Mo Tu We Th Fr Sa Su

Date

→ PARSE TABLE:

① grammar: ..... A → (A) ..... A → a .....

STATE	ACTION (	a	)	$	GOTO A
0	S2	S3			1
1				acc	
2	S2	S3			4
3			r2	r2	
4			S5		
5			r1	r1	

② 1. S → L=R 3. L → *R 5. R → L

2. S → R 4. L → id

multiple entries ↓ not SLR

GOTO ↗

STATE	ACTION =	*	id	$	GOTO S	L	R
0		S4	S5		1	2	3
1				acc			
2	S6/r5			r5			
3				r2			
4		S4	S5			8	7
5	S4			r4			
6		S4	S5			8	9
7	r3			r3			
8	r5			r5			
9				r5			

Memo No.

Date / /

Mo Tu We Th Fr Sa Su

Q. $S \to ABCDE$ $A \to a|\varepsilon$ $B \to b|\varepsilon$ $C \to c$
$D \to d|\varepsilon$ $E \to e|\varepsilon$

A. First (S) = {a, b, c} First(C) = {c}
First (A) = {a, ε} First(D) = {d, ε}
First (B) = {b, ε} First(E) = {e, ε}

Follow (S) = {$}
Follow (A) = {b, c}
Follow (B) = {c}
Follow (C) = {d, e, $}
Follow (D) = {e, $}
Follow (E) = {$}

Q. $S \to Bb|Cd$ $B \to aB|\varepsilon$ $C \to cC|\varepsilon$

A. First (S) = {a, b, c, d}
First (B) = {a, ε}
First (C) = {c, ε}

Follow (S) = {$}
Follow (B) = {b}
Follow (C) = {d}

Mo Tu We Th Fr Sa Su

Memo No.

Date / /

Q. $S \to ACB \mid CbB \mid Ba \quad A \to da \mid BC$

$B \to g \mid \varepsilon \quad C \to h \mid \varepsilon$

A. First (S) = { d, g, h, b, a, ε }

First (A) = { d, g, h, ε }

First (B) = { g, ε }

First (C) = { h, ε }

Follow (S) = { $ }

Follow (A) = { h, g, $ }

Follow (B) = { $, a, h, g }

Follow (C) = { b, g, $, h }

$A \to A\alpha \mid \beta$

$\hookrightarrow A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

Memo No. _____

Date 16 / 4 / 24

Mo Tu We Th Fr Sa Su

CC

→ LEFT RECURSION:

- if there is a non-terminal such that there i
a derivation:

$$A \Rightarrow A\alpha \quad \text{for some string } \alpha:$$

- top down parsers can't parse left recurse gramma

* Immediate Left Recursion:

rule: $A \to A\alpha \mid \beta$ ← β does not start w/ A.

new: $\begin{cases} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{cases}$ ← new grammar is equivalen

general form: $A \to A\alpha_1 \mid A\alpha_m \mid \dots \mid \beta_1 \mid \dots \mid \beta_n$

$$A \to \beta_1 A' \mid \dots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \dots \mid \alpha_m A'$$

Q. $E \to E+T \mid T \quad T \to T*F \mid F \quad F \to id \mid (E)$

A. ① $E \to E+T \mid T \Rightarrow E \to TE'$

$$E' \to +TE' \mid \varepsilon$$

② $T \to T*F \mid F \Rightarrow T \to FT'$

$$T' \to *FT' \mid \varepsilon$$

Mo Tu We Th Fr Sa Su

Memo No.

Date 24 / 04 / 24

→ Symbol Table:

- data structures used by compilers to store info about in program constructs
- program is defined & separated into blocks - using '{ }' as delimiters.

↳ table stores scope of variables/identifier, value, etc.

