COMPILER DESIGN NOTES

SEM-6

R.MADHULLIKA
106121071
CSE-A'25

Scanned with CamScanner

Lexical Analysis :

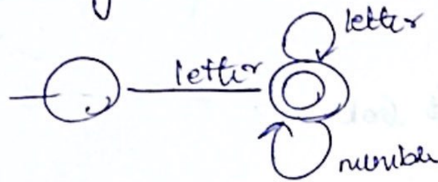
Scan the given input
line by line
char by char

Scan left to right & find words in the sentence

Valid statement : I is in CSPC62

acc to lex analyser a = b+c;

In terms of compiler, var is called an identifier

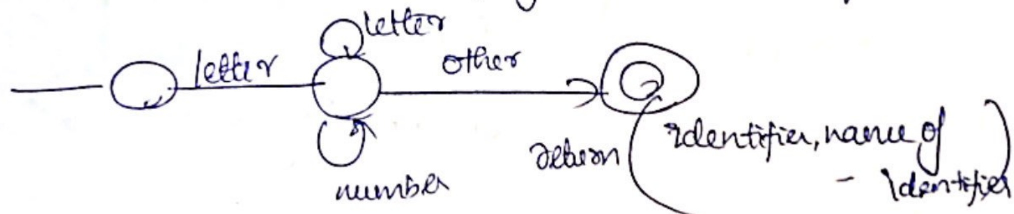


identifier :
letter (letter/number

When a letter/number is not scanned, due to white space or rel op then the analyser will stop and take var as everyth before it .

Doesnt decide on anythg as correct/incorrect → next phase role

Just scan, create the words & give it to next phase.



If not an identifier or a white space, it needs to be scanned and given a tokenization

the pointer needs to be moved back ↴
whatever scanned extra is given back to the input & scanned sequentially
↓
others

Scanned with CamScanner

Hence,
the unnecessary spaces will be removed & only information
info is passed to next phase

Syntax Analyser :

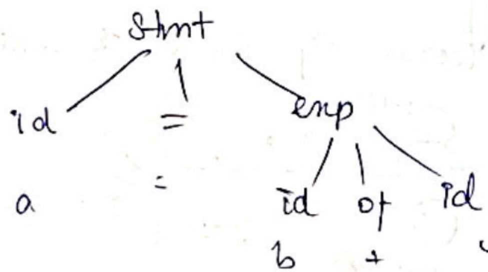
whether syntax is fine w.r.t the written language
format of the information

not look at the meaning, just check whether the
specifications are correct.

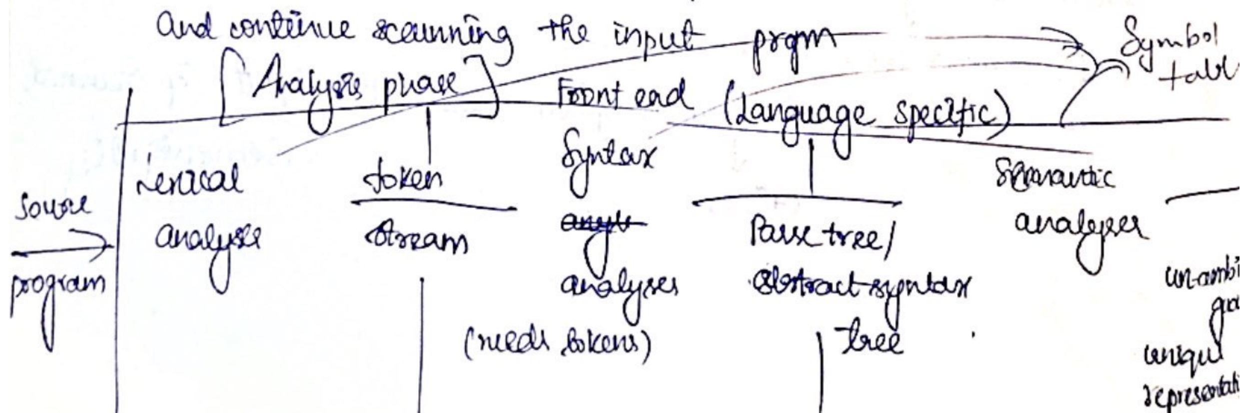
meaning - semantics × doesn't look
syntax - structure

parse tree - 3 line structure ~~exp → id of id~~ + id

Ex: stmt → id = exp
exp → id of id



When an error is detected, it reports or skips the error
and continue scanning the input prgm



Semantic analyser :

The type checking can be done but not all meanings

produce statement with only one interpretation without ambiguity

1. If c_1 then if c_2 then else c_3 → dangling else problem

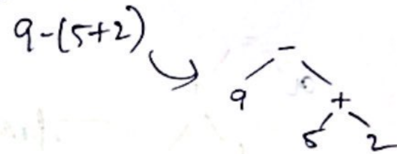
9 - 5 + 2

} ambiguous statements

9 - 5 * 2 → not ambiguous since * has higher precedence

but -, + has same precedence ∴ It can be (9-5)+2

9-(5+2)



the semantic analyser should be fed the correct representation

dead code - code part which will never be executed

Optimiser : (optional phase) → can be skipped

removes extra parts of the program

↓

dead code

common subexpression elimination - some expressions which are repeated

$X = Y+Z$

$W = Y+Z+K$

compiler sees Y+Z calculated twice

$W = X+Z$

code motion ← a variable which is of no use in the calculation of loop, the declaration goes out of loop by compiler

reduce the computation effort in the loop

copy propagation $x = 5*2 \rightarrow x = 10$

constant folding → whenever constant calculation comes up compiler puts the result into var

strength reduction

↓

$X = X * 2$

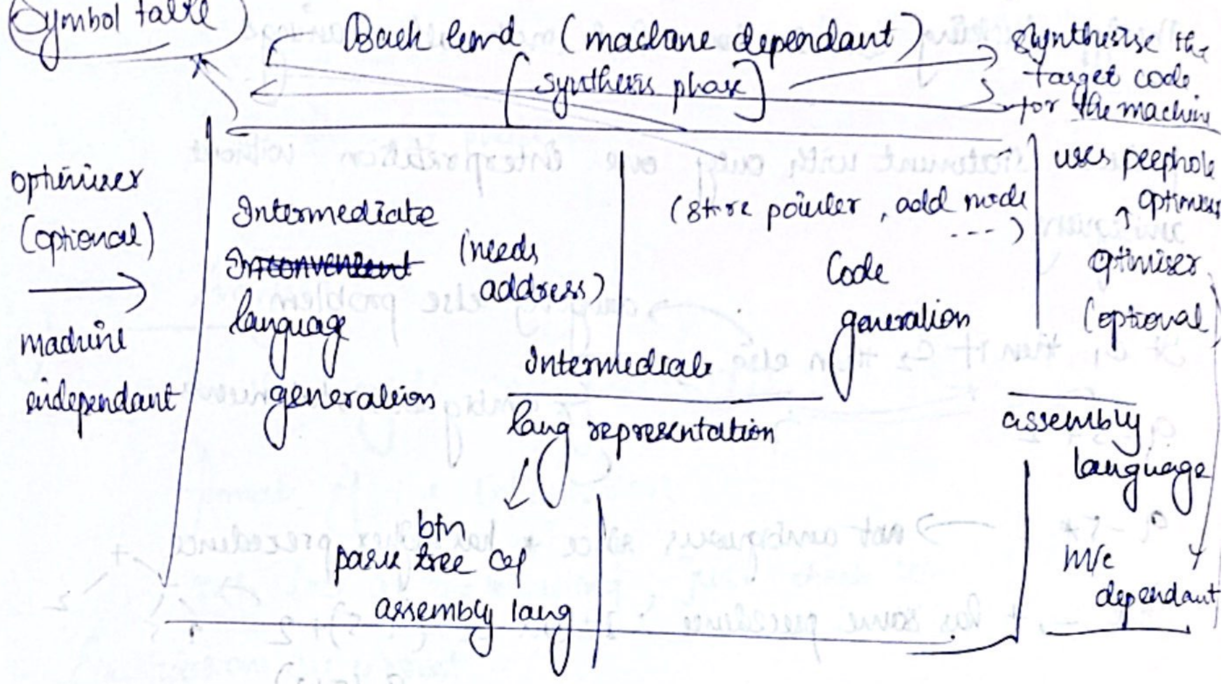
↳ costlier operation

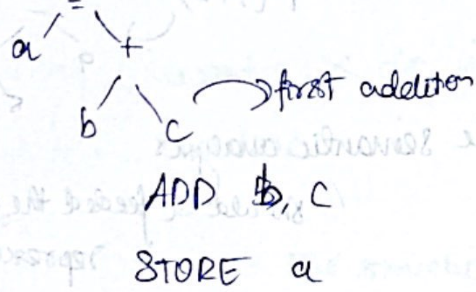
can be replaced by

X << 1

X+X

Scanned with CamScanner





postfix, prefix - stack notation are intermediate languages

It,

a[i] → [Add [a[0]) + i location].

redundancy occurs with repeated calculations
↓
can occur in code generation stage

one optimiser atleast is recommended

int count = 0;

<id, count>

The information produced by compiler needs to be stored
∵ the code can be executed only after passing other phases (comp, assembler)

Needs to be stored:

- Type of token
- name " "
- type information
- amount of storage needed,
- Address,
- pointer

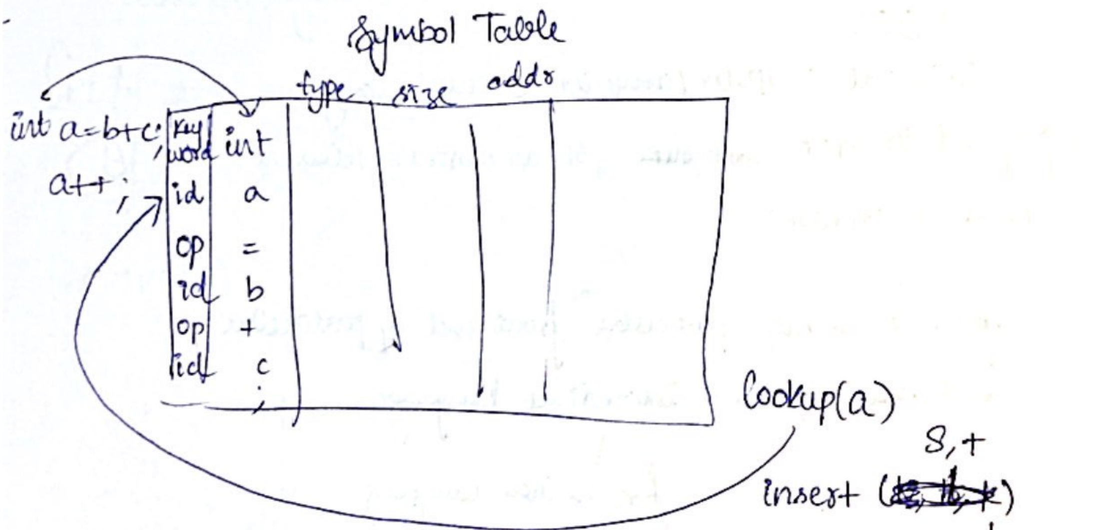
target language → code

(H.T, array)

Whenever a Data Structure is used to store, the problem is when a value is changed, it needs to be changed at all instances each time by the compiler

Hence a global table for all phases, where every phase can store its structure in the table

↓



location

token along w addr
token id " "
string " "

S - seq of char - a
t - token - id

compiler gives the line number of the error to debug the code

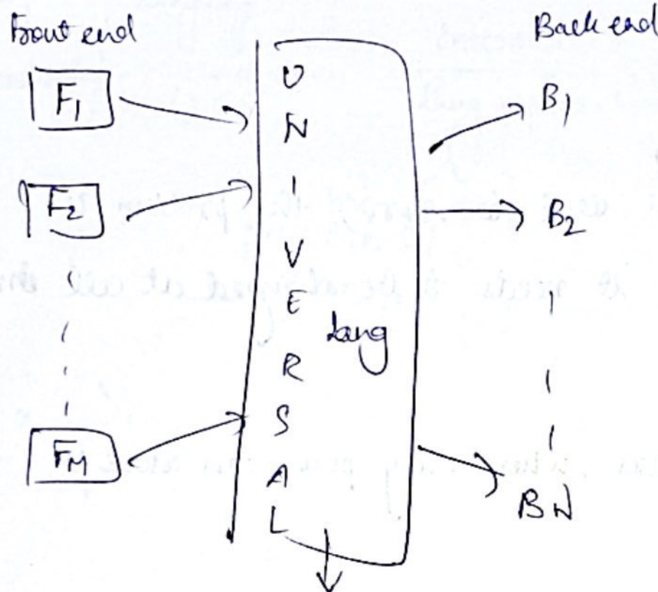
Many logical errors cannot be detected by compiler

Scanned with CamScanner

M * N

no. of prog lang * no. of architecture - not possible

" + "



UCOL

Universal computer/compiler oriented lang

proposed in 1961 ~~but dien~~, its an abstract idea and never materialized

- compiler to map particular front end of particular back end op has a intermediate language
  ↓
  for similar language

Test Suite: large no. of programs specifically written to ~~check~~ test the entire program
↓
costly operation

Lexical Analyser → creates stream of tokens

→ remove white space, comments etc

Token - unit of lex analyzer → keyword, identifier, num, op

Lexeme - corresponding sequence of characters for each token

Pattern - Rule that matches each lexeme to a token

every seq of char follows a pattern to identify which lexeme it belongs

If there is a white space the lexeme counting stops

The keywords are dictionaries kept in symbol table and hence not interpreted as identifier.

reserved keyword policy by preloading symbol table with the keywords.

[PL/I]

(if) then (then) then else (else) else then (if) if (then) then + 1

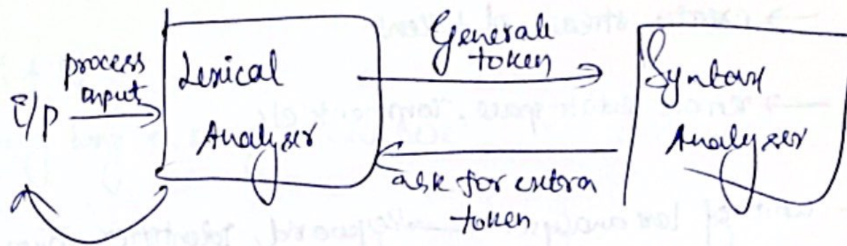
[FORTRAN]

DO 10 I = 1.25 ⟹ DO10I = 1.25 (1.25 → decimal)

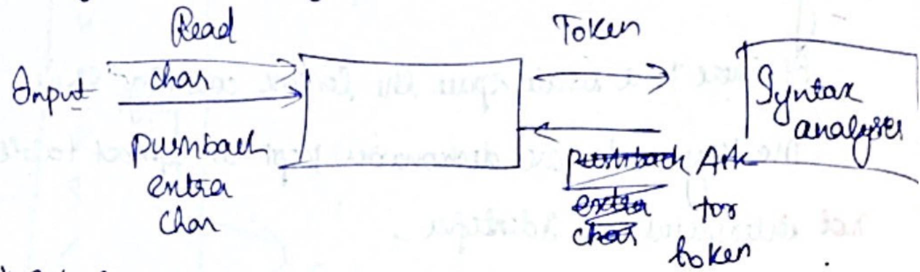
DO 10 I = 1, 25 ⟹ DO10I = 1,25

(DO → do; 10 → 10th line; 1, 25 → iterative)

Try and get the longest match of seq of character & the meaning is interpreted. - ~~contiguous~~ longest match policy followed by lex. Analyzer



30/1/24 Implementing Lex. Analyser :



Let

E = m * c * c

- input / output operation which requires disc access are costly

Hence A block of data is stored in a buffer.

The whole E = m * c * * 2 will be in the same block

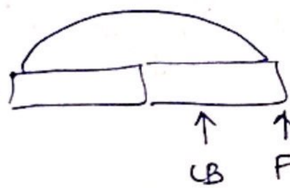
one var - Start of lexeme

another var - to indicate the end

Suppose the lexeme starts at the end of a buffer

Hence lookahead - req extra info
↓
another buf stores extra char

When buffer ends, we need to reload the buffer



Hence a circular loop is implemented

→ even then the lexeme is not decided btw

If there are more than 2 buffer of lexeme length, then this Strategy fails

Block size = 4096 bytes

We need to check:

1. whether the char is end of buffer
2. the char which ends the lexeme

2 io operations

∴ use a special char which marks the end of char

When a variable eof is loaded, it either marks the end of buf or the end of lexeme.

eof - either not need to do anytg or load a buffer

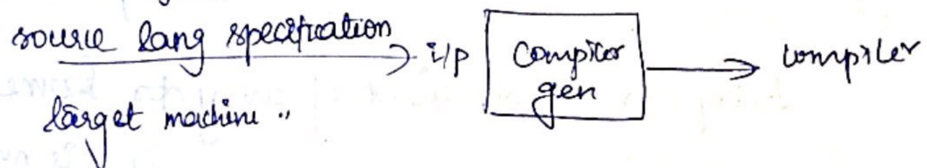
Implementation approaches

→ To write an assembly lang for the io

when to start, read, end ---

→ take 2 arrays, 2 pointers and implement in high level lang

→ Tools for compiler generator — Lex ⇒ Flex (Fast Lex)



Lex (Lex analyser generator)

YACC (Yet another compiler compiler)

Tools gives a particular structure to use it.

Target of any compiler is to move ~~any a~~ as much closer to assembly lang as possible

Scanned with CamScanner

Construct a lexical analyser

The way of specifying the specs is the challenge

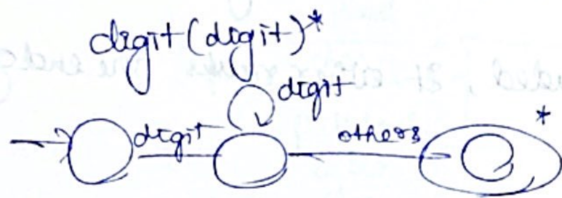
Returns tokens to the syntax analyser

A finite set of token is defined

The code given is trying to detect whether the lexeme is a number (digit)+

while(1) ⟶ Inf loop cos our lexeme can have any no. of char

digit (digit)*



Token value is returned as lexeme

tokenval - lexeme associated w token not the num

Symbol table:

Stores information of every token

Insert(s, t) :: save lexeme s and token t and return pointer

Lookup(s): return index of entry for lexeme s or 0 if s is not found

the structure of a symbol table should be

token	Value	type	Address
Id			
number			
keyword			
operator			
punctuation			

If there are m tokens, need $\log_2 m$ bits

↓ can take 4 bits

∴ 15 tokens can be represented

32 bytes - general max length of a variable

$(k+32) \times N$

K = max allowed size of the token

N = no of variables defined

(k+32) → size of 2 col

Ex: 4 tokens

		var size
00	id	32B
01	num	16 B
10	op	8 B
11	key word	32B

∴ the size is wasted

allocate memory dynamically

define what is a letter & digit

id = letter (letter/digit)* → letter = A|B| - - - |Z|a|b- - - |z

digit = 0|1| - - - |9

↓ defining a regular expression with a name - regular definitions

later we can replace the R.E with the name

Ex:

country → (digit)~

area → (digit$^+$)

exchange → (digit)^3

phone → (digit)^4

fax no. → country - area - exchange - phone

Ex: mail id

letter → a| b| - - -|z|A|B|- - -Z

name → (letter)$^+$

Address → name @name . nam

Floating point

digit → 0|1 - - - |9 e → empty string

digits → digit$^+$

optfraction → . digits | e

optexponent → (E(+|-|e) digits) | e

number → digits optfraction optexponent

123.45
123.45E
12E-3
0.012

? → optional

Use of shorthand notation

digit → 0-9

digits → digit$^+$

number → digits(. digits)? (E [+-] ? digits)?

