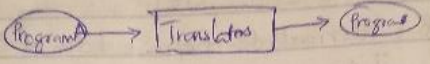
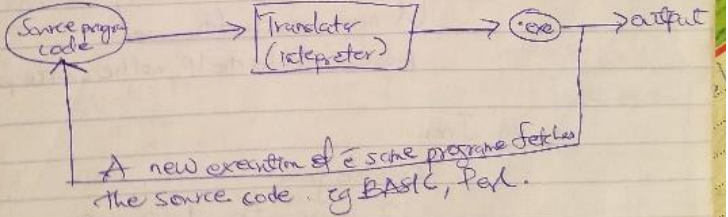
INTRODUCTION

TRANSLATORS : is a program that transforms a program written in one form

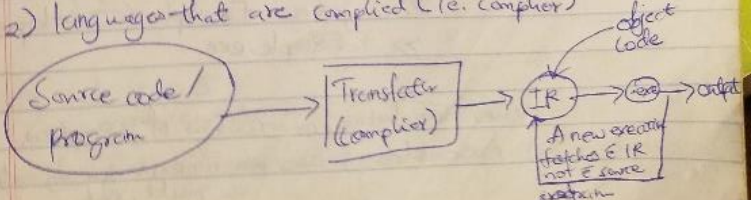


TYPES OF TRANSLATORS

Interpreters : executes program on the fly ie as statement are fetched results are gotten. transforms a program written in a high-level language into an executable form by fetching statements from the source program and executing them "on the fly"; statement-by-statement.



2) languages that are complied (ie. complier)



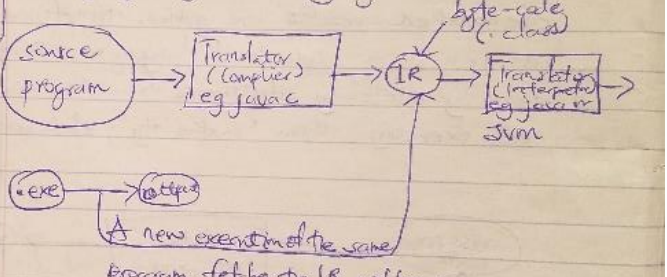
The syntax of a language is defined by the grammar of the program.

$G = (S, N, T, P)$ a grammar is defined by a starting symbol S, a nonterminals N, Terminals and production rules P.

All statements are transformed into IR before execution begins. eg C/C++, Cobol, Pascal, Ada, Visual Basic .NET

IR = Intermediate Representation of the source code or Intermediate code.

3) Languages generated by byte-code.



eg Java

Example.java

>> javac Example.java ← compiler

Example.class ← IR

>> java Example ← interpreter

>> Example.exe

Other translators, or cousins of compilers.

i) Assemblers : use mnemonics eg addl, mult, sub

signed mult and unsigned mult

int x = 5 ;

delimeter

literals

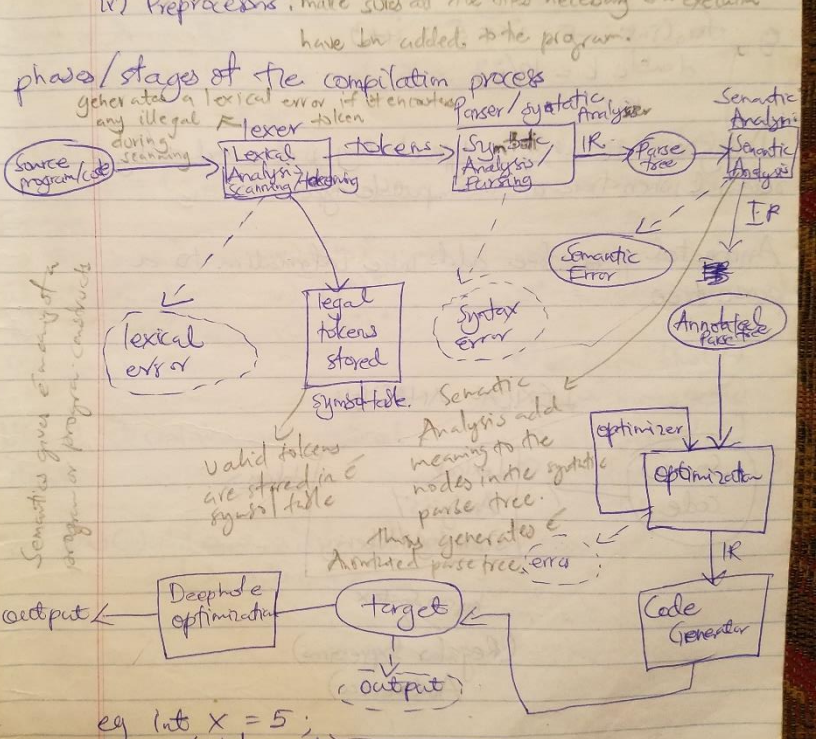
keyword identifier operator

ii) Loader : reflects class code to check for malicious code.

iii) Linker : link a program file during execution.

iv) Preprocessor : make sures all the files necessary for execution have been added to the program.

