LAB 1

Title: String Manipulation and Identifier Validation in C.

Theory:

In C programming, a string is a sequence of characters stored in a contiguous block of memory and terminated by a null character ('\0'). String manipulation involves various operations on these sequences, such as copying, concatenating, comparing, and searching. Identifier validation is the process of checking if a given string conforms to the rules for naming variables, functions, and other program elements.

String Manipulation

C does not have a built-in string data type. Instead, strings are implemented as character arrays. The string.h library provides a set of standard functions for manipulating these arrays. The core idea behind C strings is that they are null-terminated, which allows functions to determine the end of the string without needing an explicit length parameter.

Common string manipulation functions include:

- strlen(str): Returns the length of the string, excluding the null terminator.
- strcpy(dest, src): Copies the string src to dest. Be cautious to ensure dest has enough allocated memory to prevent buffer overflow.
- strcat(dest, src): Concatenates src to the end of dest. Again, ensure dest is large enough.
- strcmp(str1, str2): Compares two strings. It returns an integer value:
    - 0 if the strings are identical.
    - A negative value if str1 comes before str2 in lexicographical order.
    - A positive value if str1 comes after str2.
- strstr(haystack, needle): Searches for the first occurrence of the substring needle within the string haystack. It returns a pointer to the beginning of the found substring or NULL if not found.

Identifier Validation

An identifier is a name used to identify a variable, function, or other program entity. C has strict rules for what constitutes a valid identifier. Validating a string as an identifier involves checking it against these rules.

The rules for a valid C identifier are:

1. Starts with a letter or an underscore (_): The first character cannot be a digit.
2. Subsequent characters can be letters, digits, or underscores: After the first character, numbers are permitted.
3. Case-sensitive: myVar and myvar are considered different identifiers.
4. No special characters or spaces: Identifiers cannot contain symbols like !, @, #, $, %, or spaces.
5. Not a C keyword: The identifier cannot be one of C's reserved keywords, such as int, for, while, or return.

To implement identifier validation in C, you typically need to:

- Check the first character: Use functions from the <ctype.h> library like isalpha() and a check for _ to see if the first character is valid.
- Iterate through the rest of the string: Loop through the remaining characters and use isalnum() (which checks for either letters or digits) and a check for _ to ensure they are all valid.
- Check for length and other constraints: Although C standards don't specify a maximum length, most compilers have a limit.

- Check against reserved keywords: This can be done by storing keywords in an array and using a function like strcmp to compare the input string against each keyword.

By combining string manipulation techniques with character-level validation from <ctype.h>, you can build a robust function to check if a string is a valid identifier.

Program 1: WAP to find prefix, suffix and substring from given string.

Source code:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[100]; int
    i, j, len;
    printf("Sulav  \n");
    printf("Enter a string: ");
    scanf("%s", str);
    len = strlen(str);
    printf("\nPrefixes:\n"); for(i
    = 1; i <= len; i++) {
        for(j = 0; j < i; j++)
            { printf("%c", str[j]);
        }
        printf("\n");
    }
    printf("\nSuffixes:\n"); for(i
    = 0; i < len; i++) {
        for(j = i; j < len; j++)
            { printf("%c", str[j]);
        }
        printf("\n");
    }
    printf("\nSubstrings:\n"); for(i
    = 0; i < len; i++) {
        for(j = i; j < len; j++) { for(int k
            = i; k <= j; k++) {
                printf("%c", str[k]);
            }
            printf("\n");
        }
    }
    return 0;
}
```

Output and Discussion:

This C program prints all prefixes, suffixes, and substrings of a user-input string. It uses nested loops to iterate through character positions and display combinations. The strlen function determines string length, and scanf captures input. It demonstrates string manipulation fundamentals and loop control in a clear, structured, and educational format.