Implementation of specification

1. Write RE for lexemes of each token

number → digit$^+$

Identifier → letter (letter (digit)*

2. Construct e matching all lexemes of all tokens

$R = R_1 + R_2 + R_3 + \cdots$

(Checking if any part of it matches with RE)

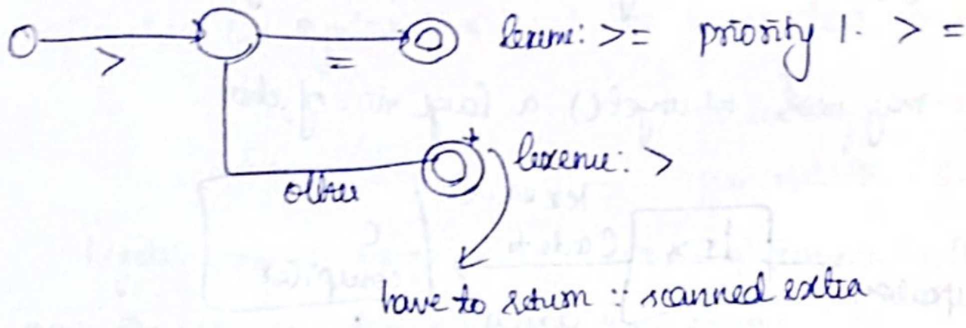
3. Let input be $x_1, \ldots x_n$

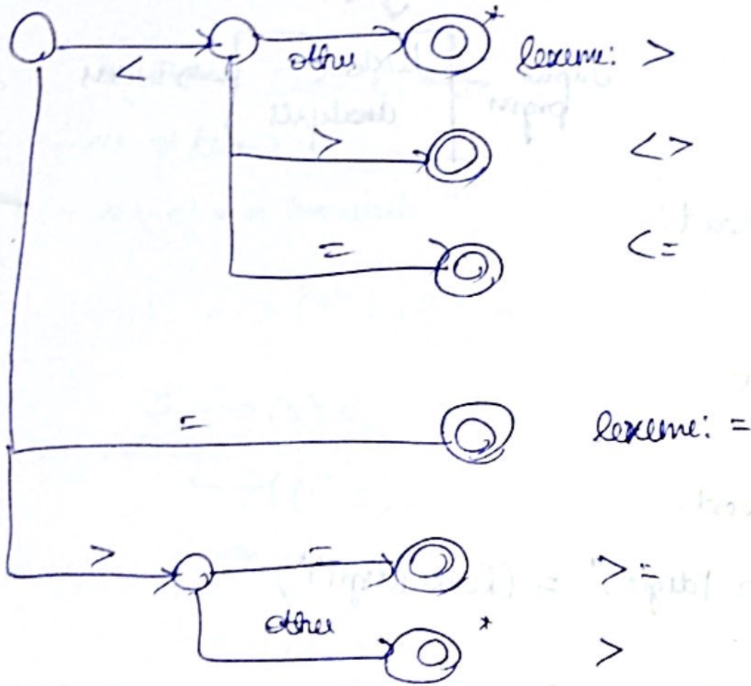
check $x_1 \ldots x_n \in L(R)$

4. $x_1 \ldots x$

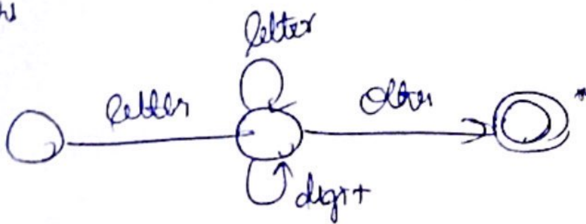
Relational operators

relop → < | <= | = | <> | >= | >





Identifiers



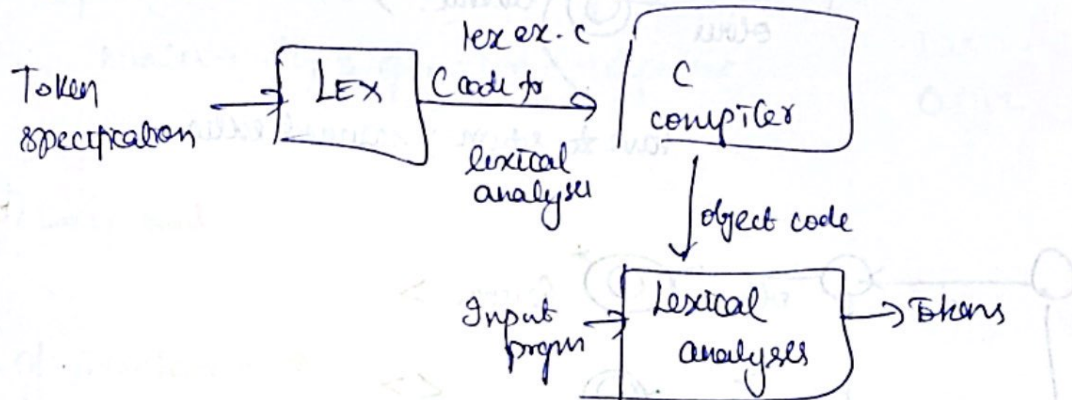
White spaces

Scanned with CamScanner

For each state, add switch cases

Complexity of transition diagram inc the error prone of code.

Tradeoff: may need to unget() a large no. of char



Implementing lookahead:

DO 10 I = 1.25

DO 10 I = 1, 25

Spec for DO as keyword:

DO / (letter | digit)* = (letter | digit)*,

Structure of LEX progm:

declaration

translation rule

auxiliary func

Translation rule

Pattern {Action}

installID() returns a pointer to symbol table placed in yyval

yytext: pointer to beginning of lexeme

yylen: length of lexeme

yyval - global variable

Scanned with CamScanner

SYNTAX ANALYSIS

- Task of this phase is to check the syntax of the grammar.
- To check the syntax, we need the parse tree of table drive PDA
  ↓
  to look up ∴ more powerful than the RE

Regular grammar cannot count, counting lemma is needed to count the opening, closing braces, indentation

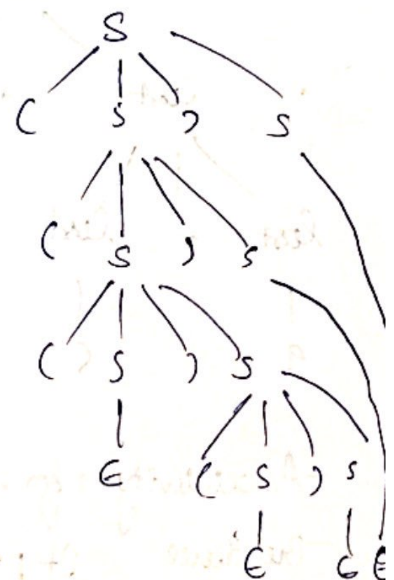
S - start symbol
T - set of terminals
NT - set of non terminals

$S \rightarrow (S)S \mid \epsilon$

((( )))

((()())) — sentential form

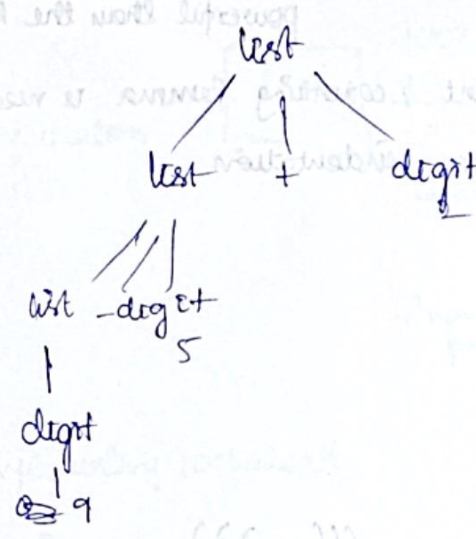
$S \rightarrow (S)S$
$\rightarrow ((S)S)S$
$\rightarrow (((S)S)S)S$
$\rightarrow ((()S)S)S$
$\rightarrow ((()(S)S)S)S$
$\rightarrow ((()()))$

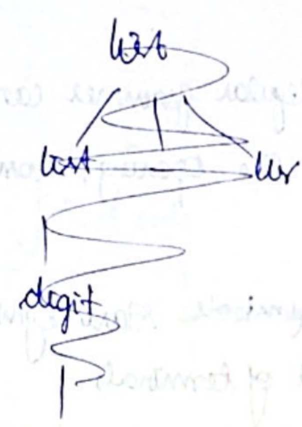


9 - 5 + 2 → parse tree for this using grammar 1 & 2

1. list → list + digit |
   list − digit |
   digit

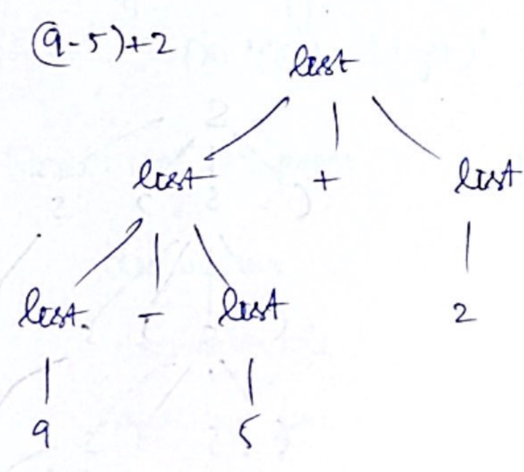
digit → 0 | - - - | 9



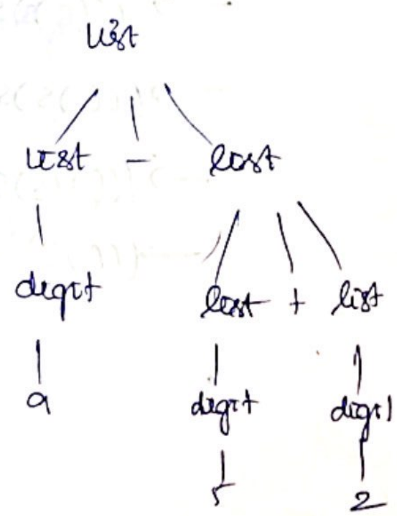


2. list → list + list | list − list |
   list | 0 | - - - | 9 |

(9-5)+2



9 − (5+2)



- Associativity : for every operator which values to be taken
- Precedence   If precedence are same then check for associativity

right associativity

left associativity also cannot be compared

9^5^2 → 9^(5^2) ✓

↓

(9^5)^2

Scanned with CamScanner

$S \rightarrow SaS|b|c|d|e$

$\rightarrow SaA$

$A \rightarrow b|c|d|e$

If $S \rightarrow b|c|d|e$ the A which is replacing S should produce b|c|d|e

list → list + list | list - list |

list 10 - 9

digit → [0 - 9]

Another ex:

$S \rightarrow SaS|b$ [Removing left recursive grammar]

$S \rightarrow SaA|b$

$A \rightarrow b$

Syntax Analyser:

WWR
DPDA X

CFG: $wA\gamma \rightarrow w\beta\gamma$

↑
string of terminals

$\alpha, \gamma, \beta \rightarrow$ string of terminals or non-terminal

If $A \rightarrow \beta$

In lex analyser generated string of tokens → terminals

$x = a_1 a_2 \ldots a_n$

w → empty string / string of terminals

$E \rightarrow E+E | E-E | id$

$E \rightarrow E + E$

$\rightarrow id + E$

$\rightarrow id + E - E$

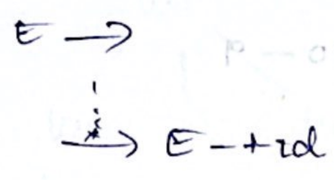
Scanned with CamScanner

In the parser, we check everytime whether the lexeme is in sentential form or not

If not valid then report error

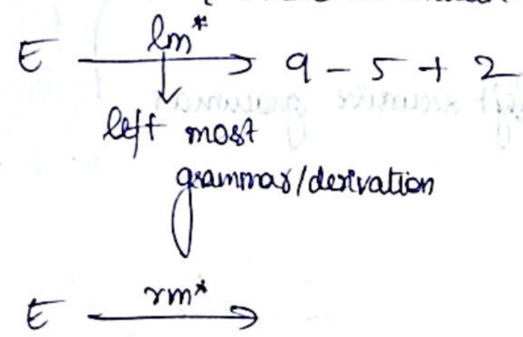
Panic mode: to complete whatever in the subsequent part of code

Skip one invalid part and take a valid sentence



The given grammar is ambiguous, any derivation has 2 parse trees

→ there are multi steps and not specified



> Note: revise
> NFA → DFA → min DFA

The compiler needs to remove the ambiguity

Top down parsing or Bottom up parsing generates diff parse trees

In a parse tree,

Start symbol - root

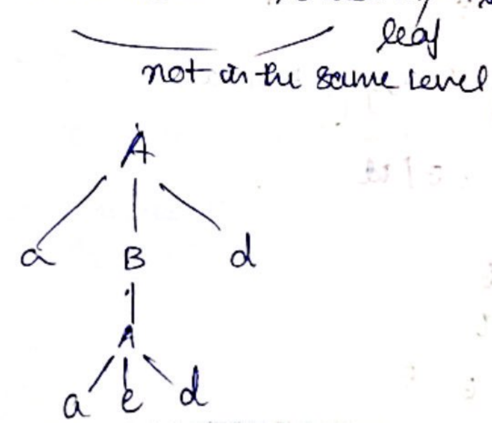
intermediate nodes - non terminals , children/leaf - terminals

not in the same level

B → A

A → aBd|e

aaedd



Scanned with CamScanner

Associativity & precedence

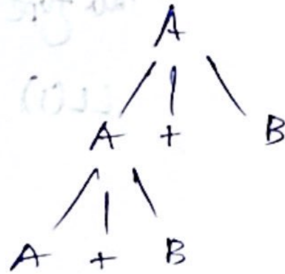
In the lex progrm, we define the ass & precedence

%% *, / left
%% +, - left

Lets take the grammar, E → E+E | E-E | id
E → E*E | E↑E

(9-5)+2 - left associative
9↑(5↑2) - right "

A → A+B
→ A+B+B
→ A+B+B+B
⋮



left heavy / associative

any operator
A → B∘A

Right heavy

needs to be left associative

E → E+A | E-A | id

9-5+2

F → id
F → E↑F | id → right heavy
T → T*F | F | T/F
E → E+T | E-T | T

our order of exec is ↑
go to nxt level
Introduce — *, /
F
+, -
↳ introduce N-T T
go to nex level

Writing 3 set of prod rules to indicate 3 levels of precedence / associativity

Scanned with CamScanner

The above is one way of removing ambiguity

Let 9*5 - 2

E → E - T

top down parsing ↓

→ T - T

→ T * F - T

→ F * F - T

~~→ id * id - id~~

→ id * F - T

→ id * id - T

→ id * id - F

Bottom up ↑

→ id * id - id

Lookahead :

Look ahead one symbol to decide on the prod rule.

This type of grammar is called LL(1)

Top-down parsing :

E → E+E | E*E | (E) | E↑E | id

We have 4 diff levels of precedence here

E → E+F | F

F → F*T | T

T → Q↑T | Q

Q → (E) | id

the tree goes deeper as we derive each level deeper

we start from left to right that

Hence deeper is given more precedence

This non-ambiguous grammar is Not suitable for top down parser

↓

have specific set of rules

Scanned with CamScanner

Left recursion

$A \to A\alpha \mid b$

Remove the left recursion

$A \to bA'$

$A' \to \alpha A' \mid \epsilon$

↓ to end the no. of α's

$A \to A\alpha$

$\to A\alpha\alpha$

$\to b\alpha \dots \alpha$

After removing left recursion,

$E \to FE'$

$E' \to +FE' \mid \epsilon$

$F \to TF'$

$F' \to *TF' \mid \epsilon$

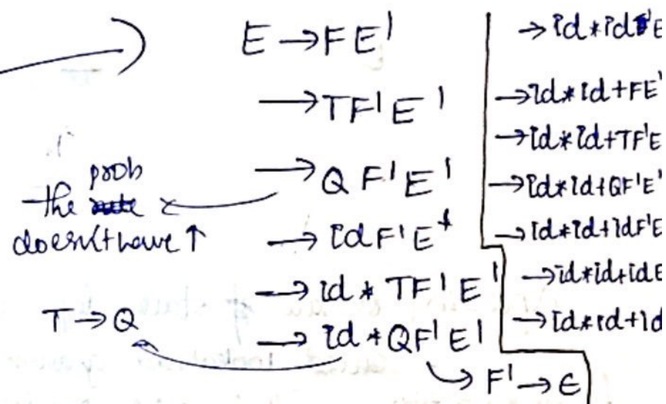
$T \to Q$

$Q \to id$

Let the,

$9 * 5 + 2$

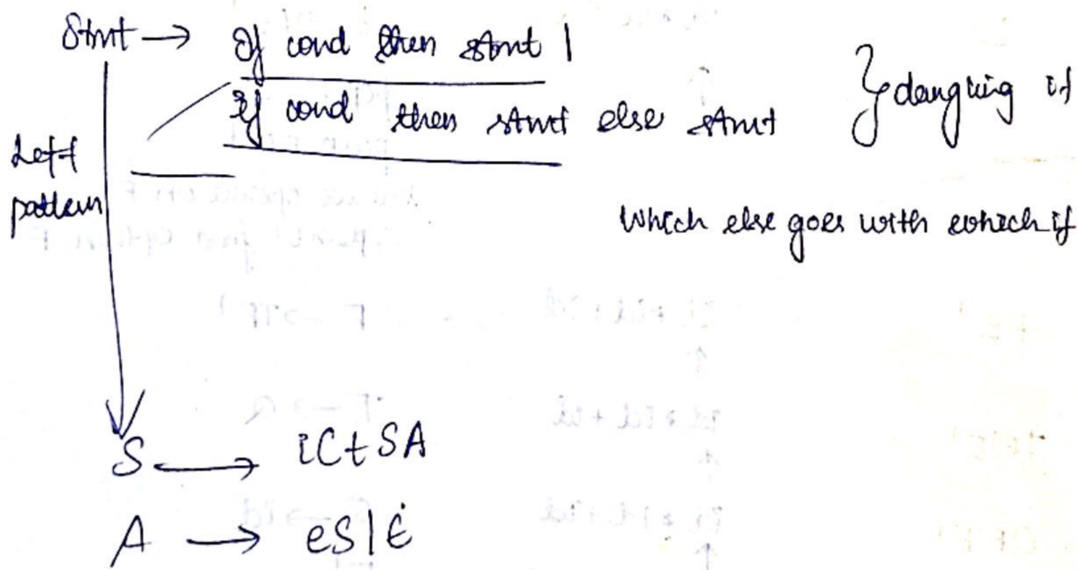
id * id + id



Stack	I/p	Action
E	id * id + id ↑	$E \to FE'$ pop E push FE' but we operate on F ∴ push E' first & then F
FE'	id * id + id ↑	$F \to TF'$
TF'E'	id * id + id ↑	$T \to Q$
QF'E'	id * id + id ↑	$Q \to id$
F'E'	* id + id ↑	$F' \to *TF'$

	Stack	I/P	Action
* is processed	TF'E'	Id + Id	T → Q
	QF'E'	Id + Id	Q → Id
	F'E'	+ Id	F' → ε
	E'	+ Id	E' → +FE'
+ is processed	FE'	Id	F → TF'
	TF'E'	Id	T → Q
	QF'E'	Id	Q → Id
	F'E'	—	F' → ε
	E'	—	E' → ε
	—	—	HALT

Depending on the stack top and the input, the compiler decides on action. called lookahead grammar (LL(1)) — left to right production with 1 position lookahead



Every then there is an associated else → matched stmt

Stmt → matched stmt | unmatched stmt

We removed ambiguity, now
apply the condition every else goes with immediate If

Matched stmt → If cond then matched stmt
else matched stmt | Others

unmatched stmt → If cond then matched stmt
else unmatched stmt |
If cond then stmt

↓
extra else
or
some If not
matched with any else

associate the else with the nearest If

From the top down parsing,
how to know the input has exhausted (I/O op are costly)
Use some marker to indicate the input has exhausted
also " " " stack " "

X
stack top

a is current element
↓
I/P symbol
(string of terminals)

If X = a (a terminal)
Just pop X and move read pointer to the next symbol

a = $ X = $
↳ E operation performed

stack input
*TF' *
We directly skip the * but it takes one more step, we push * into stack and now stack top & symbol are equal then we move the read pointer.

a is a terminal → pop stack top & progress I/P pointer
↳ X = a
F → *TF' (pop)

X is a non-terminal [X, a] ... (apply ... rule which can give N.T)

a is always a terminal $\Rightarrow$

but x can be non terminal

then apply the rule and reach to the id (terminal)

Recursive descent parsing } ANTLR } tools to generate recursive descent parsing

→ backtracking

(not a good approach)

→ Predictive parsing

predict depending on the input to be processed

↳ Lookahead

LL(K) — Generally K=1

↓

no. of lookahead symbols needed to decide

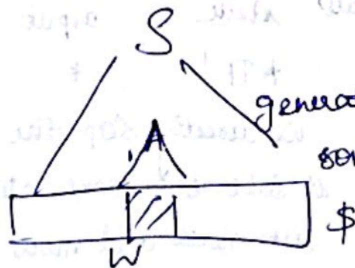
no ambiguity

Generate the parse table for every possible state and select the correct

First set } guide us on which prod rule to use
Follow set

↳ end market following the sentence should be reached (for start symbol)

First set:



generate a string w

some NT 'A' which produced a part of

1. Follow (S) contains $

2. $A \rightarrow \alpha B \beta$

Follow(B) contains first (β)

~~First set~~:

If we apply A, what are the possible values → first(A)
rules on A.

