ANIKA ISLAM (21101293)

CSE420

LECTURE 2

SATURDAY

DATE: 30/09/23

* Symbol table contains variables.
* Compiler add all the key words in the symbol table.

* Lexical Analyzer does scanning character by character.

"result" is checked character by character and is identified as a meaningful token.

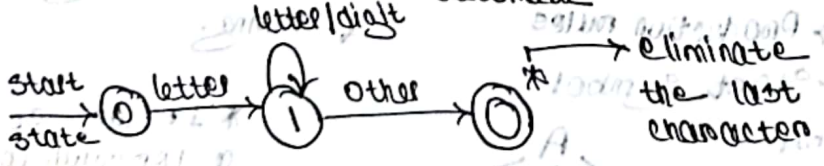
However, "result=" is not a meaningful word, so we back track by 1 word and get "result" and eliminate "=". Thus, we get a meaningful token.

* We use regular definition in compiler to denote regular pattern.

* identifier ⇒ letter (letter | digit)*

* The regular expression/definition are further converted into automata using lex compiler.

lex.l —lex compiler→ y.y.c (automata)



Token is denoted in the below format:

⟨ token name, token value ⟩

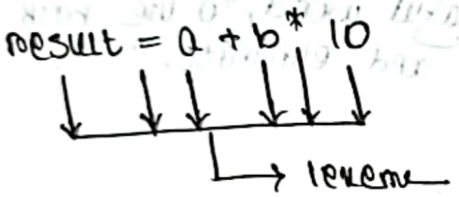
* token name of a particular identifier is id.
* Identifiers are selected from symbol table. Symbol table holds the token value of that identifier.

* Only substrings are present in symbol table.

* No number/mathematical operators are present in the symbol table. So, the number/mathematical operators themselves are the token value.

* Lexical analyzer does not work on syntax level.

* Lexeme → sequence of character that matches a pattern.
  → before tokenization.
  → eg "result="

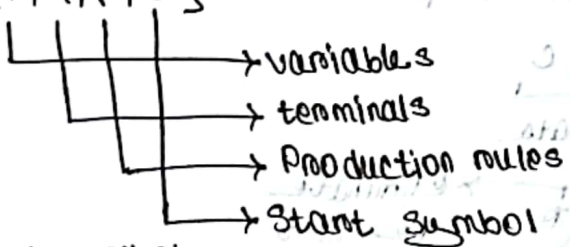


<id,1>, <op,=>, <id,2>, <op,+>, <id,3>, <op,*>, <num,10>

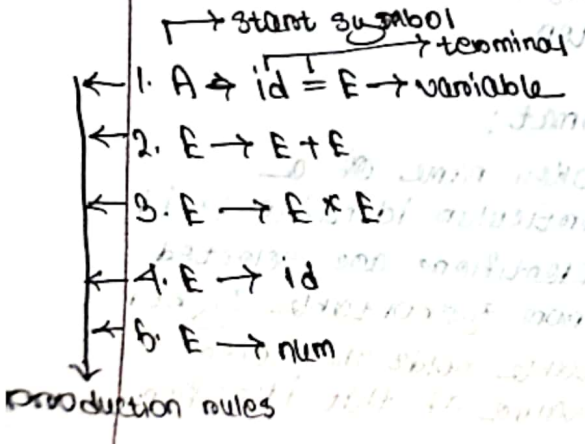
Syntax Analysis

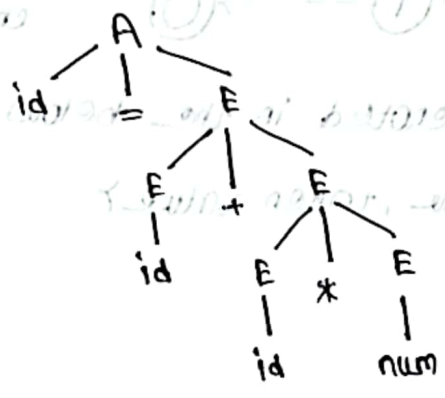
* We use CFG. From the CFG, we produce parse tree.

* G = {V, T, P, S}



* parse tree are generated by grammar
* To produce parse tree, we need grammar & string.



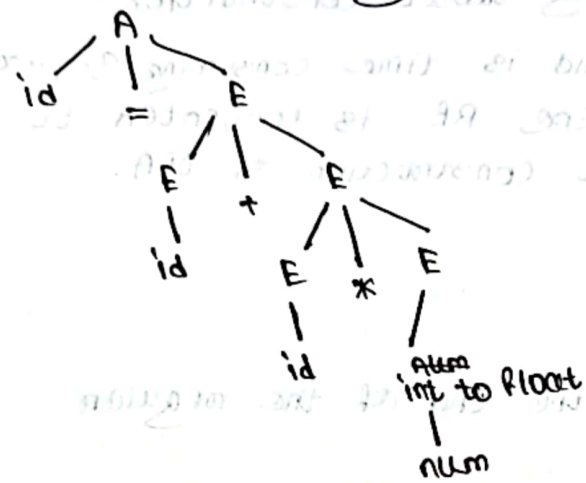


* If we get a terminal, we check it in the token produced by lexical analyzer
* If we get a variable, we split it using the other production rules.

Semantic Analysis

* The parse tree produced by syntax analyzer goes to semantic analyzer.

* Datatype can be changed here.



Intermediate code Generation

3 address code

- value
- identifier
- temporary variable → registers of hardware

Code Optimizer

* Temporary variables can be removed.

LECTURE 3

THURSDAY DATE: 05/10/23

LECTURE 2 : LEXICAL ANALYSIS

Convension of RE to DFA :-

① Convert RE to DFA

② Convert DFA to NFA using subset construction.

This takes more space and is time consuming, so, we can use "direct method" where RE is converted to DFA without going through subset construction of DFA.

We need to add # at the end of the regular expression —

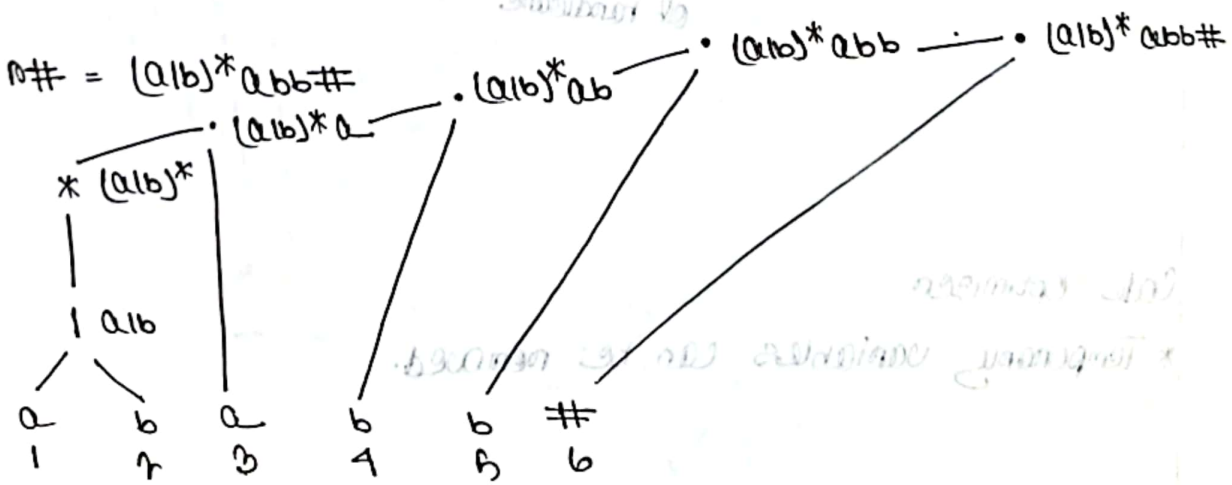
Precedence of regular expression operations:—

* → have only one child (one hand)

•, | } have two children (left hand + right hand)

* The expression in bracket will be execute first.

* We use the bottom-up method for a syntax tree.



* If multiple operators are present, then we follow the leftmost method.

* We have to give position numbers to leaf nodes. * We don't mark

* The position no. in syntax tree is the state no. in NFA.

* The outgoing arrow from that digit, marked state holds the input symbol from the leaf node of syntax tree.

* Digit marked state is important state which has at least ε transition and move to a DFA.

* Letter marked state is non-important state which does not have any ε transition and cannot move to a DFA.

* "#" is concatenated at the end of a string so that the end state is a letter marked state and the previous state is a digit marked state.

We have to implement 4 functions over the syntax tree.

nullable (n) → represents the current node
→ it's a binary function

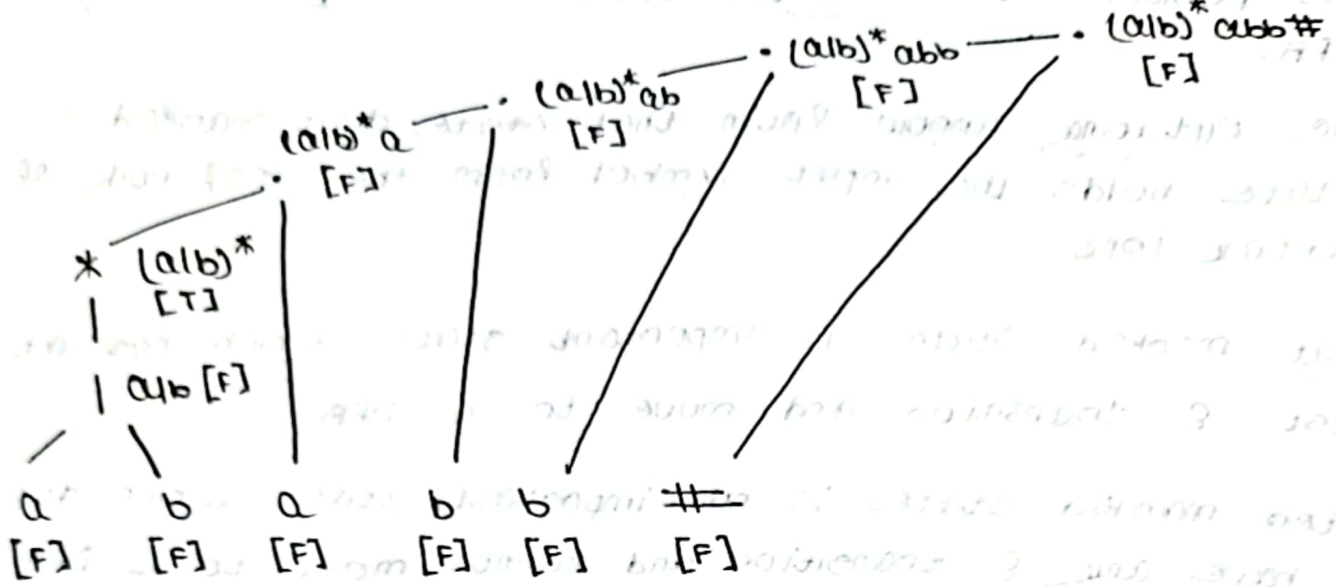
firstpos (n)

lastpos (n)

followpos (n)

* nullable : takes a leaf node and checks whether it gets a ε input or not.
If there's ε, then it returns True.
Otherwise, it returns False.