```
Sulav
Enter a string: aB1

Prefixes:
a
aB
aB1

Suffixes:
aB1
B1
1

Substrings:
a
aB
aB1
B
B1
1
```

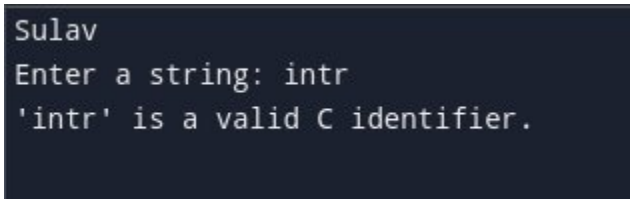
Program 2: WAP to validate C identifiers and keywords Source code:

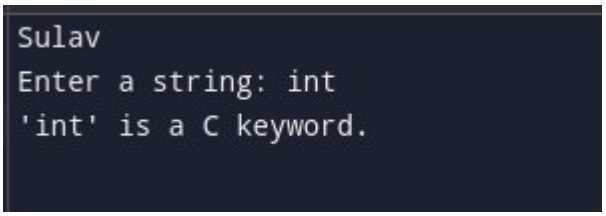
Source code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
const char *keywords[] =
   { "auto","break","case","char","const","continue","default","do",
   "double","else","enum","extern","float","for","goto","if",
   "int","long","register","return","short","signed","sizeof",
   "static","struct","switch","typedef","union","unsigned","void","volatile","while"
};
int keywordCount = 32;
int isKeyword(const char *str) {
   for(int i = 0; i < keywordCount; i++)
      { if(strcmp(str, keywords[i]) == 0)
         return 1;}
   return 0;
}
int isValidIdentifier(const char *str)
   { if(!(isalpha(str[0]) || str[0] == '_'))
      return 0;
   for(int i = 1; str[i] != '\0'; i++) {
      if(!(isalnum(str[i]) || str[i] == '_')) return
         0;
   }
```

```
    return 1;
}
int main() { char
    str[50];
    printf("Sulav \n"); printf("Enter
    a string: "); scanf("%s", str);
    if(isKeyword(str)) {
        printf("'%s' is a C keyword.\n", str);
    }
    else if(isValidIdentifier(str)) {
        printf("'%s' is a valid C identifier.\n", str);
    }
    else {
        printf("'%s' is NOT a valid C identifier.\n", str);
    }
    return 0;}
```

Output and discussion





This C program checks whether a user-input string is a C keyword or a valid identifier. It compares the string against a list of 32 keywords and verifies identifier rules using character checks. It demonstrates string handling, validation logic, and conditional output, helping users understand lexical analysis and basic syntax rules in C programming.

Conclusion:

This lab successfully demonstrated fundamental string manipulation and identifier validation in C programming. Through two practical programs, we explored essential string operations including prefix/suffix generation and substring extraction using nested loops and the strlen() function. The identifier validation program effectively implemented C's naming rules by checking first character validity, subsequent character constraints, and keyword conflicts using ctype.h functions and string comparison. These exercises reinforced core concepts of null-terminated strings, character arrays, loop structures, and lexical analysis fundamentals. The lab provided hands-on experience with string handling techniques crucial for understanding C programming and compiler design principles.

LAB 2

Title: To design and implementation of Deterministic Finite Automata (DFA) for various language.

Theory:

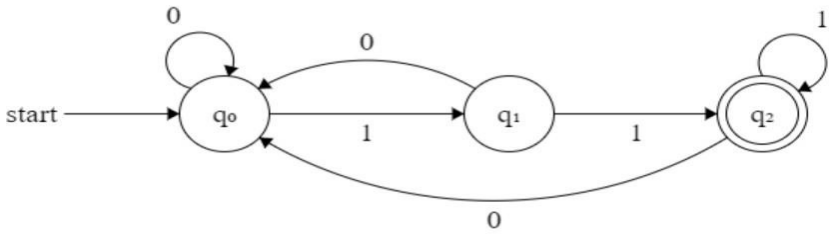
A Deterministic Finite Automaton (DFA) is a mathematical model of computation. It is a simple machine that reads a string of input symbols and decides whether to accept or reject the string. The term "deterministic" means that for each input symbol, there is exactly one state to which the machine can transition from its current state. DFAs are used to recognize regular languages.

Components of a DFA:

A DFA is formally defined as a 5-tuple: (Q,Σ,δ,q0,F), where:

- Q is a finite set of states. These represent the machine's configuration at any given time.
- Σ (Sigma) is a finite set of input symbols called the alphabet. This is the set of all possible characters the machine can read.
- δ (Delta) is the transition function. It's a mapping from a state and an input symbol to a single next state, denoted as δ(q,a)=q′, where q and q′ are states and a is an input symbol.
- q0 is the start state, an element of Q. This is where the machine begins processing the input string.
- F is a set of final (or accepting) states, a subset of Q. If the machine finishes reading the entire string and lands in a state from this set, the string is accepted.