phases / stages of the compilation process



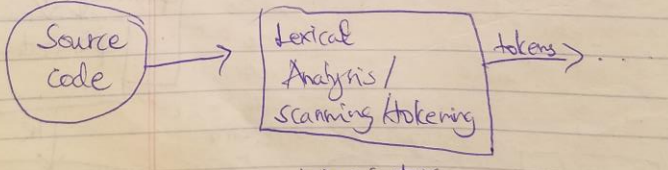
Semantic Analysis

eg `int x [5.0] = {1,2,3,}; // Semantic error`
`int x [3] = {1,2,5}; // Correct`

eg
```
for (int i=5, i<=5; i++)
    double k = 10/i;
        10/0
```

- During semantic analysis type checking is done and a flag is released when there are incompatible types
- Annotated parse tree adds more information to a parse tree

LEXICAL ANALYSIS



PROPERTIES OF REGULAR EXPRESSIONS

i) The symbol "x" is a regular expression (RE)

2) "xyz" is a ~~string~~ which is x followed by y

3) $\emptyset$ or { } or the ~~Empty~~ or null is a RE

4) ε or λ is RE. ε or λ is an empty string

~~5) ε or λ is RE~~

5) (concatenation (.) $a.b = ab$ is a RE

NB ~~ab~~ $ab \neq ba$

6) Union

$a \cup b = \{a, b\}$ is a RE

7) Kleene star (*):

eg a^* means zero(0) or more a's

$a^* = \{a^0, a^1, a^2, a^3, \ldots\}$

$a^* = \{\lambda, a, aa, aaa, \ldots\}$

~~Example~~

i) $a.(b \cup c)$

ii) $a^*.(b \cup c)$

iii) $a.(b \cup c)^*$

iv) $(10)^*10$ ✓

v) $(r^*)^*$ ✓

vi) $(\varepsilon^*)^*$ ✓

vii) $a.(b \cup c)$

8) ~~RE~~ Kleene Plus (+)

eg a^+ means one (1) a

$a^+ = \{a^1, a^2, a^3, \ldots\}$

$\Rightarrow a^+ = \{a, aa, aaa, \ldots\}$

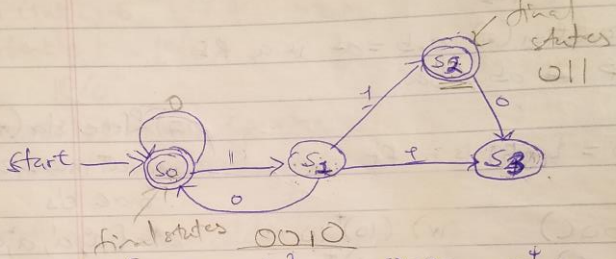
NB: A set contains well-defined objects

$a^*.(b \cup c) = \{a^0, a^1, a^2, a^3, \ldots\}.\{b \cup c\}$

$= \{\lambda, a, aa, aaa\}$

A LEXER is implemented as an automaton. Automaton is a finite state machine (FSM) with inputs, but no output

eg.



States = $S = \{S_0, S_1, S_2, S_3\}$
Final states $F = \{S_0, S_2\}$ 0010

i) 0000 $= 0^4$ is accepted or recognized

ii) 0011 $= 0^2 1^2$ is accepted at state S_2.

iii) 00101 not accepted since 00101 ends at state S_1 which is not a final state.

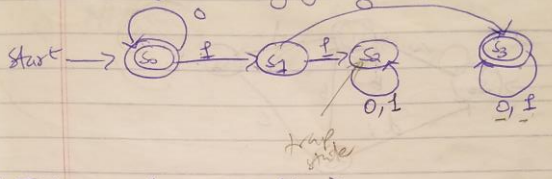
An automaton can be deterministic or Non deterministic

FINITE-STATE AUTOMATA (FSA)

Definition: A finite-state automaton $M = (S, I, f, s_0, F)$ consists of a finite set S of states, a finite input alphabet I, a transition function f that assigns a next state to every pair of state and input, an initial state s_0, and subset F of S consisting of final states.

Assignment

Find $L(M_3)$ the language of M_3.



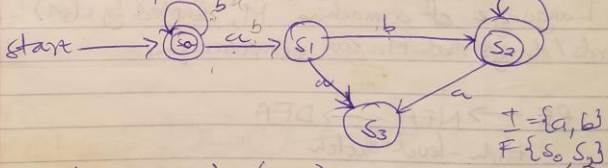
2) TYPES AUTOMATA (FSA):

1) Deterministic Finite-state Automata (DFA)

Given a (state, input) we can easily determine the next state.

NB: For multiple paths/edges inputs should be unique

eg.

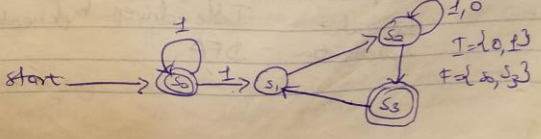


ii) There is/are no λ-(or ε) transition(s)

ii) Non deterministic Finite-state Automata

i) Given a pair of (state, input) there is no unique next state.

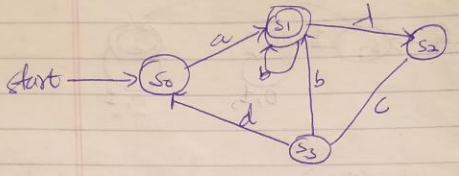
eg 1)



RE - Regular Expression.

ii NFA may contain λ-(or ε) transitions

eg.



NFA and DFA are equivalent.

