Theory of Computation

Capstone Assignment

Name : Vaishnavi

Roll No : 2301010372

Class : B Tech CSE sec-F

Submitted to : Mansi Kajal Ma'am

26/11/25

UNIT-1 Finite Automata & Regular Expression

1. Regular Expression for Valid Identifiers

Let Σ_A be the set of alphabets & Σ_D be the set of digits. The regular expression R for all valid identifiers (alphabet followed by any sequence of alphabets or digits) is

$$R = (\Sigma_A)(\Sigma_A \cup \Sigma_D)^*$$

The keywords (for, while, if) are excluded in the lexical analysis phase following token recognition.

2. Design a DFA equivalent to R

The DFA $M = \{Q, \Sigma, \delta, q_0, F)$

$Q = (q_0, q_1, q_f)$

q_0 = start state

q_1 = accepting state (valid identifier started)

q_f = dead state (invalid start)

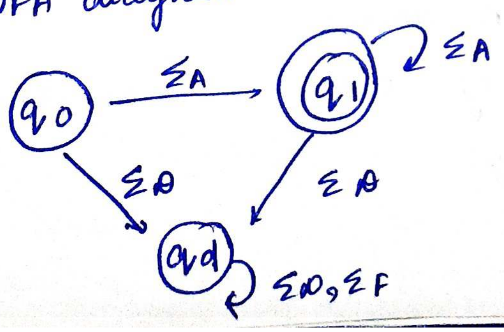
$\Sigma = \Sigma_A \cup \Sigma_D$

$F = \{q_1\}$ (final state)

Transition Table

State	Input Σ_A	Input Σ_D	Input (other)
q_0	q_1	q_d	q_d
q_1	q_1	q_1	q_1 (loop)
q_d	q_d	q_d	q_d

DFA diagram :-



3) Embedding the DFA in a lexical Analyzer

The DFA acts as the state machine for recognizing the pattern of an identifier

1) DFA recognition : The lexer consumes input characters, tracking DFA's state. When the input stream forces the DFA out of q_1 (encountering a space or operator) the operator reader to that point is identified as a potential token.

2) Keyword check : The recognized string is then checked against a small, finite list of reserved keywords (for, while, if). This is typically done via a fast hash table lookup

3) Token Generation :

- If the string is found in the keyword list, a KEYWORD token is generated.
- Otherwise, an IDENTIFIER token is generated & its entry (lexeme & type) is stored in the symbol table.

CO2 UNIT - 2 PDA & Context free language

1) Formulate a CFG for well formed queries

Let O = '<open>' & C = '</close>'. The grammar G models balanced nesting

$$S \rightarrow OSC \mid SS \mid \epsilon$$

$S \rightarrow OSC$ handles nested structure (eg <open> ... </close>)

$S \rightarrow SS$ handles concatenated structure (eg <open></close><open></close>)

$S \rightarrow \epsilon$ handles empty query

2) Construct a PDA that accept such queries

The PDA accepts the languages by empty stack. It uses the stack to track unmatched <open> tags

$M = (\{q_0\}, \{O, C\}, \{Z_0, X\}, \delta, q_0, \phi\}$

ite	Input	Top of stack	New state	stack Operation	Rationale
q_0	0	Z_0	q_0	XZ_0	push X for first 0
q_0	0	X	q_0	XX	push X for nested 0
q_0	c	X	q_0	ϵ	pop X for matching c
q_0	ϵ	Z_0	q_0	ϵ	Accept by empty stack

3) Demonstrate the parse tree

Query : <open> <open> </close> </close>

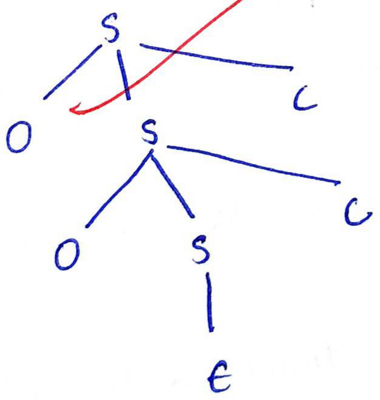
Derivation

$S \Rightarrow OSC$

$S \Rightarrow O(OSC)C \quad (S \to OSC)$

$S \Rightarrow O(O\epsilon C)C \quad (S \to \epsilon)$

parse tree :-



Q3 UNIT-3 Turing Machine & Chomsky Hierarchy

1) Justify why $L = \{a^n b^n c^n \mid n \geq 1\}$ is not context free

We use pumping lemma for CFLs

1) Choose String s: let p be the pumping length choose $s = a^p b^p c^p \in L$

2) Decompose s: $s = uvwxy$ where $|vwx| \leq p$ & $|vx| \geq 1$.