Program 1: Design and Implementation of a DFA to accept strings over {0, 1} that ends with 11.



Source code:

```
#include <stdio.h>
#include <string.h> int
main() {
   char str[100];
   int state = 0; int
   i;
   printf("Sulav \n");
   printf("Enter a binary string: ");
   scanf("%s", str);
   for (i = 0; i < strlen(str); i++) { if
      (str[i] == '0') {
         if (state == 1) state = 0;  else
         if (state == 2) state = 1; else
         state = 0;
      }
      else if (str[i] == '1') {
```

```
            if (state == 0) state = 1;  else
            if (state == 1) state = 2; else if
            (state == 2) state = 2;
        }
        else {
            printf("Invalid input! Only 0s and 1s are allowed.\n"); return
            0;
        }
    }
    if (state == 2)
        printf("String Accepted: Ends with 11\n"); else
        printf("String Rejected: Does not end with 11\n"); return
    0;
}
```

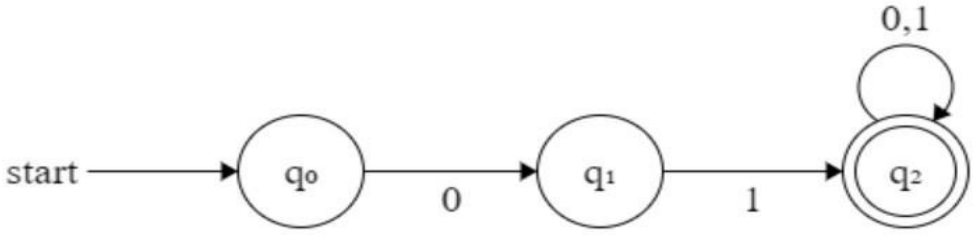
Output and Discussion:

```
Sulav
Enter a binary string: 1011
String Accepted: Ends with 11
```

```
Sulav
Enter a binary string: 10110
String Rejected: Does not end with 11
```

This C program simulates a finite state machine that checks whether a binary string ends with "11". It transitions through states based on input characters and validates binary format. If the final state equals 2, the string is accepted. Otherwise, it is rejected. It demonstrates basic automata logic and input validation effectively.

Program 2: Design and Implementation of a DFA to accept strings over {0, 1} that start with 01.



Source code:

```
#include <stdio.h>
#include <string.h>
int dfa_start_with_01(char str[]) { int
    state = 0;
    for (int i = 0; str[i] != '\0'; i++)
        { char ch = str[i];
        switch (state)
            { case 0:
                if (ch == '0')
            state = 1; else
                    state = 3;
                break;
            case 1:
                if (ch == '1') state
                    = 2;
                else
                    state = 3;
                break;
            case 2:
                state = 2;
                break;
```

```
            case 3:
                state = 3;
                break;
        }
    }
    return (state == 2);
}
int main() {
    char str[100];
    printf("Sulav\n");
    printf("Enter a binary string: ");
    scanf("%s", str);
    if (dfa_start_with_01(str))
        printf("ACCEPTED: The string starts with 01\n"); else
        printf("REJECTED: The string does not start with 01\n");
    return 0;
}
```

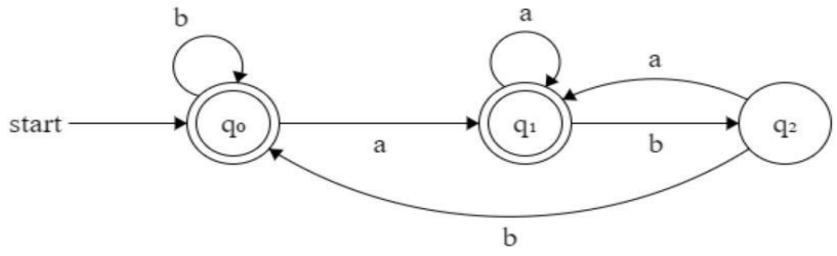
Output and discussion

```
Sulav
Enter a binary string: 0111
ACCEPTED: The string starts with 01
```

```
Sulav
Enter a binary string: 10001
REJECTED: The string does not start with 01
```

This C program uses a deterministic finite automaton (DFA) to check if a binary string starts with "01". It transitions through states based on character input and ends in an accepting state if the condition is met. The dfa_start_with_01 function handles logic, while main manages input and output. It illustrates automata principles and string validation effectively.