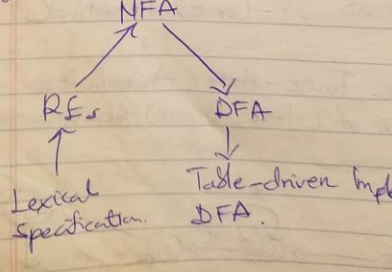
NFA $\Longleftrightarrow$ DFA

NFA and DFA machines can be designed to recognise or accept the same languages

Language of a machine M, denoted by L(M) is all words / strings that M can accept/recognise.

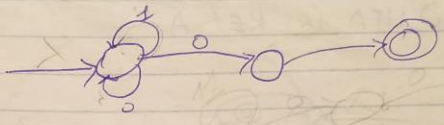
RE $\longrightarrow$ NFA $\longrightarrow$ DFA

High-level sketch

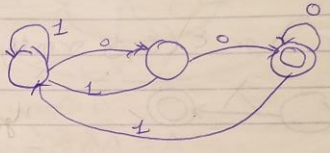


For a given language the NFA can be simpler than the DFA

eg NFA :



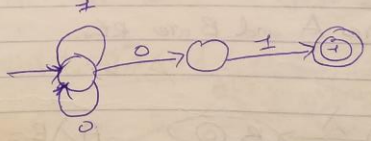
DFA:



This implies NFA $\Leftrightarrow$ DFA

DFA can be exponentially larger than NFA.

NFA:
put: 101

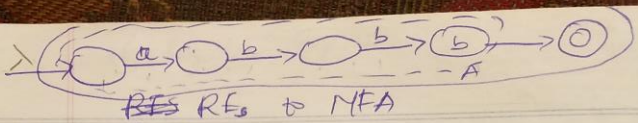


101 is recognized/accepted by the NFA

i) NFA's and DFA's recognize/accept the same set of languages. (regular languages generated by regular expression

ii) DFA's are ~~easily~~ easier to implement. → There are no choices to consider. ie no multiple paths with same input

Date Completed: .....

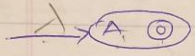


~~RES~~ RE's to NFA

For each kind of RE, define an NFA:

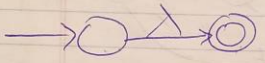
Notation:

i) NFA for RE A.



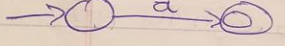
eg $A = ab^2 = abb$

ii) NFA for λ or ε (Empty or null string)



λ is recognised by a FSA

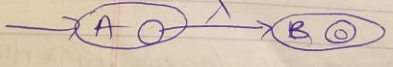
iii) for input a



a is recog by a FSA

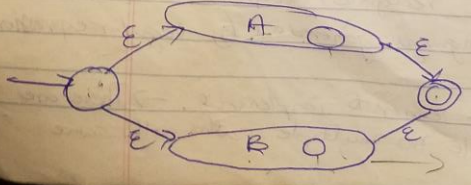
iv) for AB, where A and B are REs.

AB is recognised by an FSA $A\lambda B \Rightarrow AB$



$A\lambda B = AB$

v) for A|B ie A or B but not both.



$\varepsilon A \varepsilon = A$

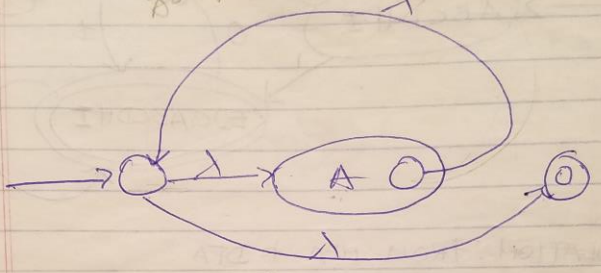
$\varepsilon B \varepsilon = B$

$\varepsilon A \varepsilon \Rightarrow A$

$\varepsilon A \lambda B \varepsilon \Rightarrow AB$

vii) for A^*, where A is a RE.

$A^* = \{\lambda, A, AA, AAA, \ldots\}$

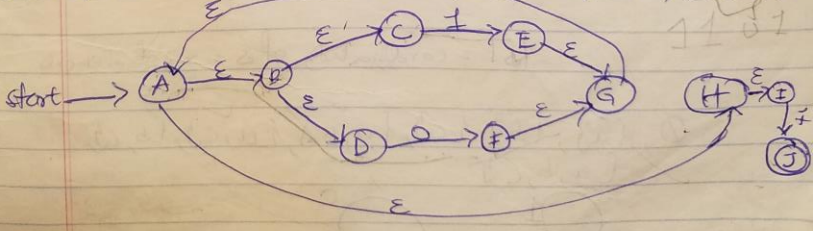


$\lambda A \lambda \lambda = A$

$\lambda A \lambda \lambda A \lambda \lambda = AA$

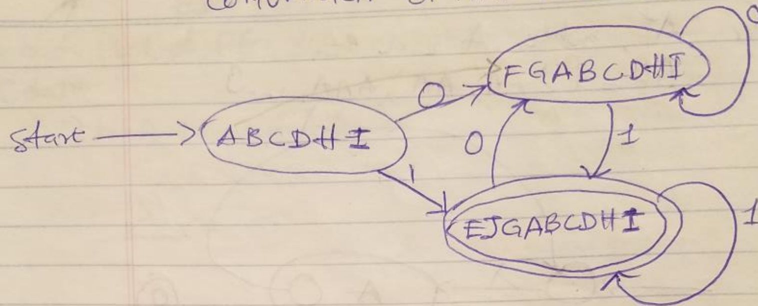
Q4) Give the NFA for the RE $(1|0)^*1$.

