FAFL LAB COMPONENT

Implement using C/C++/JAVA/Python

1. Design a Finite State Machine (FSM) that accepts all strings over input symbols {0, 1} having three consecutive 1's as a substring.
2. Design a Finite State Machine (FSM) that accepts all strings over input symbols {0, 1} which are divisible by 3.
3. Design a Finite State Machine (FSM) that accepts all decimal string which are divisible by 3
4. Design a Push Down Automata (PDA) that accepts all string having equal number of 0's and 1's over input symbol {0, 1} for a language 0n1n where n >= 1.
5. Design a Program to create PDA machine that accept the well-formed parenthesis.
6. Design a PDA to accept WCWR where w is any binary string and WR is reverse of that string and C is a special symbol.

Design a Finite State Machine (FSM) that accepts all strings over input symbols {0, 1} having three consecutive 1's as a substring.

A = {111, 0111, 1110, 0101011110101,...} means any string should be declared valid if it contains 111 as a substring.

Let M be the machine, M(Q, Σ, δ, q0, F) where

Q: set of states: {A, B, C, D}

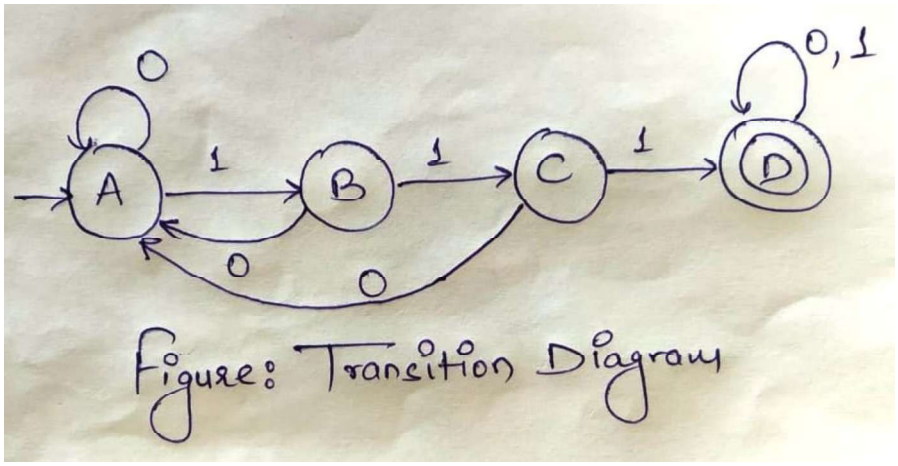
Σ: set of input symbols: {0, 1}

q0: initial state (A)

F: set of Final states: {D}

δ: Transition Function:

State	Input	
-	0	1
A	A	B
B	A	C
C	A	D
D	D	D




```
#include <iostream>
#include <string>
```

```
using namespace std;

enum State {
   S0, S1, S2, S3
};

bool acceptsString(const string& input) {
   State currentState = S0;

   for (char c : input) {
      switch (currentState) {
         case S0:
            if (c == '1') {
               currentState = S1;
            } // else stay in S0 (on '0')
            break;
         case S1:
            if (c == '1') {
               currentState = S2;
            } else {
               currentState = S0; // on '0', go back to S0
            }
            break;
         case S2:
            if (c == '1') {
               currentState = S3;
            } else {
               currentState = S0; // on '0', go back to S0
            }
            break;
         case S3:
            // Once we reach S3, we stay in S3 (accepting state)
            break;
      }
   }
   return currentState == S3;
}

int main() {
   string input;
   cout << "Enter a binary string: ";
   cin >> input;

   if (acceptsString(input)) {
      cout << "Accepted (contains 111)" << endl;
   } else {
```

```
        cout << "Rejected (does not contain 111)" << endl;
    }

    return 0;
}
```

OUTPUT:

1. Enter a binary string: 100101101101 Rejected (does not contain 111)
2. Enter a binary string: 1010111 Accepted (contains 111)

Complexity

- Time Complexity: O(n), where n is the length of the input string (since we process each character exactly once).
- Space Complexity: O(1), since we only store a fixed number of states and variables regardless of input size.

