Scanning example:-



Sno	S	MS	Buffer	NC	LFS	MP	Act.
1	0	-	int␣int32=S;␣	i	-	-	-
2	1	i	int␣int32=S;␣n	1	1	-	
3	2	in	"	t	2	2	-
4	3	Int	int␣int32=S;␣	␣	3	3	FoundINT
5	0	-	␣int32=S;	␣	-	-	Discard
6	0	-	int32=S;␣	i	-	-	-
7	1	i	int32=S;␣n	t	1	1	

Semantic Analysis:-

compile time errors

link time errors

Runtime errors

- undefined behaviour : unchecked prohibited (warnings) (Java has no warnings; these also errors)

  unspecified behaviour : implementation defined (well-defined by compiler)

  Exceptions: prohibited checked (errors)

→ SDDS :-

for generating IRs:-

$S \rightarrow id = E$ $\quad C_1 = gen(id.place, =, E.place)$

$S.code = E.code \| C_1$

$E \rightarrow E_2 \; op \; E_3$ $\quad t_1 = getNewTemp();$

$C_1 = E_2.code; \; C_2 = E_3.code$

$C_3 = gen(t_1, =, expr(E_2.place, OP, E_3.place))$

$E_1.code = C_1 \| C_2 \| C_3$

$E_1.place = t_1$

$E_1 \rightarrow (E_2)$ $\quad E_1.code = E_2.code$

$E_1.place = E_2.place$

$E \rightarrow id$ $\quad E_1.code = NULL$

$E_1.place = id.name$

$E_1 \rightarrow E_2 ? E_3 : E_4$ $\quad t_1 = getNewTemp(); \; t_2 = getNewTemp()$

$l_1 = getNewLabel(); \; l_2 = \_\_\_(\,);$

$C_1 = E_2.code \| gen(t_1 = E_2.place) \| gen(if \; t_1 \; goto \; l_1)$

$C_2 = E_3.code \| gen(t_2 = E_3.place) \| gen(goto \; l_2)$

$C_3 = gen(l_1:) \| E_4.code \| gen(t_2 = E_4.place)$

$C_4 = gen(l_2:)$

$E_1.code = C_1 \| C_2 \| C_3 \| C_4$

$E_1.place = t_1$

$S_1 \rightarrow$ while $(E)\ S_2$

$t_1 = getNewTemp();$

$l_1 = getNewLabel();\ l_2 = \_\_\_\_;$

$C_1 = gen(l_1:) \,||\, E.code$

$C_2 = gen(t_1 = !E.place) \,||\, gen(if\ t_1\ goto\ l_2)$

$C_3 = S_2.code \,||\, gen(goto\ l_1)$

$C_4 = gen(l_2:)$

$S_1.code = C_1 || C_2 || C_3 || C_4$

Array access

S.code, S.place, E.place, E.code, id.name, num.value

A.name : name of array

A.offset : name of var. holding offset of A

A.code : code that access arr. elem A

A.ndim : dimension no. being considered

width(A) : no. of bytes need for each elem.

dimLimit(A,i) : no. of elem in dim i

$E \rightarrow id$ E.code = null; E.place = id.name

$E \rightarrow num$ E.code = null; E.place = num.value

$E \rightarrow A$: $t_1 = get\_\_\_\_(); \ t_2 = \_\_\_\_()$.

$C_1 = gen(t_1 =, A.offset \times width(A.name))$

$C_2 = gen(t_2 =, A.name, [, t_1,])$

E.code = A.code || $C_1 || C_2$

E.place = t_2

$A \rightarrow id[E]$ A.name = id.name

A.ndim = 1

A.offset = E.place

A.code = E.code

$A_1 \rightarrow A_2[E]$ $t_1 = \_\_\_\_();\ t_2 = \_\_\_\_()$

$A_1.name = A_2.name,\ A_1.ndim = 1 + A_2.ndim$

$C_1 = gen(t_1 =, A_2.offset \times dimLimit(A_1.name, A_1.ndim))$

$C_2 = gen(t_2 =, t_1, +, E.place)$

$A_1.code = A_2.code \,||\, E.code \,||\, C_1 \,||\, C_2$

$A_1.offset = t_2$

Field access: Approach 1 → no F.code since compile time calculation.