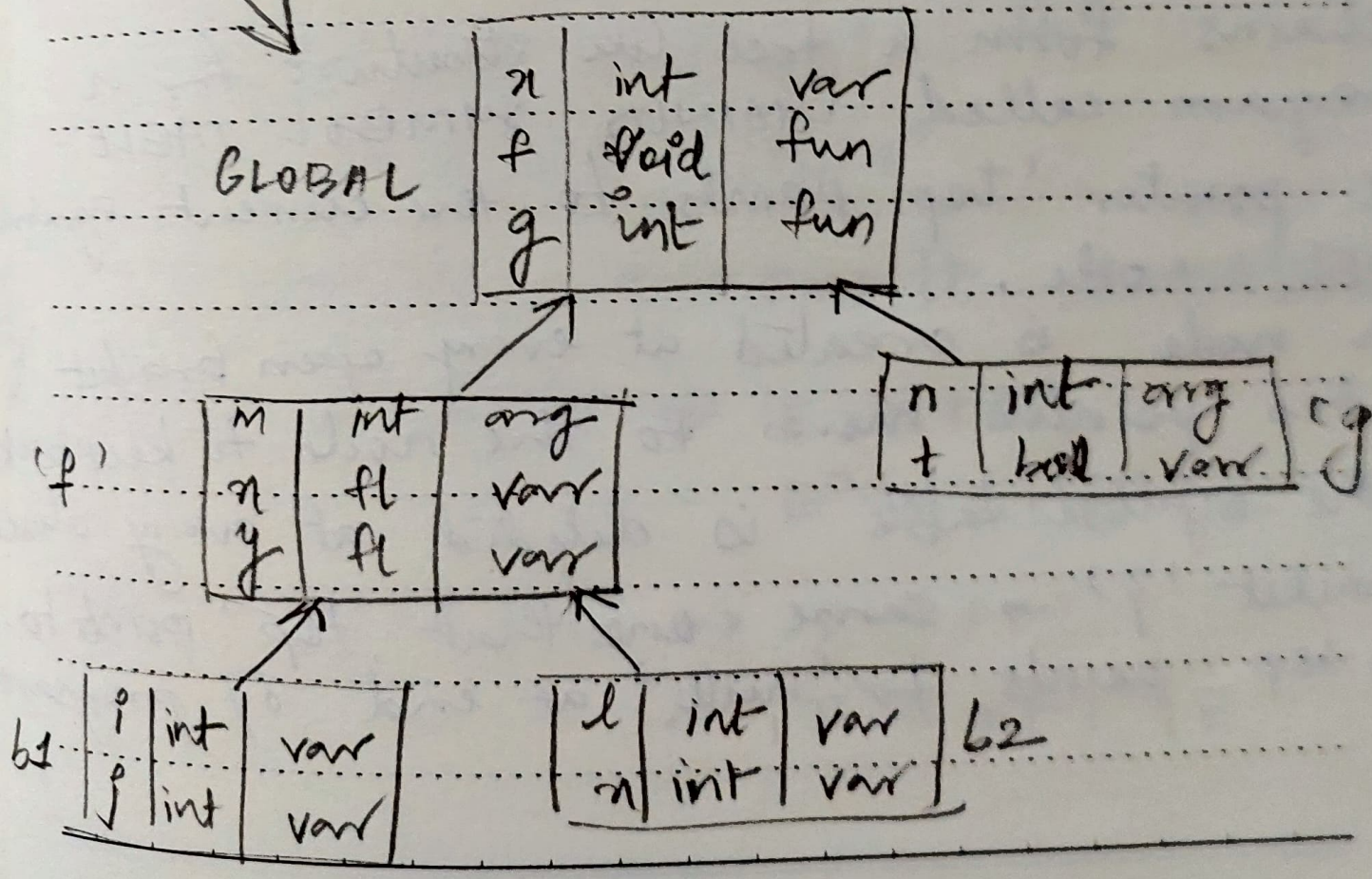
- find the declaration of a variable in a block by moving outward to the program enclosing block → FINDING SCOPE OF IDENTIFIER.
- separate symbol tables for each block in a source program → each table is chained together.
- chains form a tree like structure for a program called CHAINED SYMBOL TABLE.
- a pointer 'top' points to the current symbol table node.
- a node is created at every open bracket '{'
  ↳ top pointer moves to the node to keep track.
- the symbol table is deleted at every closed bracket '}' → same one that 'top' points to.
  ↳ top points to 'null' at end of program

Memo No.

Date

Example:

```
int x;
void f(int m) {
float x,y;
...
{ int i,j; ... ;}
{ int x,l ... ;}
}
int g(int n){
bool t;
...;
}
```



Mo Tu We Th Fr Sa Su

Date 29 / 4 / 24

CC

SYNTAX DIRECTED DEFINITION:

- SDD is a CFG + semantic rules & attributes
  (semantic rules → productions; attributes → grammar symbols)

q: if 'X' is a Symbol and 'a' is attribute
↳ X.a → value of 'a' @ 'X'.