$(1|0)^*1 = \{1, 0)^*1 = \{1, 11, 01, 111, 001, 101, 011, \ldots\}$



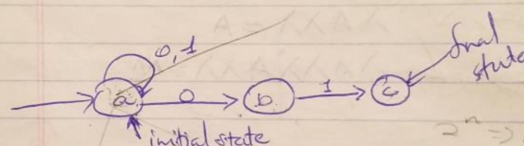
$11 = 1\varepsilon\varepsilon 1\varepsilon 1\varepsilon = 11$, $13\varepsilon = 1?$

CONVERSION OF NFA TO DFA



REAL CALCULATION: FROM NFA to DFA

eg.
M_1:

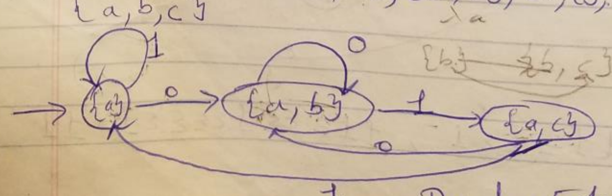


set of states = S = {a, b, c}
Form subsets of S:
No. of subset of S = $2^{|S|} = 2^3 = 8$

$|S|$ = cardinality of S = no. of elements in S

∅, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}

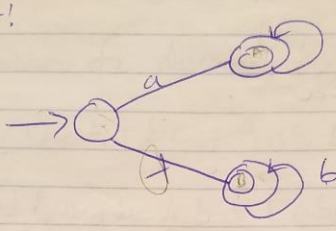
M_2:
DFA



The other 5 states are unreached

Eg

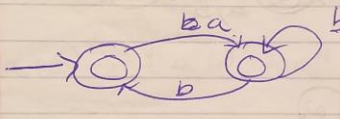
NFA:



Input alphabet

$\Sigma = \{a, b\}$

DFA:



Both the NFA and DFA accept the language $\{b^n, ab^n : n \geq 0\}$

$\Rightarrow$ NFA $\Leftrightarrow$ DFA

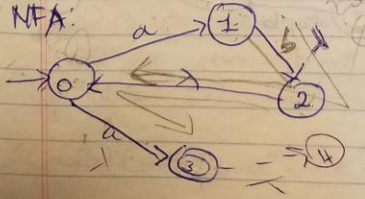
when $n = 0 \Rightarrow b^0 = \lambda$

$ab^0 = a\lambda = a$

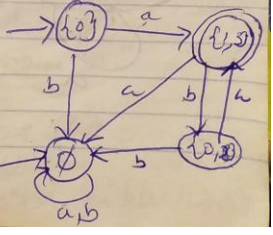
$n = 1 \Rightarrow b^1 = b$

$ab^1 = ab$

Eg NFA:



DFA:



Both NFA and DFA accept $L(ab)^n a : n \geq 0$

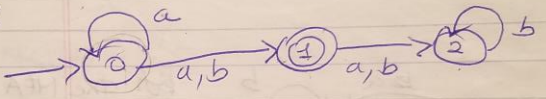
NFA: $n = 0$

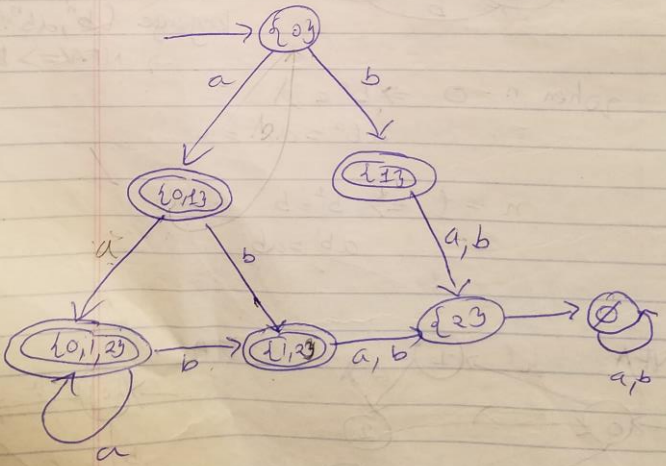
$(ab)^0 a = \lambda a = a$

$(ab)\, a = aba$

eg

NFA:





MORE ON REGULAR EXPRESSIONS

Mechanism for describing languages!

eg. i) $ab + bc = \{ab, bc\}$

ii) $\{ab+bc\} \cdot b = \{abb, bcb\}$

$\{ab, bc\} \cdot b$ ← concatenate.

iii) $(a+b)^* = \{a,b\}^*$ = all strings on $\Sigma = \bar{\Sigma} = \{a,b\}$

iv) $(a+\lambda) \cdot c = \{ac, \lambda c\} = \{ac, c\}$

$(a, \lambda) \cdot c \quad \{ac, c\}$

v) $L((a+\lambda)(b+\lambda)) = \{ab, a\lambda, \lambda b, \lambda\lambda\}$

$= \{ab, a, b, \lambda\}$

$(a,\lambda)(b,\lambda)$

Formally:

i) $+$ is Union $\{ab, a\lambda, \lambda b, \lambda\lambda\}$

ii) $\cdot$ is concatenate

iii) $*$ star-closure (Kleene star)

Note: $(\cdot)$ has higher priority than "+"