F.name : name of struct var.

F.offset : offset for the field

F.type : type of struct var.

offset(T, f) : offset of type f in struct T.

type(T, f) : type of f in struct T.

E → F : $t_1 =$ ____ $t_2 =$ ____ $t_3 =$ ____

C_1 = gen(t_1, =, &F.name)

C_2 = gen(t_2, =, t_1 + F.offset)

C_3 = gen(t_3, =, *t_2)

E.code = C_1 || C_2 || C_3 ; E.place = t_3

F → id1.id2 F.name = id1.name

F.type = type(id1.type, id2.name)

F.offset = offset(id1.type, id2.name)

F_1 → F_2.id F_1.name = F_2.name

F_1.type = type(F_2.type, id.name)

F_1.offset = F_2.offset + offset(F_2.type, id.name)

Approach 2 : runtime computation:

- F.code is there.
- F.address is there....
- F.type

E → F $t_1 =$ ____ ; E.place = t_1

E.code = F.code || gen(t_1, =, *F.address)

F → id1.id2 $t_1 =$ ____ $t_2 =$ ____

F.type = type(id1.type, id2.name)

C_1 = gen(t_1, =, &id1.name)

F.code = C_1 || gen(t_2, =, t_1 + offset(id1.type, id2.name))

F.address = t_2

$F_1 \rightarrow F_2.id$ $t_1 = \_\_\_()$

$F_1.type = type(F_2.type, id.name)$

$C_1 = gen(t_1 =, F_2.address + offset(F_2.type, id.name))$

$F_1.code = F_2.code \,||\, C_1$

$F_1.address = t_1$

$F \rightarrow id_1 \; '\rightarrow id_2$ let T be $id.type = pointer(T)$

$t = \_\_\_()$

$F.type = type(T, id_2.name)$

$F.code = gen(t_1 =, id.name + offset(T, id_2.name))$

$F.address = t_1$

$F \rightarrow F_2 \; '\rightarrow id$ T be such ~~that~~ $F_2.type = pointer(T)$

$t_1 = \_\_\_(), t_2 = \_\_\_()$

$F_1.type = type(T, id.name)$

$C_1 = gen(t_2 =, * F_2.address)$

$F_1.code = F_2.code \,||\, C_1 \,||\, gen(t_2 =, t_1 + offset(T, id.name))$

$F_1.address = t_2$

→ Syntax directed translation schemes :-

synthesized attributes (normal)	inherited attributes (from top & left)

* type is inherited to varlist.

* An SDD is S-attributed if all are synthesized.

is L-attributed if ~~using~~ inherited attributes that depend on some symbol on left.

$X \rightarrow Y_1 Y_2 X_3 X_4$

X_3 can depend on X, Y_1, Y_2

SDTS:

Decl → Type { Varlist.type = Type.name } Varlist.

actions.

declaration processing.

Type Analysis :-

- type expressions
- type equivalence
- type checking
- type inference

just if else

1) **type expressions**

- basic types eg. bool, int, float ..
- user defined types name.
- type constructor applied to T.

array (I, T)

pointer (T)

struct ((f1, T1) (f2, T2) ...)

name.

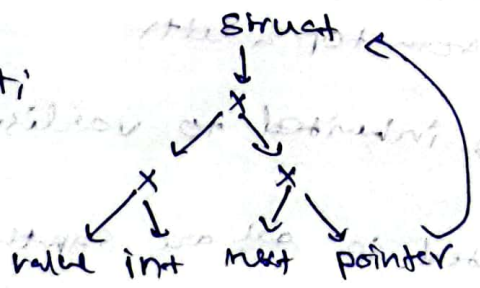
$T_1 \rightarrow T_2$ function.