  * If any one of child gives nullable, then '|' becomes nullable.
  * '*' is also nullable as * always gives ε.
  * If both child gives nullable, then '.' becomes nullable.



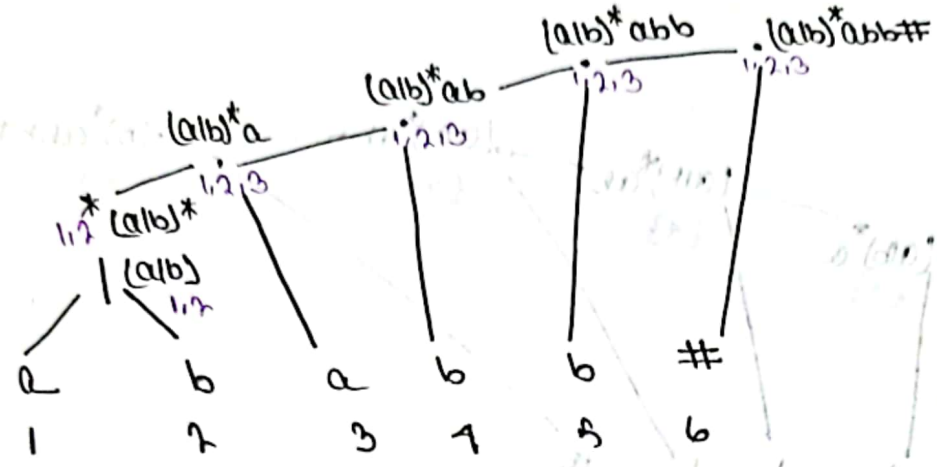
F → no ε transition (not nullable)

T → ε transition (nullable)

Continues LECTURE 4

SATURDAY DATE: 07/10/23

First pos



* For example, (a|b)*.a :- left child has *, so we have to concatenate parent of * (a|b) and right child (a), to get (a|b|c). So, first pos will be 1,2,3.
* First pos of (a|b is 1,2 and of * is 1,2.

In first pos, we take a node and return the possible position number of the left nodes for strings which can be created from the available letters.

First pos starts with an input symbol.

Rules of first pos:-

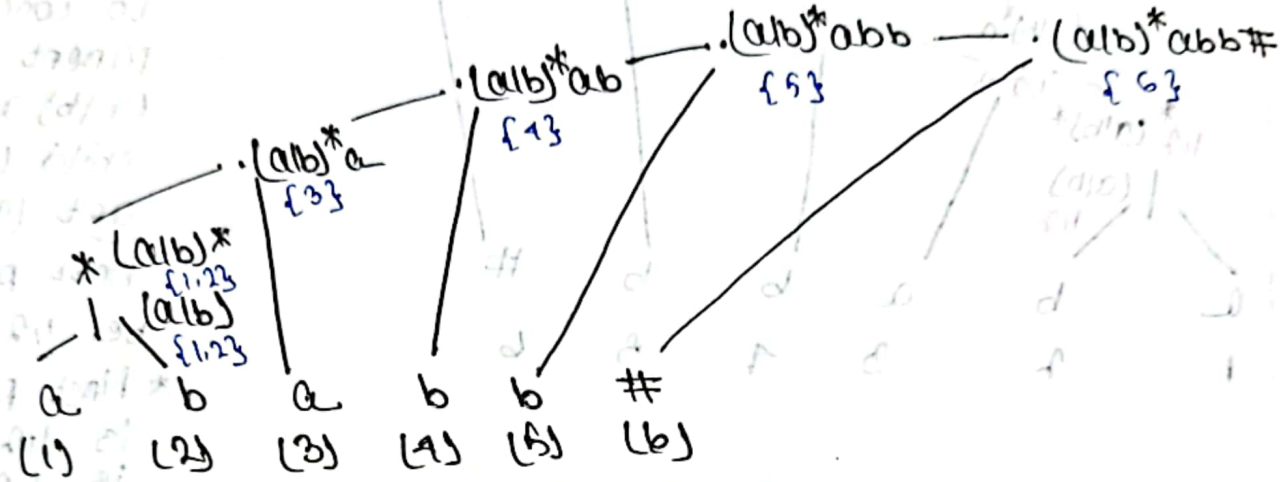
(1) First pos of any leaf node will be its position number.

(2) In case of union (|) operators, first pos will be the first pos of both of its child.

(3) In case of * operators, first pos of * will be the first pos of its only child.

(4) In case of cat (.) operators, if the left child is nullable (has *), then we have to concatenate the left child and right child and get the first pos of them. Otherwise, we have to get the first pos of the left child.

Last pos



Last pos is the last position number of the symbol.

Rules of last pos:-

(1) Last post of any leaf node will be its position number.

(2) In case of union (|) operator, last pos of union will be the last pos of both the children.

(3) In case of * operator, last pos of * will be the last pos of its only child.

(4) In case of cat (·) operator, if the right child is nullable, then last pos of cat operator will be the last pos of both its children. Otherwise, the last pos of right child will be the last pos of cat operator.

Follows pos

We look for * and cat (.) operations.

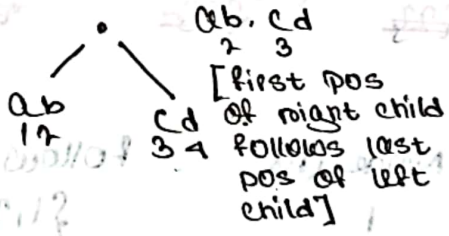
For * operation, the symbols present in the operation will be distributed to all its leaf nodes.

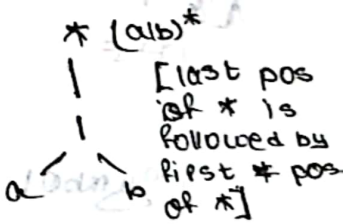
For cat (.) operation, the left last pos of the right child is the follows pos of the left child.

* Follows pos is the adjacent symbol in position number.

Node	follow pos
1	1,2,3
2	1,2,3
3	4
4	5
5	6
6	∅

* Follows pos is the position number of any symbol that follows a symbol in a possible combination.





* First pos, last pos and follows pos of ε will be ∅ as it does not have any position number.

LECTURE 5

THURSDAY

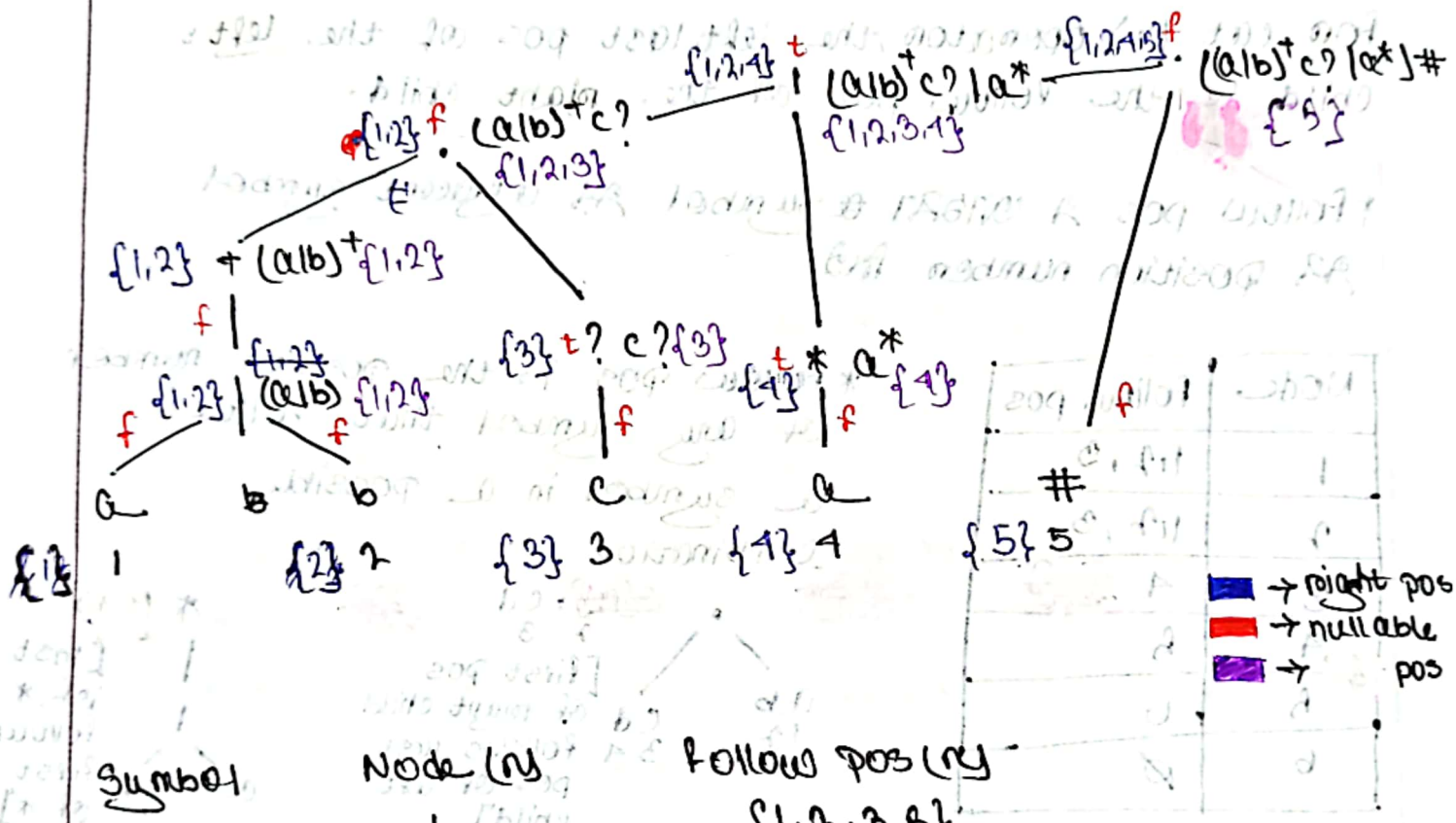
DATE: 12/10/23

$(a|b)^+c?|a^*$

$(a|b)^+c?|a^*)\#$

$\Sigma^+ = \Sigma^* - \epsilon$

$n? = n|\epsilon$

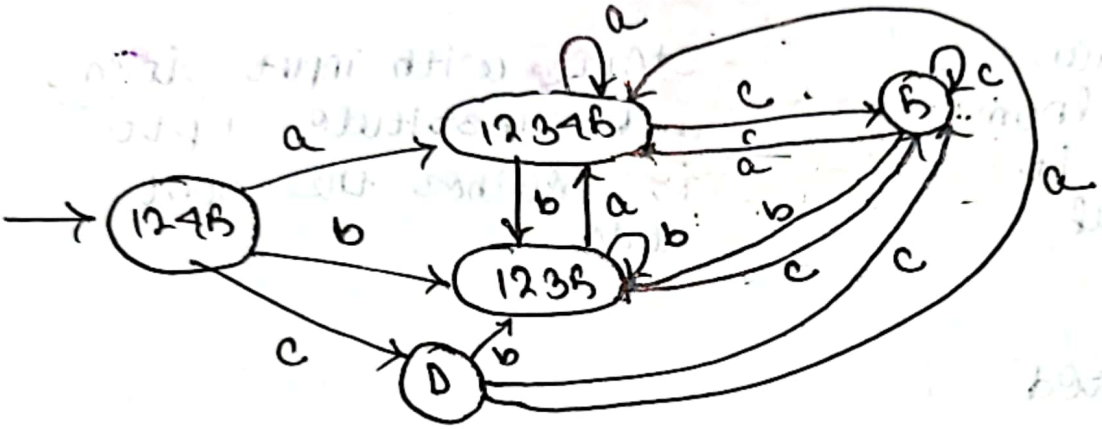


Symbol	Node (n)	Follow pos (n)
a	1	{1,2,3,5}
b	2	{1,2,3,5}
c	3	{5}
a	4	{4,5}
#	5	∅

Starting state : First pos of root node

Transition for input = number of symbol

The state of a specific symbol is its follow-pos or the union of all its possible follow pos.