Grammar:	Semantic Rules
E → E+T	E.val = E.val + T.val

predefined attribute 'val'

- Synthesized Attribute: takes values from its child nodes, i.e, value depends on child attributes

ex: A → BCD

∴ A.S = B.S
A.S = C.S
A.S = D.S + A.S

→ semantics

(A) → synthesized

- Inherited Attribute: takes values from parent or siblings or itself.

ex: A → BCD

C.i = B.i → left sibling
C.i = D.i → right sibling
C.i = A.i → parent

- terminals can have synthesized att, not inherited
- no rules in SDD to find value of terminal
↳ att. of terminals are lexical values given by lexical analyser.

Mo Tu We Th Fr Sa Su

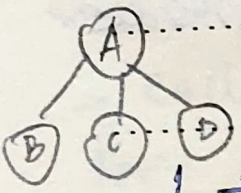
Memo No.

Date / /

- 'print' is a side effect of the grammar. → not calculating value of attribute.

eg: Expression Grammar.

GRAMMAR	SEMANTICS
F → digit	F.val = digit.lexval
L → E return	print(E.val)



↓ by analyser.

- Two types of SDD:

① S-attributed: contains only synthesized (LR parser) attributes

② L-attributed: contains syn. + restricted inherited attributes

* Evaluation of SDD:

- usually done in a bottom up fashion, using postorder traversal. (left-right-root)
- issues that can arise: cyclic dependency among inherited & synthesised attribute.

* Annotated Parse tree: parse tree that shows values of each node.

- dependency graph determines the eval. order of the semantic rules.

Memo No. _____

Date / /

Mo Tu We Th Fr Sa Su

↳ Steps to evaluate an input:

① draw parse tree → annotate it

② Annotation is done in bottom up fashion → compute terminal / leaf values then compute values of parent nodes till root node.

Q: grammar: Semantics:

$T \rightarrow FT'$ $T'.inh = F.val$ → inherited

$T.val = T'.syn$ → synth

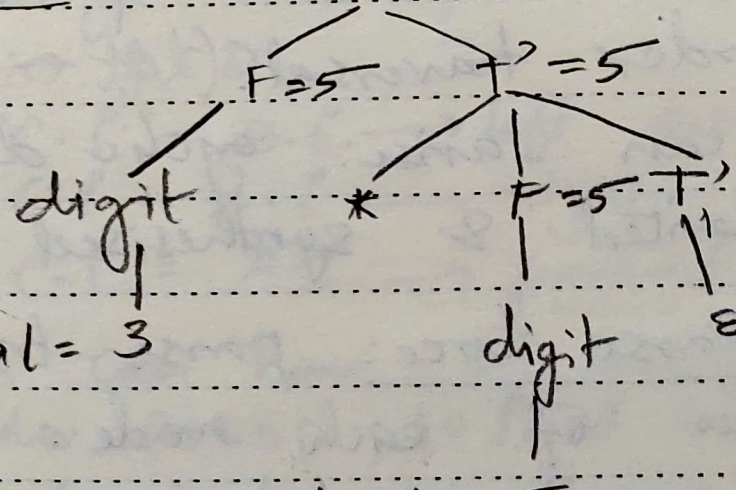
$T' \rightarrow *FT'_1$ $T'_1.inh = T'.inh \times F.val$ → inherited