$T_1 \times T_2 \times T_3$: product of types
: list.

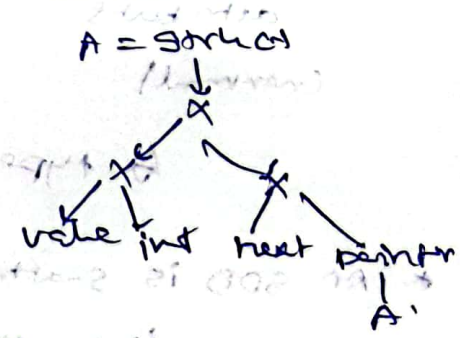
- name equivalence for types
- eg apple a_1, a_2
  banana b_1, b_2 ✓
  a_1, b_1 ✗
- **structural equivalence**
  - same basic types are equiv.
  - * C uses structural equiv. for everything except structs

```
struct A {
    int val;
    struct A* next;
}
```

type graph



type tree:



A = struct ((val; int)(next pointer(A)))

Name & Scope analysis

> stack of symbol tables.

Static scooping ✓

dynamic scooping :

- name declared in procedure enclosing X; in a call chain reaching X.

(easy) no of vars.

* Scope analysis – SDTS

translation rules.

Program → {push_new_symtab();} DL SL

DL → DL D | ∈

D → T id {addvartosymtab (id.name, T.type);}

D → T id {(add_ ,} {push_new_symtab} '(' PL ')' '{' DL SL '}' {popsymtab}

integer — action specification

T → int {T.name = int} | void {T.name = void}.

PL → PL, P | P

P → T id {add param tosymtab(id.name, T.name)}

SL → SL call | ∈

call → id {lookup(id.name)} (AL);

AL → AL, id {lookup(id.name)} | id {lookup(id.name)}

DL: list of declarations.
T: type.
PL: list of formal parameters
SL: list of statements.
AL: actual parameters

Declaration Processing :-

int a[20][10];

