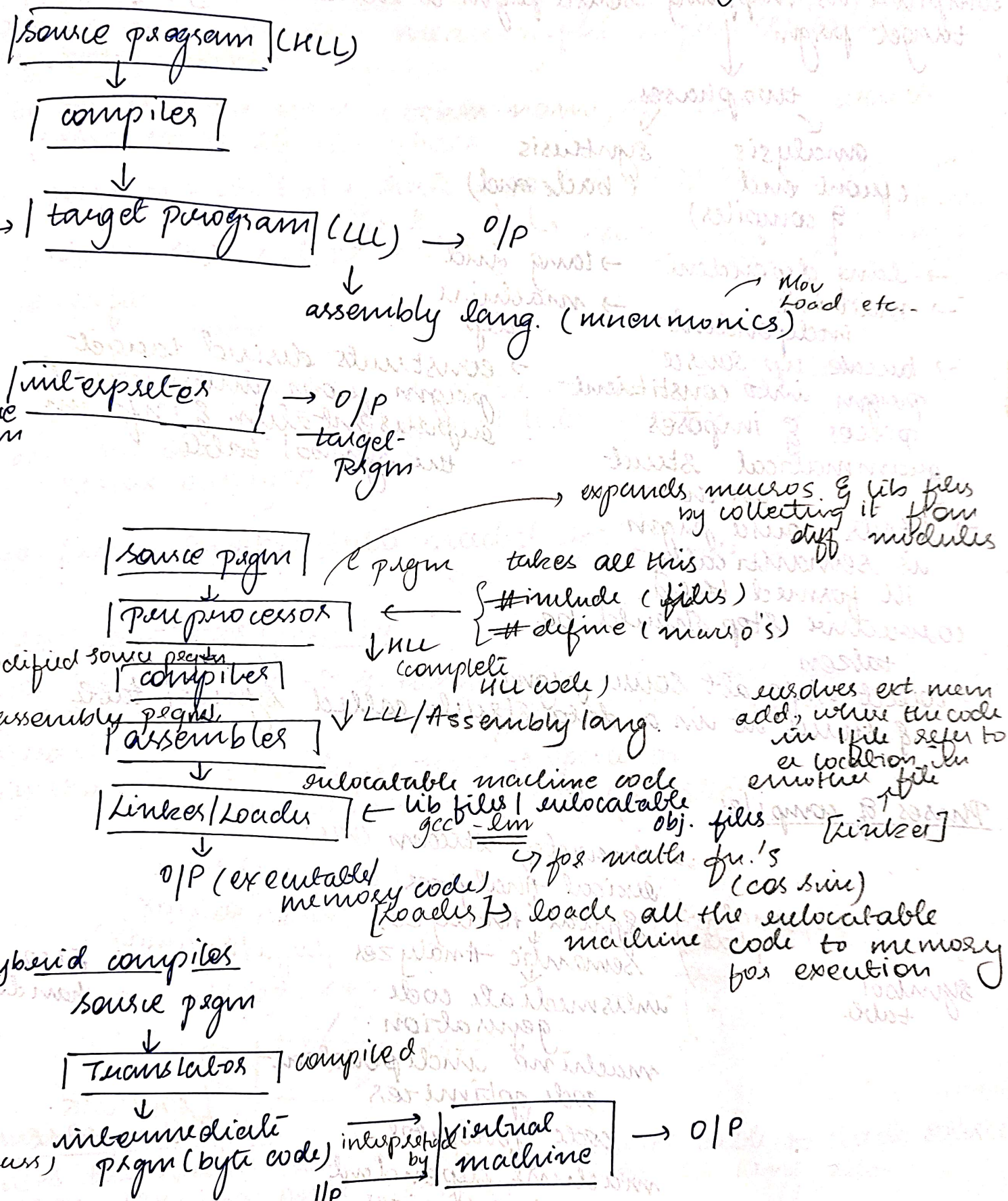
Automata theory & Compiler design



Java combines compilation & interpretation
bytecode compiled on one machine can be interpreted on another machine

Structure of a compiler

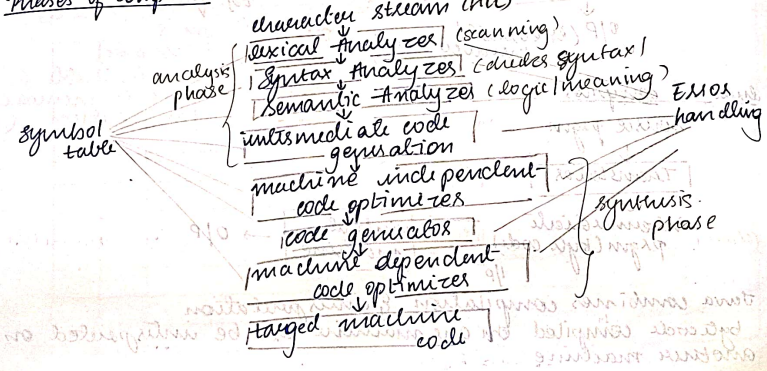
compiler is mapping source prgm to semantically equivalent target prgm

two phases
- analysis (front end of compiler)
- synthesis (back end)

→ lang dependent-machine independent
→ breaks up source prgm into constituent pieces & imposes grammatical struct on them
→ detects source prgm is semantically ill formed then corrective step should be taken
→ collect info abt source prgm & stores it in a data struct. called symbol table

→ lang and machine dep
→ constructs desired target prgm from intermediate representation & info in the symbol table

Phases of compiler



Lexical analysis / Scanning

→ reads stream of chars making up the source prgm & groups the chars into meaningful sequences called lexemes (token)

→ token is of the form <token name, attr. name> which passes on to syntax phase

Eg. position = initial + rate * 60] this step converts them into tokens

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

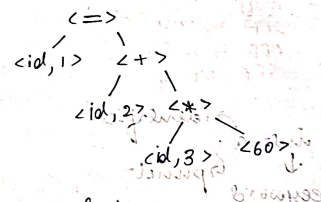
→ identifying lexeme
→ removes spaces
→ finds errors's

<token, attr> → points to an entry in the symbol table for this token
↓ abstract symbol used during syntax analysis

→ info from symbol table needed for → semantic → code generation

Syntax analyzer / parsing

→ uses first component of token to create a tree like intermediate representation that depicts grammatical struct of token stream.
→ syntax tree → interior node → operation
→ children → args. of operation



Semantic analysis

→ uses the syntax tree & info in symbol table to check source prgm for semantic consistency with the lang defn.
→ gathers info
→ type checks, coercion (conversion)

a = b + c
int int float (conversion)