$T'_1.syn = T'.syn$ → syn

$T' \rightarrow \varepsilon$ $T'.syn = T'.inh$ → synth

$F \rightarrow digit$ $F.val = digit.lexval$ → synth

input = 3*5. $T = 3*5 = 15$ → draw



directed graph / dependency graph

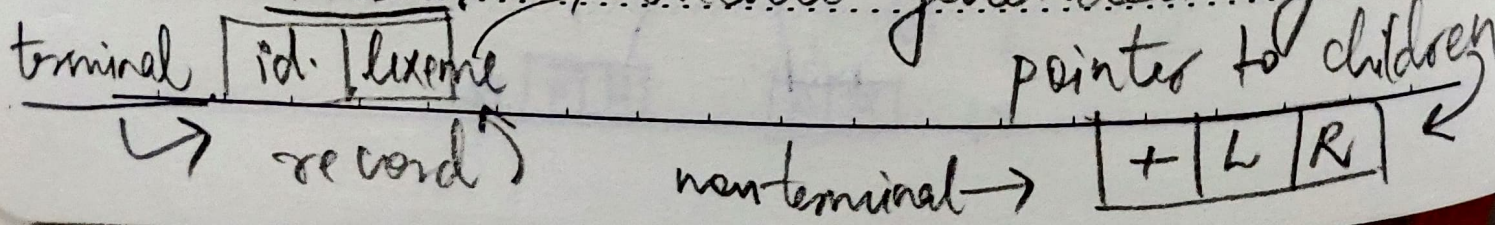
CC → SYNTAX DIRECTED TRANSLATION:

- CFG, along w/ semantic actions which can be present anywhere on the RHS of the rule
- evaluation is top-down → infix to postfix
- semantic actions are represented as individual nodes in the parse tree.
- print output / perform action during eval.

→ SYNTAX TREES:

- DAG: directed acyclic graph is a variat of syntax trees → used to identify common subexpressions.
- if a node 'N' in a DAG has more than one parent then 'N' is a common subexpression

NODE → lexeme generated by attribute



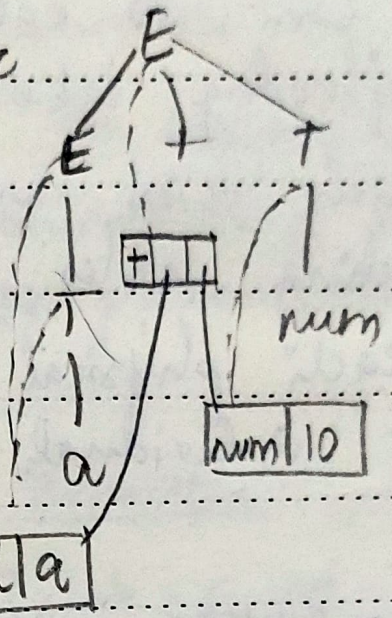
Mo Tu We Th Fr Sa Su Date / /

example: a + 10 (expr. grammar DAG)

→ how semantic rules are linked.

□ → nodes

— → syntax tree.



order of generation:

① leaf (id, a) = P1 → child nodes

② leaf (num, 10) = P2

③ Node (+, P1, P2) = P3 → 2 pointers.

→ L-attributed SDD req. dependency graph.

Q: a+a-10

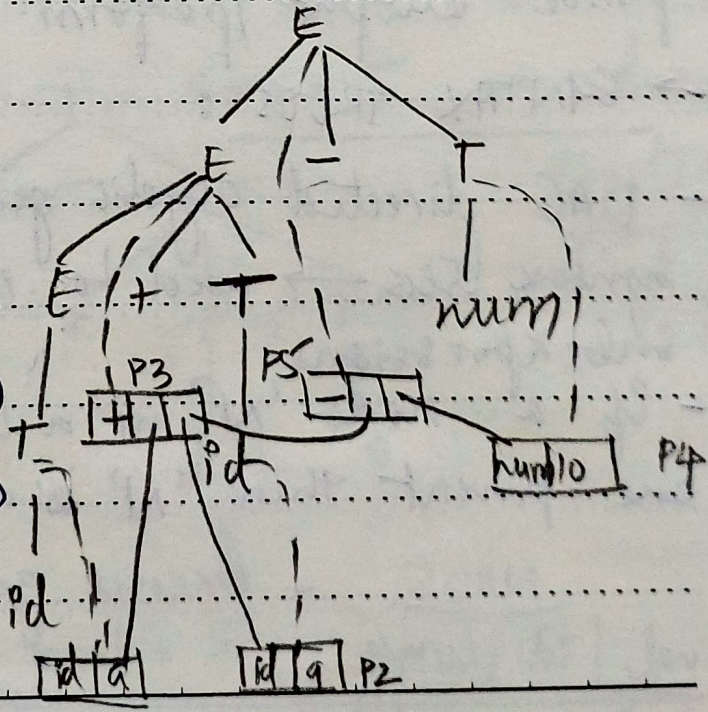
① P1 = leaf (id, a)

② P2 = leaf (id, a)

③ P3 = Node (+, P1, P2)