Program 3: Design and Implementation of a DFA to accept string over {a, b} such that string does not end with ab.



Source code

```
#include <stdio.h>
#include <string.h>
#define Q0 0
#define Q1 1
#define Q2 2
int dfa_accepts(const char *str) { int
    state = Q0;
    int i;
```

```
        for (i = 0; i < strlen(str); i++)
            { char ch = str[i];  switch(state)
            {
                case Q0:
                    if (ch == 'a') state = Q1;
                    else if (ch == 'b') state = Q0; else {
                        printf("Invalid character: %c\n", ch);
                        return 0;
                    }
                    break;
                case Q1:
                    if (ch == 'a') state = Q1;
                    else if (ch == 'b') state = Q2; else {
                        printf("Invalid character: %c\n", ch);
                        return 0;
                    }
                    break;
                case Q2:
                    if (ch == 'a') state = Q1;
                    else if (ch == 'b') state = Q0; else {
                        printf("Invalid character: %c\n", ch);
                        return 0;
                    }
                    break;
            }
        }
        return (state == Q0 || state == Q1);
    }
int main() {
        char str[100];
        printf("Sulav\n");
        printf("Enter a string over {a,b}: ");
        scanf("%s", str);
        if (dfa_accepts(str)) printf("String
            is Accepted\n");
        else
            printf("String is Rejected\n");
        return 0;
    }
```

Output and discussion

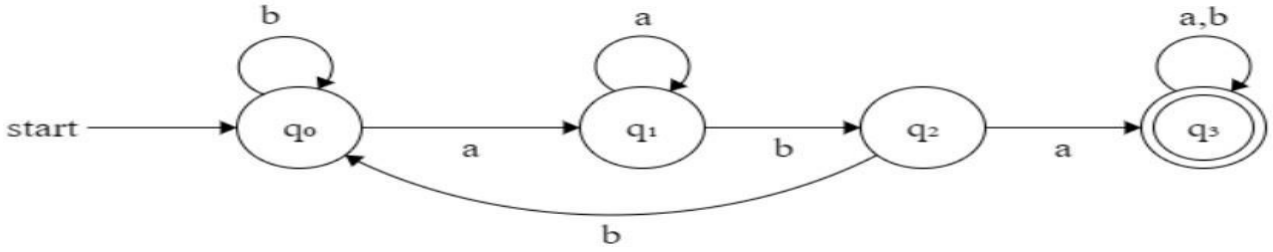
```
Sulav
Enter a string over {a,b}: aba
String is Accepted
```

```
Sulav
Enter a string over {a,b}: abab
String is Rejected
```

```
Sulav
Enter a string over {a,b}: 22
Invalid character: 2
String is Rejected
```

This C program implements a deterministic finite automaton (DFA) to evaluate strings over the alphabet {a, b}. It transitions through states Q0, Q1, and Q2 based on input characters. Invalid characters are flagged immediately. The DFA accepts strings that end in either Q0 or Q1, rejecting those that terminate in Q2. The logic is encapsulated in dfa_accepts, while main handles input and output. This program demonstrates fundamental concepts of automata theory, including state transitions, input validation, and final state acceptance. It is a practical example for understanding how DFAs process strings and determine language membership based on defined rules.

Program 4: Design and Implementation of a DFA to accept string over {a, b} such that each string contain aba as substring.