Intermediate code generation

→ After syntax & semantic phase, the source pgm, many compilers generate an LL representation
↳ pgm for an abstract machine

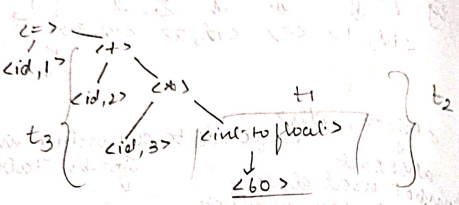
→ two properties
↳ easy to produce
↳ easy to translate

→ uses three address instr

t_1 = int to float (60)
t_2 = id_3 * t_1
t_3 = id_2 + t_2
id_1 = t_3

↓ optimizing

t_2 = id_3 * 60.0
id_1 = id_2 + t_2



mul id3 #60.0 t2
add id2 t2 id1

Code generation

→ takes as I/p an intermediate representation of the source pgm. & maps it to target lang.
→ if target lang is machine code, registers for mem loc are selected for each of the variable used by the program. Then they're translated into sequences of machine instr. that perform same task

t_1 = id_3 * 60.0
id_1 = id_2 + t_1

↓ representation of reg. exp

```
LDF  R2, id3
MULF R2, R2, #60
LDF  R1, id2
ADD  R1, R1, R2
STF  id1, R1
```

Token/ pattern/ lexeme

int a;

keyword	int	int
identifier	a	a
punct	;	;

int a;
↳ identifies
↳ punct
keyword

token → abstract symbol for representing lexical unit
lexeme → actual char sequence in pgm

printf ("total = %d", sum);

token	pattern	lexeme
keyword	printf	printf
punct	(	(
~~literal~~ string	total = %d	total = %d
op operator	=	=
operator	%	%
punct	"	"
punct	,	,
~~literal~~ identifier	sum	sum
punct	)	)
punct	;	;

↳ literal
↳ anything written in " " is a string literal

```
#include <stdio.h>
int main() {
    int a, b;
    int c = a+b;
    printf("Sum of two no.'s is: %d", c);
}
```

token are specified using regular exp. → words (set of chars)
↳ specifications of token: alphabet, string, lang.

finite set of symbols
sequence of symbol chosen from alphabet
countable set of string over fixed alphabet

Operations on string

prefix: banana → ban (elimination at end)
suffix: banana → ana
substring: x → can't be an empty string / string → ban, ana, nan
↳ empty string = ε (string with length = 0)
subsequence: → bna → banana (not continuous)

Operation on languages

$\Sigma = \{0,1\}$ $S = \{a,b,c\}$
Union: $\Sigma \cup S = \{0,1,a,b,c\}$
concatenation: ΣS = ~~{abc}~~ {0a, 0b, 0c, 1a, 1abc .... etc

Kleen closure $\Sigma = \{0,1\}$
$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11\}$

Positive closure
$\Sigma^+ = \{0,1,00,11,01,10\}$
↳ $\Sigma^* - \epsilon$

$r^* = L(r)^*, \epsilon$
$r^+ = L(r) + a$
$r|s = L(r) \cup L(s)$
$rs = L(r) L(s)$

Regular defn.

→ giving name to the regular exp.

$d_1 \to r_1$
$d_2 \to r_2$
⋮
$d_n \to r_n$

} d_1, d_2 → distinct name

identifier
↳ (letter) (letter / digit)
↳ union (∪)

letter: A|B|... z|a|b|....|z → [A-Z a-z -]
digit: 0|1.....|9 → [0-9]
↳ letter = [A|... Z, a|... z]
digit = [0|... 9]

3.9 E+2

Shorthand

One or more occurrence → +
Zero or one instance → ?
char class → [ ] ↳ A or B or

→ unsigned
digit ↳ 0|1 9
digits → digit(digit)* → 1 or more digits (digit +)
optional fractions → . digits | ϵ
optional exponents → (E (+ | - | ϵ) digits) | ϵ
unsigned no. → digits, optional fraction optional exponent

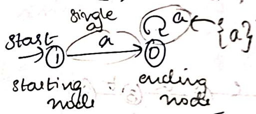
Expression	Matches	Example
c	the one non-operator character c	a
\c	character c literally	*
"s"	string s literally	"**"
.	any char but newline	a.b
^	beginning of a line	
$	end of a line	abc$
[s]	any one char of string	a|b|c

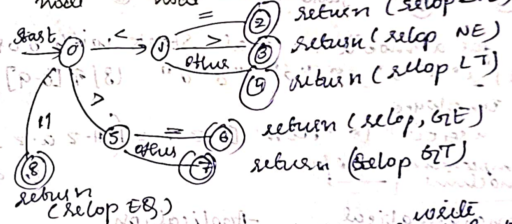
Expression	Matches	Example
[^s]		[^a|b|c]
r*		
r+	one/more occurrence	
r{m,n}		
$r_1 r_2$		
$r_1 \| r_2$		
(r)		

Write a complete line that doesn't contain a lowercase vowel
^[^aeiou]*$

relational operators → < | > | <= | >= | = | <>

lexeme	token	attr value
<	rel op	LT
>	— " —	GT
<=	— " —	LTE
>=	— " —	GTE
=	— " —	EQ
<>	— " —	NE





Q) Suppose the i/p char is A & B, ~~we want~~ write the regular exp.
1) which starts with A
2) Ending with b
↳ (a|b)*b

$\Sigma = \{a, b\}$
L = {a, aa, aaa, ab, abb, aabb

Soln. a(a|b)* → multiple appearance
↳ can be a or b

3) ab as substring

$(a|b)^* ab (a|b)^*$

4) Start with a & end with a

$a (a|b)^* a | a$

5) Start & end should be same

$a (a|b)^* a | a | b (a|b)^* b | b$

6) Third symbol from RHS is a

$(a|b)^* a (a|b) (a|b)$

7) Even length of string

$((a|b)(a|b))^*$ (OR) $(ab|ba|aa|bb)^*$

8) odd length

$((a|b)(a|b))^* (a|b)$

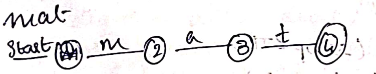
9) contains exactly 2 a's $==2$ $b^* a b^* a b^*$

atleast 2 a's $>=2$ ~~$b^* a b^* (a|b) (b)^*$~~ ~~$aa(a|b)^*$~~ $(a|b)^* a (a|b)^* a (a|b)^*$