COMPILER DESIGN
LECTURE 6

FRIDAY

DATE: 13/10/23

PARSING (SYNTAX ANALYSIS)

Parser

Top down parser

- works as human
- takes a grammar and substitutes from root node upto it reaches the leaf nodes
- not viably accepted as it accept very small group of grammars.

Bottom up parser

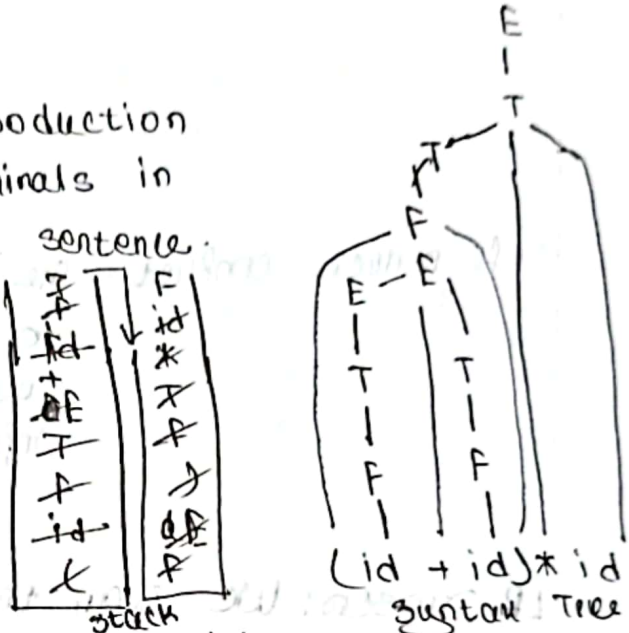
- opposite
- starts with input strings and substitute upto it reaches the root node.

Bottom up parser

input string : (id+id) * id

- traverse from left to right
- ~~check for string wit~~
- check for production rule containing the traversed part only.
- we ~~substitute head~~ replace ~~head~~ body of the production rule with the ~~body~~ head → production.
- ~~We~~ The traversed part which get reduced is called a handle.

- We use a stack to store the traversed part of the input string.
- Each line of derivation process is called sentential form
- The last state of the production rule contains only terminals in the body and is called sentence.



Sent
(id+id)*id

id gets replaced by F and is popped out from stack

(F+id)*id

F is replaced by T and F is popped out off stack as F is the handle.

(T+id)*id

T is replaced by E as T is the only handle with E as head.

(E+id)*id

E is not a handle so we traverse further E+ is not only present in any body.

(E+id)*id

id acts as the handle in the production body of F+id, so is replaced by F.

(E+F)*id

F acts as handle in the production body of T, so

(E+T)*id

(E)*id

E+T is reduced to E as it has the longest string in production body.

E*id

(E) is replaced by F

F*id

F is reduced to T

T*id

T can be reduced to E, but doing so our syntax tree will not match. So, we shift to * and id. We use id as the handle and reduce id to F to get the correct syntax tree.

T*F

T*F is reduced to T | T → T is reduce

Shift reduce conflict - The parser cannot decide whether to shift to the next input symbol or reduce the traversed input symbol.

LR parser: We scan the input string from left to right → L
We do the right-most derivation in reverse → R

Ambiguity in bottom up parser -

Reduced - reduced conflict: We cannot figure out which production rule to reduce ~~to get the need for~~ a traversed part.

Handle pruning: Repeating changing the handle in the parser.

Shift - Reduce parser : Bottom - up parser is called shift - reduce parser.

Steps for parser →
- shift the input symbol
- reduce the existing traversed input symbol to the production head.

Shift - reduce conflict : We can either use the traversed part as the handle and reduce it to production head OR shift the input symbol and then reduce it.

LECTURE 7

SATURDAY

DATE: 14/10/23

SLR (1) PARSING

SLR (1) Parsers → look ahead symbol

↓ Simple

- category of bottom up parsers
- can only look ahead of one symbol during traversal.

* input is kept in temporary memory
* A '$' symbol indicates the end of a string.
* Parse Table = Action Table + Goto Table
  - Parse Table: ↓ constructed by automata
  - Action Table: ↓ terminals of CG
  - Goto Table: ↓ non-terminals of CG

Item — Any production rule having a '.' in the production body.

- '.' is the position number and anything before the '.' is part of the parse tree.
- We have to work for the remaining part after
- Item in LR(0) automation is LR(0) item.

LR (0) AUTOMATON → Semi DPA Automata

Closure – Takes an item and gives a state

CLOSURE (I) = I_K

Rule :-

① I must be in I_K

② I_0 $A \rightarrow \alpha . B\beta$ (α → is already been parsed)

(conventions [strings of CG])

$B \rightarrow . \gamma$ [is in I_K]

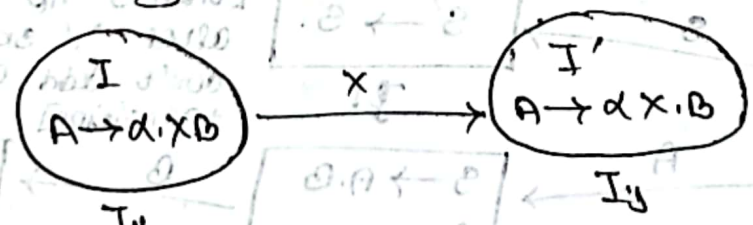
Keep on calling rule ② upto there's a terminal after the '.'.

Automata → states → closure helps to compute states

Automata → transitions → GOTO helps to compute transition

GOTO – GOTO (I_K, X) = I_y

We go from I_K state to I_y state.



*α, β are used to denote string of Grammar symbol

['.' do shifts one place right to denote X is part of parse tree]

– state is not changed when '.' is at the end of the production body.

① Augmate the grammar with a new start symbol

- create new production rule

② Call CLOSURE( ) function

$S' \rightarrow S$
$S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

CLOSURE (I_0) = I_0

I_0 $S' \rightarrow .S$

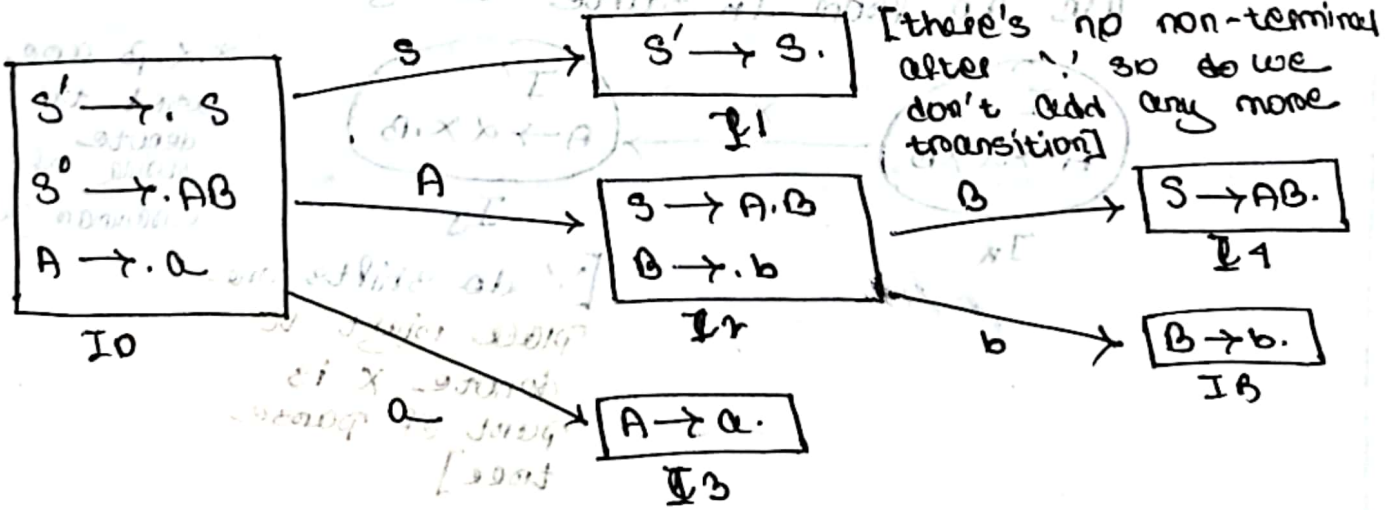
$S \rightarrow .AB$

$A \rightarrow .a$

[CLOSURE ( ) is done as there's a terminal after '.']

③ Call GOTO( ) function

- Three transitions in closure.



* Items where the '.' is the need of the production body, that items

LECTURE 8

THURSDAY

DATE: 19/10/23

SLR (1) Parse table.

* S' is not included in the parse table as it is not included in the CFG

Action (i, a) → i: state, a: terminal

Operations can be

- **shift**
  - if we encounter an input symbol, there's a shift
- **reduce**
  - if the dot is at the end of production body, the items are reduced state.
  - state
    - in which state reduced occurs
  - input symbol
  - terminals
    - act as a look-ahead symbol
- **accept**
  - first state is the accepting state.
- **error**
  - blank entries (where no shift/reduce occurs)

eg. $S \to AB.$ (I4) → state no.

$S' \to S$ (0)
$S \to AB$ (1)
$A \to a$ (2)
$B \to b$ (3)

→ production rule

For $A \to a.$
row = 3 → state no.
column = b → ~~production no.~~ follow of A

~~state~~ production no. = R2

First (α) ← string of grammar

- generate set of terminals

$S \to AB$ $A \to a$ $B \to b$ FIRST (AB) $= \{a, b\}$	$S \to AB$ $A \to a \mid \epsilon$ $B \to d$ FIRST(AB) $= \{a, d\}$	$S \to AB$ $A \to a \mid c$ $B \to d$ FIRST(AB) $= \{a, c\}$ FIRST(ϵB) $= \{\epsilon\}$

① If α is beginning with terminal, that is the first of α.

② If α has no terminal at the beginning, then we have to check what the first non-terminal produce. If it produce a terminal, then that terminal is the first of α.

③ If α has non-terminal at beginning and it produces a ϵ and nothing else or α has nothing in its beginning, then ϵ is the follow pos.