C array size calculation :-

X.bt base type
X.dt derived type
X.v value
X.s size
X.nm name
X.w width

D → T { I.bt = T.bt; I.w = T.w } I; { Enter_in_symtab (I.nm, I.dt, I.s) }

T → int { T.bt = int; T.w = 4 }

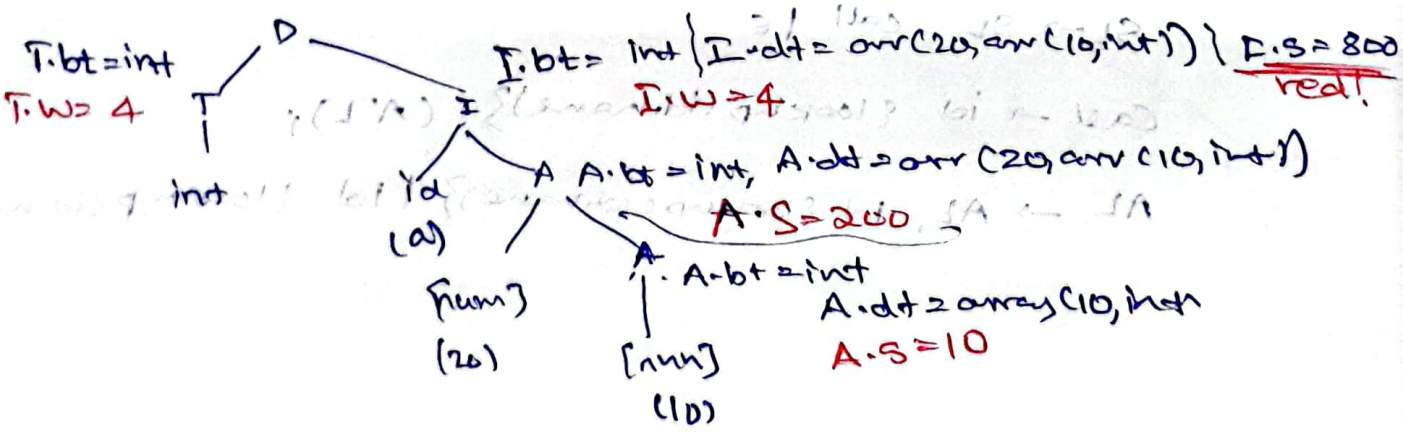
T → double { T.bt = double; T.w = 8 }

I → id { I.dt = I.bt; I.nm = id.nm; I.s = I.w }

I → id { A.bt = I.bt } A { I.dt = A.dt, I.nm = id.nm, I.s = A.s x I.w }

A → [num] { A.dt = array(num.v, A.bt); A.s = num.v }

A1 → [num] { A2.bt = A1.bt } A2 { A1.dt = array(num.v, A2.dt); A1.s = A2.s x num.v }



For pointers:

only additional rules.

Item1 → '*' {item2.bt = Item1.bt} Item2 { }

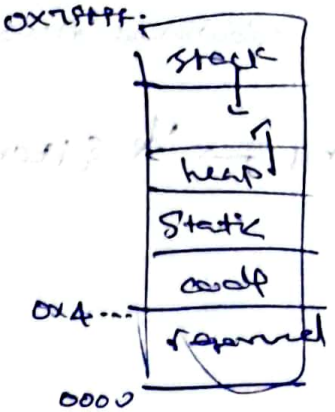
I1.dt = pointer(I2.dt);

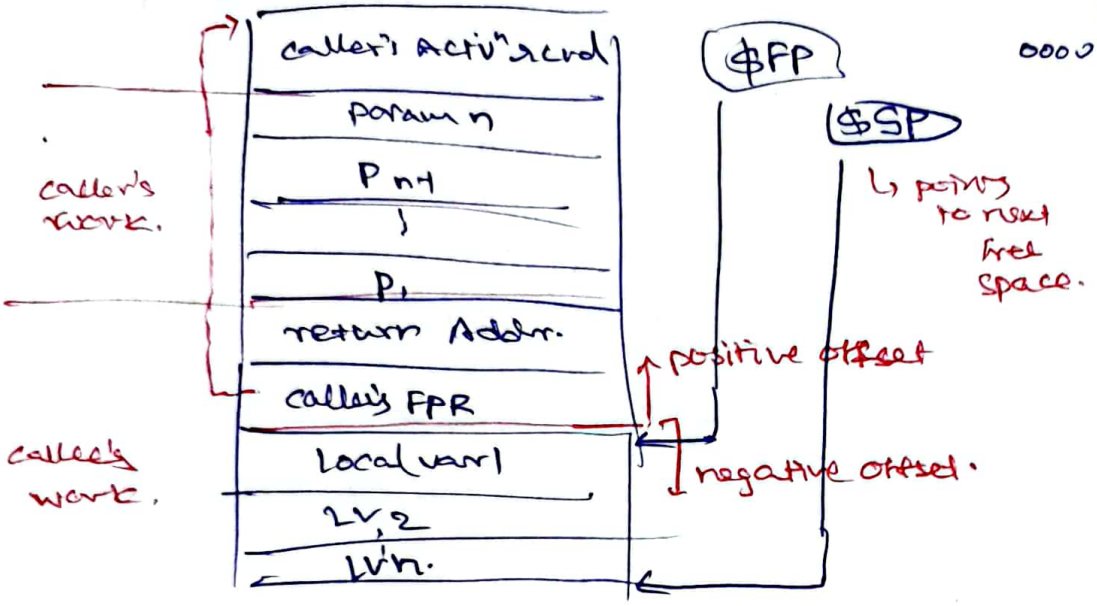
I1.s = 4

Runtime support :-

→ procedure calls :