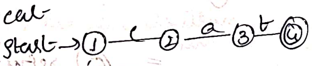
atmost 2a's $<=2$

$b | b^* a b^* a b^* | b^* a b^* | b^*$

mat



cat



Write a regular exp for a mobile no. that starts with 8 or 9

~~(8|9) d* d* d* d* d* ...~~ $(8|9) [0-9] \{9\}$

SELECT * FROM Employee

"SELECT" * "FROM" [A-Z a-z] [A-Z a-z [0-9]]

I/P → | Abs. machine | → O/P (implements)

Automata → mathematical model

finite / infinite

models implements with finite set of state

Application:

→ Design of a compiler

→ pattern matching

→ spell checking

Alphabets → {0,1} (Σ) String → {0, 1, 001, 111, ...} $w = \{010101\}$ $|w| = 6$

Lang: $\Sigma^3 = \{100, 011, 001, ...\}$

FA → DFA, NFA

$F = \{Q, \Sigma, \delta, q_0, F\}$

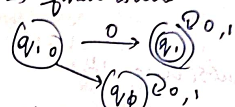
Q → finite set of states

Σ → finite set of I/P symbol

δ → transition fn.

q_0 → initial/start state

F → final state



$Q = \{q_0, q_1\}$

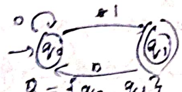
$\Sigma = \{0, 1\}$

$\delta =$ ~~δ~~ $Q \times \Sigma \to Q$

$q_0 = \{q_0\}$

$F = \{q_1\}$

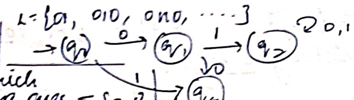
Q	0	1
q_0	q_0	q_1
q_1	q_0	q_1



{0, 01, 00, 011, 000, 001, 010, ...}

	0	1
q_0	q_1	q
q_1	q_1	q

L = {01, 010, 0110, ...}

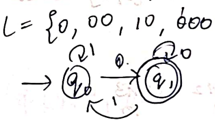


construct DFA which accepts all string over Σ={0,1} where it starts with 01 |=2

State	0	1
q_0	q_1	q_ϕ
q_1	q_ϕ	q_2
* q_2	q_2	q_2

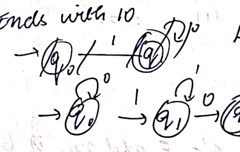
Q construct DFA which accepts all string over Σ={0,1} where it end with 0 |=1=2

L = {0, 00, 10, 000, 010, 100, 110 ....}



δ	0	1
q_0	q_1	q_0
q_1	q_1	q_0

Ends with 10 L = {10, 10, 110, 010 ....}



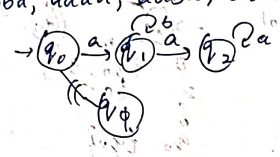
δ	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2		

$L = awa \quad \Sigma\{a,b\}$

$w = \{a/b\}^*$?

$L = a\{a/b\}^*a$

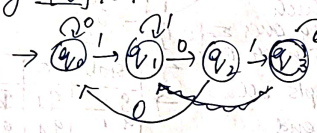
$= aa, aaa, aba, aaaa, aaba, abaa \ldots$



construct the DFA which consists of a substring '101'

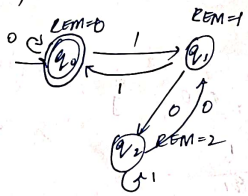
$\Sigma = \{0,1\}$? $L = \{101, 0101, 1010, 1011, 11010$

substring = ~~{0}~~ 101



construct a DFA which accepts all strings over $\Sigma = \{0,1\}$ where binary integers divisible by 3

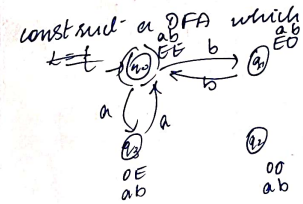
$L = \{0, 11, 110, 1001, 1100, 1111, \ldots\}$



REM 1	REM 2	REM 3
0	01	10
11	100	1011
110	111	1000

construct a DFA which accepts even no. of a's & odd no. of b's

$L = \{\epsilon, aabb, ababab\ ab \ldots\}$



NFA

construct NFA which accepts all strings over the alphabet $\{0,1\}$ where the string starts with 0

ends with 0

construct NFA with reg. exp $(a/b)^*$

$L = \{\epsilon, a, b, ab,$ min states = 0

$(a/b)^*b$ ends with b $\{b, ab, bb, \ldots\}$ → min st. → 1 → add 1 → 2 states

δ	0	1
$\to q_0$	q_0	$\{q_0, q_1\}$
q_1	q_2	q_2
$*q_2$	—	—

δ	0	1
$\to q_0$	q_0	$[q_0, q_1]$
$*[q_0 q_1]$	$[q_0 q_2]$	$[q_0 q_1 q_2]$
$*[q_0 q_2]$	q_0	$[q_0, q_1]$
$*[q_0 q_1 q_2]$	$[q_0 q_2]$	$[q_0 q_1 q_2]$

$\delta'[q_0 q_1], 0) = \delta(q_0, 0) \cup \delta(q_1, 0) = [q_0, q_2]$

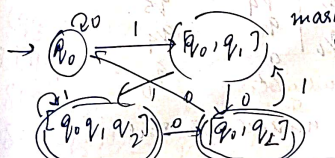
$\delta[q_0, q_1, 1) = \delta(q_0, 1) \cup (q_1, 1) = [q_0 q_1 q_2]$

$\delta(q_0, q_2, 0) = \delta(q_0, 0) \cup \delta(q_2, 0) = [q_0]$

$\delta(q_0, q_2, 1) = \delta(q_0, 1) \cup \delta(q_2, 1) = [q_0, q_1]$

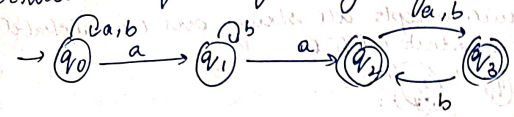
$\delta([q_0 q_1 q_2], 0) = \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0) = [q_0, q_2]$