Design a Finite State Machine (FSM) that accepts all strings over input symbols {0, 1} which are divisible by 3

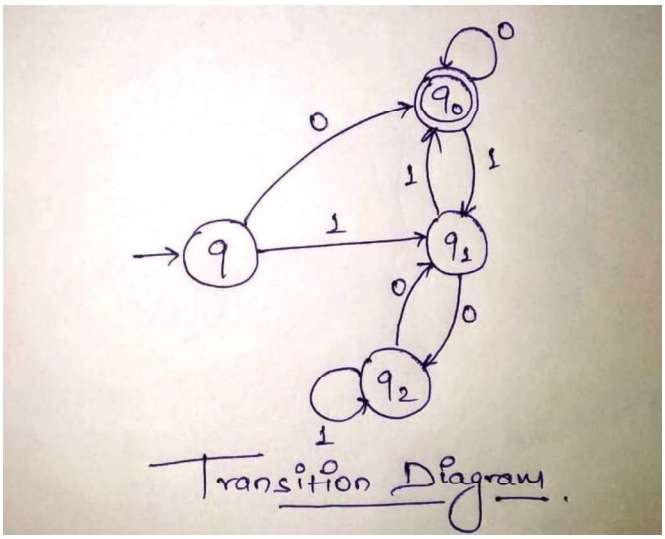
A = {0, 00, 000, 11, 011, 110, ...}means any binary string that
when divide by three gives remainder zero.
Let M be the machine for above, hence it can be define as M(Q, Σ, δ, q0, F) where
Q: set of states: {q, q0, q1, q2}
Σ: set of input symbols: {0, 1}

q0: initial state (q)

F: set of Final states: {q0}

δ: Transition Function:

State	Input	
-	0	1
q	q0	q1
q0	q0	q1
q1	q2	q0
q2	q1	q2




```
#include <iostream>
#include <string>
using namespace std;

enum State {
   S0, S1, S2
};

bool isDivisibleBy3(const string& input) {
   State currentState = S0; // Start in state S0 (remainder = 0)

   for (char c : input) {
      switch (currentState) {
         case S0:
            if (c == '0') {
               currentState = S0; // Stay in S0 (remainder 0)
            } else if (c == '1') {
               currentState = S1; // Move to S1 (remainder 1)
            }
            break;
         case S1:
            if (c == '0') {
               currentState = S2; // Move to S2 (remainder 2)
            } else if (c == '1') {
               currentState = S0; // Move to S0 (remainder 0)
            }
            break;
         case S2:
            if (c == '0') {
               currentState = S1; // Move to S1 (remainder 1)
            } else if (c == '1') {
               currentState = S2; // Stay in S2 (remainder 2)
            }
            break;
```

```
        }
    }
    // Accept if the FSM ends in state S0 (remainder 0, divisible by 3)
    return currentState == S0;
}

int main() {
    string input;
    cout << "Enter a binary string: ";
    cin >> input;

    if (isDivisibleBy3(input)) {
        cout << "Accepted (the number is divisible by 3)" << endl;
    } else {
        cout << "Rejected (the number is not divisible by 3)" << endl;
    }

    return 0;
}
```

OUTPUT:
Enter a binary string: 1010
Rejected (the number is not divisible by 3)

Enter a binary string: 11
Accepted (the number is divisible by 3)

Time Complexity

- **Time Complexity**: O(n), where n is the length of the input string. Each character is processed exactly once.
- **Space Complexity**: O(1), since we only need a fixed amount of space for the current state.

Design a Finite State Machine (FSM) that accepts all decimal string which are divisible by 3

A = {0, 3, 6, 9, 03, 06, 09, 12, 012, ..}

means any decimal number string that when divided by three gives remainder zero.