Activation Records :-





variable	Address	size.
Param 1	8($fp)	P_1
P2	(8+P1)($fp)	P_2
P3	(8+P1+P2) ($fp)	P_3
L1	$-L_1$($fp)	l_1
L2	(-L1-L2)($fp)	l_2
L3	(-L1-L2-L3)($fp)	l_3

Example: function prologue, epilogue, call

```
int f(int a, int b)
{
    int c;
    c = a+b;
    return c;
}

int main()
{ int x,y,z;

  z = f(x,y);

  print z;
  return 0;
}
```

```
f_:
# prologue
sw $ra, 0($sp)     # save the return add.
sw $fp, -4($sp)    # save the frame ptr.
sub $fp, $sp, 4    # update frame ptr.
sub $sp, $sp, 12   # make space for locals, $ra $sp.
# prologue ends
---
---
epilogue f_:
#
add $sp, $sp, 12   # remove space of locals $ra, $sp
lw $fp, -4($sp)    # set $fp to $sp-4
lw $ra, 0($sp)     # save ra.
jr $ra             # jump to caller.
# epilogue ends
```

main function:

```
!
```

Prologue ....

Access-link:

How to access variables in scoping rules.

→ parameter passing:

```
int a
int main()
{
  a = 5;
  f(a, a+1);
  cout << a;
  return 0;
}
void f(x, y)
{
  a = a+10;
  a = a+y;
  x = x+y+100;
}
```

call by value-restore:-

```
a = 5        ; a = 5
t0 = a+1     ; t0 = 6
x = a        ; x = 5
y = t0       ; y = 6
a = a+10     ; a = 15
a = a+y      ; a = 21
x = x+y+100  ; x = 111
a = x        ; a = 111
t0 = y
```

call by name :

```
x ≡ a
y ≡ a+1        replace every occurance...
a = a+10  ; a = 15
a = a+y   ; a = 31
  ≡ a+a+1
x = x+y+100;
a ≡ a+a+1+100 ; a = 163
```

writes to (a+1) are invisible

✓ call by value : 21

✓ call by reference : 127

✓ call by value-result (copy-restore) : 111
copy back.

✓ call by name : 163
(like macros)

call by need : 143
(lazy evaluation first time)

call by need :

```
a = 5
x ≡ a   ; dont calculate yet
y ≡ a+1 ;  "
a = a+10 ; a = 15
a = a+y  ; "y = a+1 = 16
         " a = 31
x = x+y+100 ; x = a = 31
```

as a. → a = 147

only once lazy; not always.

" textual substitution of formal parameters by actual parameters, but evaluation only once "

```
g()
{
  f(e)
}
```

```
f(x)
{
}
```

call by name:

use(x) → use(e)
def(x) → def(*(&e))

call by need:

use1(x) → x = e; use(x)
use_{i>1}(x) → use(x)
def(x) → def(*(&e))

lol....

→ virtual functions:

final table

```
class A :
    virtual f()
    virtual f(string s)
    virtual g()
```

Vf
Vf(i)
Vg

V f()
V f(i)
V g()

```
class B : A
    virtual void g()
    void f()
```

Vf	f
Vf(i)	
Vg	Vg

B:f()
VA:f()
VB:g()

```
class C : B
    f()
```

VA:f()	B:f()	C:f()
VA:f(i)		
VA:g()	VB:g()	

C:f()
VA:f(i)
VB:g()

compile time activity:-

1) collect all virtual across class hierarchy
2) Ignore non-virtual functions
3) Analyze the hierarchy; for similar functions

Run time:

- dereference obj pointer to know the class & access virtual function table.
- add offset to base of the table to access function ptr
- Dereference funcptr to make call.

Code generation:-

TAC to RTL.

→ Register Allocation:-

Local reg. allocation: Sethi-Ullmann Algorithm.

- using registers to hold intermediate values of expressions

Global reg. Allocation: chaitin-briggs graph colouring.