$a^*(a+b)(a+b) = a^*(aa + ab + ba + bb)$

$= a^*\{aa, ab, ba, bb\}$

$= \{a^n aa, a^n ab, a^n ba, a^n bb : n \geq 0\}$

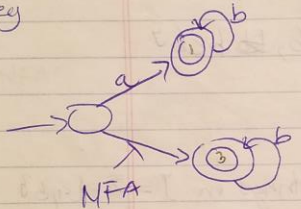
$a(a+b)^* \iff a((a+\lambda)(b+\lambda))^*$

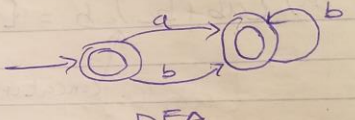
DFA $\iff$ NFA

Automata are equivalent if they accept the same langu

DFA $\subseteq$ NFA

eg





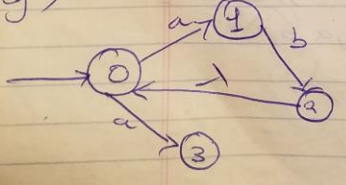
DFA

Both accept = $L = \{b^n, ab^n : n \geq 0\}$

for NFA when $n = 0$: $b^0, ab^0 = \lambda, a$

$n = 1 = b^1, ab^1 = b, ab$

eg 2)



2^4

From subsets:

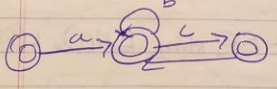
$\phi, \{0\}, \{1\}, \{2\}, \{3\}, \{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{0,1,2\}, \{0,1,3\}, \{1,2,3\}, \{0,1,2,3\}, \{0,2,3\}$

EQUIVALENCE of RE's and NFA

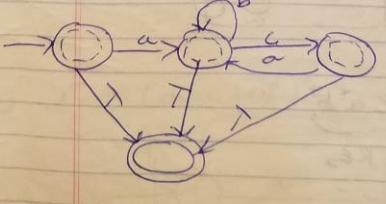
A language can be represented by a regular expression (RE) if and only if (iff) it is accepted by some NFA. In other words, REs correspond to regular languages (RLs). The proof consists of 3 parts.

1) For every NFA, there is an equivalent NFA with exactly one final state.
2) For every RE, there is an equivalent NFA.
3) For every NFA, there is an equivalent RE.

1) Given an NFA, we can construct an equivalent NFA with one final state

eg 

Make a new final state and add a λ-transition from each old final state to the new final state.



$ab^*c\lambda$
$\equiv ab^*c$
$ab^*\lambda = ab^*$
$aca\lambda = aca$

2) For every RE, there is an NFA that accepts the corresponding language.

Example:

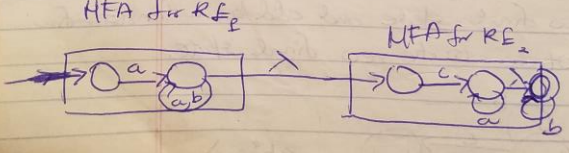
RE: $(a+b)c^*a = \{a,b\}c^*a$

NFA

This actually looks like an a DFA bear in mind that DFA is a subset of NFA thus its NFA.

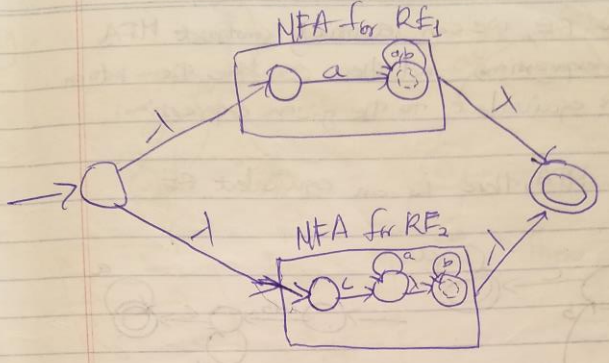
eg 2) $\underbrace{a(a+b)^*}_{RE_1} \cdot \underbrace{ca^*b^*}_{RE_2}$