④ P4 = leaf (num, 10)

⑤ P5 = Node (-, P3, P4)

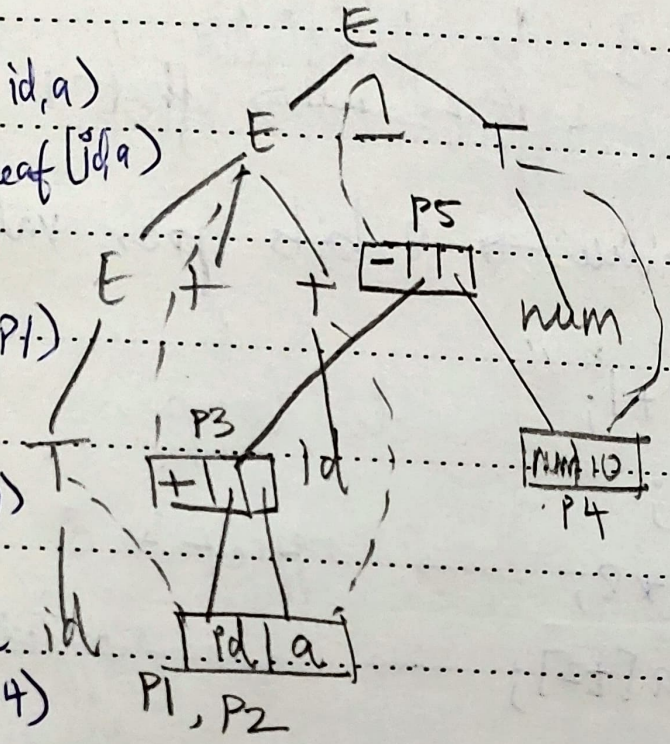


Mo Tu We Th Fr Sa Su

Memo No.

Date / /

DAG for input a+a-10:



① P1 = leaf(id, a)

② P2 = P1 = leaf(id, a)

③ P3 = Node (+, P1, P1)

④ P4 = leaf (num, 10)

⑤ P5 = Node (-, P3, P4)

* FORMS of ICG:

↳ Three Address Code

- atmost one operator on RHS
- it's the linearized representation of a Syntax tree

ex:

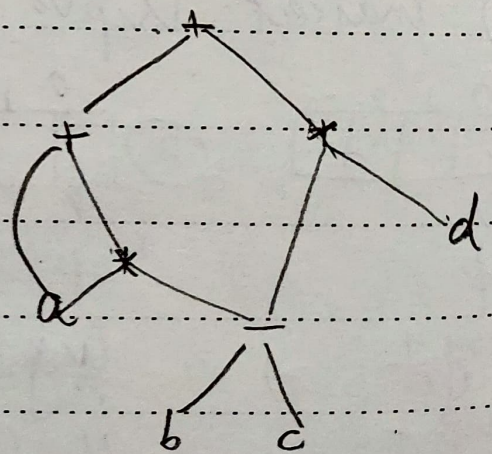
$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$



Memo No.

Date / /

Mo Tu We Th Fr Sa Su

Q. Consider: double a[10];

do i=i+1; while (a[i] < v);

index needs to be translated to the offset → offset + base add.

A: double value → 8 bits per value index.

L: t1 = i+1;
i = t1;
] do loop.