- keep the values in registers across statements.

→ chaitin-Briggs register allocator:-

- identify live ranges
- compute interference graphs
- colour the graph.

Steps

1. Coalescing:

   eliminate copy statements $x=y$ or $c=t_1$.

2. Identify live ranges:-

   - Seq of statements from definition/assignment of a variable to its last use of that value.
   - Bottom up analysis.

3. Construction of inference graph:-

   live ranges l_1, l_2 interfer if definition/assignment of a variable of l_1 occurs in range of l_2 vice versa.

   "of a variable/temp symbol V_1"
   "of a variable/temp symbol V_2"

4) simplify the inference graph?

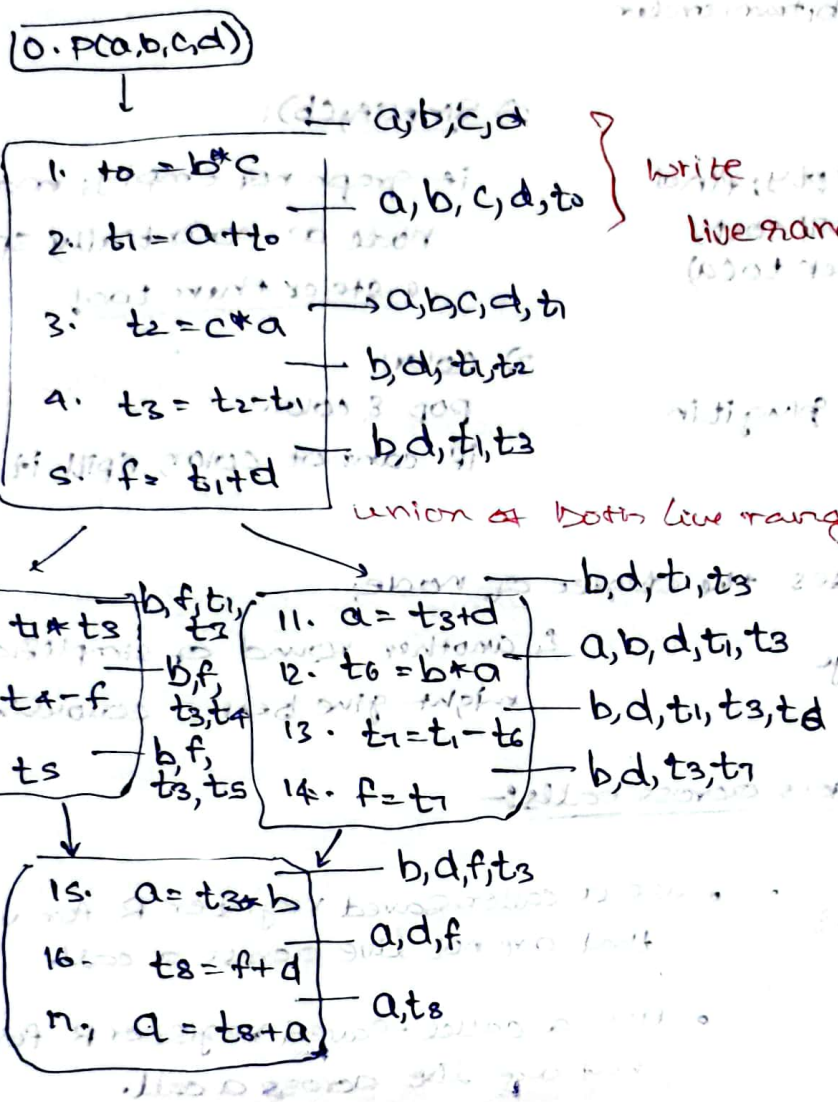
5) color the nodes.