$\delta([q_0, q_1, q_2], 1) = \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) = [q_0, q_1, q_2]$



mask starting & ending state

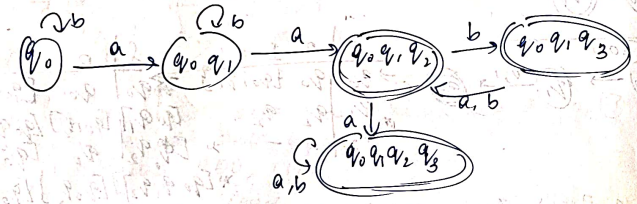
Construct DFA for the following diagram



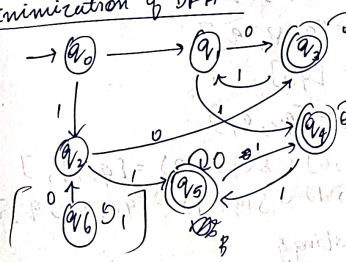
δ	a	b
q_0	$[q_0 q_1]$	q_0
q_1	q_2	q_1
q_2	q_3	$[q_2, q_3]$
q_3	–	q_2

	a	b
q_0	$q_0 q_1$	q_0
$q_0 q_1$	$q_0 q_1$	$q_0 q_1$
q_1	q_2	q_1
$q_1 q_2$	$q_2 q_3$	$q_2 q_3$
$q_2 q_3$		
$q_3 q_2$		

$q_0 q_1 q_2 q_3$



Minimization of DFA



① transition table

Σ	0	1
→ q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_3	q_5
* q_3	q_3	q_1
* q_4	q_4	q_5
* q_5	q_5	q_4
q_6	q_2	q_6

unreachable state

② find unreachable state

π_0 → zero equivalence

③ Separate into end & other states

$[\{q_3 q_4 q_5\}$ → end states, $\{q_0, q_1, q_2\}$ other states$]$

making 1 state

$\delta(q_0 q_1, 0) \Rightarrow q_1 q_3$ both in diff groups ∴ take two states from a group & check for 1 & 0

$\delta(q_0 q_1, 1) \Rightarrow q_2 q_4$ ↳ can skip ex. $(q_0 q_1, 0)$ and $q_1 q_3$ → diff grps

$\delta(q_0 q_2, 0) \Rightarrow q_1 q_3$ → diff grp

$\delta(q_0 q_2, 1) \Rightarrow$ skip

$\delta(q_1 q_2, 0) \Rightarrow q_3$ → same grp ✓ $\pi_1 = \{q_0\}\{q_1 q_2\}$

$\delta(q_1 q_2, 1) \Rightarrow q_4 q_5$ → same grp ✓ 1st equivalence

$\delta(q_3 q_4, 0) \Rightarrow q_3 q_4$ → same

$\delta(q_3 q_4, 1) \Rightarrow q_1 q_5$ → diff ∴ $q_3 \neq q_4$

$\delta(q_4 q_5, 0) \Rightarrow q_4 q_5$ } same ✓ $\pi_1 = \{q_3\}\{q_4 q_5\}$

$\delta(q_4 q_5, 1) \Rightarrow q_5 q_4$ }

$\delta(q_3 q_5, 0) \Rightarrow q_3 q_5$ → same

$\delta(q_3 q_5, 1) \Rightarrow q_1 q_4$ → diff $q_3 \neq q_5$

$\therefore \pi_1 = [q_0]\{q_1 q_2\}$ $\{q_3\}\{q_4 q_5\}$

$\delta(q_0, 0) \Rightarrow q_1$ ✓ $\delta(q_0, 1) \Rightarrow q_2$ ✓

$\delta(q_1 q_2, 0) \Rightarrow q_3$ ✓

$\delta(q_1 q_2, 1) \Rightarrow q_4 q_5$ ✓

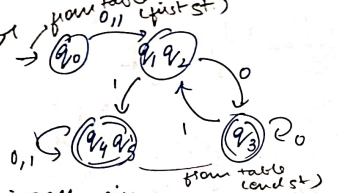
$\delta(q_3, 0) \Rightarrow q_3$ ✓

$\delta(q_3, 1) \Rightarrow q_1$ ✓

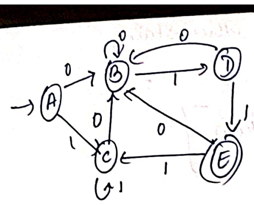
$\delta(q_4 q_5, 0) \Rightarrow q_4 q_5$ ✓

$\delta(q_4 q_5, 1) \Rightarrow q_5 q_4$ ✓

$\pi_2 = \{q_0, q_1 q_2, q_3, q_4 q_5\}$



write all pairs then use table to draw transition diag



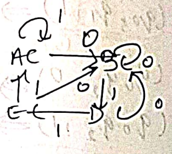
Σ	0	1
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

$\pi_0 = \{E\}\{ABCD\}$

δ(E, 0) ⇒ B ✓
δ(E, 1) ⇒ C ✓

δ(AB, 0) ⇒ B ✓
(AB, 1) ⇒ CD ✓
(BC, 0) ⇒ B ✓
(BC, 1) ⇒ D, C ✓
(CD, 0) ⇒ B
(CD, 1) ⇒ C, E ✗

	0	1	
(AC)	B	C	✓
(AD)	B	CE	✗
~~(AE)~~	~~B~~	~~C~~	
(BD)	B	DE	✗
~~(BE)~~			
~~(CE)~~			



	0	1	
ABC =	B	CD	✓
BCD =	B	DCE	✗
ACD =	B	CE	✗
ABD =	B	CDE	✗

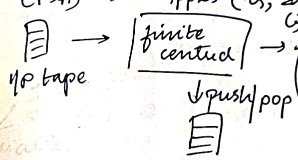
$\pi_1 \Rightarrow \{E\}\{ABC\}$
$\{E\}\{AB\}\{BC\}\{ABC\}\{AC\}$

	0	1	
AB	B	CD	✗
E	B	C	✓
BC	B	DC	✗
~~AB~~			
~~BE~~			
AC	B	C	✓

Noam Chomsky Hierarchy

Grammar type	grammar accepted	language accepted	automata
Type 0	unrestricted grammar		
Type 1	context-sensitive gram		
Type 2	context-free gram		
Type 3	regular gram		

pushdown automata = finite automata + stack (→ set of sets; → transition fn.)
(PDA) → 7 tupples (Q, Σ, Γ, δ, q_0, Z_0, F)
- Q → set of finite states
- Σ → set of i/p symbol
- q_0 → initial st.
- Z_0 → start stack symbol
- Γ → stack alphabet

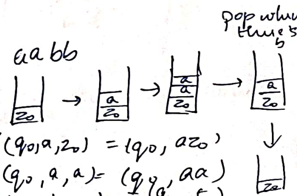


a, b/c:
- a → i/p symbol (ϵ)
- b → at the top of the stack (ϵ)
- c → i/p symbol not pushed to the stack (ϵ)