~~Follow set~~:

After applying all prod ~~rules~~ what follows A gives follow set

$E \to TE'$

$E' \to +TE' | \epsilon$

$T \to FT'$

$T' \to *FT' | \epsilon$

$F \to (E) | id$

First(E) = First(T) = First(F) = { (, id }

E is our start symbol ∴ any derivation of any N.T derives an input with (or id

First(E') = { +, ε }

Note:

If ε is the first symbol, then the next symbol acts as first

$X \to X_1 X_2 \cdots X_n$ ⇒ First(X) ≠ ε = X_2

$X_1 \to \epsilon$

If all $X_1 X_2 \cdots X_n \to \epsilon$ then only first(X) = ε

First(T') = { *, ε }

3. $A \to \alpha B \beta$ & $\beta \to \epsilon$

β is empty ∴

A ends with B whatever follow A will follow B

Follow(B) contains Follow(A)

4. $A \to \alpha B$

Follow(B) contains Follow(A)

Scanned with CamScanner

13/2/24

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid -TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid /FT' \mid \epsilon$

$F \rightarrow X \uparrow F \mid X$

$X \rightarrow (E) \mid id$

$F \rightarrow XQ$

$Q \rightarrow \uparrow F \mid \epsilon$

Remove left ~~recursion~~ factoring

Left factoring

$A \rightarrow \alpha B \beta \mid \alpha \gamma$

A takes decides the next symbol to find the prod rule

$A \rightarrow \alpha C$

$C \rightarrow B\beta \mid \gamma$

Need to delay the X to get the next symbol ↑

$F \rightarrow XQ$

$Q \rightarrow \uparrow F \mid \epsilon$

Q is following F, F also follows Q
↓
recursion

First(E) = First(T)
= First(F)
= First(X)
= {(, id}

First(E') = {+, −, ϵ}

First(T') = {*, /, ϵ}

First(Q) = {↑, ϵ}

1. Follow(S) contains $

2. $A \to \alpha B \gamma$ → Follow(B) contains First(γ) except ϵ

3. $A \to \alpha B \gamma$, $\gamma \to \epsilon$ → Follow(B) contains Follow(A)

4. $A \to \alpha B$ ↗

Followed by

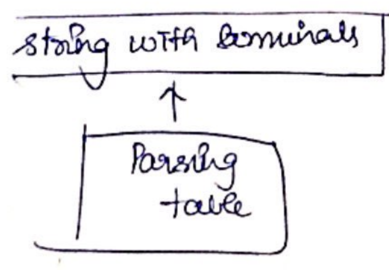
	E	E'	T	T'	F	X	Q
1.	$		~~+~~		~~*~~	$	~~$~~
2.	)		+, -		*, /	↑	
3.			$,)		+, -, $,)	*, /, +, -, $,)	
4.		$,)		+, -, $,)			*, /, +, -, $,)

$X \to \alpha$

For all $a \in$ First(α)

$[X, a] \leftarrow X \to \alpha$

the current input sees parsing table and decide which prod rule to apply



stack ↓ the input start needed to be process ⟶

↑	id	+	-	*	/	↑	(	)	$
E	E→TE'						E→TE'		
E'		E'→+TE'	F'→-TE'					E'→ε	E'→ε
T	T→FT'						T→FT'		
T'		T'→ε	T'→ε	T'→*FT'	T'→/FT'			T'→ε	T'→ε
F	F→XQ						F→XQ		
Q		Q→ε	Q→ε	Q→ε	Q→ε	Q→↑F		Q→ε	Q→ε
X	X→id						X→(E)		

Parsing table

input in the follow set of the stack top then apply ε ri

If the stack top is A

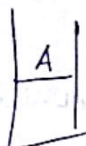
and we need to apply $A \to \epsilon$ to remove it and process follow(A)

$X \to \epsilon$

for all b in Follow(X)

$[X, b] \leftarrow X \to \epsilon$

Ex $A \to BCD$ $C \to aD$



the input needed to be processed is a

$B \to \epsilon$

Scanned with CamScanner

→ Many spaces are empty

What happens if the stack top is that cp when the input process is that, how to recover from the state

Stack	I/p (→ processing)	O/p
$E (stack top)	id + id * id $	E → TE' (put in rev order in stack to keep T on top)
$ E'T	id + id * id $	T → FT'
$ E'T'F	id + id * id $	F → XQ
$ E'T'QX	id + id * id $	X → id
$ E'T'Q id	id + id * id $	pop id, increase read pointer
$ E'T'Q	+ id * id $	Q → ε
$ E'T'	+ id * id $	T' → ε
$ E'	+ id * id $	E' → + TE'
$ ~~E'T~~ E'T+	+ id * id $	pop +, inc read pt
$ ~~TE~~ E'T	id * id $	T → FT'
$ E'T'F	id * id $	F → XQ
$ E'T'QX	id * id $	X → id
$ E'T'Q id	id * id $	pop id
$ E'T'Q	* id $	Q → ε
$ E'T'	* id $	T' → * FT'
$ E'T'F*	* id $	pop *
$ E'T' F	id $	F → XQ
$ E'T' ~~id~~ QX	id $	~~pop id~~ X → id

Stack	i/p	o/p
$E'T'Qid	id$	pop id
$E'T'Q	$	Q→ε
$E'T'	$	T'→ε
$E'	$	E'→ε
$	$	HALT Accept

One approach :

If an invalid input is processed s.t ∃ no prod rule for the symbol

then skip the stack symbols i.e from the input

till we reach the follow (stack top) is reached

pop

skip stack top

unless follow (stack top) is in the i/p under read ptr.

Stack	i/p	o/p
E	$	pop E

Synch – an entry that reports the compilation error

Scanned with CamScanner

Ex error case : id + id * + id

$ E'T'F*	*+id $	pop *
$ E'T'(F)	+id $	Synch
$ E'T'	+id $	T' → ε
$ E'	+id $	E' → +TE'
$ E'T+	+id$	~~E' → +TE'~~ process T
$ E'T	id$	~~id+id*+id~~ T → FT'
$ E'T'F	id$	F → XQ
$ E'T'QX	id$	X → id
$ E'T'Q id	id$	process id

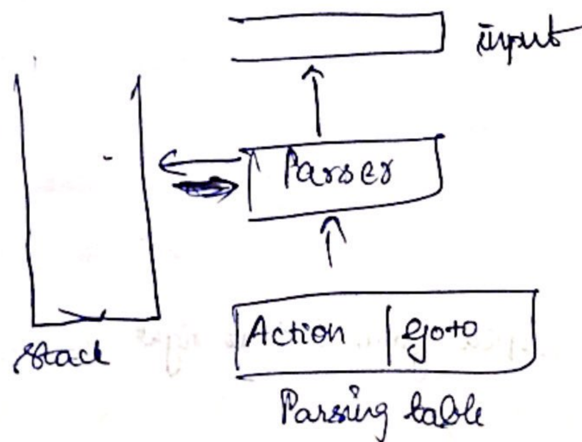
Bottom up parsing :

Start with the leaves, goes up one level, to the root

- Shift reduce parsing :
- Operator precedence parsing

Shift reduce parsing :

One type of bottom up parsing where 2 actions are there : shift, reduce



→ Terminals / tokens

→ Non terminals

→ State of parser

Parsing table of BUP :

Decide whether which action to take place or whether to go-to

Shift:

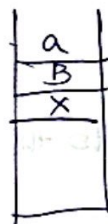
the input under read pt will be pushed inside the stack.

~~A→aBX~~

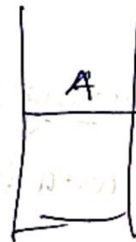
Reduce : (doing rightmost derivation in reverse order) — BUP

If A → aBX

and stack :



pop entire aBX , push A



Apply the prod rule on the input (or reduce it to start symbol)
- reduction

Ex:

S → AcBe

A → Ab | b

B → d

Input : abbcde

We always scan the input from left to right

abbcde
↑

→ aAbcde

Scanned with CamScanner

If we put one more A → b

→ aAAcde ✗ We are stuck

→ aAbcde

→ aAcde 1 shift

→ aAcde 2 shift

→ aAcBe

→ S

rightmost derivation.

S → aAcBe

→ aAcde

→ aAbcde

→ abbcde

We need a marker to specify which part of input we can reduce

Handle :

~~the string~~ which if reduced will lead one step towards the start symbol

abbcde

↓ a handle ↳ not a handle

Conditions :

→ ~~rightmost part of~~ Has to match the righthand side of a production

→ ~~It has to~~ The derivation of $A \to \beta$ should lead us one step towards the start . (if the input at rd pt is β)

Scanned with CamScanner

Process of finding handle of reducing it
↳ Handle pruning

Read marker : · [Introduced over the input]

right of · → whatever not read

·abbcde

The dot moves one step at a time and a production rule is applied after passing the symbol

Shift	·abbcde
Shift	a·bbcde
reduce ~~Shift~~	ab·bcde (handle)
shift ~~reduce~~	aA·bcde
reduce ~~Shift~~	aAb·cde
shift ~~Reduce~~	aA·cde
shift ~~Shift~~	aAc·de
reduce ~~Shift~~	aAcd·e (handle)
shift ~~Reduce~~	aAcB·e
reduce ~~Shift~~	aAcBe·
Accept ~~Reduce~~	S

} What is immediately right of · can only be shifted

[Shift αb·β
Reduce αA·β]

$A \rightarrow Ab$

$\alpha \Rightarrow \gamma a$

$\alpha \cdot b\beta$

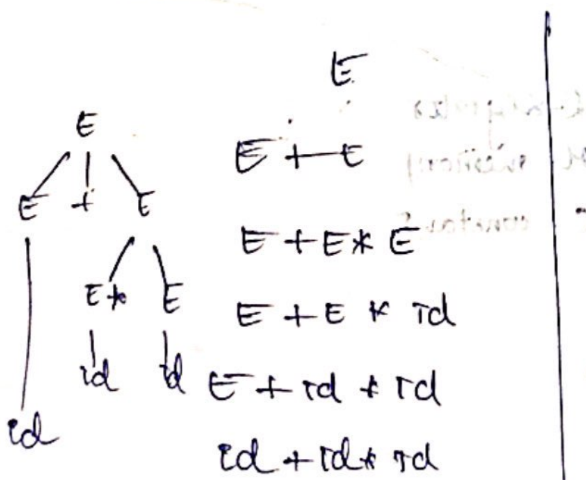
$\alpha b \cdot \beta$

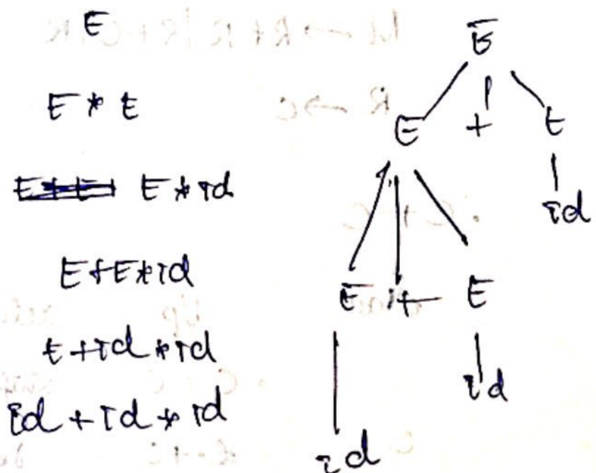
$\gamma A \cdot \beta$

$E \rightarrow E+E \mid E*E \mid id$ $\quad$ id + id * id

Action	id + id * id	
Shift	· id + id * id	
reduce	id · + id * id	$E \rightarrow id$
Shift	E · + id * id	
Shift	E + · id * id	
Reduce	E + id · * id	$E \rightarrow id$
reduce	E + E · * id	$E \rightarrow E+E$
Shift	E · * id	
Shift	E * · id	
Reduce	E * id ·	$E \rightarrow id$
Reduce	E * E ·	$E \rightarrow E*E$
Reduce	E ·	
Accept		

→ a situation where we can reduce E+E to E or to shift more or convert id → E. This situation always persists in amb grammar





Another derivation from ambiguous handle

Shift E+E . id

Shift E+E * . id

reduce E+E * id .

reduce E+E * E.

reduce E+E.

Accept E.

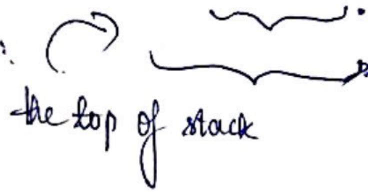
LR(K) grammar – bottom up parsing

L → left to right — more powerful & expressive

R → rightmost derivation

K → no. of lookahead symbols

A choice of applying production to a set of handles



the top of stack — we can either reduce the small set or the whole from the top of stack

Reduced

M → R+R | R+C | R

R → C

R - register

M – memory

C – constant

.C+C

Stack	I/P	action
	. C+C	shift
C	C. +C	reduce R → C
R	. +C	shift
R+	. C	shift

Scanned with CamScanner

R+C	reduce M → R+C
M	accept

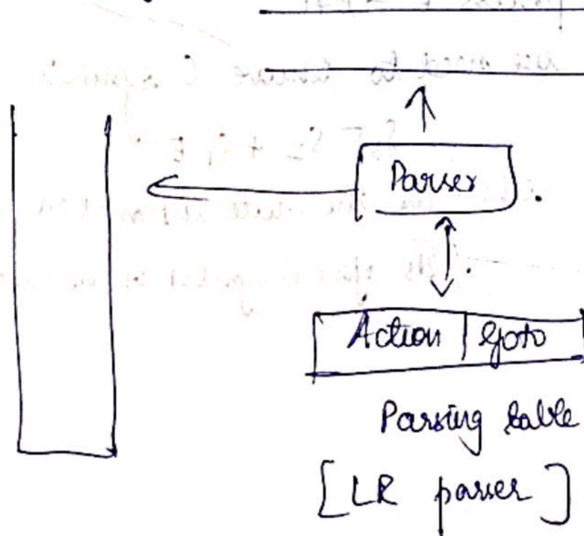
Another method

Stack	~~action~~ I/P	action
	·C+C	~~Reduce~~ Shift
.C	·+C	Reduce
R	·+C	Shift
R+	·C	Shift
R+C		Reduce R → C
R+R		Reduce M → R+R
M		accept

This is called reduced-reduced grammar
where ~~a part of~~ 2 handles and one handle is subset of other.

Which prod rule to select?

have to change the grammar to remove ambiguity



$S_0 X_1 S_1 X_2 \quad S_{n-1} X_n S_n$

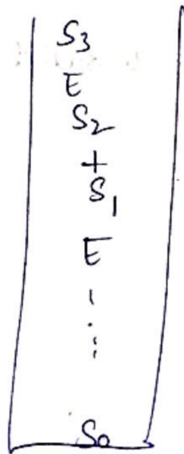
S → state symbols
→ status of the parser

Depending on state, decides when the handle has exhausted
" " " is on top of stack

location of read pt.
the part where the . has crossed → state

Store the information of whatever has been passed and whatever now in the top of stack is an handle.

S_0 start state – signifies . is at beginning
S_n final state – S.



no. of symbols to be removed = twice the no. of symbols at the ~~top of stack~~ rique of input

S – gives the state information

∴ to process E → E+C
we need to remove 6 symbols
$S_3 E S_2 + S_1 E$
which are the state information (ep
finds after 6 symbol it needs to be popped

10/12/24 Parsing table

State	id	+	*	(	)	$	E	T	F
	Action - terminals						Goto - NT		
0	S5			S4			1	2	3
1		S6				acc			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6							
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

E → E+T ——— ①

E → T ——— ②

T → T*F ——— ③

T → F ——— ④

F → id ——— ⑤

F → (E) ——— ⑥

1, 2, 3 — state symbol

S - shift

r - ~~state~~ reduce

Whenever reducing goto goto part.

id + id * id

Stack	I/P	Action
0	id + id * id $	Shift 5
0 id 5	+ id * id $	Reduce F → id
0 F 3	+ id * id $	Reduce T → F
0 T 2	+ id * id $	Reduce E → T
0 E 1	+ id * id $	~~Reduce~~ Shift 6

(margin notes: 0/3 F; r/2 T; 2 0 H E/T)

Shift 6 - S_6 - shift a, move to 6

0 E 1 + 6	id * id $	Shift 5
0 E 1 + 6 id 5	* id $	Reduce F → id
E+F · 0 E + 6 F 3	* id $	Reduce T → F
E+T · 0 E + 6 T 9	* id $	Shift 7
E+T* · 0 E + 6 T 9 * 7	id $	Shift 5
E+T*id · 0 E + 6 T 9 * 7 id 5	$	Reduce F → id
0 E 1 + 6 T 9 * 7 F 10	$	Reduce T → T * F (3 × 2 = 6 pop)
0 E 1 + 6 T 9	$	Reduce E → E + T
0 E 1	$	accept

E → E + T
→ E + T * F
→ E + T * id
→ E + F * id
→ E + id * id
→ T + id * id
→ F + id * id
→ id + id * id

How to build up the state 0 → 11

Viable prefix = set of valid symbols for the grammar

For α to be valid prefix:

αw :- sentential form

α · w a configuration of the SR parser

w is a string of terminals

Scanned with CamScanner

~~W # a~~

LR(0) Item - simple LR(SLR) parser

$A \to XYZ$

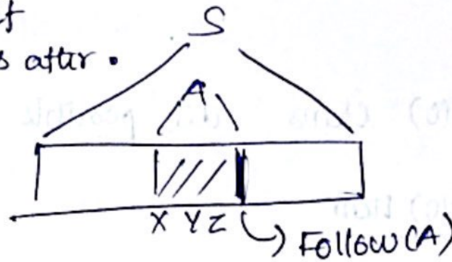
$A \to \cdot XYZ$

$A \to X \cdot YZ$ — has processed Y, now has to move to process Z

$A \to XY \cdot Z$

$A \to XYZ \cdot$ Reduction if Follow(A) is after.

You cannot reduce before follow(A) is after.



$G = (N, T, S, P)$

⇓ Augmented grammar

$G' = (N \cup S', T, S', P \cup S' \to S)$

Add $S' \to E$ — (0) to the prev grammar

Closure of LR(0) item

$A \to \alpha \cdot B\beta$ $B \to xyz$

$B \to \cdot xyz$

Ex:

$S' \to E$

2 possible LR(0) item

$S' \to \cdot E$

$S' \to E \cdot$

This whole derivation is a closure of 1st LR(0) item

2 derivation inside E

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Scanned with CamScanner

21/2/24

$S' \to E$

$E \to E+T$

$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to id$

Goto (I, X)

I: $A \to \alpha \cdot X \beta \xrightarrow{Goto(I,X)} A \to \alpha X \cdot \beta$

will have to move rel pt one step ahead

~~Sto~~ Set of LR(0) items: all possible LR(0) that can be derived from the above expression

First LR(0) item

(0)

$S' \to \cdot E$

$E \to \cdot E+T$

$E \to \cdot T$

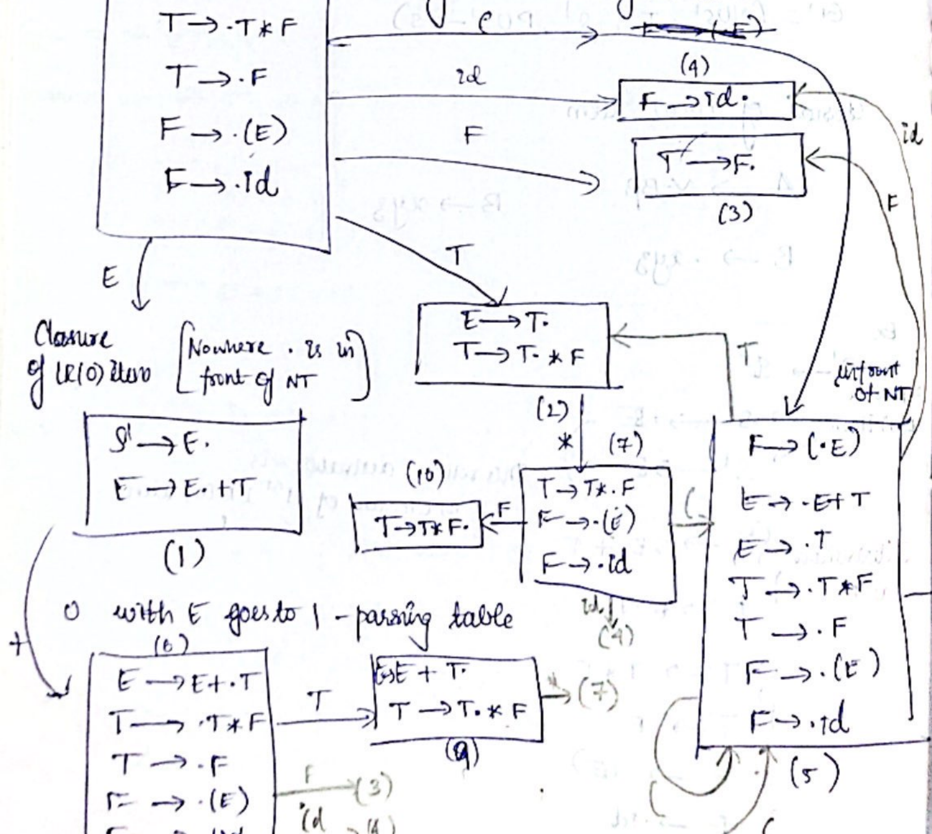
$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

Applying goto of each prod rules items and get other LR(0) grammars.



SLR parsing table:

$Goto(\overset{S_I}{I}, X) = \overset{S_I'}{I'}$

$X = a$, a is terminal

$M[S_I, a] \leftarrow$ shift S_I' — If it sees a terminal

$M[S_I, X] \leftarrow S_I'$ in goto part

↳ If it sees a terminal.