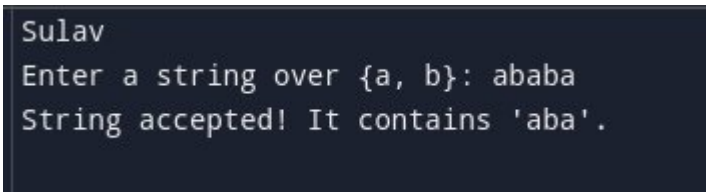


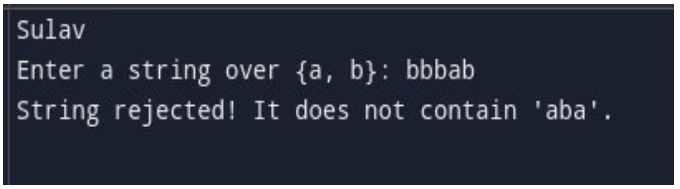
Source code

```
#include <stdio.h>
 #include <string.h>
  int main() {
            char str[100];
            int state = 0;
            printf("Sulav\n");
             printf("Enter a string over {a, b}: ");
            scanf("%s", str);
            for(int i = 0; str[i] != '\0'; i++)
               { char ch = str[i];
               switch(state)
                  { case 0:
                     if(ch == 'a')
                        state = 1;
                     else if(ch == 'b')
                        state = 0;
                     else {
                        printf("Invalid character detected.\n"); return 1;
                     }
                     break;
                  case 1:
                     if(ch == 'a')
                        state = 1;
                     else if(ch == 'b')
                        state = 2;
                     else {
                        printf("Invalid character detected.\n"); return 1;
                     }
                     break;
                  case 2:
                     if(ch == 'a')
                        state = 3;
                     else if(ch == 'b')
                        state = 0;
                     else {
                        printf("Invalid character detected.\n"); return 1;
                     }
                     break;
                  case 3:
                     if(ch == 'a' || ch == 'b')
                        state = 3;
                     else {
                        printf("Invalid character detected.\n"); return 1;
                     }
                     break;
               }
            }
```

```
if(state == 3)
                printf("String accepted! It contains 'aba'.\n"); else
                printf("String rejected! It does not contain 'aba'.\n"); return 0;
        }
```

Output and discussion





This C program checks if a string over {a, b} contains the substring "aba" using a finite state machine. It transitions through four states (0 to 3) based on input characters. Starting from state 0, it moves to state 1 on 'a', then to state 2 on 'b', and finally to state 3 on another 'a'. Once in state 3, it remains there regardless of further input. If the final state is 3, the string is accepted; otherwise, it is rejected. Invalid characters trigger an error. This approach efficiently detects the presence of "aba" in a given input string.

Conclusion:

This lab successfully demonstrated the design and implementation of Deterministic Finite Automata (DFA) for recognizing various regular languages over binary and alphabetic inputs. Four distinct DFA programs were developed: accepting strings ending with "11", starting with "01", not ending with "ab", and containing "aba" as a substring. Each implementation utilized state-based logic with switch-case structures to handle character transitions effectively. The programs highlighted fundamental automata theory concepts including state transitions, input validation, and acceptance conditions. Through practical coding exercises, we gained hands-on experience in translating theoretical DFA designs into functional C programs, reinforcing understanding of finite state machines, regular language recognition, and computational models essential for compiler design and formal language processing.

Lab 3:

Title: Design and Implementation of Nondeterministic Finite Automata (NFA) for Various Languages.

Theory:

A Nondeterministic Finite Automaton (NFA) is a model of computation that, unlike a DFA, allows for ambiguity or "nondeterminism" in its transitions. From a single state and a single input symbol, an NFA can transition to zero, one, or multiple next states. This model is conceptually more flexible than a DFA, yet both have the same computational power—they both recognize the class of regular languages.

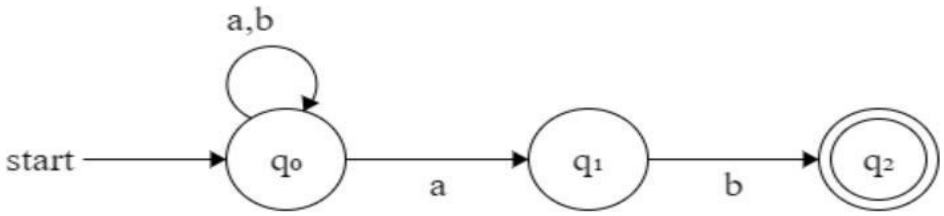
Components of an NFA

An NFA is formally defined as a 5-tuple: $(Q,\Sigma,\delta,q0,F)$, which is very similar to a DFA, but with one crucial difference:

- Q is a finite set of states.
- Σ (Sigma) is a finite set of input symbols (the alphabet).
- δ (Delta) is the transition function. This is where the nondeterminism comes in. It maps a state and an input symbol to a set of next states, rather than a single state. Formally, $\delta: Q \times \Sigma \rightarrow 2Q$, where 2Q is the power set of Q (the set of all subsets of Q).
- q0 is the start state, an element of Q.
- F is a set of final (or accepting) states, a subset of Q.