δ(q, a, X) (a → i/p, X → stack)
= δ(p, γ)
- ε → stack is popped
- X → stack remains the same
- YZ →

Design a PDA, $a^n b^n$ where $n > 0$

aabb



δ(q_0, a, Z_0) = (q_0, aZ_0)
δ(q_0, a, a) = (q_0, aa)
δ(q_0, b, a) = (q_1, ε)
δ(q_1, b, a) = (q_1, ε)
δ(q_1, ε, Z_0) = (q_2, ε) / or (q_2, Z_0)
if empty stack

$\{0^n 1^{2n} \mid n > 0\}$ $n=2$ 001111

construct a PDA where n

011

001111

$\delta(q_0, 0, z_0) = (q_0, 0z_0)$

$\delta(q_0, 0, 0) = (q_0, 00)$ →

$\delta(q_0, 1, 0) = (q_1, 0)$

$\delta(q_1, 1, 0) = (q_2, \epsilon)$

$\delta(q_2, 1, 0) = (q_1, 0)$

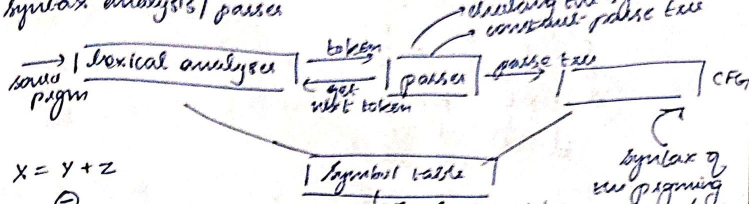
$\delta(q_1, 1, 0) = (q_2, \epsilon)$

$\delta(q_2, \epsilon, z_0) = (q_3, \epsilon)$

---

4 tupples / CFG context free grammar

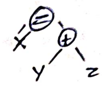
Syntax analysis / parser



↳ Top down parsing

↳ Bottom up parsing

X = Y + Z



4 tupples → {V T P S}

- V → variable / non-terminal (LHS)
- S → start symbol
- T → terminal
- P → production rule

$S \Rightarrow aSb$

$S \rightarrow (S)$ or $S \Rightarrow aSb \mid (S)$

variable

⇒ one step

$\overset{*}{\Rightarrow}$ many

lm → left most derivation

rm → right most derivation

The grammar with the following production defines a simple arithmetic exp. ~~Verify -(id+id)~~

prod. rule -(id+id) $E \Rightarrow EAE \mid (E) \mid -E \mid id$ $A \Rightarrow + \mid - \mid * \mid /$

Soln. $E \Rightarrow EAE$

$E \Rightarrow (E)$

$E \Rightarrow -E$

$E \Rightarrow id$

$A \Rightarrow + \mid - \mid * \mid /$

$E \overset{lm}{\Rightarrow} -E$

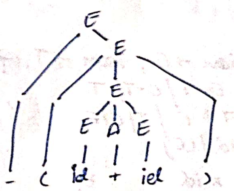
$\overset{lm}{\Rightarrow} -(E)$

$\overset{lm}{\Rightarrow} -(EAE)$

$\overset{lm}{\Rightarrow} -(id\,AE)$

$\overset{lm}{\Rightarrow} -(id+E)$

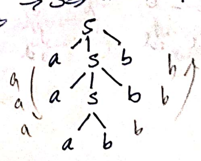
$\overset{lm}{\Rightarrow} -(id+id)$



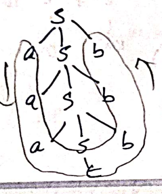
Show that the grammar which can construct the string aaabbb. Construct the parse tree to accept this string

$S \rightarrow aSb \rightarrow S \Rightarrow aSb \mid ab \mid \epsilon$

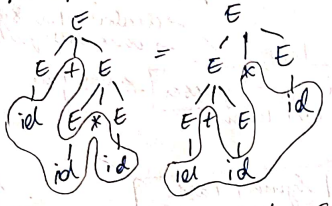
$S \rightarrow ab \mid \epsilon$



$S \Rightarrow aSb \mid \epsilon$

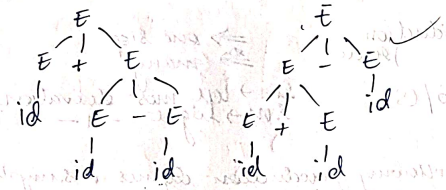


Ambiguity → more than 1 parse tree
E → E+E | E-E | E*E | id id+id*id



2+(3×6)=20 (2+3)×6 = 36 ✗

id+id-id



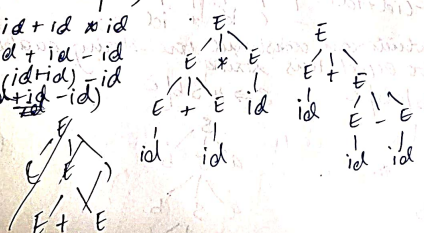
2+(6-3)=5 (2+6)-3=5

2+6-3

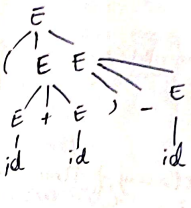
order

Expression → E, Term → T, factor → F,
E → E+T | E-T | T
T → T*F | T/F | F
F → id | (E)

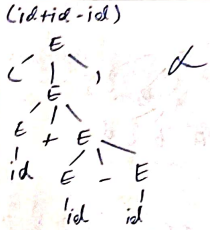
id+id*id
id+id-id
(id+id)-id
(id+id-id)



(id+id)-id



(id+id-id)



Left factoring
Non deterministic grammars → Deterministic grammars

A → αβ1 | αβ2 | αβ3 | αβ4
↳ A → αA'
A' → β1 | β2 | β3 | β4

S → aSSbS | aSaSb | abb | b
~~S → aSSbS~~ S → aS' | b
S' → SSbS | SaSb | bb
↳ S' → SS'' | bb
↳ S'' → SbS | aSb

S → bSSaaS | bSSaSb | bSb | a
S → bSS' | a
S' → SaaS | SaSb | b
~~S' → SaS'' | b~~
S'' → aS | Sb

E → T+E | T-E
↳ E → TE'
E' → +E | -E

S → iEtS | iEtSeS | a
E → b
S → iEtSS' | a
E → b S' → E | eS

S → a | ab | abc | abcd
~~S → ab~~ aS'
S' → ~~ε~~ | bS''
S'' → cS'''
S''' → d

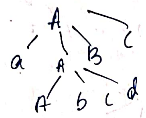
A → aAB | aBc | a
A → aA'
A' → AA' | Bc
A'' → B | c