When · is at end of prod rule

If we found the follow set of NT

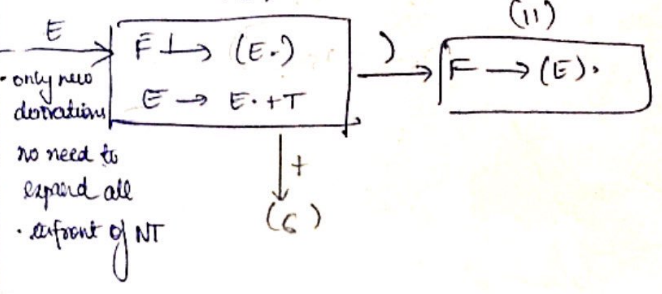
J: $A \to XYZ\cdot$ $b \in Follow(A)$

$M[J, b] \leftarrow$ Reduce by $A \to XYZ$

means we have seen the end of handle

If a set [$A \to XYZ\cdot$, $B \to CD\cdot$] cannot reduce by SLR parser

or [$A \to XY \cdot Z$, $B \to CD\cdot$] " "



Scanned with CamScanner

SLR parsing table

$S' \to E$

$E \to E+T$ — (1)

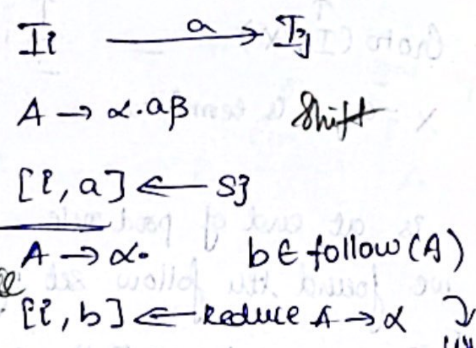
$E \to T$ — (2)

$T \to T*F$ — (3)

$T \to F$ — (4)

$F \to (E)$ — (5)

$F \to id$ — (6)



$I_i \xrightarrow{a} I_j$

$A \to \alpha . a\beta$ Shift

$[i, a] \leftarrow Sj$

Reduce $A \to \alpha .$ $b \in follow(A)$

$[i, b] \leftarrow$ Reduce $A \to \alpha$

all outgoing arrows through terminals are written with shift

Whenever we see a . at the end of a prod rule in any closure we can reduce

State	+	*	(	id	)	$	E	T	F
0			S_5	S_4			1	2	3
1	S_6	~~S_7~~				Accept			
2	r_2	S_7			r_2	r_2			
3	r_4	r_4			r_4	r_4			
4	r_6	r_6			r_6	r_6			
5			S_5	S_4			8	2	3
6			S_5	S_4				9	3
7			S_5	S_4					10
8	S_6				S_{11}				
9	r_1	S_7			r_1	r_1			
10	r_3	r_3			r_3	r_3			
11	r_5	r_5			r_5	r_5			

Scanned with CamScanner

For reducing,

Follow (E) = { $,), + }   If follow of a symbol contains $,

Follow (T) = { *, +,), $ }

{ E → E+T
→ T+T
→ T*F+T

{ F → (E)
F → (T)

{ E → T
$

T → F
(1)

{ T → T*F (1)
T → T*F*F

{ F → (E) → (T) → (F)

follow (F) = { +, *,), $ }

Follow(S') = { $ } → S' cannot come any where else

{ S' → E. ← wherever closure has this make it accept over $
Accept Follow (S') contains $

[If there is a . at the end of prod rule in any closure
ex $A \rightarrow \alpha.$
write the prod rule no. (ex r_1) in all ~~closure~~ follow set of A

[$I_i \xrightarrow{B} I_j$

$A \rightarrow \alpha . B\beta$

[i, B] ← j   Non terminals goto part   to po

From the state,
Write the closure number it redirects to on the N.T

Are all unambiguous grammar SLR grammar ?

$S' \to S$

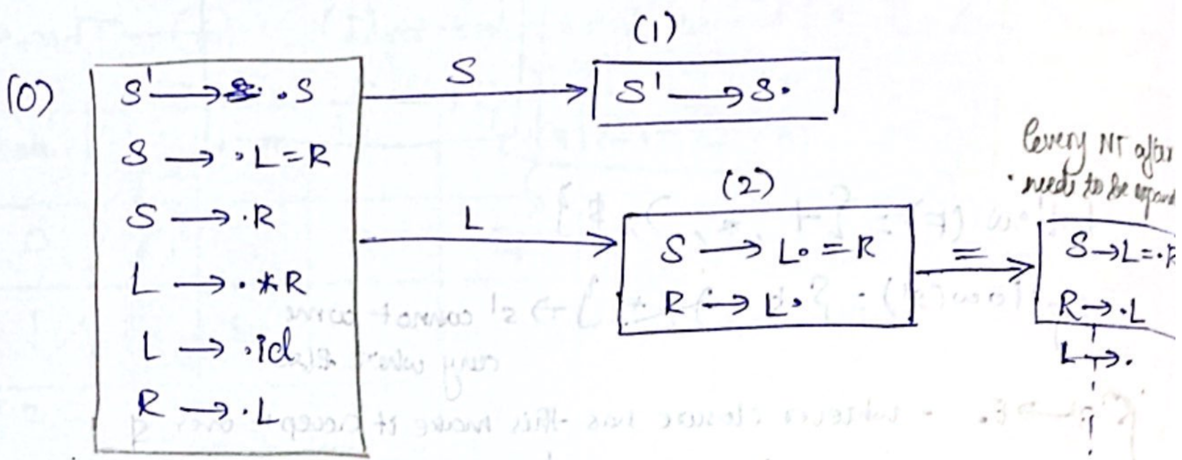
$S \to L = R$ — ①

$S \to R$ — ②

$L \to *R$ — ③

$L \to id$ — 4

$R \to L$ — ⑤



Follow(R) = {\$, =}

$S \to L = R$

$S \Rightarrow *R = R$

	=
0	
1	
2	S_3, r_5

→ 2 has $R \to L.$

follow(R) contains =

∴ add r_5 to state 2 on =

cannot be valid in SLR

From the parsing table, an example derivation :

0	id = id \$	Shift 5
0 id 5	= id \$	Reduce $L \to id$
0 L 2	= id \$	reduce $R \to L$
0 R 3	= id \$	No entry for 3 over =

Canonical LR parser :

Its LR(1) grammar

Takes the lookahead whether to reduce or shift

Scanned with CamScanner

LR(1) item

$[A \to \alpha \cdot \beta, a]$

Closure

$[A \to \alpha \cdot B\beta, a] \in I$

$B \to \gamma$, for each terminal $b \in$ first (βa)

$[B \to \gamma, b]$

Goto

$I \xrightarrow{B} J$

$[A \to \alpha \cdot B\beta, a]$

$[A \to \alpha B \cdot \beta, a]$

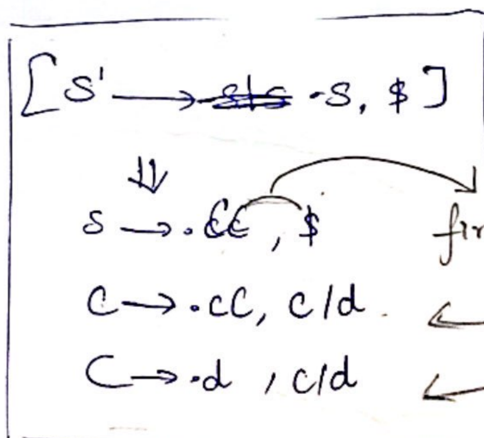
↳ the only diff is adding a

Ex:

$S' \to S$

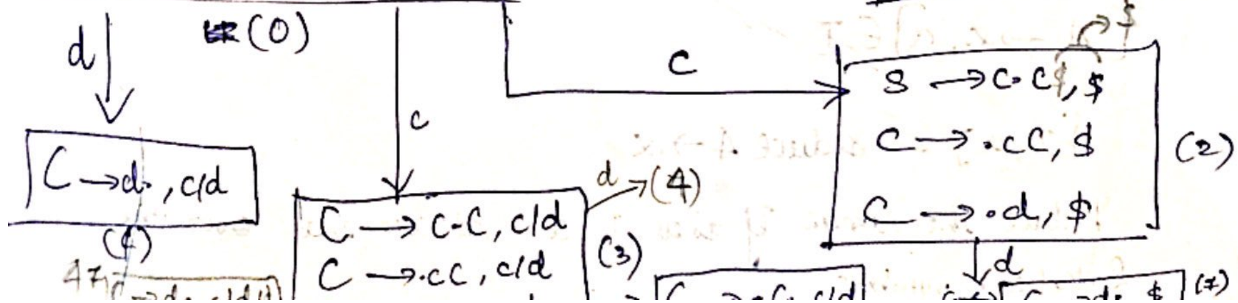
$S \to CC$ — the grammar is c^*dc^*d, atleast 2 ds

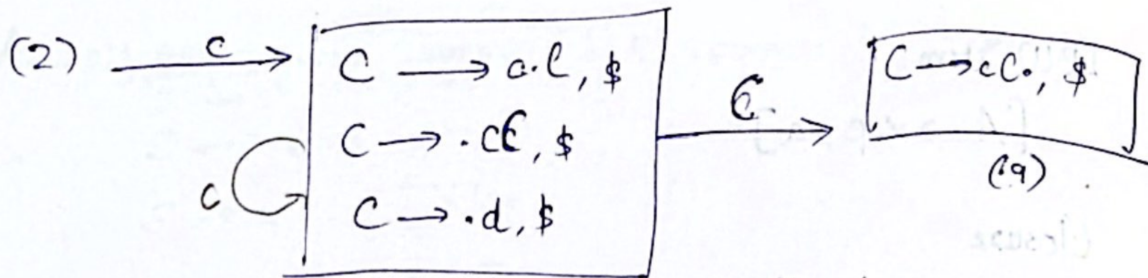
$C \to cC \mid d$

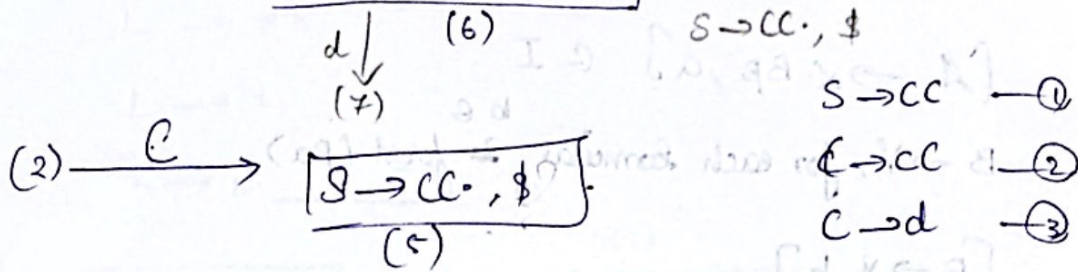


first LR(1) Item, with $ lookahead symbol

first (C$) = c/d







State	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_3		

Reduce

$[A \rightarrow \alpha\cdot, a] \in I$

$[I, a] \Leftarrow \text{reduce } A \rightarrow \alpha$

Reduce the clause if there is a · at the end over the lookahead symbols

6/3/24

SEMANTIC ANALYSER

Major roles

→ Type checking

Ex: int a , string b ;

a = b + 2 → semantic error

↓

no syntax error

→ variable used before declaration

Ex: a = b + c

↑ undefined

→ No. of arguments in function

→ return type

→ array arguments → can be 1D but then assumed as 2D

→ Accessing element / member of array incorrectly

→ Invalid operations Ex: string - string

→ Overloading of operators — For code gen phase, we need distinguish the operator distinctly

Ex: Addition, Concatenation, sign — '+' — depending on the context, meaning changes.

→ Inheritence

→ Scope checking

should not have multiple definition of the same variable within the same scope of function

→ Uniqueness of variables / function

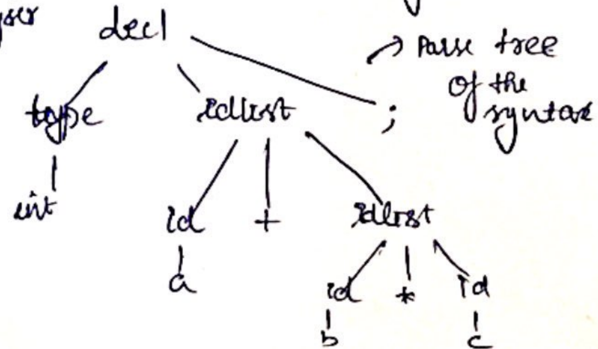
int a, b, c ; → output of lex analyser

then:

decl → type idlist ;

idlist → id , idlist | id

type → int | float | char



Scanned with CamScanner

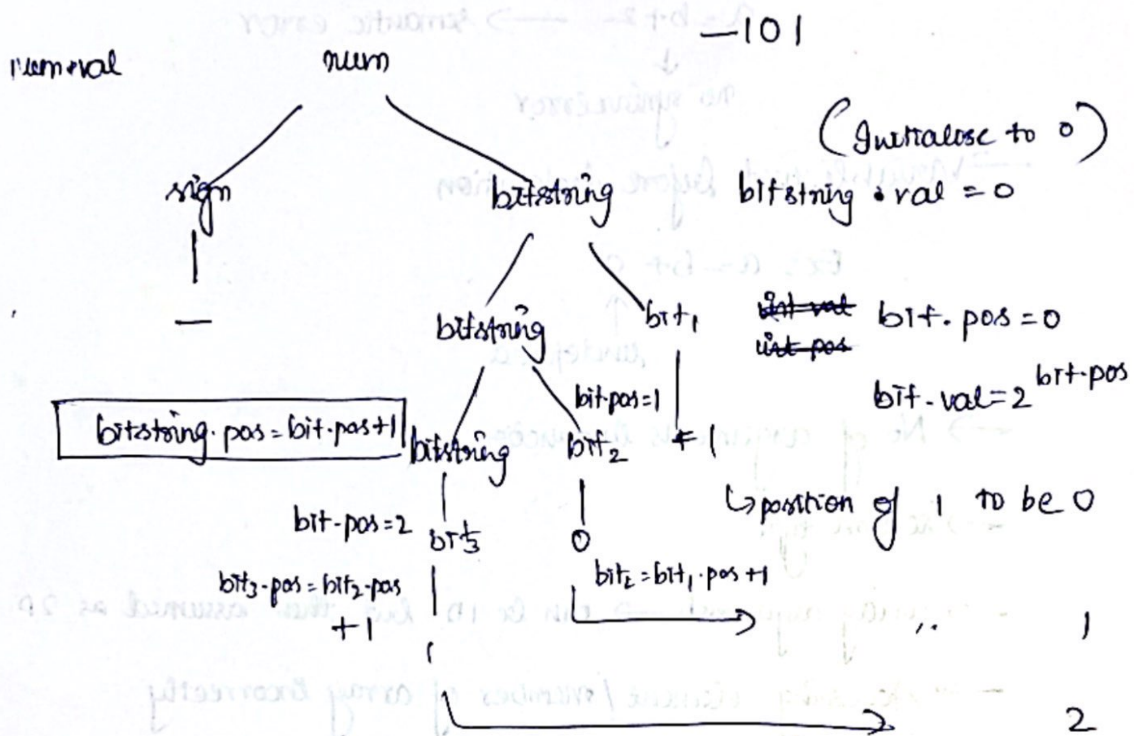
num → sign bitstring

sign → + / −

bitstring → ~~{bit {bitstring}}~~

{bitstring}{bit} | bit

bit → 0 | 1



need position of value to convert into decimal by the semantic analyser.

Attributes here are position of value

bitstring.pos = 0

We are traversing the parse tree again to find the location of variables

19/3/24

If inherited attribute,

then we need attribute ready of parents to evaluate attribute of children

If both inherited & synthesized attribute,

a top down parser is not enough to parse all the inherited attribute.

L → En

E → E+T

E → T

T → T*F

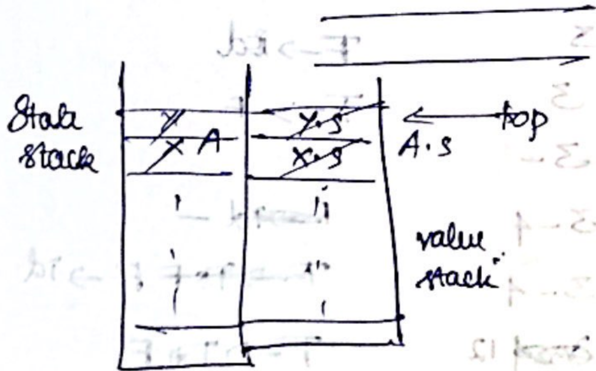
T → F

F → (E)   F.val = E.val

F → id   F.val = id.lexval

everytime root is dependant on children - synthesized attribute
↓
A grammar with all synthesized attribute → S-attributed grammar

Viable prefix : all the seen symbols are there in stack



$A.s = f(X.s, Y.s)$

$A \to XY$

$X.s = a.v$

$Y.s = b.v$

We need another stack to store val & synthesize it

Scanned with CamScanner

$Y \cdot S \leftarrow$ value[top]

$X \cdot S \leftarrow$ value[Top − 1]

nTop ~~A~~ = f (value [Top] , value[Top − 1])

updated value of top after reduction → nTop = Top − r + 1

r = | RHS of production applied for reduction |

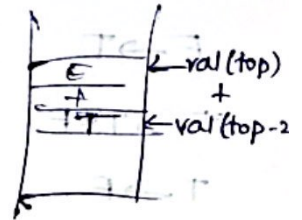
Top = nTop

L → En   print (val(top))

E → E+T   val(nTop) = val(Top) + val(Top−2)

E → T   ——

[Automatically T·val is assigned to E·val]



T → T*F   val (nTop) = val(Top) * val(Top − 2)

T → F   ——

F → (E)   val (nTop) = val(Top − 1)

F → id   ——

Value stack helps in getting the new attribute value from old "

i/p	Stack	Value	Prod
3*4+5n	—		
*4+5n	id	3	~~Shift~~
*4+5n	F	3	F → id
*4+5n	T	3	T → F
4+5n	T*	3 –	—
+5n	T*id	3 – 4	~~F→id~~ —
+5n	T * F	3 – 4	~~T→T*F~~ F → id
+5n	T	~~3–4~~ 12	T → T*F
+5n	E	12	E → T

Scanned with CamScanner

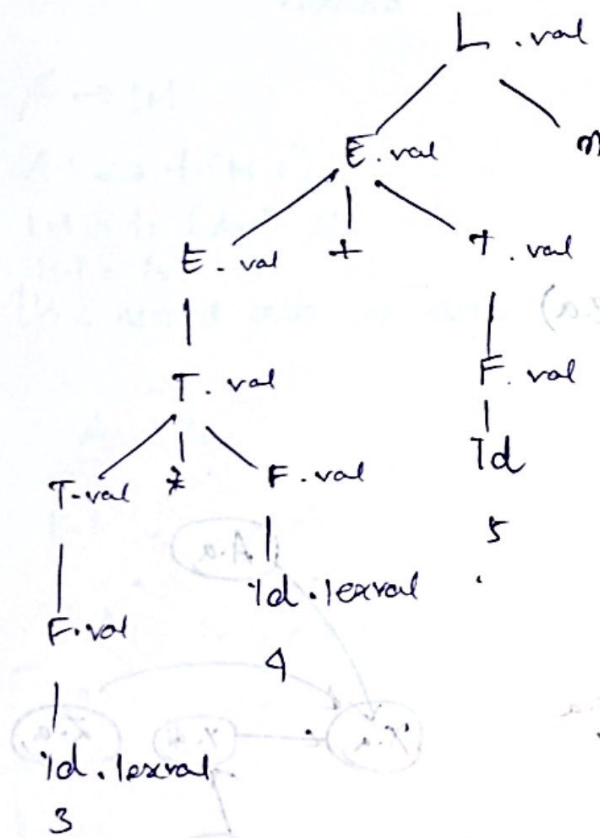
Input	Stack	Value	Production
5n	E+	12 -	—
~~5~~n	E+Id	12 - 5	—
n	E+F	12 - 5	F→Id
n	E+T	12 - 5	T→F
n	E	17	E→E+T
	En	17	—
—	L	17	L→En

↓ print

Accepting stage, point the value [top]

For the prod rules

.val for all variables



→ E.node = newnode(+, E_1node, T node)

→ E.node = T.node

→ T.node = newnode(*, T_1node, F node)

→ T.node = F node

→ F.node = E.node

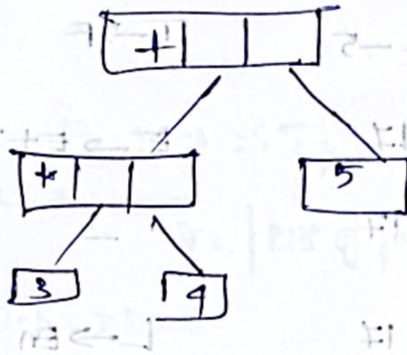
→ F.node = new node (Id)

Annotated parse tree - when there are attribute along with the variable

one version of parse tree

Syntax tree

3 * 4 + 5



Annoted parse tree
eg " "
doesn't carry forward
after semantic analy
Only syntax tree will
carry forward

For the prod rules

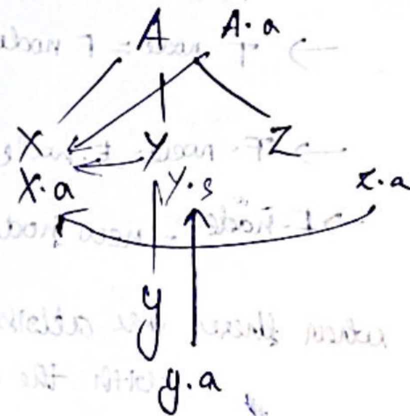
$Y \to y$

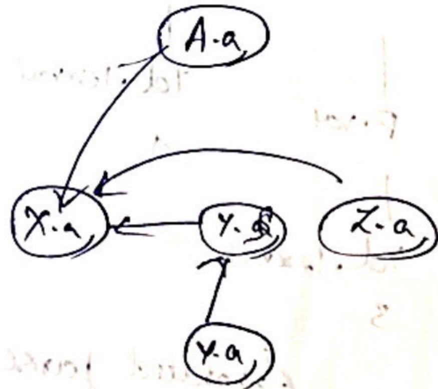
$A \to XYZ$

$Y.s \Rightarrow f_1(y.a)$

$X.a = f_2(A.a, Y.s, Z.a)$

Dependancy Graph





A graph with a node having zero indegree, can start with that

$N_1, N_2 - - -, N_K$ — nodes

→ N_j should be dependant on any nodes from N_1 to N_{j-1}
but not on N_l, $l > j$

Scanned with CamScanner

→ N_1 is zero indegree node

	inh	syn
Top	? ✓ (L-attr grammar)	??
Bottom	??	✓

⇒ Depth First search over parse tree

L-attributed grammar

$$A \to X_1 X_2 \cdots X_n$$

$$X_j.a = f(A.i, X_1.a, X_2.a, \ldots, X_{j-1}.a)$$

↑ inherited — any attribute

$A \to LM$

$A.s \Rightarrow f_1(M.s)$

$L.i = f_2(A.s)$ ✗

$M.i = f_3(L.s)$

L's inherited attr can depend only on its parents inh attribute

$A \to QR$

$R.i = f_4(A.i)$ ✓

$Q.i = f_5(A.i, R.s)$ ✗

$A.s = f_6(Q.s)$

20/3/19

AGF

- Attribute Grammar
- Translation rules -

$L \to En$ L.val = E.val ← Attribute Grammar

{print (E.val)} ← translation rule

$F \to id$ addentry(id, id.type)

Scanned with CamScanner

- Syntax directed definition (SDD) - balance btw A.G & T-Rule
- " " translation ~~rule~~ scheme (SDT) - embedding the action at a particular location where it needs to be execute

Semantic rule is L.val = E.val

but instead of it, we use print (E.val)

↓

This is called side effect.