An additional feature of some NFAs is the epsilon (ϵ) transition, which allows the machine to move from one state to another without consuming an input symbol. This further simplifies the design of certain automata.

Program 1: Design and implementation of NFA to accept strings over {a, b} that ends with ab.



Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
bool isAccepted(char *str) { int n
   = strlen(str);
   if (n < 2) return
      false;
```

```
    return (str[n-2] == 'a' && str[n-1] == 'b');
}
int main() {
    char *input = NULL;
    char buffer[1000];
    printf("Program by: Sulav\n"); printf("Enter
    a string over {a,b}: ");
    if (fgets(buffer, sizeof(buffer), stdin) == NULL)
        { printf("Error reading input.\n");
        return 1;
    }
    buffer[strcspn(buffer, "\n")] = '\0'; input
    = strdup(buffer);
    if (isAccepted(input))
        printf("Accepted: String ends with \"ab\".\n"); else
        printf("Rejected: String does not end with \"ab\".\n"); free(input);
    return 0;
}
```

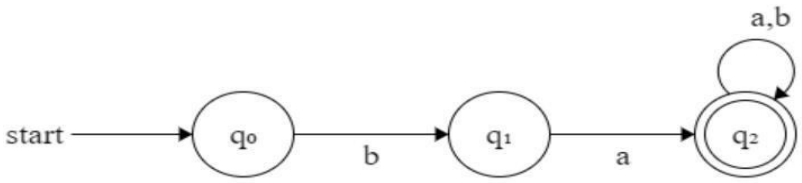
Output and discussion

```
Program by: Sulav
Enter a string over {a,b}: ababba
Rejected: String does not end with "ab".
```

```
Program by: Sulav
Enter a string over {a,b}: bab
Accepted: String ends with "ab".
```

This C program determines if a string ends with "ab". It checks the last two characters of the input string. The isAccepted function returns true if the string has at least two characters and they are 'a' and 'b' respectively, otherwise it returns false.

Program 2: Design and implementation of NFA to accept strings over {a, b} that starts with ba



Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
bool isAccepted(char *str) {
int n = strlen(str); if (n < 2)
    return false;
    return (str[0] == 'b' && str[1] == 'a');
}
int main() {
    char *input = NULL;
```

```
    char buffer[1000];
    printf("Program by: Sulav\n"); printf("Enter
    a string over {a,b}: ");
    if (fgets(buffer, sizeof(buffer), stdin) == NULL)
        { printf("Error reading input.\n");
        return 1;
    }
    buffer[strcspn(buffer, "\n")] = '\0'; input
    = strdup(buffer);
    if (isAccepted(input))
        printf("Accepted: String starts with \"ba\".\n"); else
        printf("Rejected: String does not start with \"ba\".\n"); free(input);
    return 0;
}
```

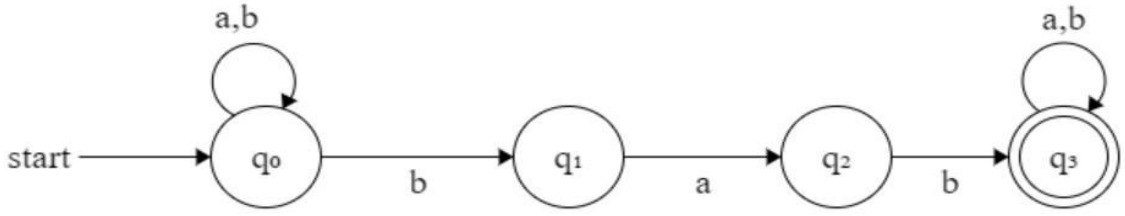
Output and discussion

```
Program by: Sulav
Enter a string over {a,b}: abbaba
Rejected: String does not start with "ba".

Program by: Sulav
Enter a string over {a,b}: babaaa
Accepted: String starts with "ba".
```

This C program checks if a string over {a, b} starts with "ba". It reads input using fgets, removes the newline, and duplicates the string. The isAccepted function returns true only if the first two characters are 'b' and 'a'. Based on this, the program prints whether the string is accepted or rejected, demonstrating simple pattern recognition.