Concatenating RE_1 strings with RE_2 string, $L(RE_1) \cdot L(RE_2)$



eg 3) RE: $\underbrace{a(a+b)^*}_{RE_1} + \underbrace{ca^*b^*}_{RE_2}$

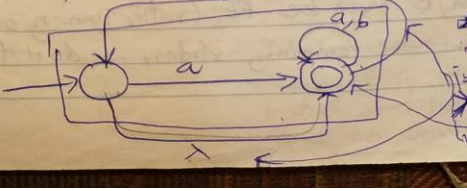
$L(RE_1) \cup L(RE_2)$



eg 4) r^*, where r is a RE

Concatenating any number of "r" strings, $\in L(r^*)$

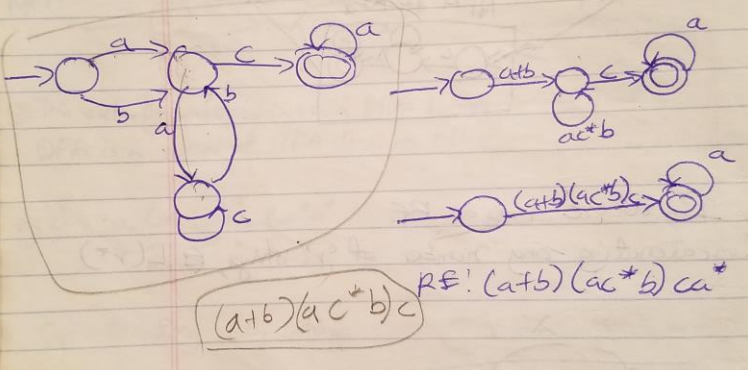
eg $r^* = (a(a+b)^*)^*$ $r = a(a+b)^*$



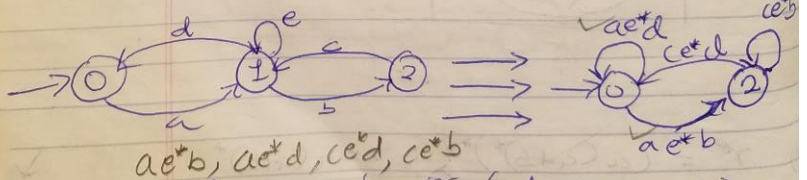
#Always remove the intermediate node.

Thus, given a RE, we can recursively construct NFAs for its sub-expressions, and then combine them into an NFA that is equivalent to the given expression.

3) For every NFA, there is an equivalent RE

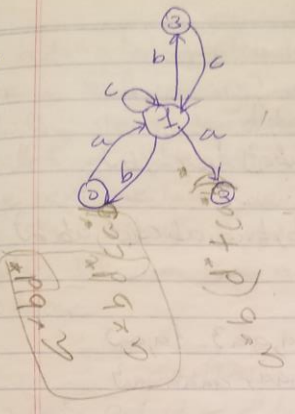


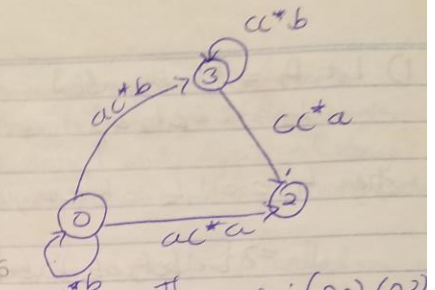
REMOVING A STATE (GENERAL CASE)



NB a, b, c, d, e may be REs (rather than symbols)
If there are more remaining states, we do it for every pair.

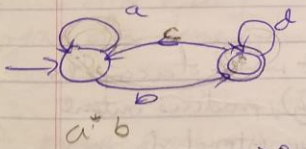
eg.





Three pairs: (0,2), (0,3), (2,3)

EXPRESSION FOR A TWO-STATE AUTOMATON (General Case)



a^*b

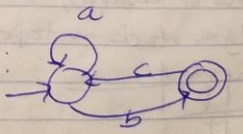
RE: $a^*b(d+ca^*b)^*$

$(a^*)^* = a^*$

0: $a^*b(d+ca^*b)^0$

$\lambda b \lambda = b$ ✓

1: $a^*b(d+ca^*b)^1$



eg: $a^*b(ca^*b)^*$

Thus given any NFA, we may remove all its states except the initial and the final states, and then convert it to a RE.

1) Let $L_1 = \{a, ba, abc\}$
$L_2 = \{bcb, b\}$

then $L_1 L_2 = L_1 L_2 = \{a, ba, abc\}\{bcb, b\}$

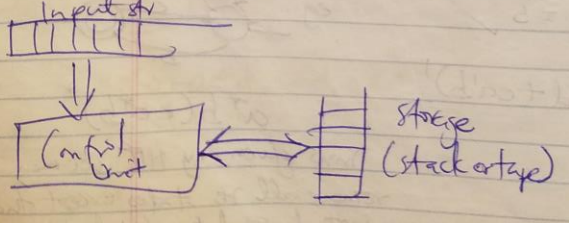
$\Rightarrow \{abcb, ab, babcb, bab, abcbcb, abcb\}$

2) Let $L = \{a, aa\}$
then $L^3 = L \cdot L \cdot L = \{a, aa\} \cdot \{a, aa\} \cdot \{a, aa\}$
$\Rightarrow \{aaa, aaaa, aaaaa, aaaaaa\}$

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots = \bigcup_{i=0}^{\infty} L^i$$

AUTOMATA

An automaton is an abstract model of a computer which accepts some input (a string), produces output (yes/no or a string), and may have internal storage (usually a stack or tape).



An automaton operates in a discrete time frame (like a real computer). A particular state of the automaton, including the state of the control unit, input and storage is called a configuration. The transaction from a configuration to the next one is a move.

If the output is yes/no, the automaton is called an acceptor. If the ~~string~~ output is a string, it is called a transducer.