Let M be the machine, hence it can be define as

M(Q, Σ, δ, q0, F) where
Q: set of states: {q, q0, q1, q2}

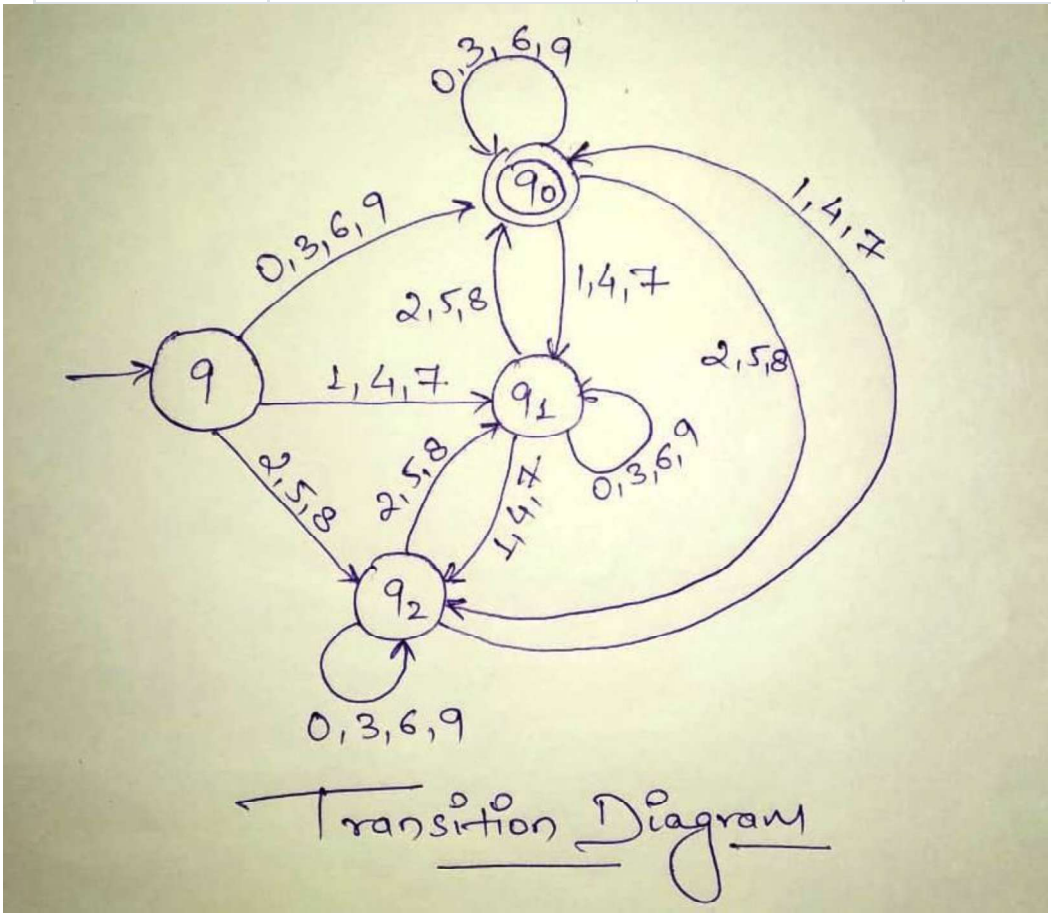
Σ: set of input symbols: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

q0: initial state (q)

F: set of Final states: {q0}

δ: Transition Function:

State	Input		
-	0, 3, 6, 9	1, 4, 7	2, 5, 8
q	q0	q1	q2
q0	q0	q1	q2
q1	q1	q2	q0
q2	q2	q0	q1




```
#include <iostream>
#include <string>
using namespace std;

enum State {
   S0, S1, S2
};

bool isDivisibleBy3(const string& input) {
   State currentState = S0; // Start in state S0 (remainder = 0)

   for (char c : input) {
      if (c < '0' || c > '9') {
         cout << "Invalid input: Non-decimal character encountered!" << endl;
```

```
            return false;
        }

        int digit = c - '0'; // Convert char to int (e.g., '3' -> 3)

        switch (currentState) {
            case S0:
                currentState = static_cast<State>((0 * 10 + digit) % 3);
                break;
            case S1:
                currentState = static_cast<State>((1 * 10 + digit) % 3);
                break;
            case S2:
                currentState = static_cast<State>((2 * 10 + digit) % 3);
                break;
        }
    }

    // Accept if the FSM ends in state S0 (remainder 0, divisible by 3)
    return currentState == S0;
}

int main() {
    string input;
    cout << "Enter a decimal string: ";
    cin >> input;

    if (isDivisibleBy3(input)) {
        cout << "Accepted (the number is divisible by 3)" << endl;
    } else {
        cout << "Rejected (the number is not divisible by 3)" << endl;
    }

    return 0;
}
```