Eg:

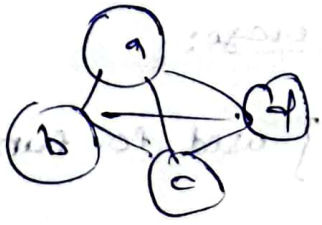
0. P(a,b,c,d) — $a_0\ b_0\ c_0\ d_0$
1. $t_0 = b * c$ — $a_0\ b_0\ c_0\ d_0\ t_1$ (Step-1)
2. $t_1 = a + t_0$ — $c_0\ b_0\ c_0\ d_0$
3. $e = t_1$ — $a_0\ b_0\ c_0\ d_0\ e_3$
4. $t_2 = c * a$ — $a_0\ b_0\ c_0\ d_0\ e_3$
5. $t_3 = t_2 - t_1$ — $a_0\ b_0\ c_0\ d_0\ e_3$
6. $c = t_3$ — $a_0 b_0 C_3 d_0 e_3$
7. $f = t_1 + d$ — $a_0 b_0 C_3 d_0 e_3 f_1$
8. $a = d + e$ — $a_3\ b_0\ c_3 d_0 e_3 f_1$ $\quad a = d + t_1$

Step-2)

- go from bottom to up.
- when in RHS, start live range.
- when in LHS, stop live range.



Eg1 if a line has a,b,c,d; then interference graph is :-



Eg2

Node	degree
a	7
b	11
t_1	8
t_2	3
⋮	⋮

→ Live range information:

→ always maintain in memory.

Live range	Degree D	Loads L	Stores S	Spill Cost C = L+S
a	9	4	3	7
b	11	0	3	3
⋮	⋮	⋮	⋮	⋮

K-coloring Chaitin:-

1) Simplify(a)

for n, D(n)<K, remove & push onto stack in arbitrary order.

2) Simplify(b)

if graph is not empty; find node with least spill cost, spill it and go back to(a)

3) color

pop from stack; plug it in graph & color.

K-coloring Briggs:-

1) Simplify(a):

for n when D(n)<K; push onto stack.

2) Simplify(b):

if graph not empty; mark the node as potentially spillable (PS) & stack them too!

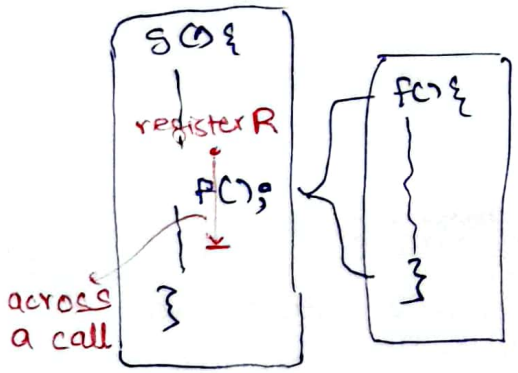
3) color:

pop & color. (in order of increasing degree)

if cannot color, spill it & go back to a.

* spilling decreases the degree of node; for a live-range & another round of simplification might give better colouring order.

→ **Managing registers across calls:-**



- use a caller-saved register R for values that are not live across a call.
- use a callee-saved register R for values that are live across a call.

* SCLP register usage:

v1
f0 (f0, f1) } used for function result.

v0
t0 - t9
s0 - s7 } int

f2 - f30 } float
(f2, f3) (f30, f31)

→ register allocation in SCLP:-

* no register is occupied across any assignment in the program.

for LHS variable, result is stored in memory.

not temporary? hmmm.... NO!

1) When temporary is assigned a register, mark it occupied.
2) When temporary is used in TAC statement, mark the register free.

TAC order for register allocation : first opd, result, second opd.

→ Integrated Instruction Selection, & register Allocation:- local. reg. allocation

Sethi-Ullman Algo:-

- Simple machine model
- optimal no. of instructions with min. no of registers & steps
- Linear in size of exp. tree.

Aho-Thonson Algo:-

- very general machine model
- linear in exp. tree size
- optimal in terms of cost of execution.

- contiguous evaluation!