SYNTATIC ANALYSIS

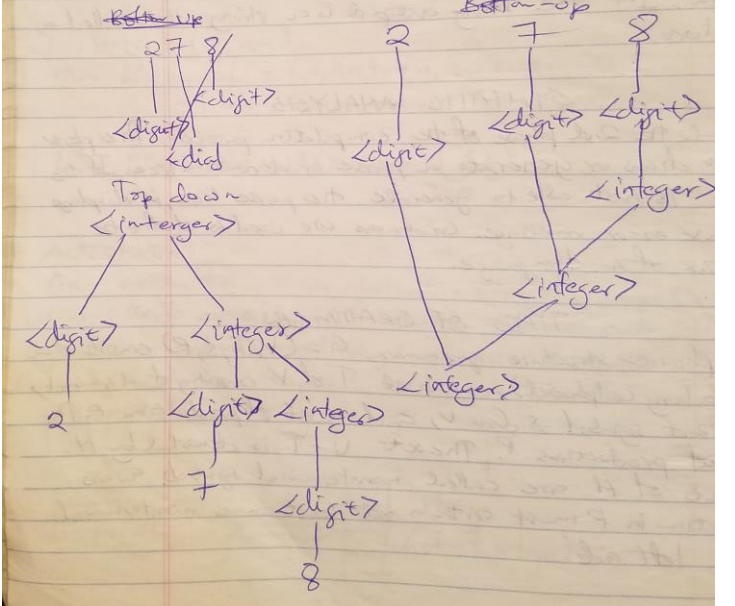
This is the 2nd phase of the compilation process. The parser tries to draw or generate a parse or derivation tree. If the parser is not able to generate the parse tree, it displays syntax error message. Grammars are used to define the syntax of a language.

TYPES OF GRAMMARS

A phrase structure grammar $G = (V, T, S, P)$ consists of a vocabulary (alphabet), a subset T of V consisting of terminal, a start symbol S from V, and a set of productions P. The set of productions P. The set $V - T$ is denoted by N. Elements of N are called nonterminal symbols. Every production in P must contain at least one nonterminal on its left side.

Top down approach: starts from the starting symbol and derive the terminals. Starting symbol is a non-terminal

Bottom-up parsing: the parse-tree is generated from terminals until the starting symbol is encountered or reached.



* If more than one parse tree can be generated for a single statement.

AMBIGUOUS GRAMMAR

Production rule:

<expression> ::= <expression> {+|-}* <term> | <term> | <expression> {+|-}* <term>

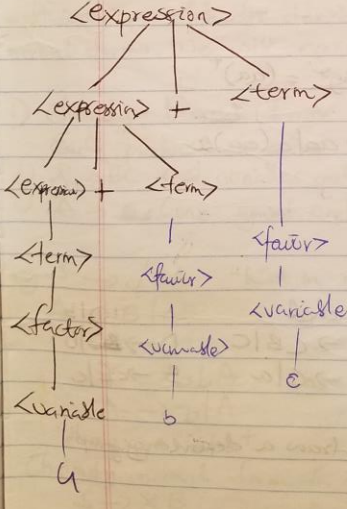
<term> ::= <factor> {*|/}* <factor> | <factor>

<factor> ::= <variable> | <literal>

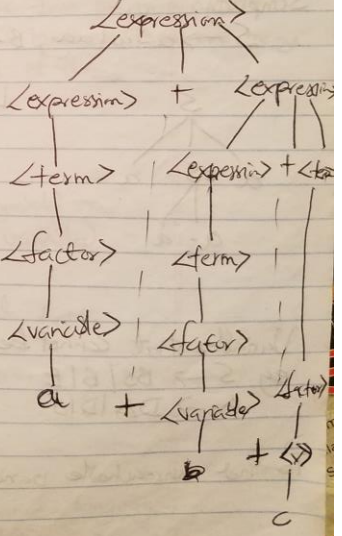
<variable> ::= a|b|c|...z|A|B|C|...Z

<literal> ::= string or numeric

Case 1: (a+b)+c



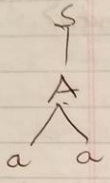
Case 2: a+(b+c)



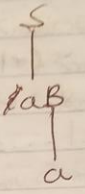
Production rule

$S \to A \mid aB \quad A \to aa \quad B \to a$

Case 1:

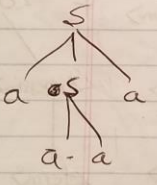


Case 2:



Simplify:

$S \to aSa \mid aa, \quad B \to b$



Language $= (aa)^+$

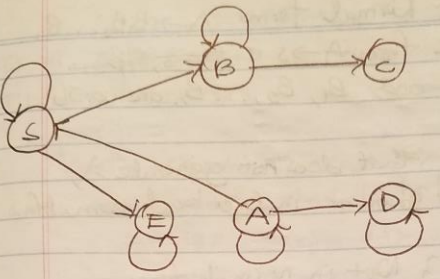
$S \to aa \mid a(aa)a$

Variables that cannot be reached

eg $S \to BS \mid B \mid E \quad B \to CB \mid C \quad D \to bD \mid b$

$A \to DA \mid D \mid S \quad C \to aC \mid a \quad E \to cE \mid c$

To find unreachable variables draw a dependency graph:



Reachable variables:
S, B, C, E

NORMAL FORMS

A normal form is a ~~reach~~ restrictive form of grammar that is sufficiently powerful for describing all languages. There are many normal forms. We will consider 2 of them.

1) Chomsky Normal Form

Each production is either

i) $A \to BC$ (2 variables and no symbols) or

ii) $A \to a$ (one symbol and no variables)

eg $S \to aBA$ Not in normal form.