OUTPUT

```
Enter a decimal string: 25
Rejected (the number is not divisible by 3)

Enter a decimal string: 27
Accepted (the number is divisible by 3)
```

Time Complexity

- **Time Complexity**: O(n), where n is the length of the input string. Each character is processed exactly once.

- **Space Complexity**: O(1), since we only need a fixed amount of space for the current state.

Design a Push Down Automata (PDA) that accepts all string having equal number of 0's and 1's over input symbol {0, 1} for a language 0n1n where n >= 1.

A = {01, 0011, 000111, ...}

means all string having n number of 0's followed by n numbers of 1's where n can be any number greater than equal tp and count of 0's must be equal to count of 1's. Block diagram of push down automata is shown in Figure 1.

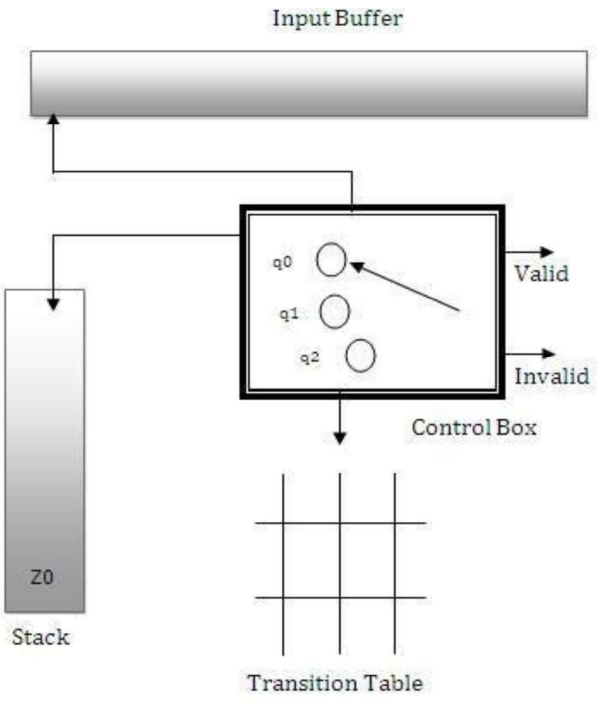


Figure 1: Block Diagram of Push Down Automata.

Input string can be valid or invalid, valid if it follows the language 0^n1^n where n >= 1 else invalid. PDA has to determine whether the input string is according to the language or not.

Let M be the PDA machine for above AIM, hence it can be define as M(Q, Σ, Γ, δ, q0, Z0, F) where

Q: set of states: {q0, q1, q2}

Σ: set of input symbols: {0, 1}
Γ: Set of stack symbols: {A, Z0}
q0: initial state (q0)
Z0: initial stack symbol (Z0)

F: set of Final states: { } [Note: Here, set of final states is null as decision of validity of string is based on stack whether it is empty or not.]
δ: Transition Function: (Transition state diagram is shown in Figure 2.)

δ(q0, 0, Z0) → (q0, AZ0)

δ(q0, 0, 0) → (q0, 00)

δ(q0, 1, 0) → (q1, ε)