3) Pumping arguement since $|vwx| \leq p$ the pumpable segments vwx can only contain symbols from atmost two blocks (only a's & b's, or only b's & c's)

- Case (vwx is in a's & b's) pumping up (setting $i=2$) ↑ the no of a's and/or b's, but leaves the number of c's fixed at p
- Resulting string $s' = uv^2wx^2y$ has unequal counts of a's, b's & c's (specifically, count(a) + count(b) > count(c))

4) Conclusion $s' \notin L$. since the conditions of the pumping lemma are violated, L can't be a CFL

2) Design a Turing Machine (TM) that accepts L

The TM mark one a, one b & one c in the cycle until all symbols are marked

- Tape alphabets Γ: $\{a, b, c, X, Y, Z, \square\}$ (X, Y, Z are markers)
- Core logic

q_0: Mark the leftmost a as X & transition to find b

q_a: Find the leftmost unmarked b & mark it as Y, then transition to find c

q_b: Find the leftmost unmarked c & mark it as Z, then transition to return.

q_{ret}: Scan left to the starting point (X)

q_{check}: After all a's are marked, scan right to ensure the rest of the tape is just Y's, Z's & finally $\square$ (B)

step by

The

1) a

Step by step configuration for 'aaabbbccc'

The TM cycles three times to mark the three pairs

1) Cycle 1 (Mark $a_1 b_1 c_1$) : q_0, aaabbbccc → q_a, Xaabbccc (Mark a) → q_b, XaaYbbcc (Mark b) → q_{ret}, XaaYbbZcc (Mark c, return left) → q_0, XaaYbbZcc (Restart)

2) Cycle 2 (Mark $a_2 b_2 c_2$) : q_0, Xaabbccc → q_{ret}, XXaYYbZZc (Tape becomes XXaYYbZZc)

3) Cycle 3 (Mark $a_3 b_3 c_3$) : q_0, XXa YYb ZZc → q_{ret}, XXX YYYZZZ

4) Final check (q_{check}) : q_0 reads the marked a's (X's), transitions to q_{check}. q_{check} scans over Y's & Z's until it hits ☐. q_{check}, XXX YYY ZZZ ☐ → q_{accept} (Accept).

Q4 UNIT : 4 - Code generation & optimization

Expression (A+B) * (C-D) + E

1) Syntax - Directed Translation scheme (S attributed) using a simple precedence - based grammar

Production	Semantic Rules
E → E_1 + T	E.addr = new_temp(); E.code = E_1.code ‖ T.code ‖ E.addr = E_1.addr + T.addr
T → T_1 * F	T.addr = new_temp(); T.code = T_1.code ‖ F.code ‖ T.addr = T_1.addr * F.addr
F → (E_1)	F.addr = E_1.addr ; F.code = E_1.code
F → id	F.addr = id.lexeme , F.code = ε

2) Generate three address code (TAC)

The TAC is generated based on expression's evaluation order (
precedence : parenthesis → multiplication → addition

1) $t_1 = A + B$
2) $t_2 = C - D$
3) $t_3 = t_1 * t_2$
4) $t_4 = t_3 + E$

3) Optimize the generated TAC

There is no duplicate expression (common subexpressions) in lines 1 & 2. The code is already optimal wrt to CSE

Dead Code Removal

Assume the final result t_4 is used, all intermediate variables (t_1, t_2, t_3) are necessary inputs for subsequent lines. No dead code can be removed

Optimized TAC (unchanged)
1) $t_1 = A + B$
2) $t_2 = C - D$
3) $t_3 = t_1 * t_2$
4) $t_4 = t_3 + E$

Q5 (Cumulative - Advanced Reasoning & Application)

Language L = Equal no of 0's & 1's & no prefix has more 1's than 0's (Dyck Paths)

1) Prove that L is context free but not regular
- Not Regular : Use the pumping lemma for regular languages. Choose $S = 0^P 1^P$. Pumping down ($i = 0$) gives $0^{P-b} 1^P$ ($b \geq 1$) which has unequal counts voilating L. Thus L is not Regular
- Context free : The language L is accepted by a pushdown Automaton (PDA) shown below) which demonstrates its context free nature. The PDA's stack is essential for counting & comparing the non local dependencies (0s vs 1s)

CFG for
or 4

Provide a CFG for this language (L)

The Grammar G must enforce that every 1 is matched by a preceding 0

$$S \rightarrow 0S1S \mid \epsilon$$

This CFG generates all valid Dyck paths

3) Design a PDA & Trace '0011'

A) PDA Design (M)

M accepts by empty stack, using X to count the excess no of 0's

Start Z_0 : $\delta(q_0, 0, Z_0) = \{(q_0, XZ_0)\}$

Push 0 : $\delta(q_0, 0, X) = \{(q_0, XX)\}$

pop 1 (prefix check) : $\delta(q_0, 1, X) = \{(q_0, \epsilon)\}$ (pops only if 0's are in excess

Accept : $\delta(q_0, \epsilon, Z_0) = \{(q_0, \epsilon)\}$

B) Trace the acceptance of '0011'

Input	State	Stack (L → R)	Transition	Condition check (0's ≥ 1's)
0011	q_0	Z_0	push 0	
0011	q_0	XZ_0	push 0	$2 \geq 0$
0011	q_0	XXZ_0	pop 1	$2 \geq 1$
0011	q_0	XZ_0	pop 1	$2 \geq 2$
0011	q_0	Z_0	Empty stack	$2 = 2$
0011ϵ		ϵ	accept	