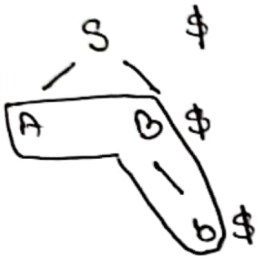
FOLLOW (A)

- generate a set of terminals
- The look ahead is the follows

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$



Follow (A) $\rightarrow$ {b}

A will be followed by b

$AS \rightarrow ABd$

$A \rightarrow a$

$B \rightarrow b|c$

Follow (A) = {b, c}

$S \rightarrow A$

$A \rightarrow a$

Follow (A) = $

$S \rightarrow Bd$

$B \rightarrow A$

$A \rightarrow a$

Follow (A) = d

Follow for a non-terminal is the immediate next terminal

$S \rightarrow ABd$

$\rightarrow$ we can denote the remaining part after A to be β

① Follow of A is the first of β

Follow of S is $

② Follow of any non-terminal cannot be ϵ

③ If β is ϵ, then we have to check the head of the production rule. The head becomes the follow.

Online
LECTURE 9

SATURDAY

DATE: 21/10/23

Why do we prefer to use LR(1) instead of SLR(1) for parsing?

SLR(1) parser uses LR(0) Automation, whereas LR(1) parser uses LR(1) automation.

Similarity: LR(0) and LR(1) are semi-deterministic automation.

Difference: LR(0) contains LR(0) items $[A \to \alpha .]$

LR(1) contains LR(1) items $[A \to \alpha ., a]$

core, lookahead

* ~~In LR(1),~~ there is one lookahead symbol and that's why there is 1 in LR(1). and 0 in LR(0).

LR(1)

CLOSURE (I) = I_K

* closure takes an item (I) and returns a state (I_K) where K is an integer number from 0 to n.

(1) I is in I_K

(2) $I : A \to \alpha . B\beta, a$ → comes from follows set of A

↳ there's a non-terminal after dot, so we have to add the production rule for the non-terminal in I_K and '.' is put before the production body.

$B \to .\gamma, b$

↳ need to find lookahead for B to add it in I_K.

↳ one/more steps

$S^* \Rightarrow A$

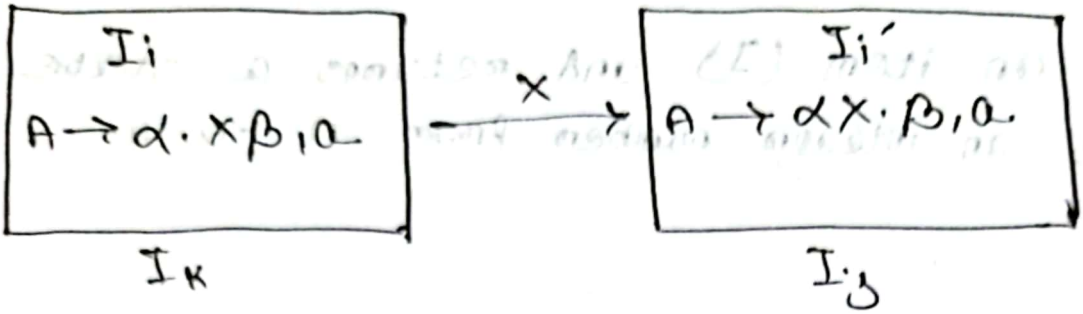
$S^* \Rightarrow SAax$

need to expand; + is already expanded in terminal forms

$\Rightarrow \delta\alpha B\beta ax$

Lookahead of B = First(βa) = b

GOTO(I_K, X) = I_j



* When we are changing state from I_K and I_j, lookahead does not change.

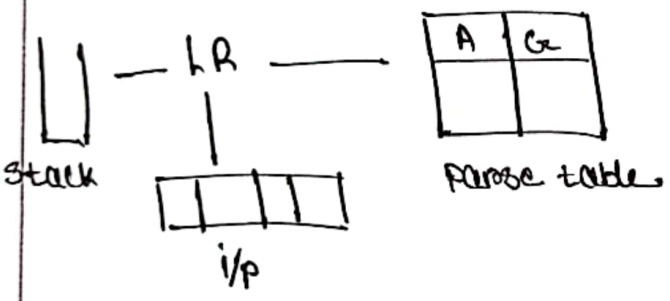
(online) LECTURE 11 | SATURDAY | DATE: 28/10/23

* যখন A একটি state কে reduced form এ থাকে, A reduced form এর item এর operation কে record করে।
* একই non-terminal এর follows এর জন্য GOTO table. এমনভাবে, if we encounter

rows = state no.

column = production rule no.

↳ lookahead দেখে decide করে



Parsing Algorithm

pointers { Stack — s
i/p — a

* Every iteration parse table এ যাব, action table এ কি লেখা রয়েছে তা দেখব

while {

action (s, a) → if shift

action (s, a) → if reduce

action (s, a) → if accept

action (s, a) → if error

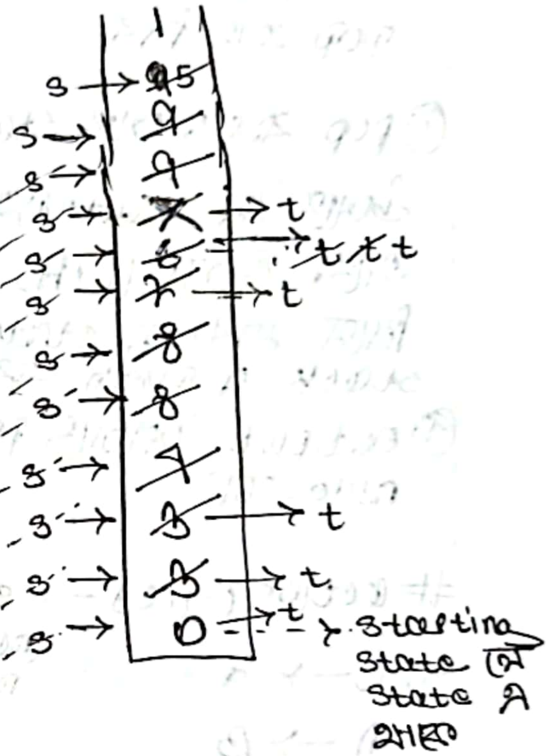
// update s

}

* action table এ ৭ টা কলাম হয়, তাই ৭ টা বার action call হয়েছে

Simulation of LR(1) Parser

ccdcd$ → input string
↑
a



action (0, a) ~~→ s3~~

* Action table এ দেখব (0, a) এ কি আছে
(a = c)

(0, a) → ~~if~~ shift 3 (s3)

* shift হলে কিছু কাজ করা লাগে :-

① যে state এ যাব, state টা stack এ push

② i/p pointer (a) টা বাড়াব

input
ccdcd$
cdcd$
dcd$
cd$

State = 3 টা stack এ push

ccdcd$
↑
a = c

action (3, a) → shift to c. state 3 (s3)

state = 3 টা stack এ push করব

ccdcd$

action (3, d) → s4

state = 4 টা push করব

ccdcd$
↑ a

action (4, c) → r3

* reduce using rule = 3

3. C → d

* reduce হলে

① কোন rule দিয়ে reduce করবি, তারে $A \to \beta$ দিয়াতে থাকে

② β তে যতটা symbol থাকে, ততটা symbol stack থেকে pop করে দিবি

③ pop করার পরে stack এর উপরে যে state টা থাকবে তারে GOTO(t, A) তে দিয়ে দেখবি, যে state পাবি stack এ push করবি

GOTO(t, A) → production rule এর need
stack এর top after pop

④ output হিসেবে production rule টা

o/p → 1. C → d
2. C → cC
3. C → cC
4. C → d
5. C → cC
6. C → d

#action(4, c) → r3 (reduce using rule = 3)

3. C → d

$A \to \beta$

$\therefore \beta = d$

cardinality বা $|\beta|$ include করবি, কয়টা symbol আছে count করবি

Here, $|\beta| = |d| = 1$ → একটা state pop করে দিবি

GOTO(t, A) → GOTO(3, C) → state = 8

i/p এর ccdcd$

production rule: C → d

C
1

ccd এর পরে d দেখে C দিয়ে reduce করা লাগে ccdcd$

action (8, $) → R2
(reduce using rule)

2. C → c C

A → β

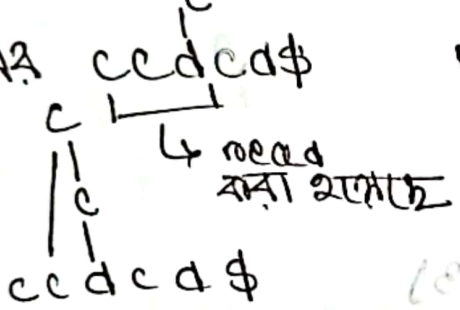
β = cC

|β| = |cC| = 2 → 2 ta state pop করবে

t = 3

GOTO (3, C) → 8

i/p AS ccdcd$ production rule
2. C → cC

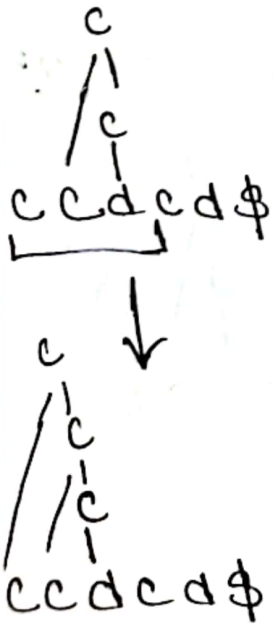


read করা হয়েছে

action (8, C) → R2 (same production rule as b4)
2 ta state pop 2a

GOTO (0, C) → 2

production rule
2. C → cC



action (2, c) → 36
(6 এ push করতে হবে)

ccdcd$
↑
a

ccdcd$
reduce করা যাবে
a=c

action (6, d) → 37
(7 এ push করতে হবে)

ccdcd$
↑
a

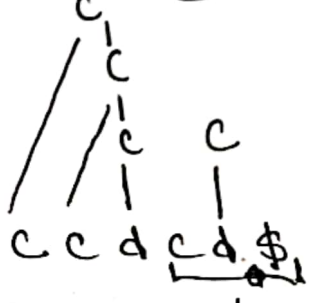
action (7, $) → r3
(reduce using rule = 3)

3. C → d
A → B

|B| = |d| = 1 → 1 টা state pop out হয়ে যাবে

t = 6

GOTO (6, C) → 9



↳ read করবি

~~# action (9, $) → r~~

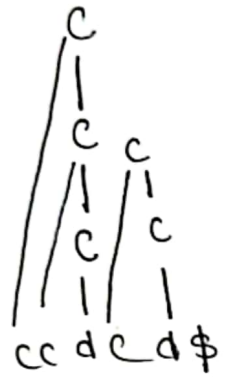
action (9, $) → r2
(reduce using rule 3)