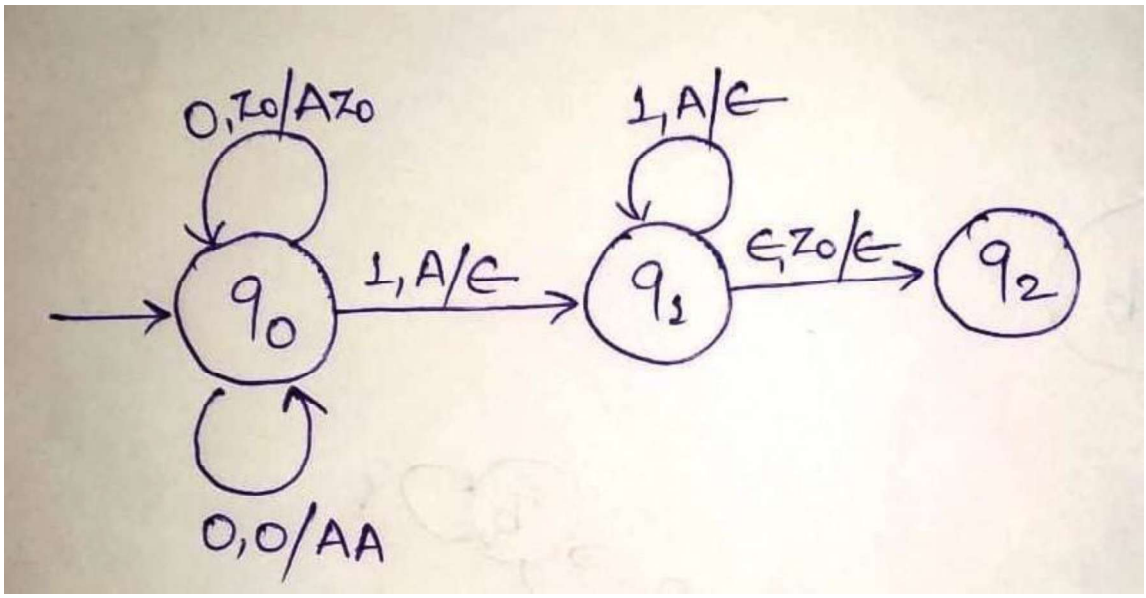
δ(q1, 1, 0) → (q1, ε)

δ(q1, ε, Z0) → (q2, ε)

Rules for implementing PDA for a given language

Initial Setup: Load the input string in input buffer, push Z0 as an initial symbol in stack and consider the machine at initial state q0.
Rules:

1. **It is must that the first symbol should be 0. If the first symbol of input string is zero then push symbol 'A' into stack and read the next character in input string.**
2. **If next character is again 0, then push A again in stack and repeat the same process for all consecutive 0's in input string.**
3. **If next character is 1, then pop A and change its state from q0 to q1.**
4. **If again the next character is 1 and top symbol of stack is 'A', then pop A and repeat the same process for all consecutive 1's in input string.**
5. **If all the characters of input string are parsed and stack top is Z0, it means string is valid, pop Z0 from stack and change the state from q1 to q2.**




```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

bool isValidString(const string& input) {
   stack<char> pdaStack;
```

```
    // State 0: Process '0's and push them to the stack
    int n = input.length();
    int i = 0;

    // Push all 0's onto the stack
    while (i < n && input[i] == '0') {
        pdaStack.push('0');
        i++;
    }

    // Now, process '1's and pop them from the stack
    while (i < n && input[i] == '1') {
        if (pdaStack.empty()) {
            // There is no matching '0' for this '1', reject the string
            return false;
        }
        pdaStack.pop();
        i++;
    }

    // If there are still characters left or the stack is not empty, reject the string
    return i == n && pdaStack.empty();
}

int main() {
    string input;
    cout << "Enter a string of 0's and 1's: ";
    cin >> input;

    if (isValidString(input)) {
        cout << "Accepted: The string has equal number of 0's and 1's in the form 0^n1^n." <<
       endl;
    } else {
        cout << "Rejected: The string does not have equal number of 0's and 1's in the form
       0^n1^n." << endl;
    }

    return 0;
}
```

OUTPUT

```
Enter a string of 0's and 1's: 0011
Accepted: The string has equal number of 0's and 1's in the form 0^n1^n.

Enter a string of 0's and 1's: 0011101
Rejected: The string does not have equal number of 0's and 1's in the form 0^n1^n.
```

Time Complexity:

- **Time Complexity**: O(n), where n is the length of the input string. We process each character exactly once.

- **Space Complexity**: O(n), since in the worst case, we may need to store up to n 0s in the stack.