→ LA → parser →
CFG → parse tree

Error recovery strategy
→ panic mode recovery
→ phrase level recovery
→ error production
→ global correction

Parser
top down
recursive descend
non recursive descend
LL(1)
↳ LMD
Bottom up
op. precedence
LR
LR(0) SLR(1)

S → aABe
A → Abc | b
B → d
w → abbcde



Soln. S → aABe
A → Abc
S → aAbcBe
S → abbcBe
B → d S → abbcde

Q.) S → aBDh
B → cC
C → bC | ε
D → EF
E → g | ε
F → f | ε

	First	Follow
S → aBDh	{a}	{$}
B → cC	{c}	{g, f, h}
C → bC / ε	{b, ε}	{g, f, h}
D → EF	{g, f, ε}	{f, h}
E → g / ε	{g, ε}	{f, h}
F → f / ε	{f, ε}	{g, f, h}

when ε appears D → EF: FIRST(D) = FIRST(E) ∪ FIRST(F)

Soln. follow: $ → start symbol; S not present in ②

if nothing follows C then write prev follow.

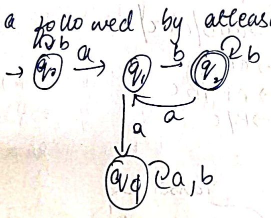
B → D
first of D → {g, f, ε}
since ε is there so B → h

Q.) E → TE'
E' → ε / +TE'
T → FT'
T' → ε / *FT'
F → id / (E).

Soln.

	First	Follow
E → TE'	{id, (}	{), $}
E' → ε / +TE'	{ε, +}	{), $}
T → FT'	{id, (}	{), $, +}
T' → ε / *FT'	{ε, *}	{), $, +}
F → id / (E)	{id, (}	{), $, +, *}

each a followed by atleast 1b



post fix

RE → (a|b)* b (a|b)*
S → AbA
A → aA / bA / ε

([0-2][0-9] / [3][0-1]) [1-] [Jan / Feb / Mar / Apr / May / Jun / Jul / Aug / Sep / Oct / Nov / Dec] [1-] [0-9][0-9][0-9][0-9]

consider the CFG given
S → 0S1 | 01

check whether the G is suitable for predictive parsing (non recursive). If not make it suitable and write modified CFG. Compute FIRST and FOLLOW set of all non-terminals in the CFG

S → 0S'
S' → S1 | 1

Non. terminal	First	Follow
S	{0}	{$, 1}
S'	{0, 1}	{1, $}

first term so add $; check RHS & see where S appears & check what follows that – S1

S → 0S': S' appears at end so check follow of its head

Predictive parse table

NT \ I/P	0	1	$
Non terminal S	S → 0S'		
S'	S' → S1	S' → 1	

I/P + $ ($ indicates end of I/P)

Blank spaces are errors

CFG represented as A → α

context free: LHS has no context

For every production of form A → α (check if its of A → α form)

① (S → 0S' (find first of RHS) find first of this → 0
∴ M [S, 0]

② (S' → S1 → first(S1) = 0 first(S)
∴ M [S', 0]

③ (S' → 1
∴ M [S', 1]

1. For each terminal 'a' in FIRST(α), add A → α to M[A, a]
2. If ε is in FIRST(α), then for each terminal 'b' in FOLLOW(A), add A → α to M[A, b]
   If ε is in FIRST(α) and $ is in FOLLOW(A), add A → α to M[A, $] as well
3. All blank entries are errors

S → AaAb / BbBa   its of the form A → α
A → ε
B → ε

NT \ I/P	a	b	$
S	S→AaAb	S→BbBa	
A	A→ε	A→ε	
B	B→ε	B→ε	

S → AaAb
first( ) = a
M[S, a]

S → BbBa
first( ) = b
M[S, b]

A → ε   first of (A) = ε
so check follow of A = a, b   S → AaAb
M[A, a], M[A, b]

B → ε
first(B) = ε
so check follow(B) = b, a
M[B, a] M[B, b]

E → TE'
E' → +TE'/ε
T → FT'
T' → *FT'/ε
F → (E) / id

Compute FIRST & FOLLOW of all

NT \ I/P	+	*	(	)	id	$
E	~~E→TE'~~	~~E~~	E→TE'		E→TE'	
T			T→FT'		T→FT'	
E'	E'→+TE'			E'→ε	~~E~~	E'→ε
T'	T'→ε	T'→*FT'		T'→ε		T'→ε
F			F→(E)		F→id	

E → TE'
first(TE') → (, id
M[E, (], M[E, id]

E' → +TE'
M[E', +]

E' → ε,), $
M[E',)]
M[E', $]

T → FT'
M[T, (], M[T, id]

T' → *FT'
M[T', *]

T' → ε
follow(T') → $, +,)
M[T', $] M[T', +] M[T',)]

E' → ε
follow(E') → follow(E) → start symbol so '$'
↳)

F → (E)
M[F, (]

F → id
M[F, id]

follow(T') → $
↳ follow(T)
↳ follow(E')
↳ follow(E)
↳)

Q) S → iEtSS'
S' → eS | ε

NT \ I/P	i	t	e	$
S	S → iEtSS'			
S'			S' → eS S' → ε	S' → ε

Q) S → (L) | a
L → L, S | S

Q) S → CC
C → cC | d

Q) S → aBa
B → bB | ε

NT \ I/P	(	)	a	,
S	S → (L)		S → a	
L				

Eg'
S → AB
A → 0S1 | 1S
B → 0S1 | 1S1

I/P = 0S1S1

$	0S1S1$
$0	S1S1$
$0S	1S1$
→ conflict	
$A	1S1$
$A1	S1$
$A1S	1$
$A1S1	$
$AB	$
$S	$

S → iEtSS'   S' → eS
first(   ) → i   ε

follow(S$)
↳ ntg, start sym
$, follow(S')
ntg, follow(S')
ntg
first(S')

S →

S → (L)   S → a
(a
M[S, (]
L → L, S

Stack	I/P	Action
$	id*id$	shift
$id	*id$	F → id
$F	*id$	T → F
$T	*id$	shift
$T*	id$	shift
$T*id	$	F → id
$T*F	$	T → T*F
$T	$	E → T
$E	$	accept

E → E+T | T
T → T*F | F
F → (E) | id

LR parsing ⟶ no backtracking, fast error detection, more general than LL parser, efficient
L → left to right
R → rightmost derivation in reverse

A → .XYZ
shows wt has to be passed
→ X.YZ (X passed)
→ XY.Z (XY —"—)
→ XYZ. (XYZ —"—)

LR(K) → constant usually 1

① S → L=R
② S → R
③ L → *R
④ L → id
⑤ R → L

- Augment the grammar
↳ S' → .S

I₀ state
S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

S → I₁: [S' → S.] (dot gets shifted)
L → I₂: [S → L.=R, R → L.]
R → I₃: [S → R.]
* → I₄: [L → *.R, R → .L, L → .*R, L → .id] — non terminal aft dot so include NT's production
id → I₅: [L → id.]