3. C → cC

|β| = 2 → 2 টা state pop out করবে

t = 6

GOTO (6, C) → 9



action (9, $) → r2

3. C → cC

β = 2

t = 6

GOTO (6, C) → 5

action (5, $) → r1

1. S → CC

β = 2

t = 0

GOTO (0, S) → 1

* reduce + shift করতে করতে ও আসলে i/p ২ $ ৭ stack ২ 1 আসলে → STOP

LAB 1

MONDAY DATE: 09/10/23

$x = 42 + y$

~~* meaning~~

* lexeme
  → smallest meaningful element
* token – category

print ("5+3", a)
(1) print, (2) "5+3", (3) a
Total token = 3

+ – + + =
1 2 3 4
Total token = 4

* Regex এ second brac দেওয়া লাগবে না
* Regex definition এ second brac দেওয়া লাগবে

yylex()
↳ input file থেকে প্রত্যেকটা জিনিস token হিসেবে read করে

① Definition
② Rules & Action
③ User code → main function

Folder create করে input file + student version file নিবে। Right click করে git bash এ click করবে।

flex <lex file_name> → code convert (compile) করে → lex.yy.c create করে

g++ lex.yy.c -o <desired file_name> → compiler এ যাবে file যদি ঠিক তো exe file এ convert করে
↳ exe file create করে

./<desired file_name>.exe <input_file> → exe file input এ file এ দিবে to read exe file.
↳ input file read করে using lexical analysis from .exe file

- %option noyywrap → prevents wrapping of input stream when it reaches the end
  → needed for single file
  → returns 1
  - when the input reaches its end, this function starts reading from the very beginning.
  - can control parsing
  - can shift to new file.
  - returns 0

- #include<bits/stdc++.h> → C++ library import করেছি

- using namespace std → std initiate করে দিলে, std::cout এর জায়গায় just cout দিলেও পারি

- ofstream yyoutlog → out file create করে
  ↓
  (ofstream object)
  → যত গুলো token read হচ্ছে তাদের info write করে এই file এ

- string logList = " " → empty C++ string
  → token গুলো নিবে and এই string এ token + token info store হবে
  → এই string টাকে yyoutlog এ দিয়ে write হয়

- { } → regular expression is defined
  - → identifiers, digit, binaries, hexadecimals, floats, tabs, newlines
  - → pattern match হলে necessary action নেয়া define করি
- < , > → Token Type
- (Lexeme) → actual lexeme টা
- yytext → contains the actual lexeme (যে text pattern follow করেছে regex এর)
- loglist = " . . . . " → loglist এর মধ্যে যা store করতে চাই লেখা
- yyoutlog << loglist → loglist এ যা যা store করেছি তাকে, to yyoutlog file এ write করছি

input → in → lexer → matches pattern of lexeme → token is created <token, value>
↓
Output

- %% → 2 টা section এর মধ্যে boundary create করি
  - → rules and actions stage & definitions stage এর end এ দিই

- yylex() → used to perform lexical analysis
  - → scans input character wise
  - → identifies & matches patterns
  - → token এর সাথে token value return করে

LAB 2

MONDAY DATE: 30/10/23

Syntax Analysis

→ Lexical analysis থেকে পাওয়া token এর জন্য check করে কোন rule এ আছে and then parse tree generate করে।

→ Token এর জন্য rule print করো (Lab 2 task)

eg.

extended version
আছে, এর declaration rule
তাতে
non-terminal

terminal

int ab;

INT ID SEMICOLON → ~~expres~~ var_declaration: type_specifier declaration_list SEMICOLON

type_specifier: INT → declaration_list: ID

From the given grammar, we have to check the rules for the tokens.

~~For a token~~, when the rule matches ~~for we~~ for a token, we print the rule.

* Grammar ~~এর~~ এর file থেকে উপরে এর check করবে (bottom up parsing)

Grammar file এ কোন edit হবে না
+ lex file

Ambiguity handle করা লাগবে

eg. if
else
[else] → এটা C তে থাকলে না handle করা লাগবে

%nonassoc ELSE → use করে ambiguity handle করে
↓
Prio প্রায়োরিটি সম্পর্কে

Grammar check করার জন্য yacc file ব্যবহার করা হয়।
↳ Bison

lex file থেকে token yacc file এ যায় for grammar check.

যেসব token read করা হচ্ছে in a line, সেগুলো store করা লাগবে।

lexeme গুলোকে list এ store করতে

int *ptr = &a

↓

memory location store করে।

y.tab.h → define না করেও token generate করে নেয়।

extern FILE *yyin;

↓

যদি file এ yyin initiate করেছি তবে extern দিয়ে access করতে

Terminal ও non-terminal + ambiguity handle করার + rule define করি।

FINAL

LECTURE 12

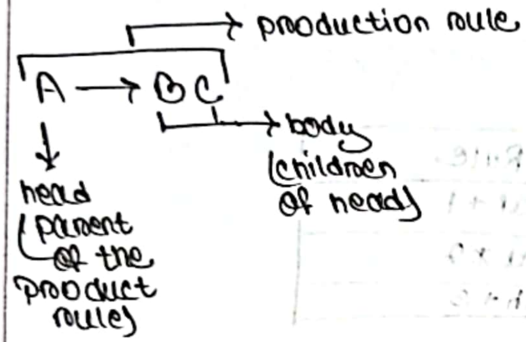
THURSDAY DATE: 16/11/23

Work of semantic analysis:

① Type checking

② Type formation

Semantic analyzer deals with type mismatches.



Semantic analysis এখন আমরা যদি প্রত্যেক টা non-terminal এর সাথে at least one attribute থাকে

eg. of attribute : for A, A.syn
for B, B.var
for C, C.var

* dependency create করতে পারি attribute দিয়ে।

Rule:

① A.syn = B.var + 1
* A.syn এর যে value পাবে সেটা B.var + 1 থেকে আসবে

② C.var = B.var * 2
* C.var এর value আসবে B.var * 2 থেকে

③ C.var = A.inh + 3
* C এর attribute এর value তার parent থেকে আসছে।

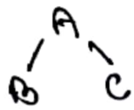
→ Temporal dependencies বানাচ্ছি

↓

2 types of attributes [syntax derivated sentence এ থাকবে]

synthesized
- parent এর value child থেকে আসে
  eg. Rule ①
  A is parent which is dependent on its child B.

inherited
- sibling এর value আরেক sibling নেয়
  eg. Rule ②
  B and C are siblings
- child এর value parent থেকে আসে
  eg. Rule ③



* B and C are in same level, so are siblings

Common feature between synthesized & inherited attributes:
Non-terminal এর যেকোন attribute এর আওতায় attribute এর উপর dependent হলে তাকে when the non-terminal for both the attributes is same.

eg. A.syn = A.inh | A.inh = A.syn
Also

*syn এর value inh থেকেও আসতে পারে

Production Rule	Semantic Rule
A → BC	(1) A.syn = B.val + 1
	(2) C.val = B.val * 2
	(3) C.val = A.inh + 3

* Production Rule and Semantic Rule নিয়ে syntax Directed Definition (SDD) বানানো

Example:

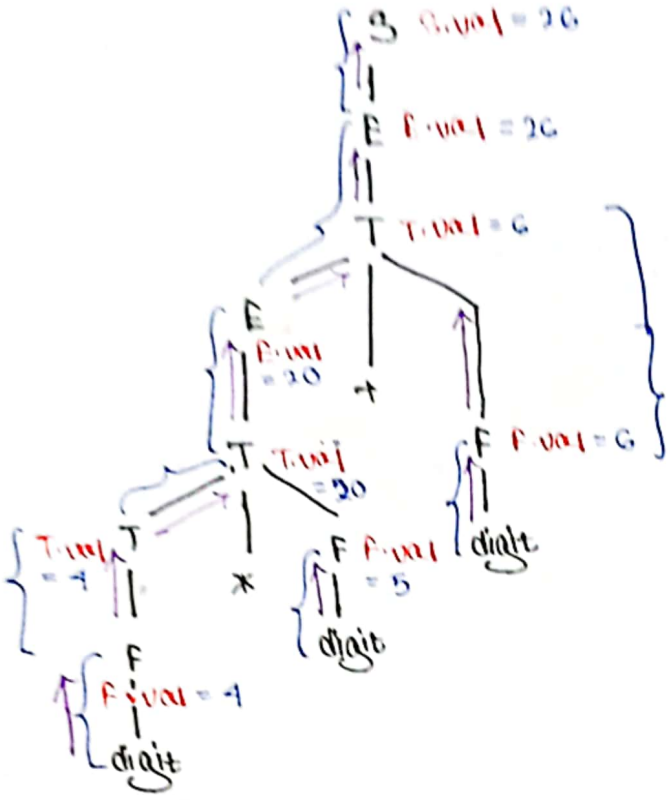
Production Rule	Semantic Rules
S → E	S.val = E.val
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → f	T.val = f.val
f → digit	f.val = digit.lexval

[parent → T and T1 → same, child. T child and parent হিসাবে আছে, so T1 ও child হবে]

lexval → lexical analyzer থেকে token আসাকে (terminal থেকে আসে)

input = 4 * 5 + 6

① Construct a parse tree using production rule.



② Semantic Rules দিয়ে evaluate করি

- যে attribute নিয়ে চিন্তা non-terminal এ করবে
- attribute এর value compute করে in a bottom-up approach.
- evaluation order is bottom-up

প্রত্যেকটা attribute তার child এর value নিচ্ছে, so this is SDD is called S attributed SDD as every attribute is synthesized. (Parent value তার child এর)

* SDD Type:
S attributed SDD

* toParse Tree Type:
Annotated Parse Tree
(Parse tree এর প্রত্যেক attribute এর value দেয়া আছে)

* Flow of operation:
dependency graph (arrow দিয়ে যা দেখানো, তার উপর যা dependent থাকবে)

LECTURE 13

SATURDAY DATE: 18/11/23

L ATTRIBUTED SDD

→ used for inherited attribute

→ এই sibling/parent থেকে value নেয়

Production Rules	Semantic Rules
1. T → FT'	T'.inh = F.val T.val = T'.syn
2. T' → *FT1'	T1'.inh = T'.inh * F.val T'.syn = T1'.syn
3. T1' → ε	T'.syn = T'.inh
4. F → digit	F.val = dig.lexval

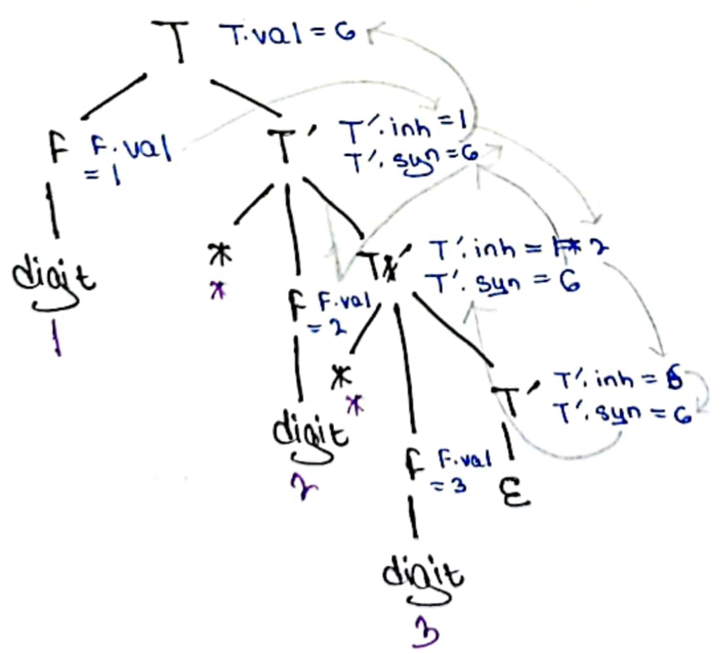
(Rule 1: seman…; semantic rules per production)

(T' and T1' same)

(T1' — T' এর child)

(Rule 2) → more than one non-terminal থাকে in the body.

input : 1*2*3



* Bottom up এর semantic rules থাকবে

* এখানে left to right evaluation technique use হয়

* Evaluation technique টা আগে হয় from left to right AND bottom up.