Program 3: Design and Implementation of NFA to accept string over {a, b} such that each string contain bab as substring.



Source code:

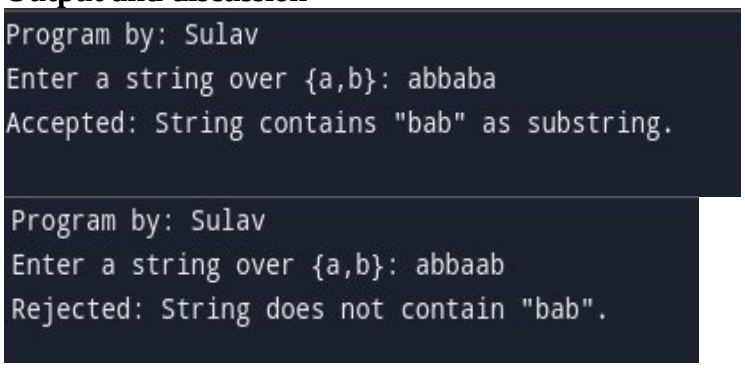
```
#include <stdio.h> #include
<string.h> #include
<stdbool.h>  bool
isAccepted(char *str){
```

```
    return strstr(str, "bab") != NULL;
}
int main()
{
    char buffer[1000];
    printf("Program by: Sulav\n"); printf("Enter
    a string over {a,b}: ");
    if (fgets(buffer, sizeof(buffer), stdin) == NULL)
 {
        printf("Error reading input.\n"); return
        1;
    }

    buffer[strcspn(buffer, "\n")] = '\0'; for
    (int i = 0; buffer[i] != '\0'; i++)
 {
        if (buffer[i] != 'a' && buffer[i] != 'b')
 {
            printf("Invalid input: only 'a' and 'b' are allowed.\n"); return 1;
        }
    }

    if (isAccepted(buffer))
        printf("Accepted: String contains \"bab\" as substring.\n"); else
        printf("Rejected: String does not contain \"bab\".\n"); return 0;
}
```

Output and discussion



This C program validates input to ensure it solely comprises 'a' and 'b'. It then checks if the input string contains "bab" as a substring. The isAccepted function leverages strstr to efficiently locate the substring. If "bab" is found, the string is accepted; otherwise, it is rejected. Error handling for invalid characters and input reading is included.

Conclusion:

This lab demonstrated the design and implementation of Nondeterministic Finite Automata (NFA) for recognizing regular languages over {a, b}. Three NFA programs were developed to accept strings ending with "ab", starting with "ba", and containing "bab" as a substring. Unlike DFAs, NFAs allow multiple state transitions, offering design flexibility while recognizing the same regular languages. Using simple string processing, the programs checked suffixes, prefixes, and substrings with strstr, ensuring valid input with error handling. These exercises showcased NFA concepts and their translation into functional C programs for pattern recognition.

Lab 4

Title: Design and implementation of Push down Automata.

Theory:

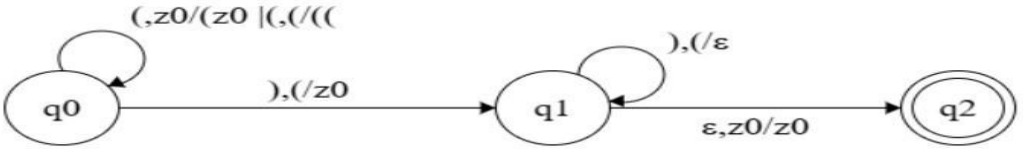
A Pushdown Automaton (PDA) is a computational model that extends the capabilities of a Finite Automaton by adding a stack. This stack acts as an auxiliary memory, allowing the PDA to recognize a broader class of languages known as context-free languages. Unlike the finite memory of a DFA or NFA, the stack provides an unlimited, last in, first-out (LIFO) memory, which is essential for handling nested structures like balanced parentheses or nested loops.