I₂ = → I₆: [S → L=.R, R → .L, L → .*R, L → .id]

From I₄ (unique symbols):
R → I₇: [L → *R.]
L → I₈: [R → L.]
* → I₄: L → *.R, R → .L — already done so prev bcms I₄
id → I₅: id. already there

I₆
S → L=.R
R → .L
L → .*R
L → id

R → I₉: [S → L=R.]
L → I₈
* → I₄
id → I₅

	S	L	R
First	{*, id}	{*, id}	{*, id}
Follow	{$}	{$, =}	{$, =}

Steps	Action			Goto		
	=	*	id	$	S	L	R
0		S_4	S_5		1	2	3
1				accept			
2	S_6, S_5			S_5			
3				S_2			
4		S_4	S_5			8	7
5	S_4			S_4			
6		S_4	S_5			8	9
7	S_3			S_3			
8	S_5			S_5			
9				S_1			

Construct a TM for a fn. of $x+1$

$f(x) = x+1$

$q_0 000B$

	0	B	B
q_0	$(q_0, 0, R)$	$(q_1, 0, R)$	

$q_0 000B$
$0\ q_0 00B$
$00\ q_0 0B$
$000\ q_0 B$
$0000B$

0/0 → R/0

$w = ab$
$w^R = ba$
$ww^R = abba$
$wcw^R = ababa$
$C = a/b = abbba$

a b B abbaB

q_0 (q_1, B, R) (q_4, B, R)

q_1 (q_1, a, R) (q_1, b, R) (q_2, B, L)

q_2 (q_3, B, L)

q_3 (q_3, a, L) (q_3, b, L) (q_0, B, R)

q_4 (q_4, a, R) (q_4, b, R) (q_5, B, L)

q_5 — (q_3, B, L) (q_6, B, L)

X → grammar symbol a → attri X.a → value at node X

$E \Rightarrow E+T$ E.val → E.val + T.val

Synthesized
$A \to BC$) A → parent
A.S = B.S
A.S = C.S
A.S = D.S
parent node A take from B, C, D

Inherited
$C_i = A.i$
$C_i = B.i$
$C_i = D.i$
node C takes values from parent-A & siblings B, B

Types of SDD

S-attributed SDD
A → BC)
A.S = B.S
A.S = C.S
A.S = D.S
Bottom-up parsing
semantic rule
@ end of prod

L-attributed SDD
A → XYZ
Y.S = A.S
Y.S = X.S
top down, left-right
inherit from parent/left sibling
& rules placed anywhere only

$E \to E+T$ E.val → E.val + T.val
$E \to T$ E.val → T.val
$T \to$ digit T.val → digit.lexval

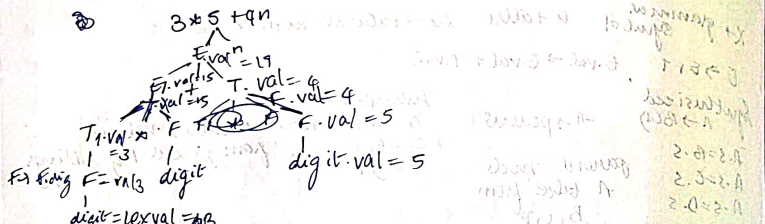
$E \to TE'$
$E' \to +T\{E'.inh = E.inh + T.val\}E' \mid \epsilon$
$T \to$ digit {T.val = digit.lexval}

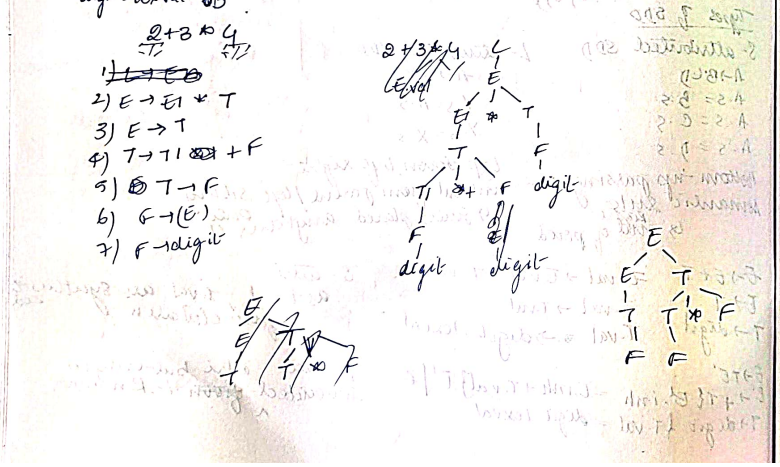
S-attri
as T.val, T.val are synthesized from children

L-attri attri but only is inherited L-R manner

SVDD ~~to create~~ a simple desk calc for eval of exp / annotated parse tree for 3*5+4n

Production	Semantic Rules
1) L → En	L.val = E.val
2) E → E1 + T	E.val =
3) E → T	
4) T → T1 * F	
5) T → F	
6) F → (E)	F.val = E.val
7) F → digit	F.val = digit.lexval



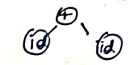


Application of syntax directed translation

Const. of syntax trees

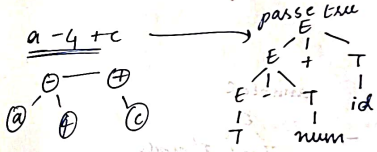
- syntax tree node representation E1+E2 has a label + & two children
- if node is leaf, an additional field hold the lexical value for the leaf.
- Construction fn. leaf(op, val)
- if node is an int. node, construction fn. node(op, c1, c2, c3, ...cn)