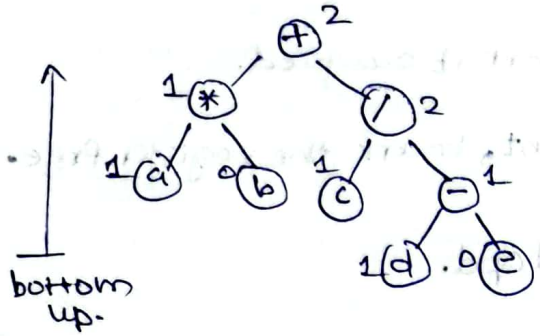
Instructions:-

r ← m
m ← r
r ← r1 op r2
r ← r1 op m

Algorithm:-

1. Traverse tree bottom up & label each node with min. no. of registers needed to evaluate the subexpression.
2. Traverse top-down & generate code.

$$label(n) = \begin{cases} 1, & n \text{ is leaf \& in register (left leaf)} \\ 0; & n \text{ is right leaf} \\ \max(l_1, l_2); & n \text{ has two children with } l_1 \neq l_2. \\ l_1+1; & \text{when } l_1 = l_2. \end{cases}$$



Always first generate RTL for post with more label.

→ Generating code:-

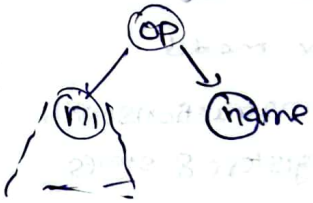
case-1:

name n

gencode(n) is

emit (top(rstack) ← name)

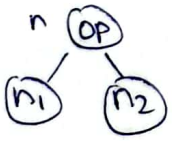
case-2:



gencode(n):

```
gencode(n1)
emit(top(rstk) ← top(rstk) op name)
```

case-3:



label(n1) ≥ label(n2) & label(n2) < k

gencode(n):

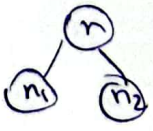
```
gencode(n1)
R = pop(rstk)
gencode(n2)
emit(R ← R op top(rstk))
push(R, rstk)
```

case-4:

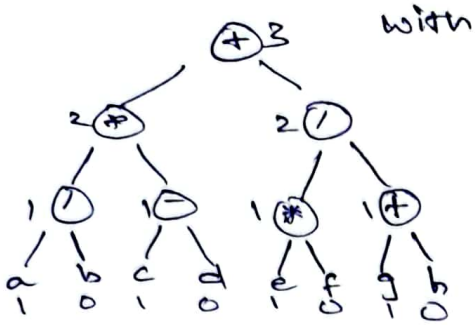
label(n1) < label(n2) & label(n1) < k

```
swap(rstk)
gencode(n2)
R = pop(rstk)
gencode(n1)
emit(top(rstk) ← top(rstk) op R)
push(R, rstk)
swap(rstk)
```

case-5: label(n1) ≥ k & label(n2) ≥ k




```
gencode(n2)
T = pop(tstack)
emit(T ← top(rstk))
gencode(n1)
emit(top(rstk) ← top(rstk) op T)
push(T, tstack)
```



with 2 registers:-

K=2

$r_0 \leftarrow e$
$r_0 \leftarrow r_0 * f$
$r_1 \leftarrow g$
$r_1 \leftarrow r_1 + h$
$r_0 \leftarrow r_0 / r_1$
$t_0 \leftarrow r_0$

right subtree.

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$
$r_0 \leftarrow r_0 * r_1$

left subtree.

$r_0 \leftarrow r_0 + t_0$

→ Dense nodes & Major Nodes:

- if label(n) ≥ k
- if both children are dense nodes!

major node also a dense node.

- node is dense → parent is dense.
- node is major → parent is need not major.

* if we make a store, then we decrease a dense node & its parent node will no longer be major.

* A store can reduce no. of major nodes by atmost one.

& so; we call a store for every case-s

major node

& so; our no. of stores is optimal.