Design a Program to create PDA machine that accept the well-formed parenthesis.

A = {(),(()), ()(), (()), ((()(()))) ...}

means all the parenthesis that are open must closed or combination of all legal parenthesis formation. Here, opening par is '(' and closing parenthesis is ')'. Block diagram of push down automata is shown in Figure 1.

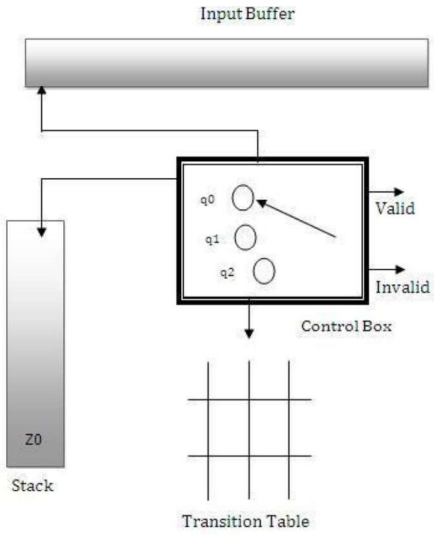


Figure 1: Block Diagram of Push Down Automata

Input string can be valid or invalid, valid if the input string follow set A (define above). PDA has to determine whether the input string is according to the language or not.

Let M be the PDA machine for above AIM, hence it can be define as M(Q, Σ, Γ, δ, q0, Z0, F) where

Q: set of states: {q0, q1}

Σ: set of input symbols: {(,)}
Γ: Set of stack symbols: {(, Z}
q0: initial state (q0)
Z0: initial stack symbol (Z)

F: set of Final states: { } [Note: Here, set of final states is null as decision of validity of string is based on stack whether it is empty or not. If empty means valid else invalid.]
δ: Transition Function: (Transition state diagram is shown in Figure 2.)

$$\delta(q0, (, Z) \rightarrow (q0, (Z)$$

$$\delta(q0, (, () \rightarrow (q0, (()$$

$$\delta(q0,), () \rightarrow (q0, \varepsilon)$$

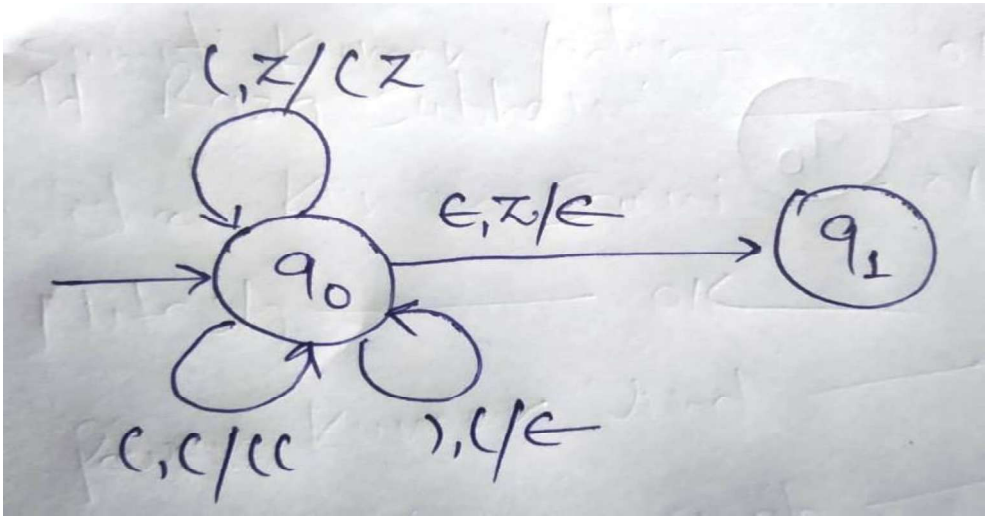
$$\delta(q0, \varepsilon, Z) \rightarrow (q1, \varepsilon)$$

Rules for implementing PDA for a given language

Initial Setup: Load the input string in input buffer, push Z as an initial stack symbol and consider the machine in at initial state q0.
Rules:

1. **It is must that the first symbol should be '('.**

2. **If the input symbol of string is '(' and stack top is Z then push symbol '(' into stack and read the next character in input string.**
3. **If next character is again '(', then push '(' again in stack and repeat the same process for all '(' in input string.**
4. **If character is ')' and stack top is '(', then pop '(' from stack.**
5. **If all the characters of input string are parsed and stack top is Z, it means string is valid, pop Z from stack and change the state from q0 to q1.**




```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// Function to check if the string of parentheses is well-formed
bool isWellFormedParentheses(const string& input) {
    stack<char> pdaStack;

    // Traverse the input string
    for (char c : input) {
        // If we encounter an opening parenthesis, push it onto the stack
        if (c == '(') {
            pdaStack.push(c);
        }
        // If we encounter a closing parenthesis, pop from the stack
        else if (c == ')') {
            if (pdaStack.empty()) {
                // If the stack is empty, there is no matching opening parenthesis, reject the
string
                return false;
            }
            pdaStack.pop();
        }
        // If the character is neither '(' nor ')', we reject the string
        else {
```

```
            return false;
        }
    }

    // The string is well-formed if the stack is empty at the end
    return pdaStack.empty();
}

int main() {
    string input;
    cout << "Enter a string of parentheses: ";
    cin >> input;

    if (isWellFormedParentheses(input)) {
        cout << "Accepted: The parentheses are well-formed." << endl;
    } else {
        cout << "Rejected: The parentheses are not well-formed." << endl;
    }

    return 0;
}
```

Output

Enter a string of parentheses: (()())
Accepted: The parentheses are well-formed.

Enter a string of parentheses:)()((
Rejected: The parentheses are not well-formed.

Time Complexity:

- **Time Complexity**: O(n), where n is the length of the input string. The program processes each character once.
- **Space Complexity**: O(n), since in the worst case, all opening parentheses (could be stored in the stack.

Design a PDA to accept WCWR where w is any binary string and WR is reverse of that string and C is a special symbol.

A = {0C0, 1C1, 011000110C011000110, 101011C110101,.....}

means string must have some binary string followed by special character 'C' followed reverse of binary string that appears before 'C'. Block diagram of push down automata is shown in Figure 1.

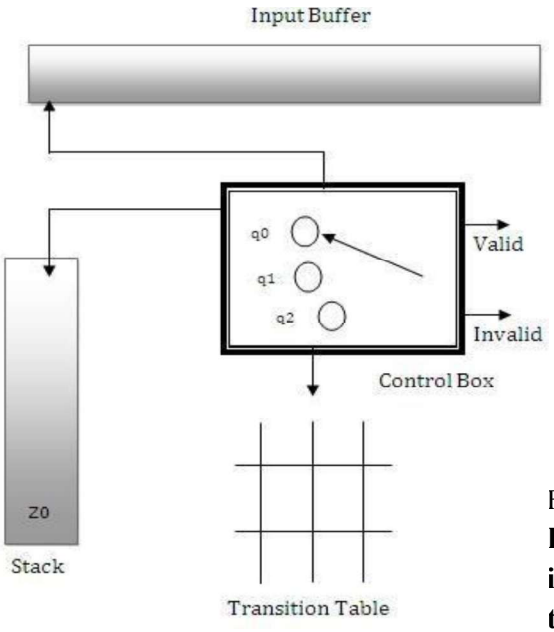


Figure 1: Block Diagram of Push Down Automata

Input string can be valid or invalid, valid if the input string follow set A (define above). PDA has to determine whether the input string is according to the language or not.