LECTURE 19

SATURDAY

DATE: 25/11/23

SYNTAX DIRECTED TREE

SDD V/S SDT

SDD is all about semantic rules, whereas SDT is all about semantic actions.

$E \rightarrow K * t$

E.inh = K.syn * T.syn → concatenated output চাই

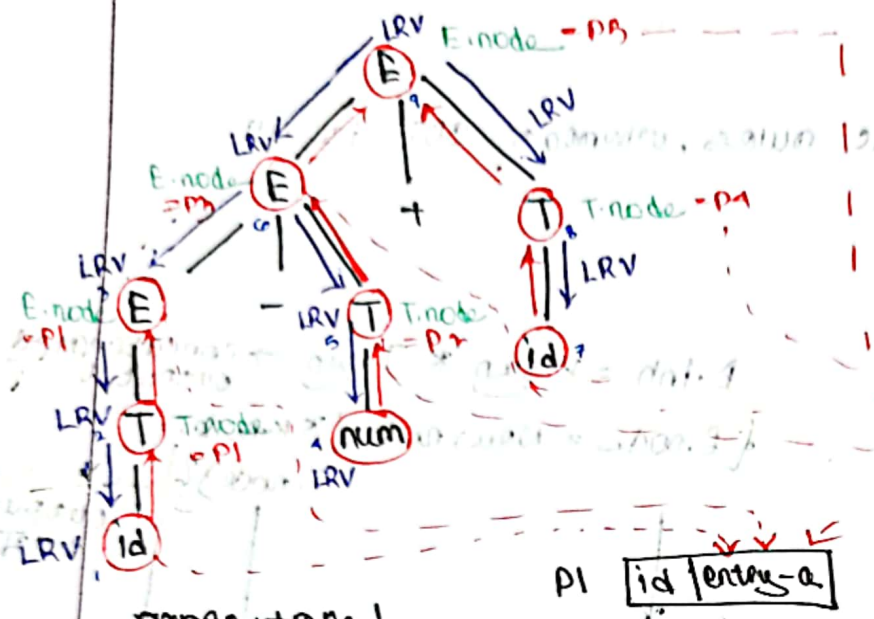
{E.node = New node (*, K.node, T.node)} → direct output চাই

E.node → operator

K.node, T.node → sub-nodes

* methods call করে বা object নিয়ে কাজ করতে হবে

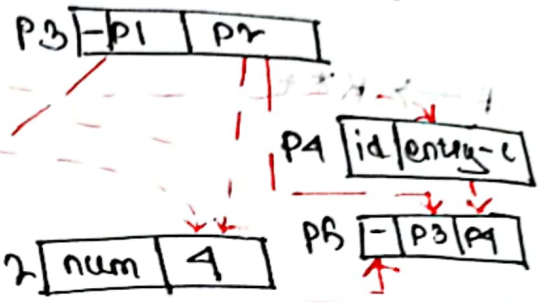
a - 4 + c



parse tree / concrete syntax tree

sequentially with steps post order traversal

P1 = new (id, entry-a)
P2 = new (num, 4)
P3 = new (+, P1, P2)
P4 = new (id, entry-c)
P5 = new (–, P3, P4)



abstract syntax tree / syntax tree

(Q) sequentially with steps

→ semantic rules ⓐ sequentially follows ZAA

→ tree ⓑ post order traversal use ZAA

in order (LSR) - left → self → right

pre order (SLR) - self → left → right

post order (LRS) - left → right → self

Concrete syntax tree / Parse tree — parse tree for left side grammar
Abstract syntax tree / Syntax tree — parse tree for right side program

* Syntax tree is shorter than parse tree.
* No helper production in syntax tree
  $T \to f$
* No ϵ production in abstract syntax tree
  $f \to \epsilon$

LECTURE 18

SATURDAY DATE: 30/11/23

DAG (DIRECTED ACYCLIC GRAPH)

Parse tree (concrete tree)

→ grammar use করে parse tree করা।

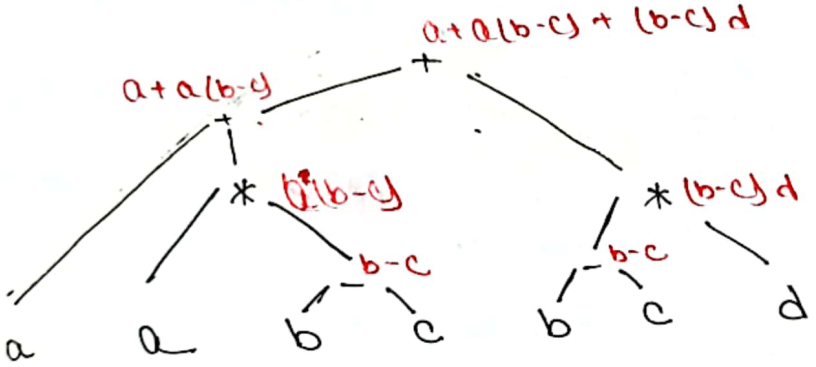
Abstract syntax tree

→ binary form এ থাকে

→ input use করে তৈরি

→ precedence of operator follows করে তৈরি

a + a * (b - c) + (b - c) * d → addition, left associative so এভাবে লেখা যায়

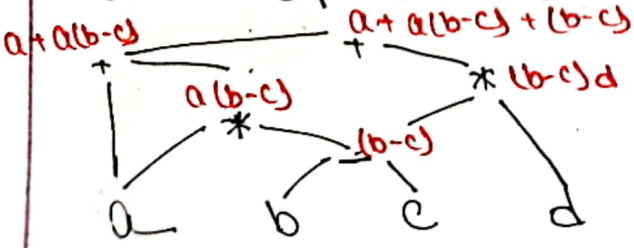


DAG

→ short part of abstract syntax tree

→ common variable কে uniquely treat করে তৈরি (এক বার নিয়ে variable এ)

→ common expression ও uniquely চিনতে হয়



Three address code

Three address code can be :-

1. variable
2. temporary register $(t1, t2)$
3. constant $(4, 5, 61, 2.0)$

* per line A max 3 টা three address code থাকতে পারে

three address code for AST :-

$t1 = b - c$

$t2 = b - c$

$t3 = a * t1$

$t4 = t2 * d$

$t5 = a + t3$

$t6 = t5 + t4$

three address code for DAG :-

$t1 = b - c$

$t2 = a * t1$

$t3 = t1 * d$

$t4 = a + t2$

$t5 = t4 + t3$

* Time complexity & space complexity কমবে for DAG as less no. of lines used. (in three address code)
* Three address code কোথায় data structure A store হয়
* Data structure — Quadruples & Triples
  — Both are table type data structure

Quadruples

(+/-) — OP; left (1st) — arg1; right (2nd) — arg2; result save হয় — result

OP	arg1	arg2	result
−	b	c	t1
−	b	c	t2
*	a	t1	t3
*	t2	d	t4
+	a	t3	t5
+	t3	t4	t6

* ~~ind~~ instruction array use করে triples করলে তাকে আমরা directed triples বলি

Triples

line no. + (+/-) — OP; 1st — arg1; 2nd — arg2

* result column নাই

OP	arg1	arg2
(0) −	b	c
(1) −	b	c
(2) *	a	(0)
(3) *	(1)	d
(4) +	a	(2)
(5) +	(4)	(3)

* No temporary variable so space is reducing.
* execute faster as line থেকে আনছি

* Disadv:—
code optimizer এর মাধ্যমে line reduce করে ফেললে, so line number changes তাই track হারায় থাকে, তাই ~~এই~~ আলাদা array use করা

instruction array (RAM এর মতো)

0
1
2
3
4
5

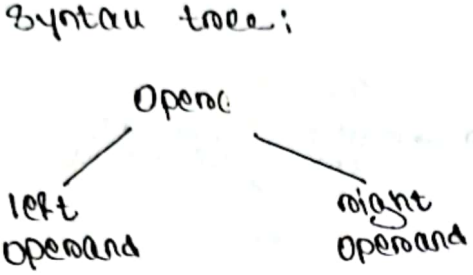
code optimizer এ line reduce হলেও, info এই array তে থাকে, তাই triples access পারে variable এর,

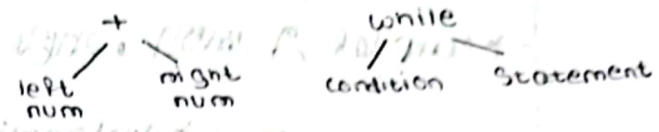
LECTURE 16

SATURDAY DATE: 2/12/23

Type Expressions

Abstract Syntax tree:

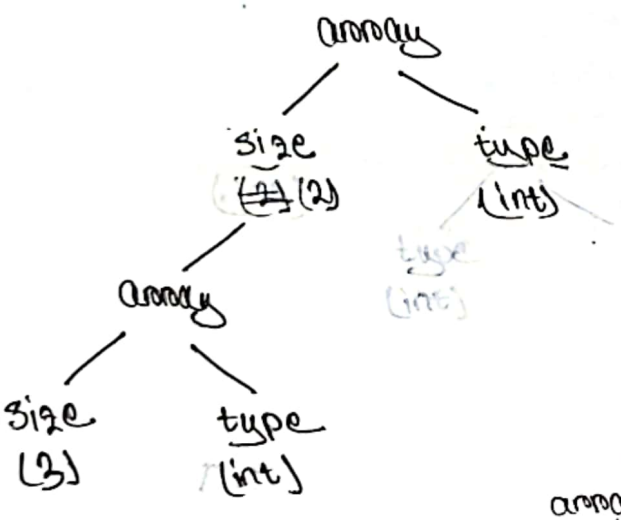




*Binary representation define ~~AST~~ since 2 child ~~hai~~

* Type expression ~~ko~~ tree ~~mein~~ represent ~~karte~~ hai;

int[2][3] → array type operation



record{t}
record{float x;
int[2] y}t

symbol table
float
int[2] y

name of symbol table

array(2, array(3, int))

- basic type — boolean, char, int, void, float
- type name is a type expression

public class Node{} — Node n

- array type construction is a ty
- record is a type expression. record(t)

*Symbol table, a record ~~hai~~, field name and type ~~ke~~ ~~liye~~ ~~hai~~ construction implement ~~hai~~.