SDT - to have translation rule with controlled side effect.

S-attributed SDD ⇒ bottomup parser

↳ Post order traversal of parse tree ✓

$A \rightarrow \{\} X_1 \{\} X_2 \{\} X_3 \{\}$ ↳ actions | SDT

$A.s = f(X_1.s, X_2.s, X_3.s)$

The action is put at the rightmost end of the production rule in a synthesized attribute grammar.

SDT:

- L → En { print (E.val) }
- E → E+T { E.val = E₁.val + T.val }
- E → T { }
- T → T*F { }
- T → F { }
- F → (E) { }
- F → digit { F.val = digit.lexval }

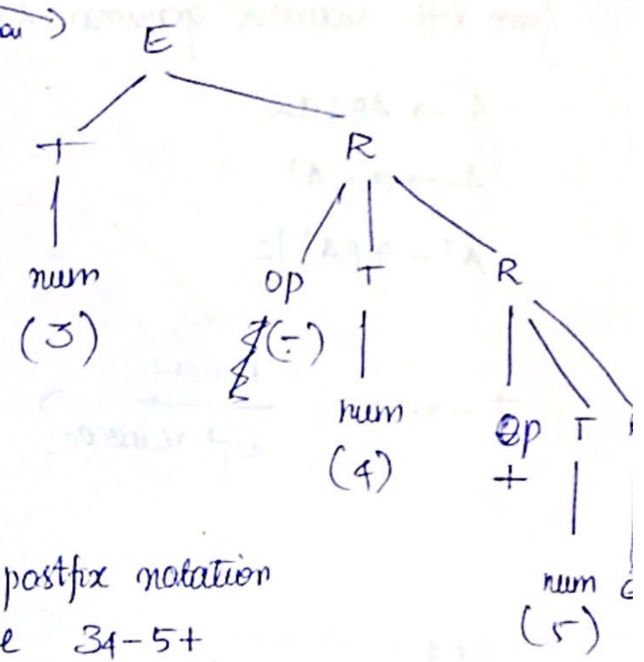
$E \rightarrow TR$

$R \rightarrow op\,TR \mid \epsilon$

$T \rightarrow num$

$op \rightarrow + \mid -$

parse tree for this grammar



To get the foll expression,

$3-4+5$

Add the action s.t it reads the postfix notation

i.e $34-5+$

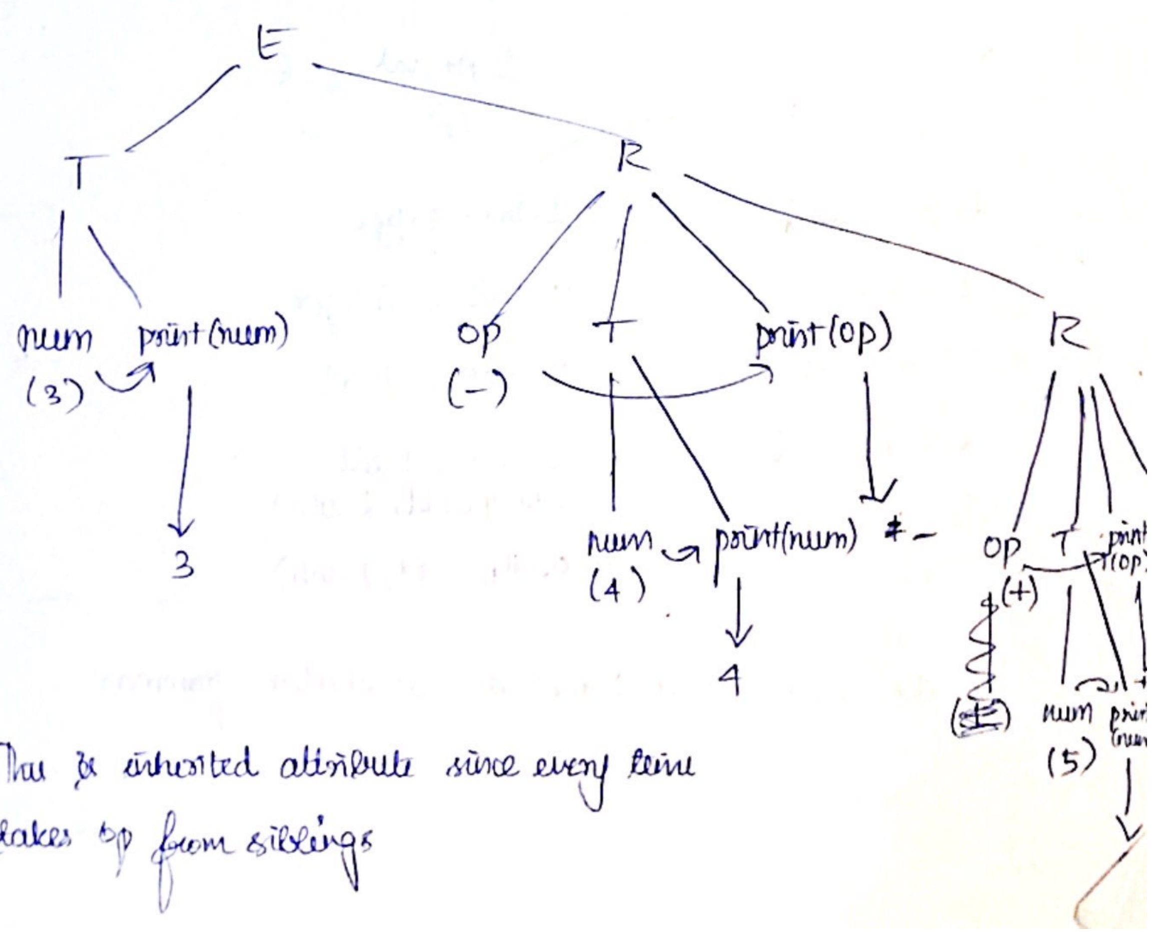
$E \rightarrow TR$

$R \rightarrow op\,T\ \{print(op)\}\ R \mid \epsilon$

$T \rightarrow num\ \ \{print(num)\}$

$op \rightarrow + \mid -$

∵ after every T converted to num in R, we need to print op symbol



This is inherited attribute since every time it takes op from siblings

form left recursive grammar, we cannot construct top down parser

$A \rightarrow A\beta \mid a\alpha$

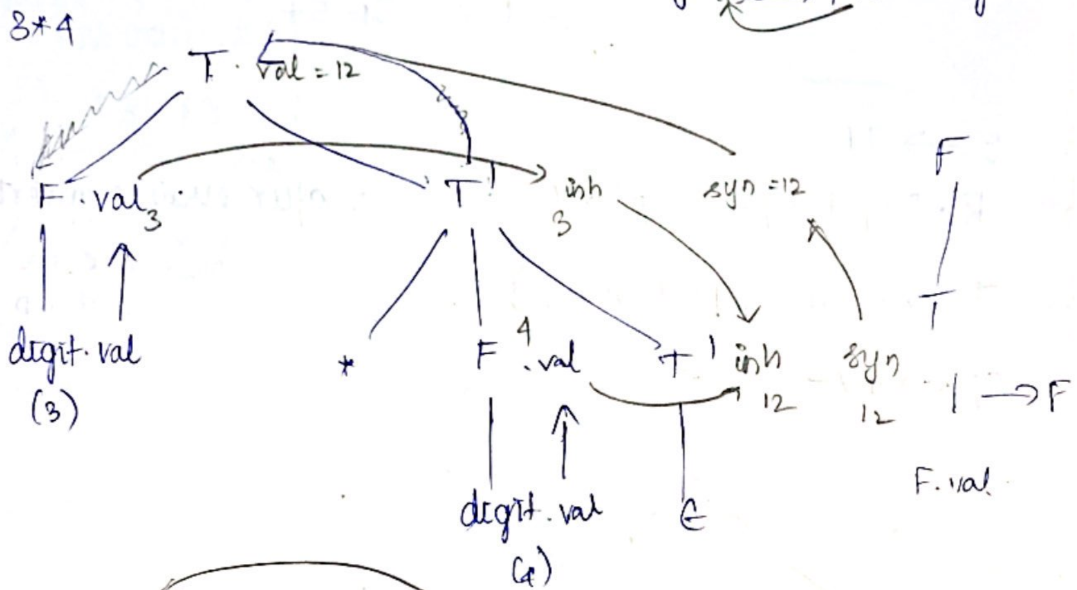
$A \rightarrow a\alpha A'$

$A' \rightarrow \beta A' \mid \epsilon$

$T \rightarrow T * F$ —Remove left recursion→

	Sem rule
$T \rightarrow F T' \{\}$	$T'.inh = F.val$; $T.val = T'.syn$
$T' \rightarrow * F T'_1 \{\}$	$T'_1.inh = T'.inh * F.val$; $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon \{\}$	$T'.syn = T'.inh$
$T \rightarrow digit \{\}$	$T.val = digit.lexval$

3*4

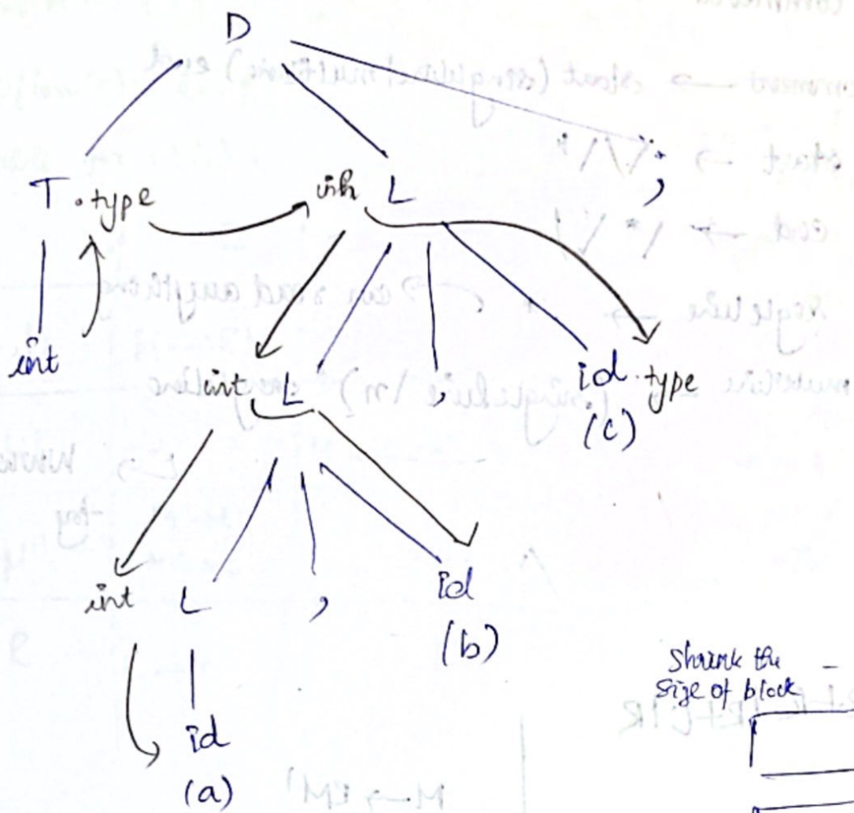


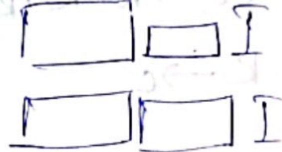
$D \rightarrow \{\} T L ;$	$L.inh = T.type$
$T \rightarrow int \{\}$	$T.type = integer$
$T \rightarrow float \{\}$	$T.type = float$
$L \rightarrow \{\} L_1 , \{\} id$	$L_1.inh = L.inh$; addtype(id, L.inh)
$L \rightarrow \{\} id$	addtype (id, L.inh)

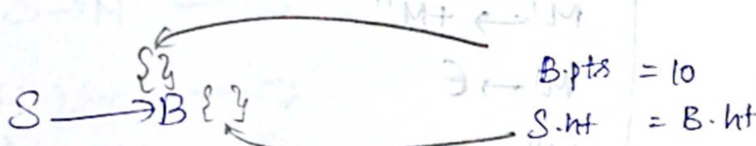
Put the action just before the attributed grammar

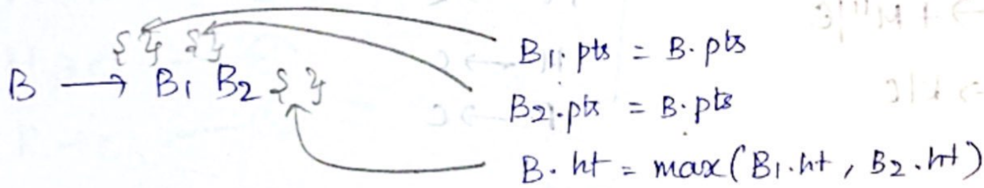
Scanned with CamScanner

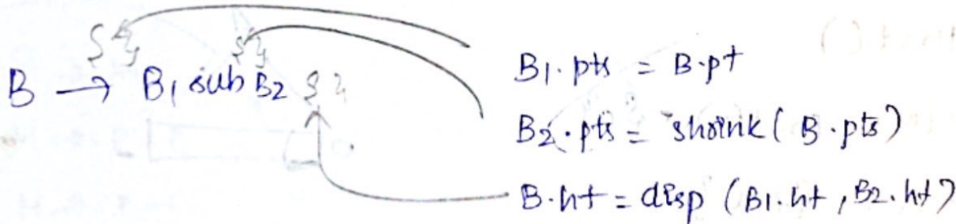
int a, b, c ;



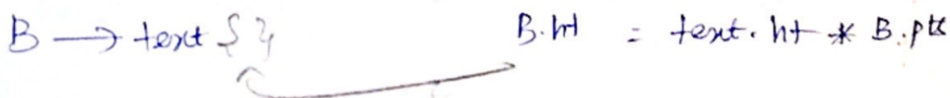








displacement : calculating subscript block size along with main block size



Scanned with CamScanner

CT

RE for comments

comment → start (singleline | multiline) end

start → \/*

end → *\/

singleline → .* ↪ can read anything

multiline → (singleline \n)* singleline

↪ www.regex101.com try

"/*"

$M \to R+R \mid R+C \mid R$

$R \to c$

$M \to RM'$

$M' \to +R \mid +C \mid \epsilon$

$M' \to +M'' \mid \epsilon$

$M'' \to R \mid C$

$M \to RM'$

$M' \to +M''$

$M' \to \epsilon$

$M'' \to R$

$M'' \to c$

$R \to c$

First( )

First(M) = First(R) = {c}

First(M') = {+, ∈}

First(M'') = First(R) = {c}

Follow set

	M	M'	M''	R
1	\$			
2				+
3				\$
4		\$	\$	

Scanned with CamScanner

Follow(M) = {$} = Follow(M') = Follow(M'')

Follow(R) = {+, $}

Parse table for LL(1):

	c	+	$
M	M → RM'		
M'		M' → +M''	M' → c
M''	M'' → R M'' → c		
R	R → c		

In the first set of the respective symbol fill the prod. that made that symbol first set

If there is ε, then see the follow set symbol & fill the prod there

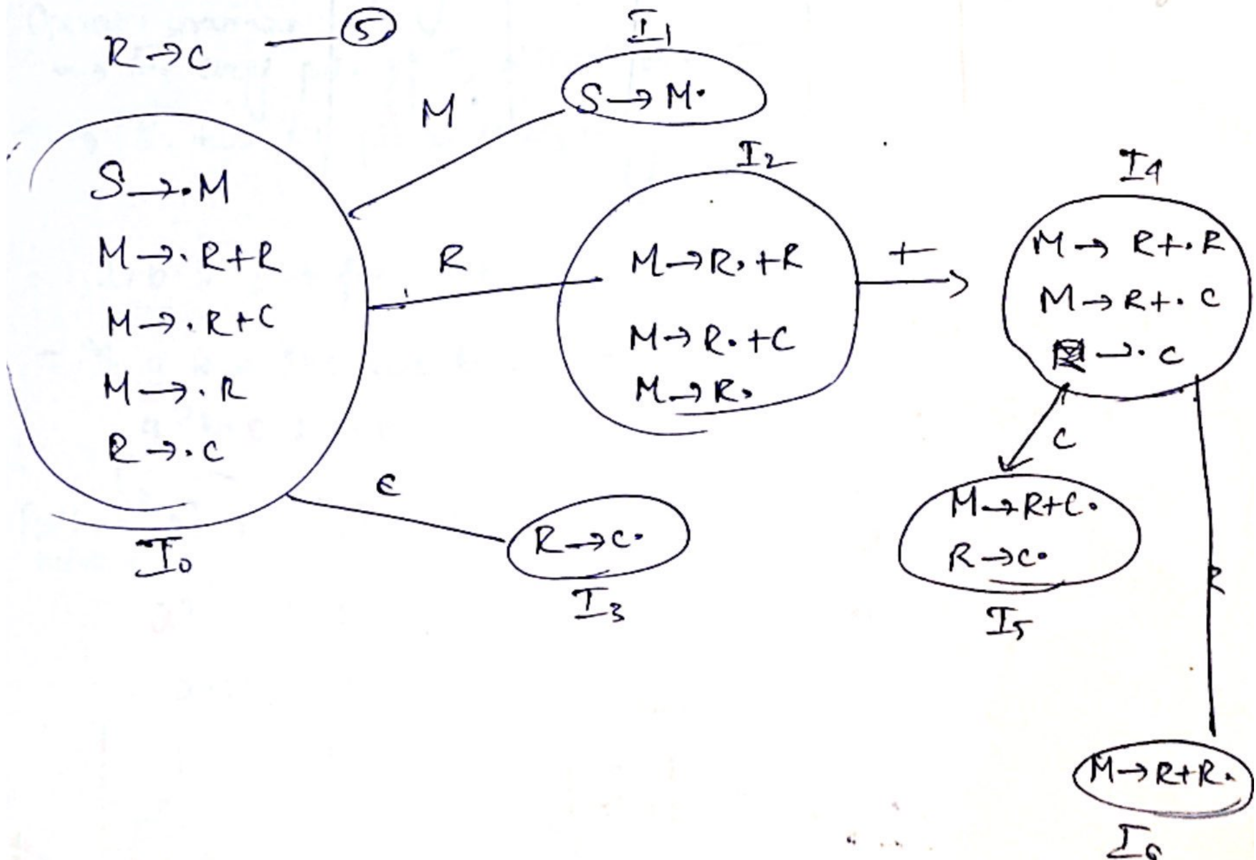
S → M — ①

M → R+R — ②

M → R+C — ③

M → R — ④

R → c — ⑤



Follow(S') = {$}

Follow (M) = {$}

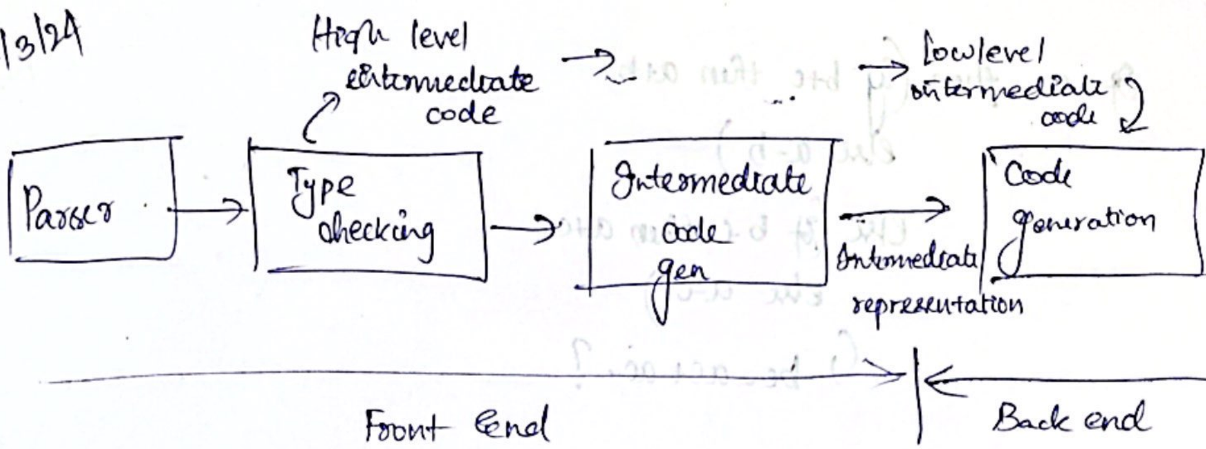
Follow (R) = {+, $}

SLR parse table

M is the start symbol ∴ S→M. is there in S1 as Follow(S) = $

States	Action: c	+	$	Goto: S	M	R
0	s3				1	2
1			Accept			
2		S4	r4			
3		r5	r5			
4	S5					6
5		r5	r3 r5			
6			r2			

27/3/24



- Post fix notations
- syntax tree
- Three address Code

$A \rightarrow \alpha a \beta b \gamma$

α, β, γ can be ϵ or a single non-terminal.