$E_1 + E_2$ id + id

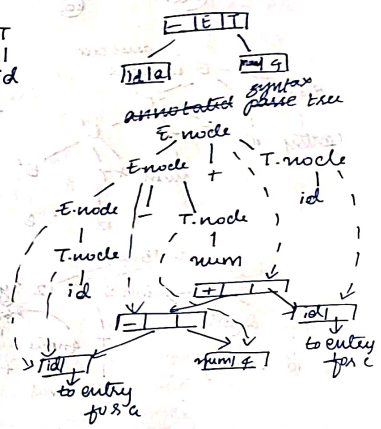


S- attributed defn.

prob:	Semantic rule:
$E \to E_1 + T$	E.node = new Node ('+', E1.node, T.node)
$E \to E_1 - T$	E.node = new Node ('-', E1.node, T.node)
$E \to T$	E.node = T.node
$T \to (E)$	T.node = E.node
$T \to id$	T.node = new Leaf (id, id.entry)
$T \to num$	T.node = new Leaf (num, num.val)



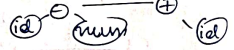
1) p_1 = new Leaf (id, entry a);
2) p_2 = new Leaf (num, 4);
3) p_3 = new ~~Leaf~~ Node ('-', p_1, p_2);
4) p_4 = new Leaf (id, entry-c);
5) p_5 = new Node ('+', p_3, p_4);

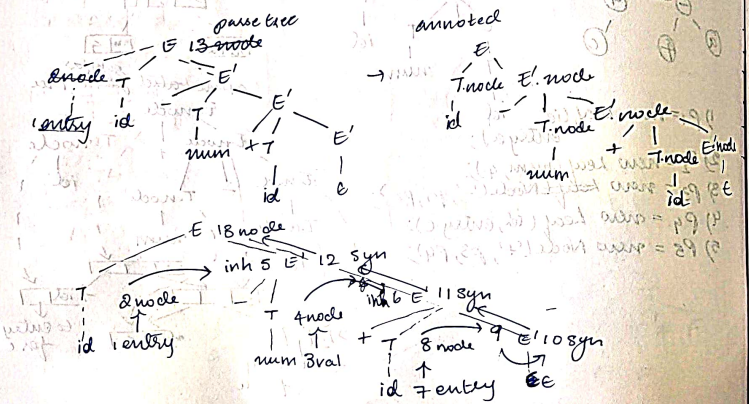


Down passing - L attributed

Production	Semantic rules
$E \to TE'$	$E.node = E'.syn$ $E'.inh = T.node$
$E' \to +TE_1'$	$E_1'.inh = new\ Node('+', E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \to -TE_1'$	$E_1'.inh = new\ Node('-', E'.inh, T.node)$ $E_1'.syn = E_1'.syn$
$E' \to \epsilon$	$E_1'.syn = E'.inh$
$T \to (E)$	$T.node = E.node$
$T \to id$	$T.node = new\ leaf(id, id.entry)$
$T \to num$	$T.node = new\ leaf(num, num.val)$

Dependency graph for a-4+c

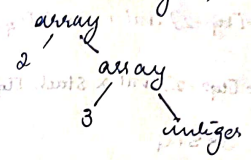




Structure of a type

int[2][3] → array of 2 arrays with 3 integers

int[2][3] = array(2, array(3, integer))



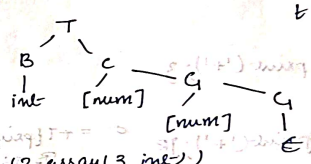
The op. array takes 2 parameters → num → type

~~if type~~

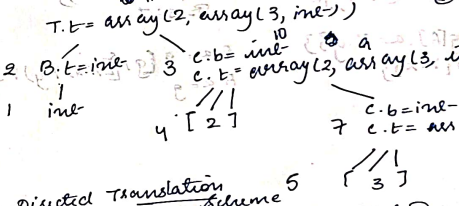
Prod.	Semantic rule
$T \to BC$	$T.t = C.t$ $C.b = B.t$
$B \to int$	$B.t = integer$
$B \to float$	$B.t = float$
$C \to [num]\ C_1$	$C.t = array(num.val, C_1.t)$ $C_1.b = C.b$
$C \to \epsilon$	$C.t = C.b$

C.b inherited

int[2][3]



T.t = array(2, array(3, int))



T → non terminals

B → one of basic types (int/float)

C → components consisting of seq. of integers

b → inh attri.

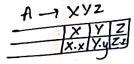
t → syn attri.

Syntax-Directed Translation Scheme

→ complementary notation to Syntax Directed Defn. (SDD)

→ SDTs is a CFG with ~~pgm fragments~~ semantic actions embedded with prod. bodies

→ can appear at any pos. within a prod. body.

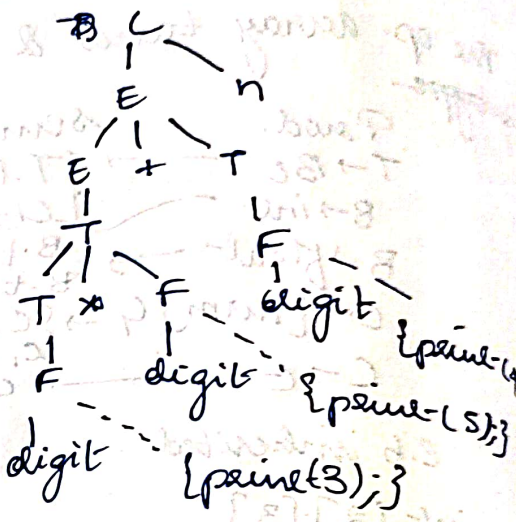


Production	Actions
$L \to En$	{print(stack[top-1].val); top = top - 1;}
$E \to E_1 + T$	{stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2;}
$E \to T$	
$T \to T_1 * F$	{stack[top-2].val = stack[top-2].val × stack[top].val; top = top - 2;}

infix $B \to X\{a\}Y$

$L \to En$
$E \to \{print('+');\}\ E_1 + T$
$E \to T$
$T \to \{print('*');\}\ T_1 * F$
$T \to F$
$F \to (E)$
$F \to digit\ \{print(digit.lexval);\}$

3*5+4



Eliminating left-recursion

$A \to BR$
$R \to \alpha R \mid \epsilon$

Eg. $E \to E + T\ \{print('+');\}$
$E \to T$

Ans: $E \to TR$
$R \to +T\{print('+');\}R$
$R \to \epsilon$

$\alpha = +T\{print('+');\}$
$B = T$

$A \to AY\ \{A.a = g[A.a, Y.y]\}$
$A \to X\{A.a = f[X.x]\}$

$A.a = g[g[f[X.x], Y.y], Y.y]$
$A.a = g$

Ans $A \to XR$
$R \to YR \mid \epsilon$