t2 = i * 8; → ~~offset * 8~~

t3 = a[t2]; → ~~base + offset~~ in t3

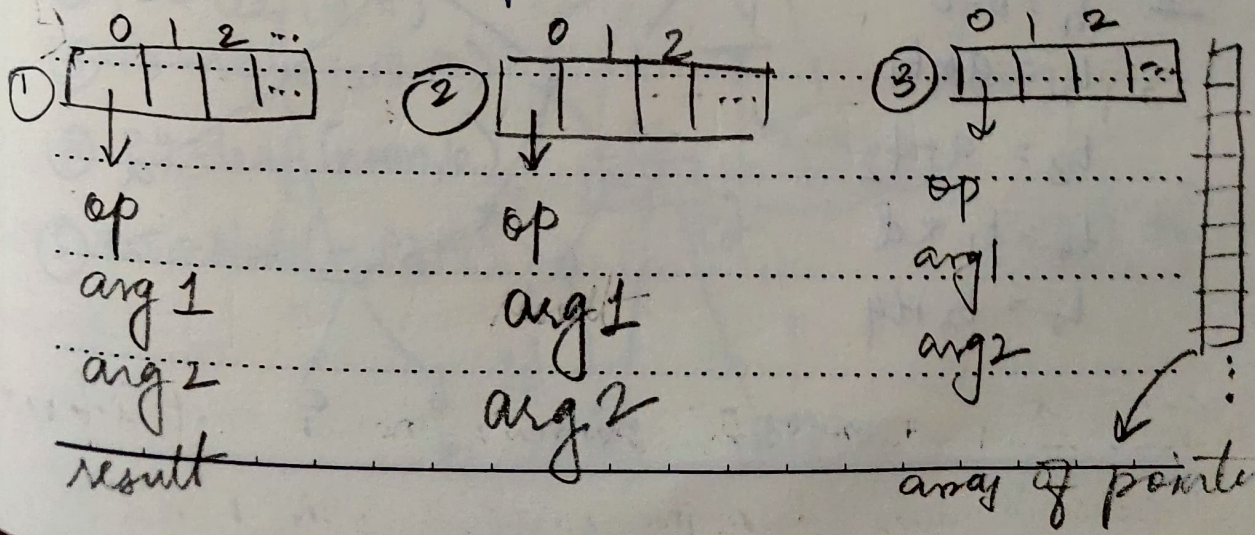
if t3 < v goto L → rel jump

- 3AC representations (array based):

① quadruples → 4 fields for each record.

② triples → 3 fields for each record.

③ indirect triples



Memo No.

Date / /

Mo Tu We Th Fr Sa Su

example: $a = b * -c + b * -c$ **QUADRUPLES**

unoptimized:

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

op	arg1	arg2	result
minus	c		t_1
*	b	t_1	t_2
minus	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a

- triples don't req. result → result marked by index of second.

TRIPLES

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

* reduces storage space
* copy/assignment op is done with result ↓ arg1 {quad → trip}

- quadruples are preferred by an optimizing compiler.

INDIRECT TRIPLES: an array of pointers to indicate & keep track of reordering of instructions in triples

Mo Tu We Th Fr Sa Su

Memo No.

Date / /

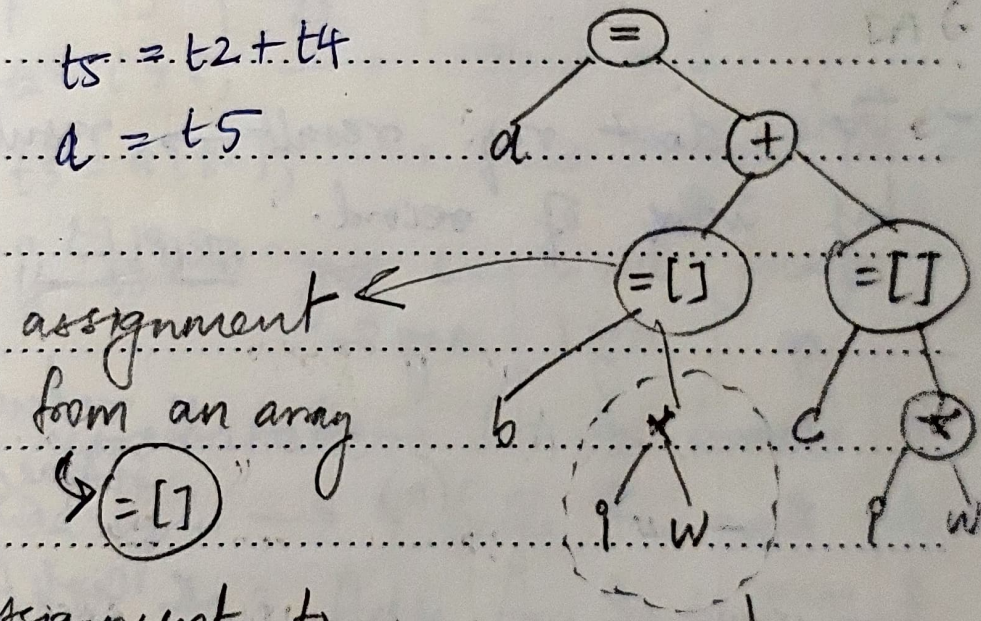
* Translating Arrays:

- base : address of 1st array location
- width : size of each array location

- ex:

3AC:

```
t1 = i * w      ---> multiplying index of width
t2 = b[t1]      ---> accessing element from array 'b'
t3 = i * w
t4 = c[t3]
t5 = t2 + t4
a = t5
```



assignment from an array → =[]

* assignment to an array → []=

offset

input ⇒ a = b[i] + c[i] → syntax tree

=[] ⇒ always has 2 children

base (array) & offset (index * width)

Memo No.