$E \rightarrow EAE \mid (E) \mid id$
$A \rightarrow + \mid - \mid * \mid \uparrow$

not an operator grammar

$E \rightarrow E+E \mid E-E \mid E*E \mid E \uparrow E \mid (E) \mid id$

operator grammar

Operator grammar :
→ For every pair of T, atleast one NT
→ No two NT placed consequently

a+b in post fix : ab+

- If a then b+c else b-c

a?b+c : b-c

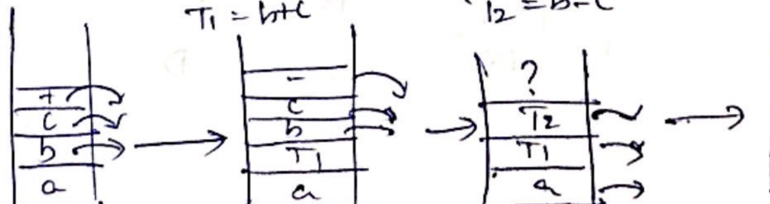
Postfix notation: abc+bc-:?

a?b+c b-c

abc+bc- ?

- If a then (if b+c then a+b) else if b-c then a+c

If a   b+c
If b+c
a+b
else if b-c
a+c



If a then (if b+c then a+b
else a-b)
else (if b-c then a+c
else a-c)

↳ bc-ac+ac-?

a bc+ab+ ab-? bc-ac+ ac-??

Quadruples

OP	ARG1	ARG2	RESULT
ADD	B	C	T1
EQ	T1		A

→

(1) ADD B C

(2) EQ (1) A

Triplets/triples

A = B * C + D

MUL	B	C	T1
ADD	T1	D	T2
EQ	T2		A

(1) MUL B C

(2) ADD (1) D

(3) EQ (2) A

Easy implementation

Space efficient

Changing an expression affects entirely

Indirect Triples

If any changes needed, the order table changing would do.

Order

seq	addr
(1)	10
(2)	11
(3)	12
(4)	13

	OP	ARG1	ARG2
10	ADD	B	C
11	EQ	10	A
12	SUB	13.	C
13	EQ	12	D

9/4/24

Jumping

unconditional   conditional

Ex: In an if else block, there exist common code

call stack

param arg 1

" " 2

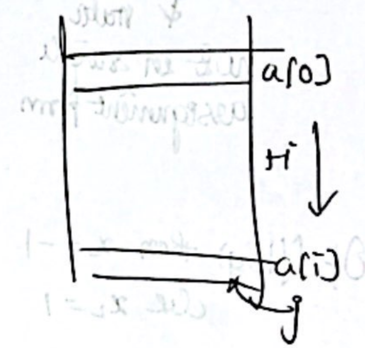
:

call (<function>, n)

<function> (arg 1, arg2, ... arg n)

a[i] = j

j = a[i]

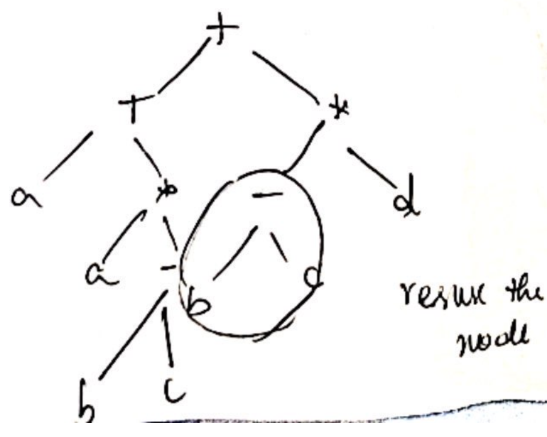


TAC for

If x > y then x = y + 10
else y = x + 10

```
    if x>y goto L1   -> conditional
    t1 = x+10
    y = t1
    goto L2          -> uncond
L1  t1 = y+10
    x = t1
L2  ...
```

⟹ $a + a*(b-c) + (b-c)*d$



Table

1.	id	a	
2.	id	b	
3.	id	c	
4.	-	2	3
5	*	1	4
6.	+	1	5
7.	id	d	
8.	*	4	7
9.	+	6	8

Static single assignment form

$p = a+b$ | $p_1 = a+b$
$q = p - c$ | $q_1 = p_1 - c$
$p = q+d$ | $p_2 = q_1 * d$
$p = e-p$ | $p_3 = e - p_2$
$q = p+q$ | $q_2 = p_3 + q_1$

↓ note: not in single assignment form

↓ SSA → can't reuse same variable without defining

If (flag) then $x = -1$
else $x_2 = 1$

$y = x * a$

∵ we can't predict the val of x in $y = x * a$, we take $x_3 = \phi(x_1, x_2)$ which outputs the correct value

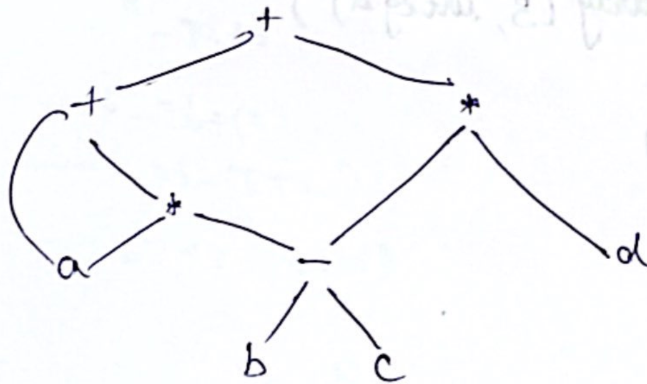
Scanned with CamScanner

10/4/24

DAG:

$a + a*(b-c) + (b-c)*d$

Start →



Three address codes:

1. Assignment : $x = y \text{ op } z$, op is arithmetic or logical operation
2. " : $x = \text{op } y$, op is unary operation
3. Copy instructions : $x = y$
4. Unconditional jump → goto L
5. Conditional jumps →
   a. If x goto L
   b. If false x goto L
   c. If x relop y goto L
6. Procedure calls & returns:

   param x1
   param x2
   ⋮
   param xn

7. Indexed copy instructions
8. Address & pointer assignments

   $x = \&y$ sets the r-value of x to be location of y

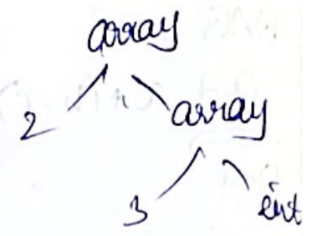
   $x = *y$ y is pointing to a value which is a address

   $*x = y$ sets r-value of the object pointed to by x to the r-value of y

Scanned with CamScanner

⇒ SSA ;

⇒ Type expressions



array = int[2][3]

array(2, array(3, integer))

x = a+i×4

x = y[i]

Ex:

int[2][3]

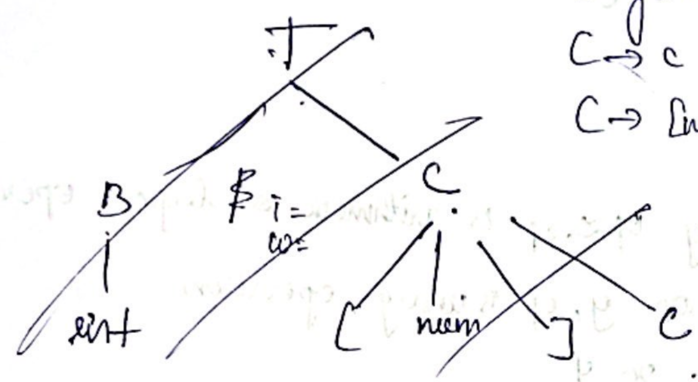
T → B {t = B.type; w = B.width}
C

B → int {B.type = int, B.width = 4}

B → float { " float, B.width = 8}

C → c {c.type = t, c.width = w}

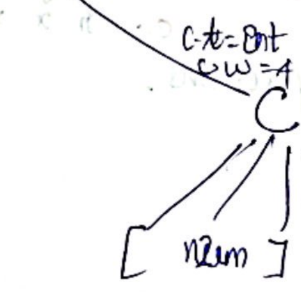
C → [num] C1 {array(num.value, C1.type), C.width = num.value × C1.width}



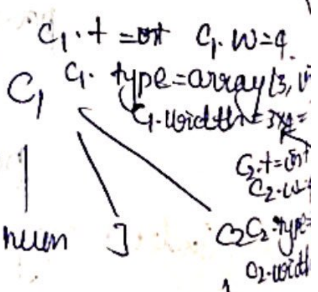
t = int
w = 4

T.type = array(2, array(3, int))
T.width = 24

B.type = int
B.width = 4



C.type = array(2, array(3, int))
C.width = 2 × 12 = 24



Seq of declarations

computing the relative addr of declared name

P → {offset = 0}
D

D → T id ; E.top = put id (lexeme)

Translation of expressions

- Operation within expr
- Incremental translation Rules

$x = -y + (w+z)$

$S \rightarrow id = E$

$\rightarrow id = -E+E$

$\rightarrow id = -id+E$

$\rightarrow id = -id+(E)$

$\rightarrow id = -id+(E+E)$

$\rightarrow id = -id+(id+id)$

$S \rightarrow id = E;$

$E \rightarrow E_1 + E_2$

$\mid -E_1$

$\mid (E_1)$

$\mid id$

S gen = 12

id — x (1)

=

E.addr = t3 = t1+t2 (11) gen

E (1. gen (4)) + E2.addr = t2 (10)

E → − E3.addr = y (3); id (2); y

E2 → (E2) gen (t2)

E4.addr = w (6) + E5.addr = w (8); id (5) — w; id (7) — z

1. T_1 = minus y (4)
2. T_2 = w+z (9)
3. $T_3 = T_1 + T_2$ (11)
4. $x = T_3$ (12)

↓ lookup.get(id.lexeme)

↓ 3AC is generated as part of SDT

↓ 3AC

t_1, t_2, t_3 - new temp ( )

Scanned with CamScanner

16/4/24

$$A[i][j] = base + i \times w_1 + j \times w_2$$

(n_1, n_2 written above the indices)

↓

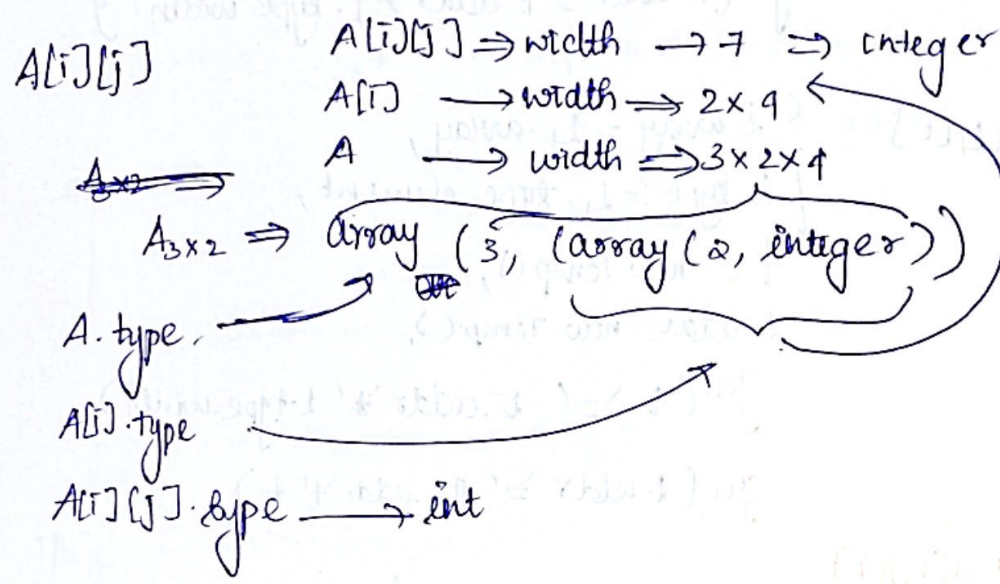
using no. of elements

↓

$$base + (i \times n_2 + j) \times w \Rightarrow$$

k-dimension ⇒ $base = i_1 \times w_1 + i_2 \times w_2 + \dots + i_k \times w_k$

$$base + ((\dots((i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k) \times w$$



$$L \to L[E] \mid id[E]$$

$$E \to E + E \mid id \mid (E)$$

S → id = E, { gen (top.get(id.lexeme) '=' E.addr); }

↓

Creating 3 addr code

| L = E { gen (L.array.base '[' L.addr ']' '=' E.addr); }

↓

base addr

E → $E_1 + E_2$ { E.addr = new Temp();
gen (E.addr '=' E_1.addr '+' E_2.addr); }

Scanned with CamScanner

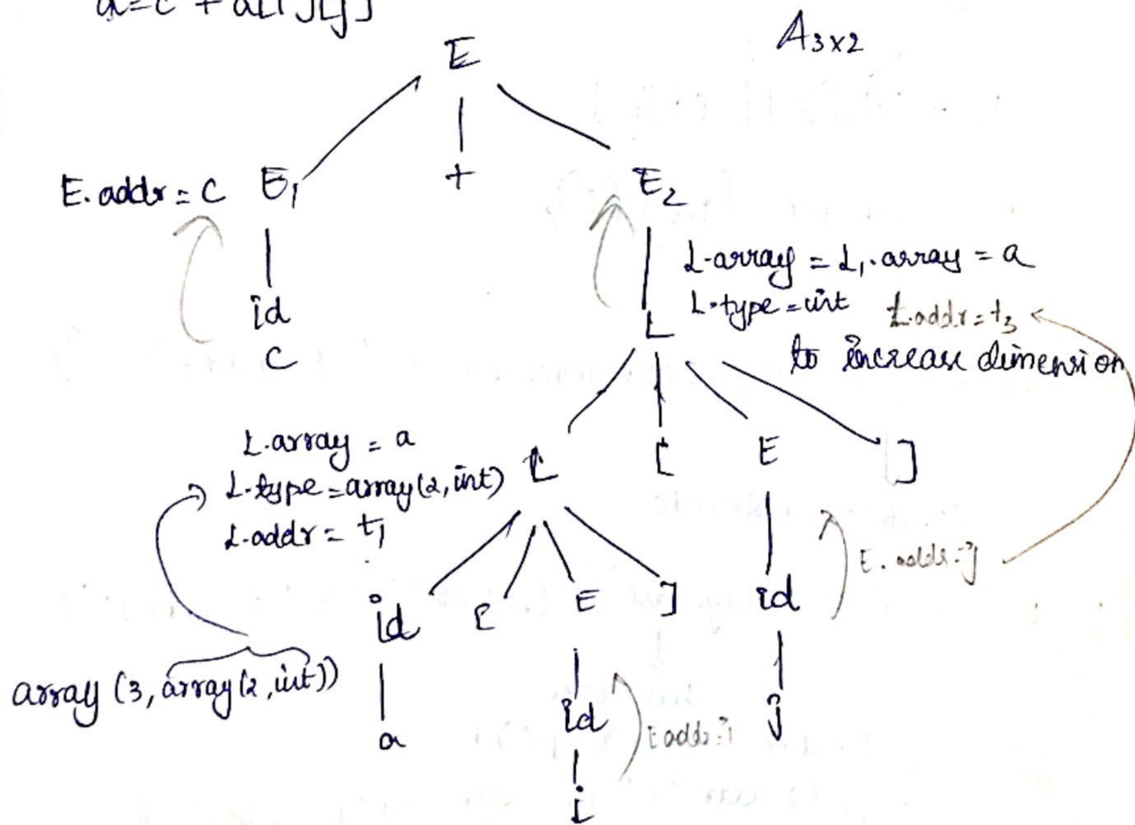
| id   { E.addr = top.get(id.lexeme); }

| L   { E.addr = new Temp();
gen(E.addr '=' L.array.base '[' L.addr ']'); }

L → id[E]   { L.array = top.get(id.lexeme) }
L.type = L.array.type.element,
L.addr = new Temp();
gen(L.addr '=' E.addr * L.type.width) }

| L_1[E]   { L.array = L_1.array,
L.type = L_1.type.element,
t = new Temp(),
L.addr = new Temp(),
gen(t '=' E.addr '*' L.type.width),
gen(L.addr '=' L_1.addr '+' t);

d = c + a[i][j]



1. $t_1 = i \times (2*4)$
   Type.width

2. $t_2 = j * 4$
   Eaddr   L.type.width (int)

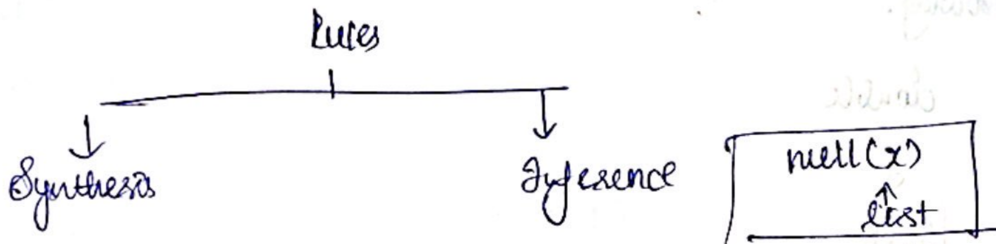
3. $t_3 = t_1 + t_2$
   L.addr

4. $t_4 = a[t_3]$
   L.array.base → L.addr

17/9/24

Type Checking

- sound type system - compile time error free
- Strongly typed - language that guarantees that there won't be any type error



ML — doesn't require var to be declared before using - dependending on context decides the type
inference type

If f has type $s \to t$ and x has type s then expression $f(x)$ has type t

$a = b + c$

If $f(x)$ is an expression

then for some α & β, f has type $\alpha \to \beta$ & x has type α

Scanned with CamScanner

finding length(x):

if null(x) then 0

else length

Type widening - default type conversion in most progr lang

not using any info, giving more space than reqd.

double
↑
float
↑
long
↑
int ← char ← byte
↑
short

Type narrowing:

double
↓
float
↓
long
↓
int → short → char → byte

Casting: explicit

programmer has to define

Coercion: implicit

Scanned with CamScanner

$E \rightarrow E_1 + E_2$ { E.type = max(E_1.type, E_2.type),
a_1 = wider(E_1.addr , E_1.type, E.type),
a_2 = wider(E_2.addr, E_2.type, E.type),
E.addr = new temp (),
gen(E.addr '=' a_1 '+' a_2), }

control flow

- Boolean variable
- Relational operator

- if
- if then else
- while loop

B → B||B | B&&B | !B | (B) | E rel E | true | false

④ ③ ② ①

rel : — allowed along with boolean operations

<, <=

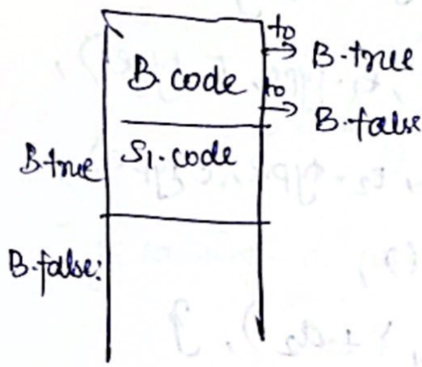
>, >=

=, !=

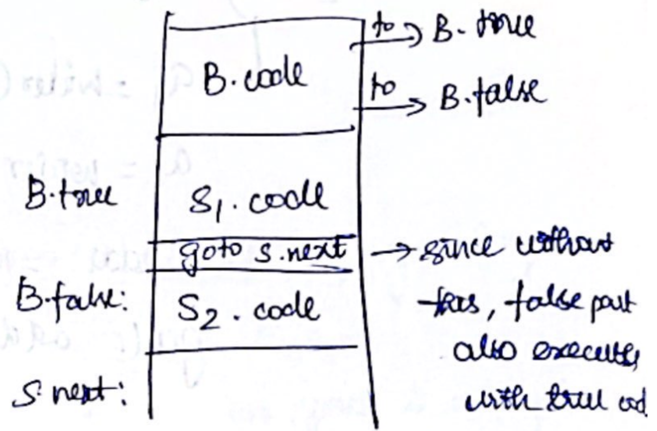
```
if (x<100 || x>200 && x!=y) then x=0;
                        ↑ more precedence
        ⇓
    if x<100 goto L2 (some level)
    ifFalse x>200 goto L1
    ifFalse x!=y goto L1
L2 : x=0
L1
```

short circuit code

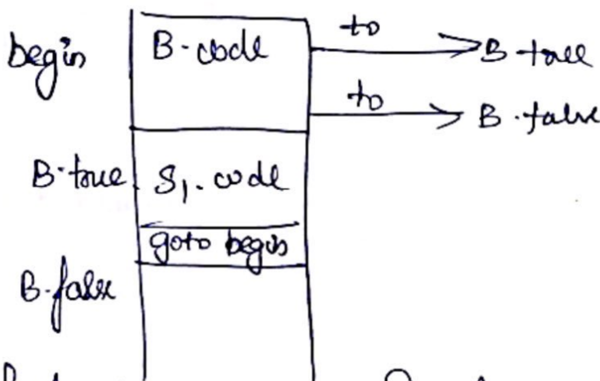
$S \rightarrow$ if (B) then S_1



$S \rightarrow$ if (B) then S_1 else S_2



$S \rightarrow$ while (B) S_1



Prod rules | Semantic rules

$P \rightarrow S$ S.next = newlabel() {,
P.code = S.code || label (S.next)