Let M be the PDA machine for above AIM, hence it can be define as M(Q, Σ, Γ, δ, q0, Z0, F) where

Q: set of states: {q0, q1, q2}

Σ: set of input symbols: {0, 1, C}
Γ: Set of stack symbols: {A, B, Z}
q0: initial state (q0)
Z0: initial stack symbol (Z)

F: set of Final states: { } [Note: Here, set of final states is null as decision of validity of string is based on stack whether it is empty or not. If empty means valid else invalid.]
δ: Transition Function: (Transition state diagram is shown in Figure 2.)

$$\delta(q0, 0, Z) \rightarrow (q0, AZ)$$

$$\delta(q0, 1, Z) \rightarrow (q0, BZ)$$

$$\delta(q0, 0, A) \rightarrow (q0, AA)$$

$$\delta(q0, 0, B) \rightarrow (q0, AB)$$

$$\delta(q0, 1, A) \rightarrow (q0, BA)$$

$$\delta(q0, 1, B) \rightarrow (q0, BB)$$

$$\delta(q0, C, A) \rightarrow (q1, A)$$

$$\delta(q0, C, B) \rightarrow (q1, B)$$

$$\delta(q1, 0, A) \rightarrow (q1, \varepsilon)$$

$$\delta(q1, 1, B) \rightarrow (q1, \varepsilon)$$

$$\delta(q1, \varepsilon, Z) \rightarrow (q2, \varepsilon)$$

Rules for implementing PDA for a given language

Initial Setup: Load the input string in input buffer, push Z as an initial stack symbol and consider the machine in at initial state q0.
Rules:

1. **If the input symbol of string is '0' and stack top is Z then push symbol 'A' into stack and read the next character in input string.**
2. **If the input symbol of string is '1' and stack top is Z then push symbol 'B' into stack and read the next character in input string.**
3. **If the input symbol of string is '0' and stack top is A or B then push symbol 'A' into stack and read the next character in input string.**
4. **If the input symbol of string is '1' and stack top is A or B then push symbol 'B' into stack and read the next character in input string.**

5. **If the input symbol of string is 'C' and stack top is A or B then change state from q0 to q1 and read the next character in input string.**
6. **If the input symbol of string is '0', machine state is q1 and stack top is A then pop stack top and read the next character in input string.**
7. **If the input symbol of string is '1', machine state is q1 and stack top is B then pop stack top and read the next character in input string.**
8. **If all the characters of input string are parsed, stack top is Z and machine state is q1, it means string is valid, pop Z from stack and change the state from q1 to q2.**

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// Function to check if the string is of the form wCwR
bool isValidWCWR(const string& input) {
    stack<char> pdaStack;
    int n = input.length();
    int i = 0;

    // Step 1: Push the first part of the string (w) onto the stack until we encounter 'C'
    while (i < n && input[i] != 'C') {
        if (input[i] != '0' && input[i] != '1') {
            // Invalid character encountered
            return false;
        }
        pdaStack.push(input[i]);
        i++;
    }

    // Check if 'C' is encountered
    if (i == n || input[i] != 'C') {
        // No 'C' found or end of string without 'C'
```

```
        return false;
    }
    i++; // Skip the 'C'

    // Step 2: Check if the second part of the string (wR) matches the reverse of the first part
    while (i < n) {
        if (input[i] != '0' && input[i] != '1') {
            // Invalid character encountered
            return false;
        }

        if (pdaStack.empty() || pdaStack.top() != input[i]) {
            // Mismatch or stack is empty (unmatched wR)
            return false;
        }

        // Pop the top of the stack if the characters match
        pdaStack.pop();
        i++;
    }

    // Step 3: If the stack is empty at the end, the string is valid (accepted)
    return pdaStack.empty();
}

int main() {
    string input;
    cout << "Enter a string (in the form wCwR, where w is a binary string and C is a special
symbol): ";
    cin >> input;

    if (isValidWCWR(input)) {
        cout << "Accepted: The string is in the form wCwR." << endl;
    } else {
        cout << "Rejected: The string is not in the form wCwR." << endl;
    }

    return 0;
}
```

OUTPUT

```
Enter a string (in the form wCwR, where w is a binary string and C is a special symbol):
1C1
Accepted: The string is in the form wCwR.

Enter a string (in the form wCwR, where w is a binary string and C is a special symbol):
```

100C01
Rejected: The string is not in the form wCwR.

Time Complexity:

- **Time Complexity**: O(n), where n is the length of the input string. Each character is processed once.
- **Space Complexity**: O(n), since in the worst case, we may need to store up to n characters in the stack.