Date / /

	op	arg 1	arg 2	result
0	*	i	w	t1
1	+	b	t1	t2
2	=[]	t2		t3
3	*	i	w	t4
4	+	c	t4	t5
5	=[]	t5		t6
6	+	t3	t6	t7
7	=	t7		a

① QUADRUPLE FOR SAMPLE 3AC

① array assignment is split into 2 steps in quadruple → similar to syntax tree

↳ not the same as 3AC (more temp variables used)

	op	arg 1	arg 2
0	*	i	w
1	+	b	(0)
2	=[]	(1)	
3	*	i	w
4	+	c	(3)
5	=[]	(4)	
6	+	(2)	(5)
7	=	a	(6)

TRIPLE FROM SAMPLE QUADRUPLE

Memo No. _____

Date / /

eg: assignment to an array ([] =)

↳ always has 3 children

array, offset, RHS value

input ⇒ a[i] = b*c + b*d

t1 = b*c

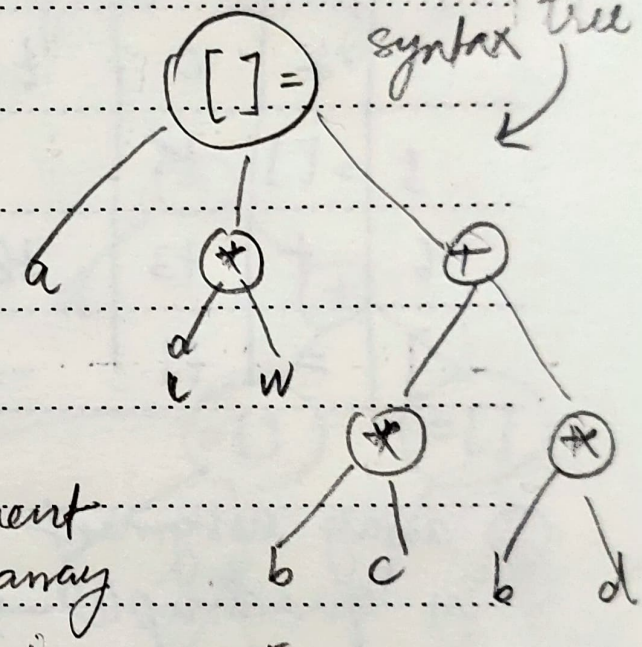
t2 = b*d

t3 = t1+t2

t4 = i*w

a[t4] = t3 → assignment to an array

offset is index



QUADRUPLE				TRIPLE			
op	arg1	arg2	res.		op	arg1	arg2
*	b	c	t1	0	*	b	c
*	b	d	t2	1	*	b	d
+	t1	t2	t3	2	+	(0)	(1)
*	i	w	t4	3	*	i	w
+	a	t4	t5	4	+	a	(3)
[]=	t3		t5	5	[]=	(4)	(2)

split into 2 steps w/ new temp variables.

Memo No. _____

Mo Tu We Th Fr Sa Su

Date / /

Ex. let a[2][3] be an int array & c, i, j are int.

3AC = c + a[i][j] {size = w = 4}

- considering row major order,

address = base + (i1 * w1 + i2 * w2)

= base + ((i * (3*4)) + (j * 4))

w1 = width of 1 row = n x w

w2 = width of 1 element = w → size of element.

op	arg1	arg2	res
*	i	12	t1
*	j	4	t2
+	t1	t2	t3
+	a	t3	t4
=[]	t4		t5
+	c	t5	t6

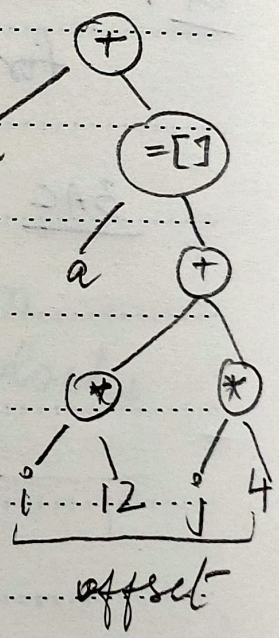
t1 = i * 12

t2 = j * 4

t4 = a[t3]

t3 = t1 + t2

t5 = c + t4



→ SHORT CIRCUIT CODE:

- jumping code, boolean expressions are converted to jumping statements (go to)
- redundent 'gotos' can be avoided using 'iffalse' → fall through helps w/ true cond.