$S \rightarrow$ if (B) then S_1 B.true = newlabel()
B.false = S_1.next = S.next
S.code = B.code || label (B.true) || S_1.code

$S \rightarrow$ if (B) then S_1 else S_2 B.true = newlabel()
B.false = newlabel ()
S_1.next = S_2.next = S.next
S.code = B.code || label (B.true)
|| S_1.code || gen (goto S.next)
|| label (B.false) || S_2.code

Scanned with CamScanner

$S \rightarrow$ while (B) S_1 begin newlabel()

B.true = newlabel()

B.false = S.next

S_1.next = begin

S.code = label(begin) || B.code ||

label(B.true) || S_1.code ||

gen(goto begin)

For comparison operators:

Prod rules   Semantic rules (Picture)

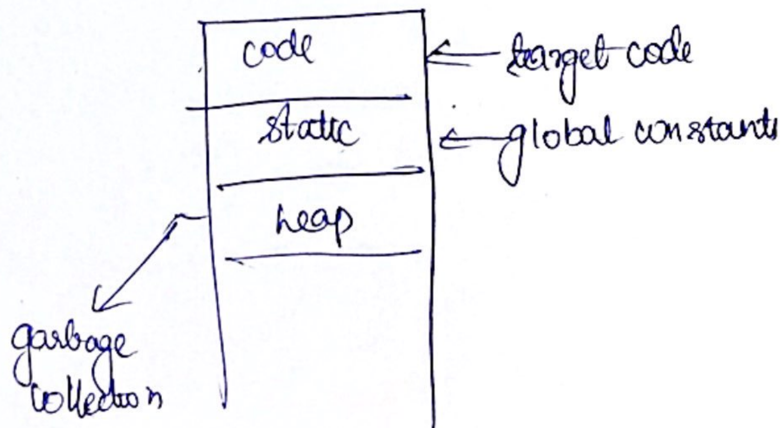
$B \rightarrow B_1 || B_2$ B_1: true

22/4/24

Run time environment

Code optimization

code generation



23/4/24
afternoon-extra

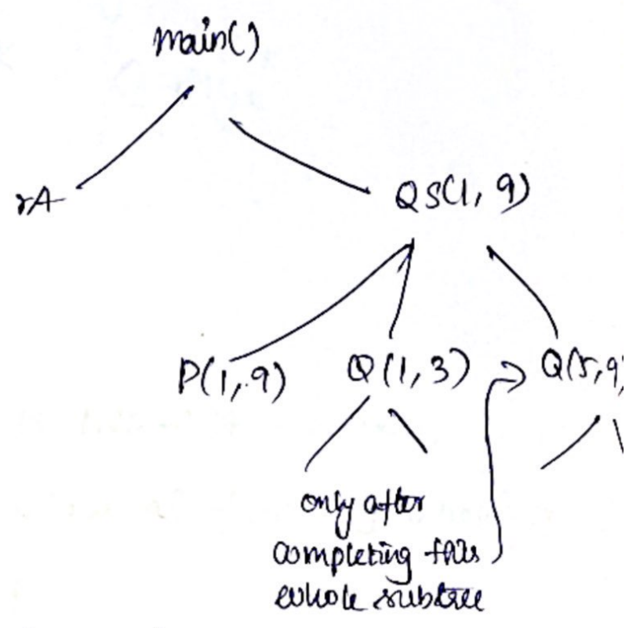
Activation - executing procedure

- main
- readArray
- Qsort (1, 9)
- partition (1, 9)
- QS (1, 3)

  ⋮

QS (5, 9)

Activation tree



pre-order traversal : the seq of procedure call

post-order " find out seq, which procedure are executed

→ normal termination
→ termination cos of execution p → q

If p is able to handle the exception returned by q

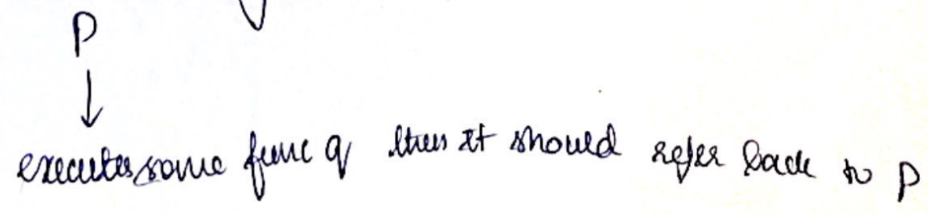
not able → then p also terminates

→ If q is aborted → p also aborted

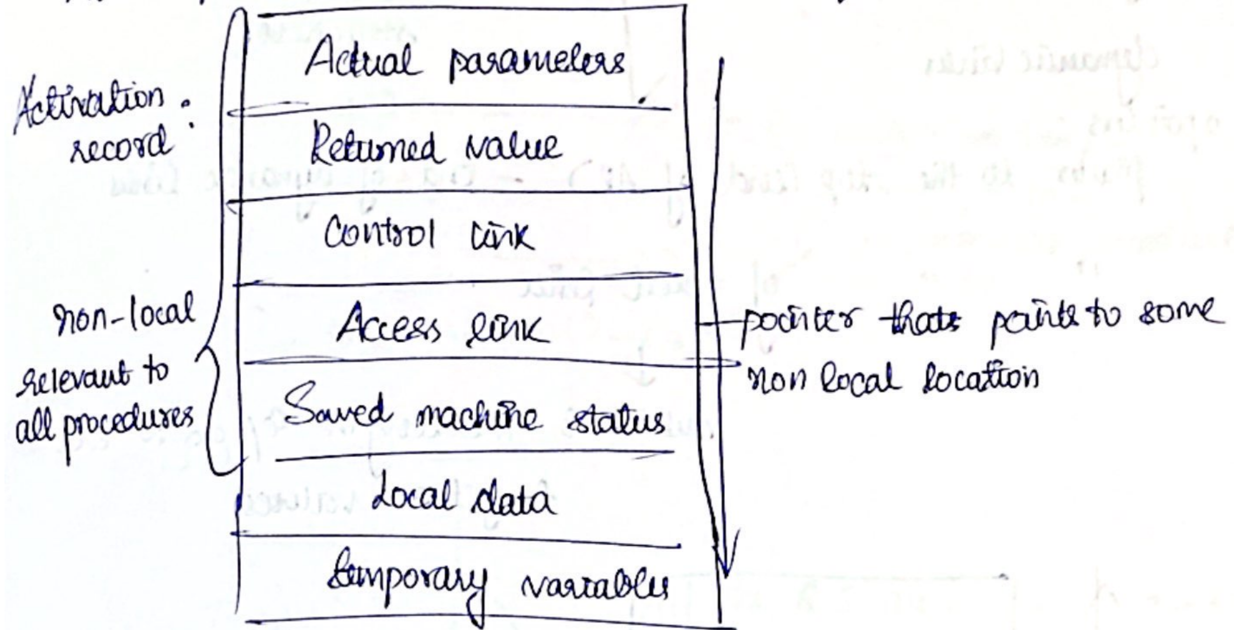
A process being terminated then the main also terminates

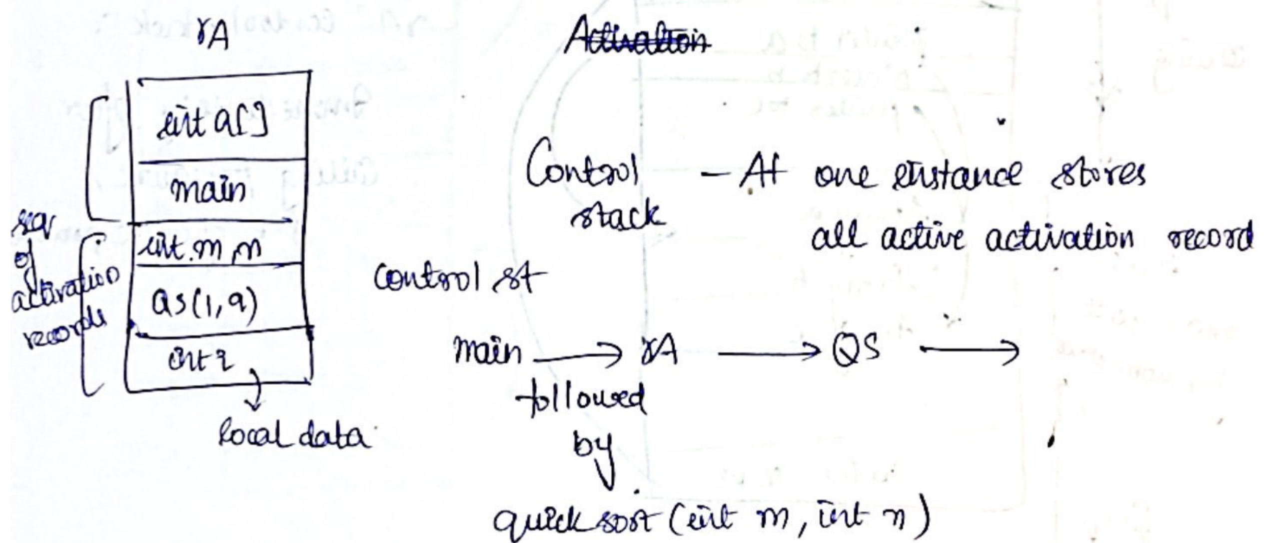
Activation record:

A DS that stores lot of info which are relevant to executing p



Actual parameters :- actual values that are passed





partition activation record comes after rA

Activation record - store the communicating parameters

↓

top of activation record

↓

are to be shared between callee procedure and calling procedure

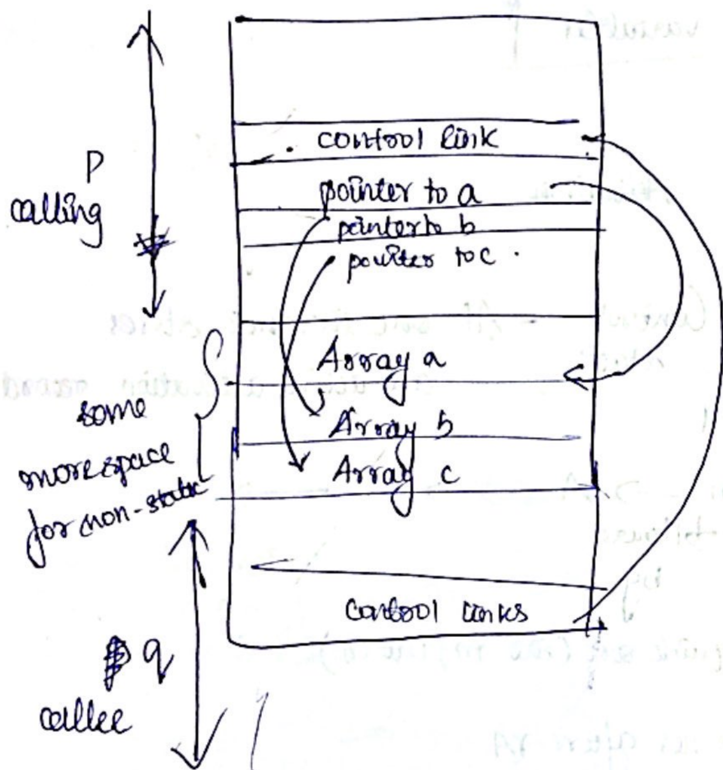
ex: actual parameters, return value -

callee q
calling p → p calls q

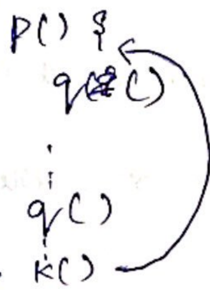
parameters to be shared
fixed links
dynamic links

activation record structure

2 pointers :
pointer to the top (end of AR) - end of dynamic links
" " of fixed link
↓
subtract the length of go to end to get values

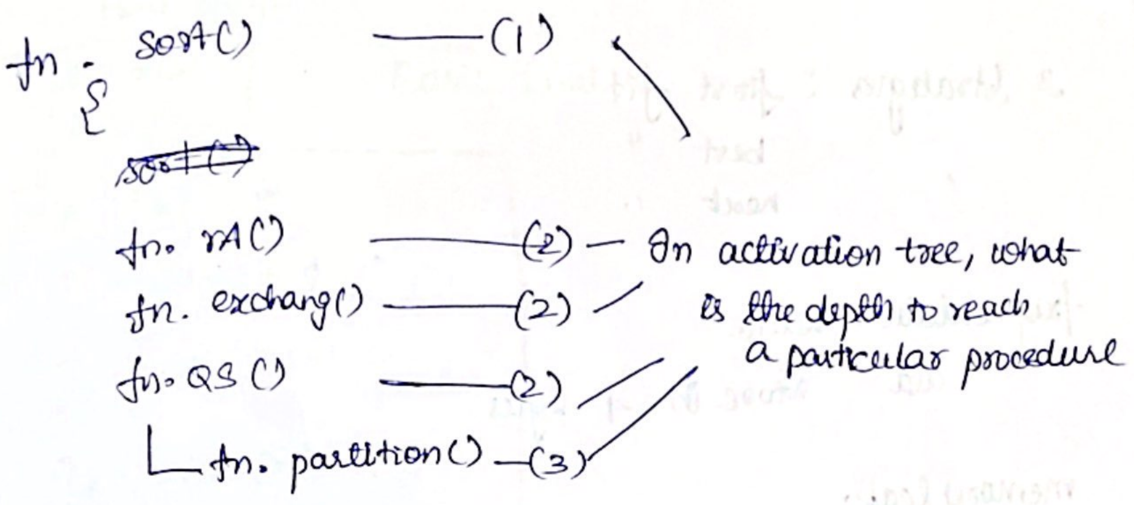


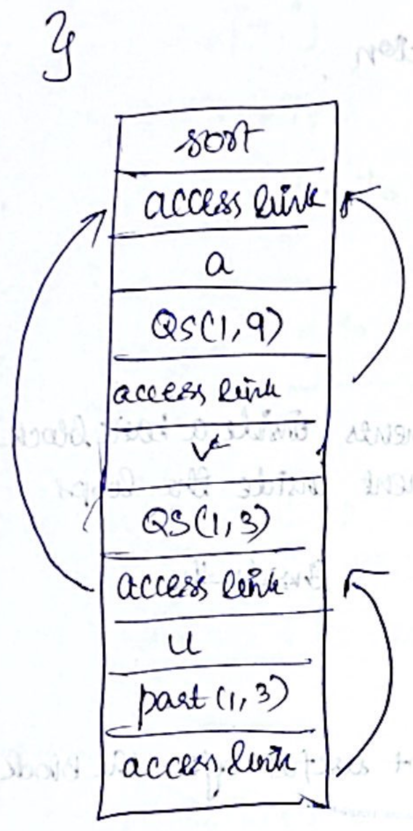
vA control stack :
Immediately after calling procedure,
A.R of callee proc



Let say, K - uses array a of procedure P ∴ goes to the access link
a proc K
pointer to a then fetches array

Using access link,
can go to any non local procedure

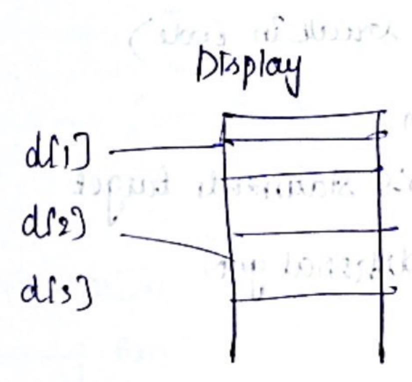




All the Q.S are under sort function

access link will point to the immediate nesting procedure

Display



temporal locality : stores the recently accessed items — has higher chance of being called next

spacial " : stores the items nearest to recently accessed

3 Strategies : first fit
best "
next "

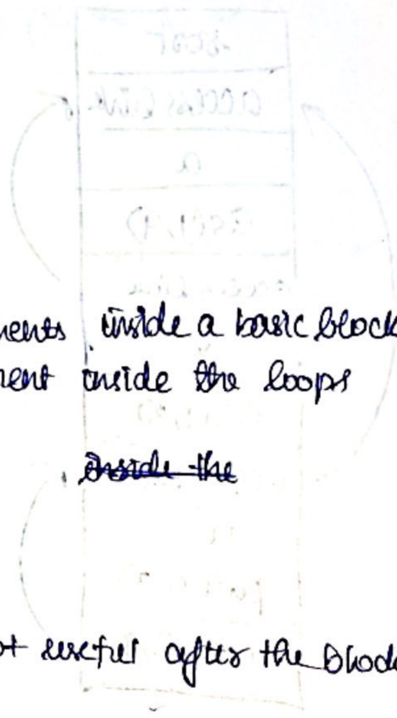
fragmentation reduce:
ala store in 4 bytes

memory leak:
not handling garbage collection.
so fragmented ∴ cannot use

Dangling pointer
excessive freeing of memory

24/4/24

Basic Blocks Eg



Code optimization → Minimise the no. of statements inside a basic block
Improve the no. of statement inside the loops

↓

code improvement → The ~~Inside the~~

no feasible approach
→ Dataflow analysis :
temp variables not useful after the block

Leader : (wherever there is break in code)

1. First statement of the prgm.
2. Conditional or uncond goto's statements target
3. Immediate next instr after conditional goto

```
prod = 0
i = 1
do
  begin
     prod = prod + a[i] * b[i];
     i++;
  end
while i ≤ 20
```

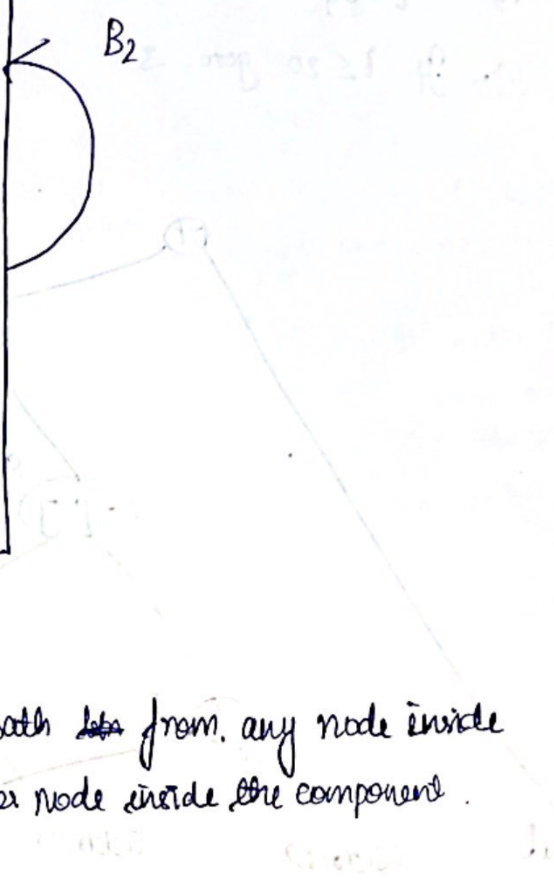
Scanned with CamScanner

1. prod = 0
2. i = 1

Basic Block B_1

3. $T_1 = 4 * i$
4. $T_2 = addr(a) - 4$
5. $T_3 = T_2[T_1]$
6. $T_4 = addr(b) - 4$
7. $T_5 = T_4[T_1]$
8. $T_6 = T_3 * T_5$
9. prod = prod + T_6
10. i = i+1
11. if i <= 20 goto 3

B_2



Strongly connected comp

There should be path from any node inside the component to any other node inside the component.

Flow graph:

→ Only one entry to go inside the component.
→ Strongly connected component

These conditions determines whether there is a loop in the flow graph.

→ 4*i need to calculated twice to write TAC but we optimized to one
→ Escaped using another temp var for assignment at ~~prod~~ (9)

3. $S_1 = 4 * i$
4. $S_2 = addr(a) - 4$
5. $S_3 = S_2[S_1]$
6. $S_4 = 4 * i$
7. $S_5 = addr(b) - 4$
8. S_6 ...