1 এই Exercise 6.3.1 type এর question আসবে

Declarations

→ Actual grammar

→ compiler এ লেখা থাকবে

$D \rightarrow T\ id\ ; D \mid \epsilon$ (T → type; id → declaration → multiple identifier identify করতে পারি)

$T \rightarrow BC \mid record\ '\{' D '\}'$

$B \rightarrow int \mid float$

$C \rightarrow \epsilon \mid [num] C$

int a ; int b; float [8] y

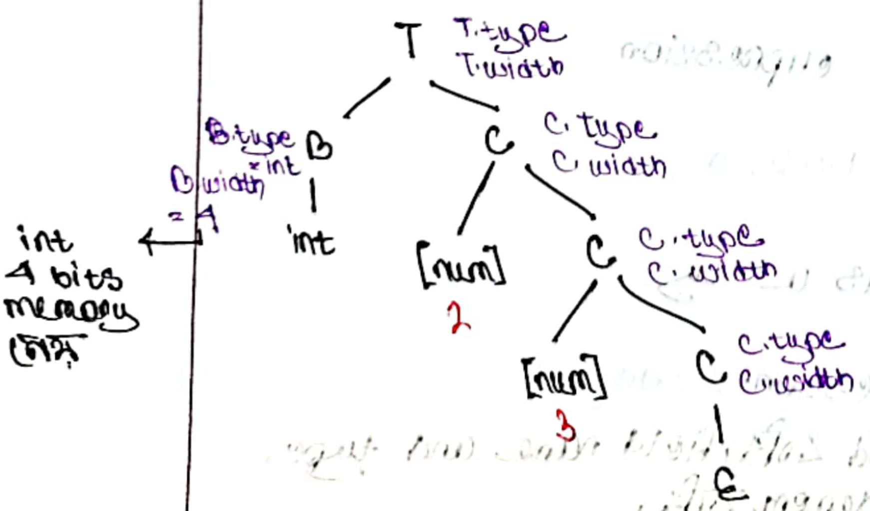
* যে কোনো int বা float type এর multiple array কে declare করবি।

* record এ float বা int type identifier কে declare করতে পারি।

Exercise for section 6.3

Exercise 6.3.1 → important

int [2][3]



$T \rightarrow B\ C$ {T.type = t; T.width = w}

$B \rightarrow int$

$B \rightarrow float$

$C \rightarrow \epsilon$

$C \rightarrow [num]\ C_1$ {C.type = array(num.value, C_1.type); C.width = num.value × C_1.width}

t, w → helping global variables

LECTURE 17

THURSDAY DATE: 07/12/23

*Procedural call এর grammar.

SDT:-

T → B {t = B.type, w = B.width}

C {T.type = C.type, T.width = C.width} → connection

B → int {B.type = integer, B.width = 4}

B → float {B.type = float, B.width = 8}

C → ε {C.type = t, C.width = w}

C → [num]C₁ {C.type = array (num.value, C₁.type); → connection

C.width = num.value * C₁.value}

↑process

P → {offset = 0;} → এর থেকে কাজ শুরু

start হয়; space allocate করে b4 compilation → global variable

D → T id; {top.put(id.lexeme, T.type, offset); → we can avoid

offset = offset + T.width;} —

D;

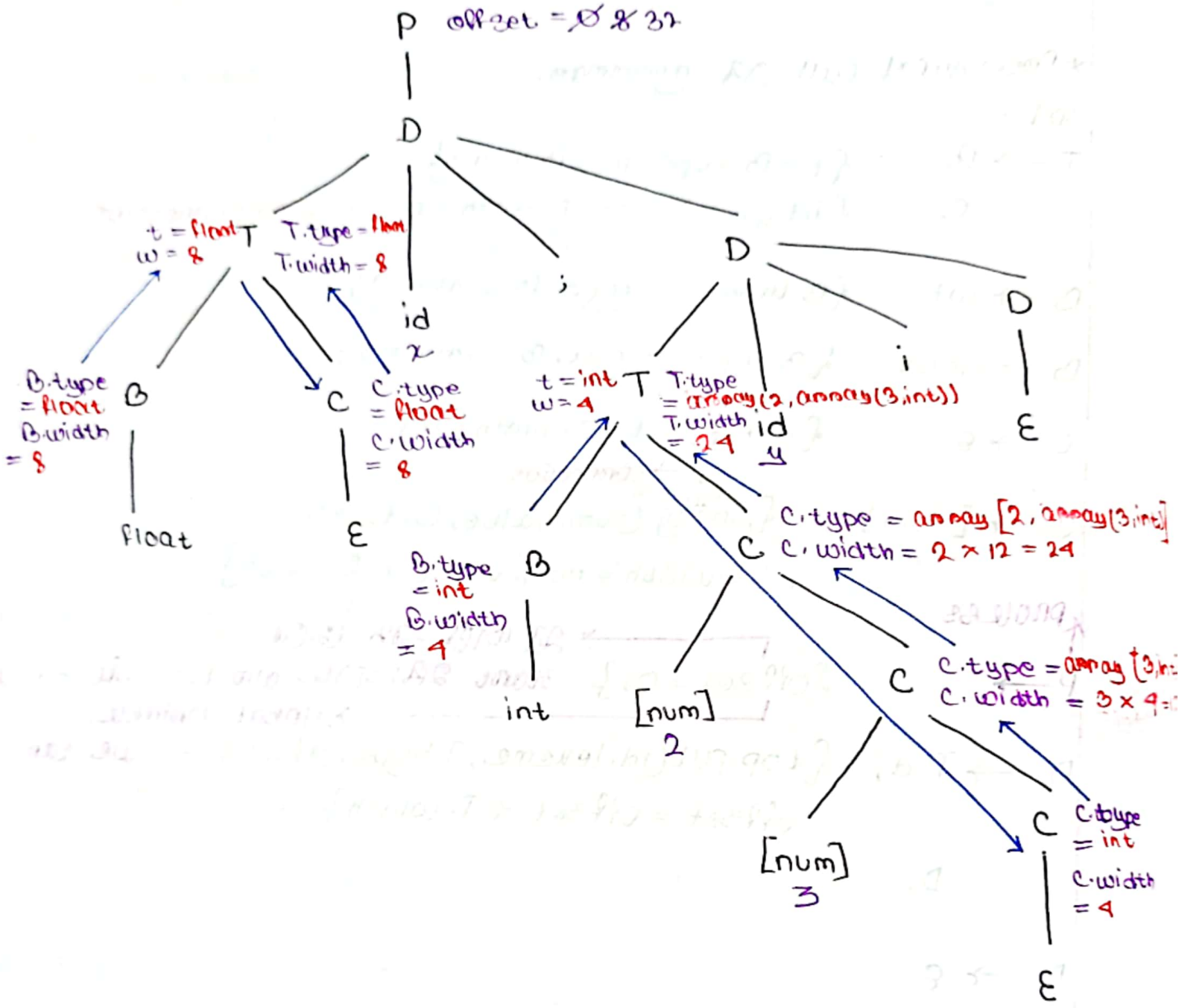
D → ε

String:-

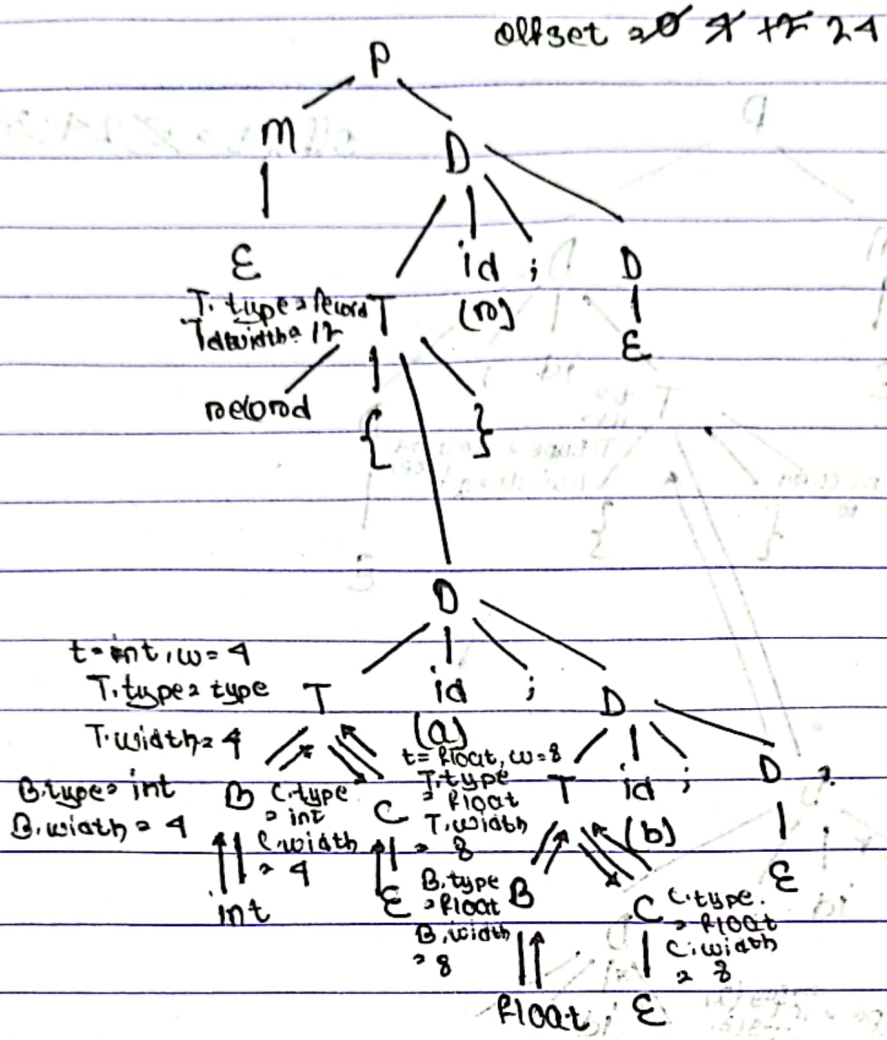
float x; int [2][3] y

*প্রত্যেকটা identifies memory location নেয়, তাই compilation এর আগে offset দিয়ে space allocate করে দিই

*Question এ SDT আর string দেওয়া থাকবে

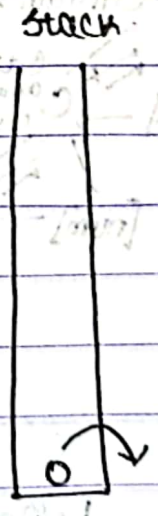

P
offset = 0 & 32
D
t = float
w = 8
T
T.type = float
T.width = 8
id
x
;
D
D
B.type = float
B.width = 8
B
C
C.type = float
C.width = 8
float
ε
t = int
w = 4
T
T.type = array(2, array(3, int))
T.width = 24
id
y
i
D
ε
C.type = array [2, array(3, int)]
C.width = 2 × 12 = 24
B.type = int
B.width = 4
B
C
int
[num]
2
C
C.type = array [3, int]
C.width = 3 × 4 = 12
[num]
3
C
C.type = int
C.width = 4
ε

record {int[2][8] a; float b;} n ;



top

lexeme	type	Offset
a	int	0
b	float	4
~~n~~	~~record~~	~~12~~



(offset value 24 here)

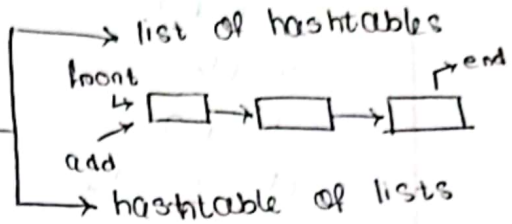
lexeme	type	Offset
n	record	12

LECTURE 18

SATURDAY DATE: 08/12/23

SYMBOL TABLE AND SCOPING

Symbol table can be implemented in two ways. (→ identifiers info track রাখা)




```
int x = 0;
int x = 0;
```
⇓
Java will say duplicate value.

```
int x = 0;
x = 2
```
⇓
x is updated

#block of code given

```
void f(int a){
    double x;
    while(...){
    int x,y;
    }
}

void g(){
    f(3);
}
```

consider the code. Draw the symbol table as it would be after processing all the declaration in the body.