Memo No.

Date / /

Mo Tu We Th Fr Sa Su

ex: while (a<b) { c=a+b; a=a+1; }

3AC:

L1: if false a<b L2

t1 = a + b

c = t1

t2 = a+1

a = t2

goto L1

L2:

ex: sum=0

for(i=0; i<n; i++;){ sum=sum+i; }

3AC: sum=0

i=0

L1: if false (i<n) L2

t1 = sum+i

sum = t1

i = i+1

goto L1

L2:

if x goto L

op	arg1	arg2	res
if	x	1	L
iff	x		L

	op	arg1	arg2
(0)	==	x	1
(1)	if	(0)	L
(0)	==	x	1
(1)	iff	(0)	L

iffalse x goto L

Mo Tu We Th Fr Sa Su

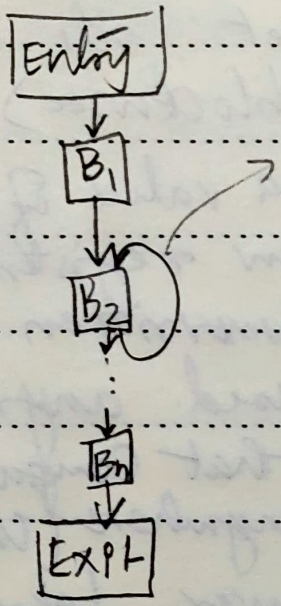
Memo No.

Date / /

→ CODE GENERATION:

- the intermediate code is partitioned into basic blocks → maximal sequences of 3AC instr.
- basic blocks become nodes of a flow graph
- R1 - the first instr of a block is the leader of the block → instr of 'goto' is a leader.
- R2 - instr following 'goto' is also a leader
  ↳ to form a block, find a leader, add instr to the block until another leader is encountered.

example:



$B_1, B_2, .. B_n$ ↳ basic blocks

- function "getReg(instr)" is used to get registers
- registers are scarce resources (limited)
  ↳ registers have reg. descriptors → which variable is stored in a particular register.
- every variable has an Address descriptor → stores ~~locations~~ of where the values of a variable then is.

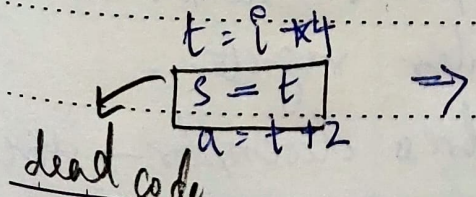
Memo No.

Date / /

Mo Tu We Th Fr Sa Su

ex: suppose a program has variable 'a',
↳ 'a' is in 'a' → denotion of location.

- for copy instructions (x=y), get Reg will always same register for both variables.
- while ending a basic block;
↳ forget about temporary variables
↳ if a variable is LIVE ON EXIT: needed outside blk
then value of variable is stored in memory
- machine level target code is ~~first~~ generated from 3AC (blockwise)
- to avoid reloading a value of a variable, store results in new registers.
- registers can be overwritten when a variable is not used anymore.
- DEAD CODE: instructions that compute a variable which isn't used anywhere (code in a branch that is never taken.)
↳ optimized code should contain no dead code
ex: copy propogation: use a variable as far as possible:



Memo No. _____

Mo Tu We Th Fr Sa Su

Date / /

- loop invariant: an expression whose value never changes despite multiple iterations.
  ↳ CODE MOTION: moving loop invariates outside to avoid extra computation.
- CONSTANT FOLDING: replacing values of constants during compile time
  ↳ ex: $u=3$, $v=u+w \Rightarrow v=3+w$
- code is optimised multiple times, repetitive steps can optimise code as much as possible.