Scanned with CamScanner

9. $S_7 = S_3 * S_6$

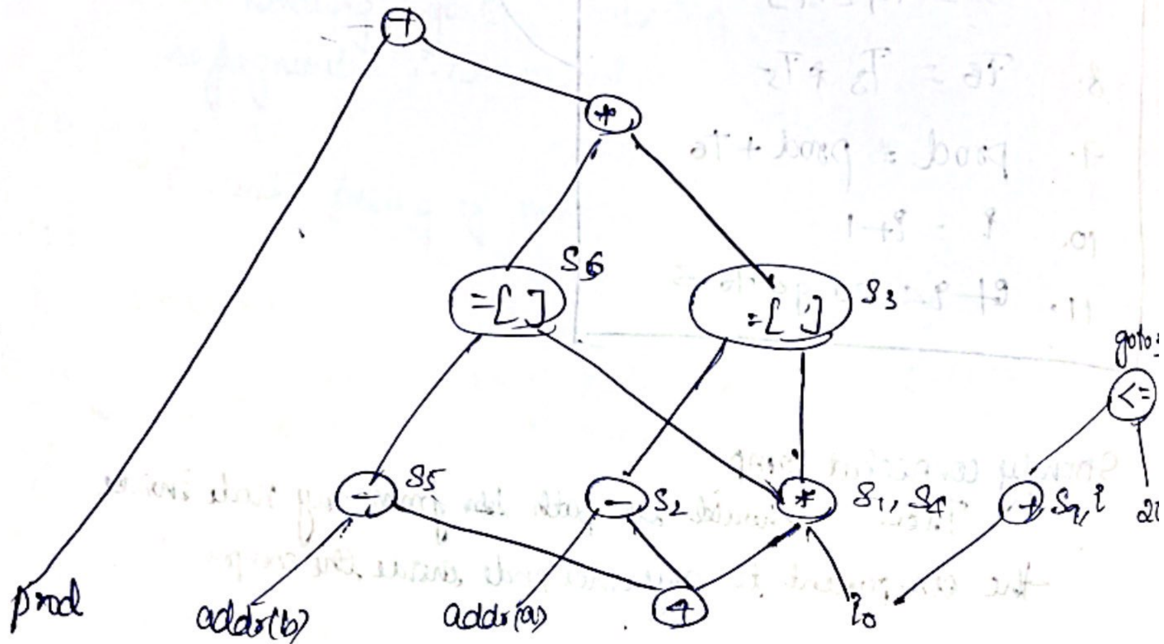
10. $S_8 = prod - S_1$

11. $prod = S_8$

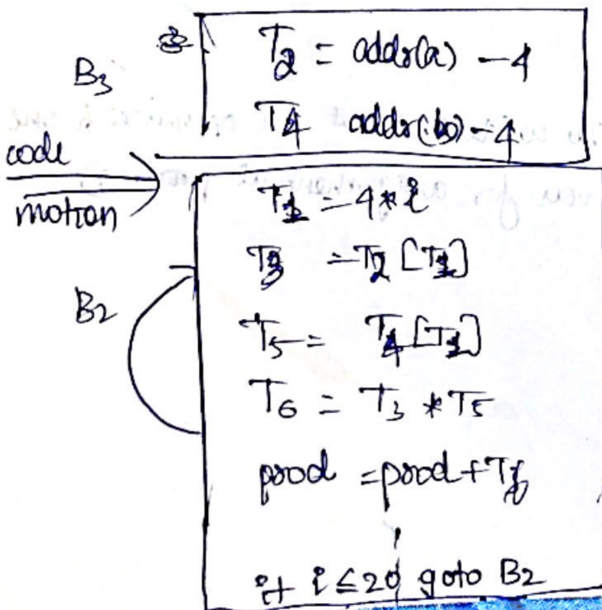
12. $S_9 = i + 1$

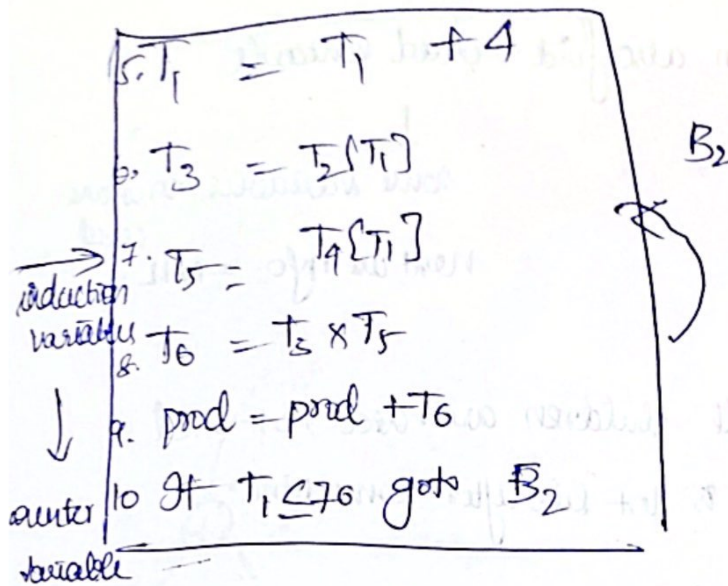
13. $i = S_9$

14. If $i \leq 20$ goto 3



Code motion





If
$i = 1, 2, \ldots\ldots 20$
$T_1 = 4, 8, \ldots\ldots 80$ ✗
only till 76

If $T_1 = 76$ it will go one more loop for 80

but 80 if gives T_1 then it will run one more iteration

$*$ – costly $\xrightarrow[\text{to}]{\text{converted}}$ $+$

B_3

26/9

Optimization : (Reductions)

→ code motion

→ induction variable

→ Reduction in Strength

→ Live variable analysis:

find the next use possible for the variable
↓
next use information

$i \; : \; x = y \text{ op } z$

line number i is the next use info

T_3 op T_5 – next use info – 8

prod, T_6 – " – 9

T_1 – " – 10

Scanned with CamScanner

Dead code elimination:

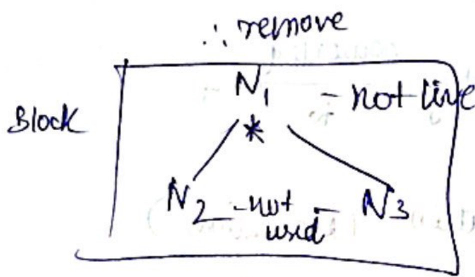
Live var analysis can also find dead variable

↓

some variable no where used

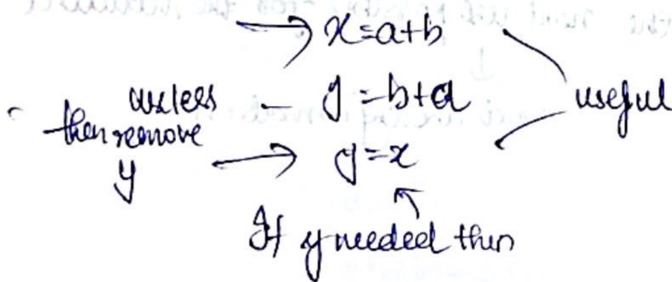
next use info = NIL

A block of code where all children are ~~dea~~ not used elsewhere & the parent is not live after some time ∴ remove the variable



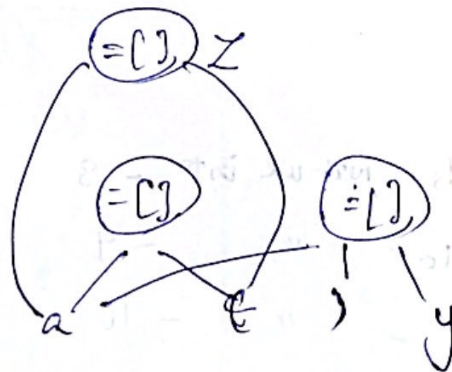
Use of Algebraic Identities

$i = i \times 1$ $\quad$ $x = x + 0$
$i = 1 \times i$ $\quad$ $x = x - 0$ $\quad$ } useless computations



$a = b + c$

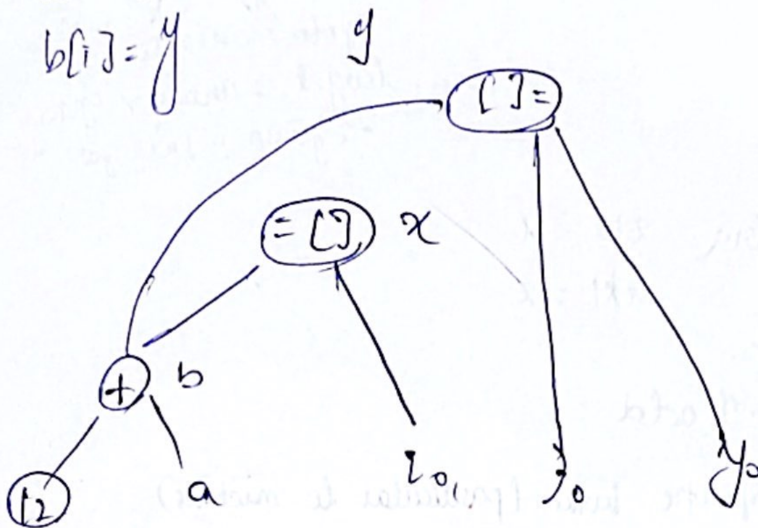
$e = c + d + b$



$b = 12 + a$

$x = b[i]$

$b[i] = y$



30/4/24

Loophole optimisation :

→ Eliminating redundant loads & stores (1)

→ Eliminating unreachable code (dead code) (2)

→ Flow of control optimisation (3)

→ Algebraic simplification (4)

→ Reduction in strength (5)

→ Use of machine idioms (6)

(1) LD a , R0
  ST R0, a
  ↓
  continuous statement
  • either of the statements may be removed

(2) if debug == 1 goto L1
  goto L2
  L1 print debugging info
  ↓ L2
  if 0 != 1 goto L2
  ↓ L2

• L1 will never be reached if debug is set to 0

• If L1 is not used anywhere, it can completely be removed

Scanned with CamScanner

(3) goto L1

L1 goto L2

⇓

goto L2

L2

Peephole optimization identifies jumps over jumps

↓

goto statements target is another goto → jump ii final goto

replace L1 with L2

(4) Apply rules of algebra $x+0 = x$

$x*1 = x$

(5) pow (x, 2) → x * x

x+2 → replace with add

(6) Inbuilt func to do specific tasks (particular to machine)

↓

underlying arch

↓

ISA

INC } increment/decrement
DRC }

instead of $x = x+1$ write INC x

Scanned with CamScanner

CODE GENERATION

Generated code should have the foll characteristics

- correctness
- easy to implement
- easy to debug
- efficient

Input to code generator : optimised intermediate representation

Assumptions

1- lexical, syntactic & semantic error ✗

2- code has been optimised (intermediate repr is efficient)

Output : target language (usually assembly)

Design Issues

→ input to code generator

→ target ~~issues~~ program
  ↳ underlying architecture : CISC/RISC
  ↳ resources of architecture

→ special case : JVM (evolved bcuz on a stack-based arch) → uses push/pop

stack-based arch: considered obsolete (almost)

JVM ↓ intermediate repr : bytecode ↓ directly convert to target (m/c) code, no intermediate (assembly) code created.

Instr selection

```
LD   R0  b        } a=b+c
ADD  R0  R0  c
ST   a   R0       → extra block as R0 contains a & can
LD   R0  a          be used directly
ADD  R0  R0  e    } d=a+e
ST   d   R0
```

limited no. of registers

need to know which registers are "safe" ← values don't change

Register allocation

allocation   assignment

identify values to be stored in the reg to concurrently perform op

which register should be assigned to which value?

↓

If a val is live, don't replace it → consider these constraints while generating code

Evaluation order

maintain flow as it is in original progm

should not give undesirable output.

→ A simple target machine model

LD

ST

OP ← ADD, SUB, MUL, DIV

BR

BLTZ r, L (if value of r is ≤ 0, Jump to L

direct addressing

4 ADD R_1, R_1, #100 ($R_1 = R_1 + 100$)

↓ immediate addressing   ↓ actual value 100

ADD R_1, R_1, R_2 $R_1 = R_1 + R_2$

LD R_1, i

MUL R_1, R_1, 8   ($i = i * 8$)

2 LD R_2, a(R_1) $b = a[i]$

ST b, R_2 R_2 = Content (a + content (R_1))   ↓ base ↓ offset   (index addr)

↑ indirect addressing

↓

3 LD R_1, *100 (R_2)

R_1 = content (content (100 + content(R_2)))

Register access has min cost

$R_1 \leftarrow R_1 + R_2$

$R_1 \leftarrow$ content (100 + content(R_2)) — 2

$R_1 \leftarrow$ " (content (100 + content (R_2)) — 3

↓ 3 content
access mem 3X

cost = 3

Exercises

1. $x = y - 5$
2. $a[j] = c$
3. $x = *p$
4. $*p = y$
5. if $x < y$ goto L

(3) LD R_1 p

LD R_2, 0(R1)

ST x R_2

Scanned with CamScanner

1/5/24

```
LD    R1 , a

ADD   R1 , R1 , R2          R1 <- R1 + R2

LD    R1 , 100(R2)          R1 <- content (100 + content(R2))

LD    R1   *100(R2)         R1 <- content (content(100 +
                                    content(R2)))

LD    R1   #100             R1 <- 100
```

Q:

```
a[i] = c                    *p = y
  ⇓                           ⇓

LD  R1, c                   LD  R1, p
LD  R2, i                   LD  R2, y
MUL R2, R2, 8               ST  0(R1), R2
ST  a(R2), R1
```

```
if (x<y) goto L

LD   R1 , x
LD   R2 , y
SUB  R1 , R2 , R3
BLTZ R1 , L
```

A Simple Code Generation

2 Data Structures used

→ Register descriptor — list of curr var in every reg

→ Address " — for every var, its location info

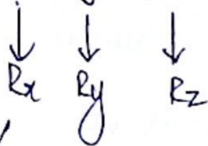
Scanned with CamScanner

Register access - min cost - unit 1

memory access > 1

∴ 2 register contents - 2 units

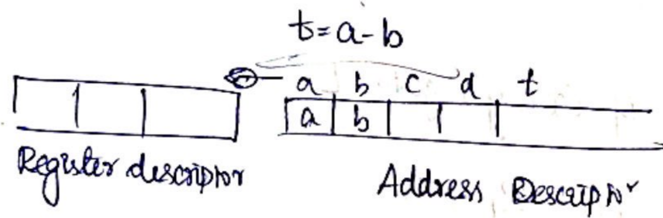
9+3 " - 3 units



Occ of Rx for any other var should be nil ∵ Rx should now be the address of x which is updated.

1) update register descriptor of Rx to hold only x & remove Rx from the address descriptor of every other variable.

2)

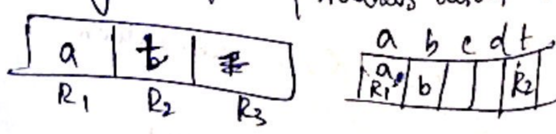


LD R1 a

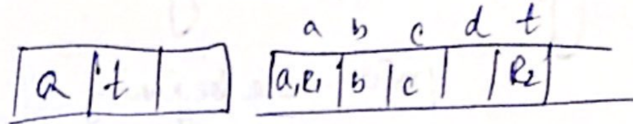
LD R2 b

SUB R2, R1, R2

update Reg descriptor & Address descriptor,

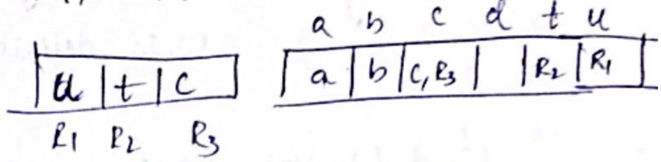


$u = a - c$



LD R_3 , c

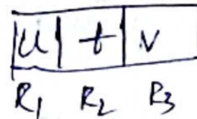
SUB R_1, R_1, R_3

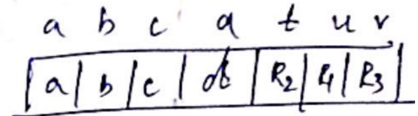


$V = t + u$

~~LD~~

ADD R_3, R_1, R_2

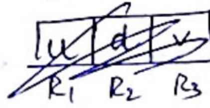


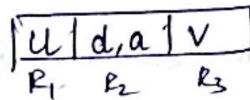


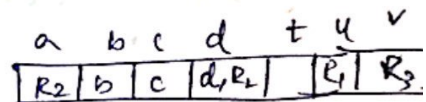
$a = d$

LD R_2 d

not doing any op for assignment, just changing descriptors







meaning both has same value

exit
While exiting store R_2 to a

ST a R_2

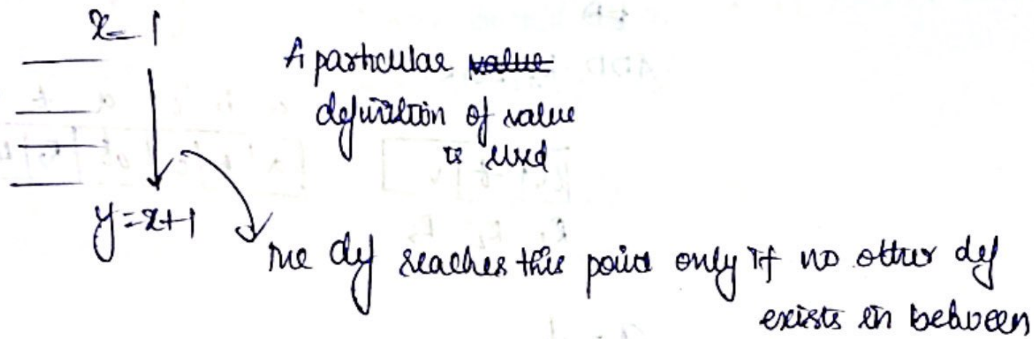
t, u, v are temp variables - no need of storing
The defined variables are need to be stored

Global Data Flow Analysis

point – the beginning of any statements or just after the statement

Used definition – when usage of a value which definition of value is active

Used definition (ud) chaining – find where the value is used which definitions are active



the def reaches this point only if no other def exists in between

Reaching definitions :

~~for~~ Every block the variables ~~that~~ are active at the start of the block ~~or~~ ~~all~~ at the end of block.

IN[ ] → definitions reaching before start of block

OUT[ ] → " " end of block.

GEN [B] → what are the definitions generated inside that blo

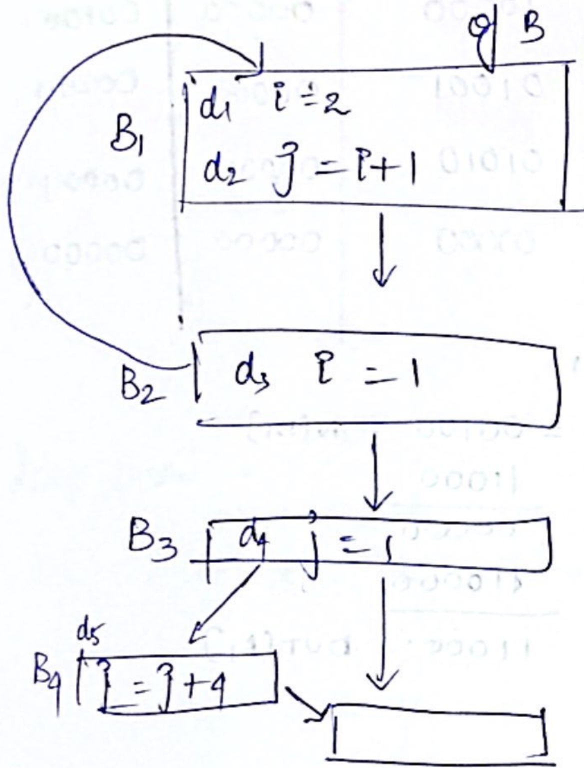
KILL [B] → what are the definitions that are killed due to re-definitions

Scanned with CamScanner

$OUT[B] = IN[B] - KILL[B] \cup GEN[B]$

$IN[B] = \cup OUT[p]$

p is predecessor blocks of B



A vector of all definitions active

	d_1	d_2	d_3	d_4	d_5
$GEN[B_1]$ =	1	1	0	0	0
$KILL[B_1]$ =	0	0	1	1	1

Kill all other def of i, j

$GEN[B_2]$ =	0	0	1	0	0
$KILL[B_2]$ =	1	0	0	0	0

$GEN[B_3]$ =	0	0	0	1	0
$KILL[B_3]$ =	0	1	0	0	1

Kill all other def of j

$OUT[B_1] = 00000 - 00111 \cup 11000$

$A - B = A \wedge \neg B$

```
  00000
∧ 11000    AND
-------
  00000
∨ 11000    OR
-------
  11000
```

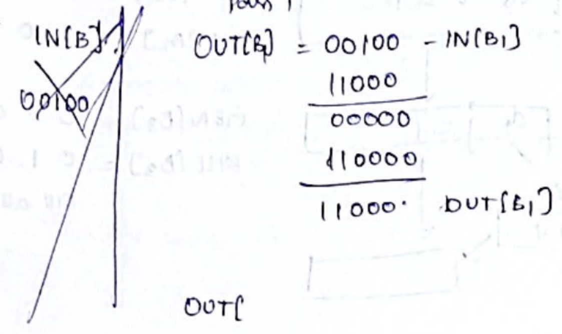
$OUT[B_1]$ = ~~10000~~ 11000

$IN[B_1]$ = 00100 - 100

$OUT[B_2]$ = 00100

Block	GEN[B]	KILL[B]	Pass 0 IN[B]	Pass 0 OUT[B]	Pass 1 IN[B]	Pass 1 OUT[B]	Pass 2 IN[B]	Pass 2 OUT[B]	Pass 3 IN[B]	Pass 3 OUT[B]
B_1	11000	00111	00000	11000	00100	11000	01100	11000	01111	11000
B_2	00100	10000	00000	00100	11000	01100	11111	01111	11111	01111
B_3	00010	01001	00000	00010	00100	00110	01111	00110	01111	00110
B_4	00001	01010	00000	00001	00110	00101	00110	00101	00110	00101
B_5	00000	00000	00000	00000	00111	00111	00111	00111	00111	00111

Pass 1



OUT[B] = 00100 - IN[B1]

11000
00000
11000
11000· OUT[B1]

Backward flow problem

→ live variable analysis

IN[B] = OUT[B] - DEF[B] ∪ USE[B]

OUT[B] = ∪ IN[S]
S ∈ successor blocks

⇕

IN[B] = $\bigcup_{P}$ OUT[P]

OUT[B] = IN[B] - KILL[B] ∪ GEN[B]

Loop unrolling :

Increase the body of loop & decrease the no. of comparison

```
for (I=1 to 100)              for (I=1 to 100
{                     →       { A[I] = 0
  A[I]=0                        I=I+1
}                               A[I] = 0
```

Loop jamming:

Same loop indices but diff body ∴ combine

```
for I=1 to 10
  for J=1 to 10
    a[I,J]=0                  for I=1 to 10
                      →         for J=1 to 10
for I=1 to 10                     a[I,J]=0
                                a[I,J]=1
  a[I,J]=1
```