$A \to B \mid AB$

or

$S \to abA$

$A \to b \mid A$

language: abb^+

Chomsky normal form for the same language:

$S \to XB$ $\quad A \to B \mid AY$ $\quad Y \to b$

$B \to YA$ $\quad X \to a$

sBi

Greibach Normal Form $A \to sB_1B_2 \ldots B_n$

Each production is of the form $A \to s$ or ~~$A \to s, sB_2 \ldots$~~

where s is a symbol and $B_1, B_2, \ldots B_n$ are variables

For every grammar that does not generate λ, there is an equivalent grammar in Greibach normal form

eg $S \to abA$
$A \to b \mid Ab$ } Not in normal form

$S \to aX_BA$
$A \to b \mid bA$
$X_B \to b$

eg $S \to aAbB$
$A \to aaA \mid a$
$B \to bB \mid b$

$S \to wX_CB$
$X_C \to Ab \mid aaA \mid a$
$B \to bB \mid b$

$S \to aAX_BB$
$A \to aX_aA \mid a$
$B \to bB \mid b$
$X_a \to a$
$X_B \to b$

PUSHDOWN AUTOMATA (PDA)

A pushdown automaton (PDA) is an automaton with

EXPRESSION

TYPES OF EXPRESSION:

PREFIX EXPRESSIONS:

Syntax:

<prefix-expression> ::= <operator><operand1><operand2>

<operator> ::= + | - | * | ↑ | ~ | ...

<operand1> ::= <operand>

<operand2> ::= <operand>

<operand> ::= <variable> | <number>

eg (i) + 2 4 = 2+4=6

(ii) * + 2 4 5

(2+4) * 5

⇒ 6 * 5 = 30

→ Scan from right to left if you see any operator put it in between two operands

2) INFIX / ALGEBRAIC EXPRESSION

Syntax:

<infix-expression> ::= <operand1><operator><operand2>

eg i) 2+4

ii) 2+4*5

2+20

= 22

3) POSTFIX/SUFFIX/REVERSE POLISH EXPRESSION

Syntax:

<operand1><operand2><operator>

eg:) 2 4 +

→ Move from left to right when you see an operand put it in between.

Operator	Precedence	Complement
Exponentiation ↑	4	0
×, /	3	1
Uniary minus ~	2	2
dyadic (+, -)	1	3
Assignment	0	4
Opening parentesis (	-10	—
closing ")	+10	—

Evaluate:

$2 + (5 - 2 * 3) + 15/3$

$2 +_1 (5 -_1 2 *_3 3)_{-10} +_1 15/_3 3$

$2 +_1 5^{+10} -_{11} \underset{1}{2} *_{13} 3 +_1 15/_3 3$

After finding the sum of complements add 1. (+1)

PARENTHESIS ALGORITHM

eg fully parenthesize the expression

$d = a + b * c - e \uparrow m$

Step 1: Assign operator precedence:

$d =_0 a +_1 b *_3 c -_1 e \uparrow_4 m$

Step 2: Find the complement of the operators:

$d =_4 a +_3 b *_1 c -_3 e \uparrow_0 m$

Step 3: Add back-to-back parentheses equal in number to the complement of the operators.

$d)))) = (((((a))) + (((b) * (c))) - (((e \uparrow m$

Step 4: Find the sum of the complements + 1

$\Rightarrow (4 + 3 + 1 + 3 + 0) + 1 = 12$

Step 5: Surround the whole expression with 12 opening and closing parentheses: Maintain what is in step 3 and add 6 brackets

$(((((((((((d)))) = (((((a))) + (((b) * (c))) - (((e \uparrow m))))))))))))$.

For human readability remove redundant parenthesis.

$(d = ((a + (b * c)) - (e \uparrow m)))$

Question $q = ab \uparrow c - r + d$

$q =_0 a_2 b \uparrow_4 -_1 /_3 m +_1 d$

$q =_4 a_2 b \uparrow_0 -_3 /_1 m +_3 d$

$q = (((a b \uparrow))) - ((( )/(m))) + (((d$

$q)))) = (((((a))) - ((b \uparrow c$

$q = \sim b \uparrow c - r / m + d$

$q =_0 \sim_2 b \uparrow_4 c -_1 /_3 m +_1 d$

$q =_4 \sim_2 b \uparrow_0 c -_3 /_1 m +_3 d$

q)))) = ((((( )) ~ ((b ↑ c))) - ((( ) / (m))) + (((d

Sum of complexity = 4+2+0+3+1+3 => ~~to~~ 14

((((((((((((((q)))) = (((( )) ~ ((b ↑ c))) - ((( ) / (m))) +
(((d)))))))))))))))

For human readability

CODE GENERATION

operator in source code	operator in generated instruction (mnemonic (op code = operational code)
+	ADD
−	SUB
*	MUL
/	DIV
↑	EXP
RECA	1/Acc [Reciprocal of contents of Acc
REVA	Acc ← − Acc
STOR	

c → content

mov, LDA load into Accumulator

The target code generated can be in assembly language (or machine code).

eg) a+b*c

a+b*c

( )

LDA b ; ← ACC ←c (b)
MUL c ; ACC ←c (ACC) *c (c)
ADD a ; ACC ←c (ACC) +c (a)
STOR α_1 ; α_1 ←c (ACC)

Prefix expression

= k + ab

= k (a+b)

LOAD a
ADD b
STOR k

b) k = -b ↑ c / -m / q * r + s

LDA b
EXP c
STOR α_1
LDA m
DIV q
MUL r
STOR α_2
LDA α_1
REVA
SUB α_2
ADD s
STOR k

Assignment:

1a) Convert the ff algebraic/~~ex~~ infix expressions to reverse polish expression/strings.

i) k = -t ↑ s - d / m ↑ c + s * w

ii) w - s ↑ c * m + q ↑ r / s + d

Generate the assembly language code fr Q1a) i and (ii)