① Scope এ new hashtable/ symbol table create হবে

② current hashtable এ variable না থাকলে, hashtable এ খুঁজি যেখানেই variable টা insert হয়ে গিয়ে

③ কোন variable আছে কি না, সেটা check করে by back tracing. variable info retrieve করি

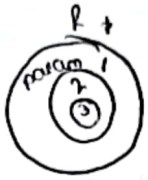
④ Scope exit হলে, counter-- করে দিব and hashtable remove করে দিব,

Env
↙ Hashtable ↘ prev

* put → Hashtable এ string টা and Symbol info store করি
(type, level)

* get → check করি string টা already hashtable এ আছে কি না

```
void f (int a){
level=1   level=2
    double x;
    level 2
    while(...){
    int x, y;
    } level 3
}
```



Using list of hashtables:-

level = 1;

When see a {, increase level;

Env e1 = new env(null);

When match type id Symbol found;

e1.put(f, void, 1);

level = 2;

Env e2 = new Env(f, void, 1);

e2.put(a, int, 2);

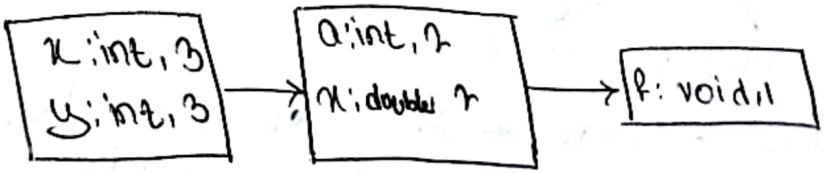
e2.put(e, double, 2);

level 3;

Env e3 = new Env(f, void, 1,
a, int, 2,
x, double, 2)

e3.put(x, int, 3)

e3.put(y, int, 3)



* method এর জন্য আলাদা scoop বানাব

* type id গুলো আলাদা scoop.

```
void f(int a) {
    int b
    double x;
    while (...) {
        int x, y;
        ...
    }
}
void g() {
    f(3);
}
```

Q. Draw the symbol table operations for the following declarations in the body of f by hashtable of lists approach.

type ↑ level ↗

	key	value (Symbol)
1	f :	void, 1
2	a	int, 2
3	b	int, 2
4	x	double, 2 → int, 3
5	y	int, 3

① Initialize level = 1

② type id search করা

③ search for available key

put hashtable এ

```
table.get(s) == null;
    table.put(s, symbol);
```

① Scope entry ---> level ++

② Variable declaration ---> level check করি

- same level এ same variable থাকলে, duplicate variable বলে
- different level এ same variable এ থাকলে hashtable টা append করে নিচে front এ।

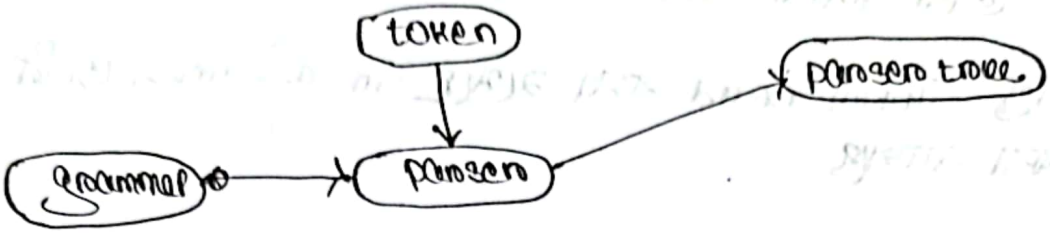
③ Variable use ---> get method দিয়ে খুঁজতে where is a variable

④ Scope exit ---> level --, current level এর variable এর

LAB 3

MONDAY DATE: 20/11/23

From LAB 02:-



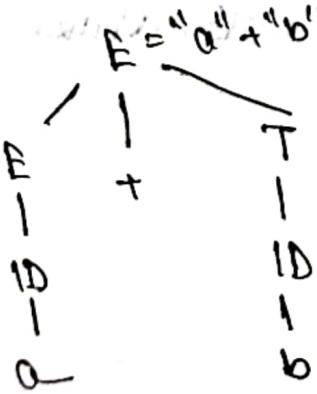
grammar দিয়ে node create করা লাগবে

Grammar লাগবে + semantic rules লাগবে → abstract syntax tree create করে

* source code থেকে abstract syntax tree বানাতে হবে।
* Instead of keeping all semantic rules, প্রত্যেক node এ আমরা values গুলো রাখব।

* প্রত্যেক semantic rule এর জন্য node create করব

$E \rightarrow E_1 + T$

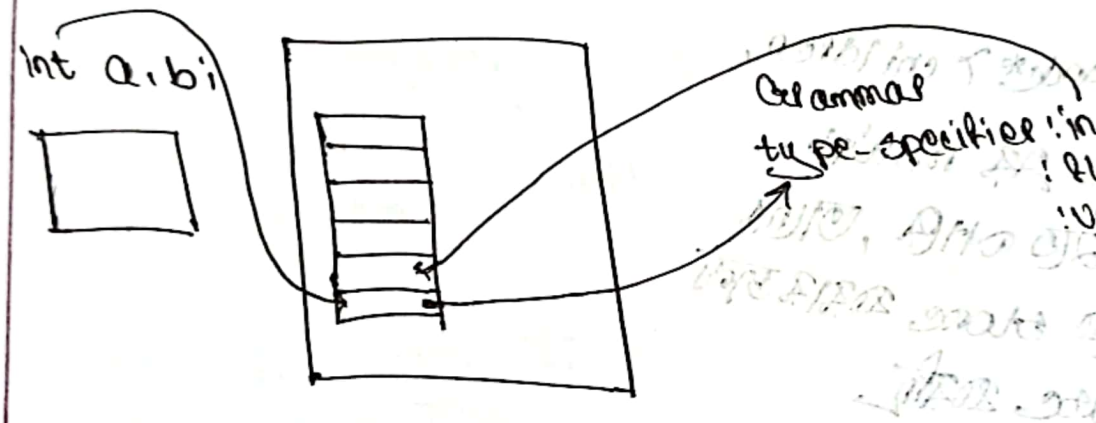


* প্রত্যেক node এ syntax store করব (token গুলো 2 attribute/syntax)
* আমরা post order traversal করলে source string আবার পাব।

input → output will be same as the input → Because we are creating AST. The node holds the attribute as a string (token value)

↓

we will do post order traversal on the AST to get back the same input.



stack এ token + object save হয়

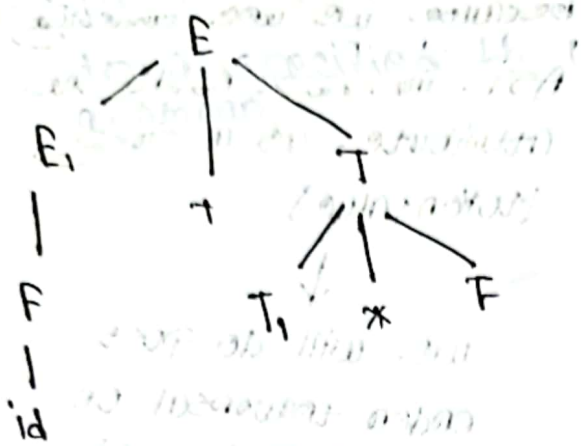
* যখন কোনো grammar এর সাথে match হয় node create হয় Tree তে add হয়ে যায়।

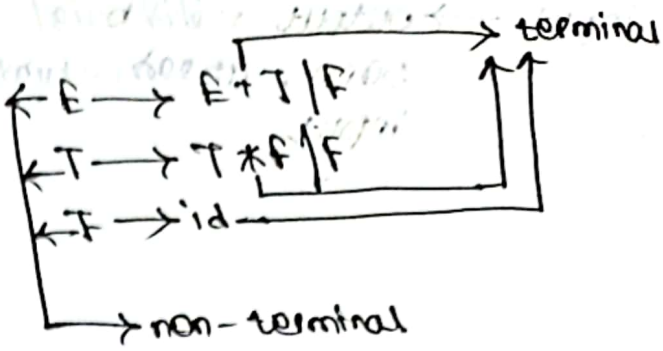
{ ⇒ node create হয়ে matching nodes

symbol-info তে থাকে →
→ type
→ name
→ node

⇓

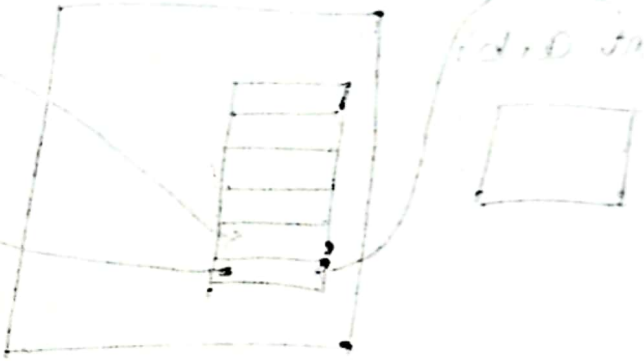
action





vector<TreeNode*> children,

↳ প্রতিটা root এর multiple children হতে পারে, তাদের children কে store করার জন্য vector use করছি।



value — syntax খুলো রাখার

type — terminal/non-terminal child

postOrderTraversal এ node গুলো traverse করি, output গুলো file এ লিখি

* start এর জন্য node করা লাগবে না

* driver code এ node কেন postordertraversal(node) call দিব।

* posttraversal function এ recursively ~~pot po~~ function টা call দেওয়া লাগবে।
* ~~যা in node~~ value গুলোকে output file এ লিখে দিই।

* Treenode create করব → node create করতেছি for head
* Treenode এর সাথে child add করব

* symbol_info ও node টা include করে দিই, stack এ save হচ্ছে।
* $$ দিয়ে access করতেছি stack এর
* rootnode হিসেবে save করব starting node এর জন্য

* Rule গুলো production head এর জন্য লিখছি

* variable : id_name
এইটার জন্য একটা node আছে
so $1 $$ = $1 এমন করে লিখতেছি।

* $$ → stack এ node save করি
$ → get call দিলে saved node in stack দিয়ে যাবে