Components of a PDA

A PDA is formally defined as a 7-tuple: $(Q,\Sigma,\Gamma,\delta,q0,Z0,F)$, where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (the alphabet).
- Γ (Gamma) is a finite set of stack symbols.
- δ (Delta) is the transition function. It maps a state, an input symbol (or ϵ), and a stack symbol to a new state and a string of stack symbols to be pushed. Formally, $\delta: Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$.
- q0 is the start state.
- Z0 is the initial stack symbol, an element of Γ. This symbol is on the stack when the PDA begins.
- F is a set of final (or accepting) states.

Program 1: Design and implementation of PDA for Balanced Parentheses.



Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#define MAX 1000  int
main() {
   char str[MAX]; char
   stack[MAX]; int top
   = -1;
   printf("Program by: Sulav\n");
```

```
    printf("Enter parentheses string: ");
    scanf("%s", str);
    for (int i = 0; i < strlen(str); i++) { if
        (str[i] == '(') {
            stack[++top] = '(';
        } else if (str[i] == ')') { if
            (top == -1) {
                printf("Rejected: Not balanced\n"); return
                0;
            }
            top--;
        }
    }
    if (top == -1)
        printf("Accepted: Balanced parentheses\n"); else
        printf("Rejected: Not balanced\n");
    return 0;
}
```

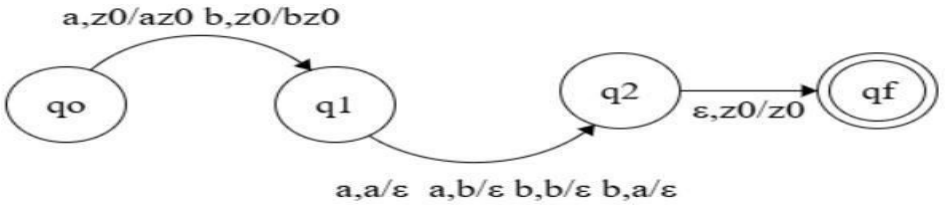
Output and discussion

```
Program by: Sulav
Enter parentheses string: (())
Accepted: Balanced parentheses
```

```
Program by: Sulav
Enter parentheses string: (()
Rejected: Not balanced
```

This C program checks whether a string of parentheses is balanced using a stack-based approach. It pushes each opening bracket '(' onto the stack and pops for each closing bracket ')'. If a closing bracket appears without a matching opening, or if any unmatched opening brackets remain after processing, the string is rejected. Otherwise, it is accepted. The program demonstrates fundamental stack operations and validates syntactic correctness of parentheses, which is essential in parsing expressions, compiler design, and evaluating mathematical formulas. It is a concise and effective implementation of bracket matching logic using arrays and control structures.

Program 2: Design and implementation of PDA for even length strings over {a,b}.



Source code:

```
#include <stdio.h>
#include <string.h> int
main() {
    char str[1000]; int
    state = 0;
```

```
    printf("Program by: Sulav\n");
    printf("Enter string over {a,b}: "); scanf("%s",
    str);
    for(int i=0; i<strlen(str);
        i++){ if(str[i]=='a' ||
        str[i]=='b'){
            state = 1 - state;
        } else {
            printf("Invalid character detected!\n"); return
            0;
        }
    }
    if(state == 0)
        printf("Accepted: String has even length.\n"); else
        printf("Rejected: String has odd length.\n");
    return 0;
}
```

Output and discussion

```
Program by: Sulav
Enter string over {a,b}: aabba
Rejected: String has odd length.
```

```
Program by: Sulav
Enter string over {a,b}: abababab
Accepted: String has even length.
```

This C program checks whether a string over the alphabet {a, b} has even length using a simple state-flipping mechanism. It starts in state 0 and toggles between states 0 and 1 for each valid character. If any character is not 'a' or 'b', it flags an error. After processing the entire string, it accepts if the final state is 0 (even length), and rejects if the state is 1 (odd length). This approach cleverly uses state transitions to determine string length parity without explicitly counting characters, making it an elegant demonstration of finite automata principles in action.

Conclusion:

This lab successfully demonstrated the design and implementation of Pushdown Automata (PDA) for recognizing context-free languages. Two programs were developed: one for validating balanced parentheses using stack operations, and another for accepting even-length strings over {a,b} using state transitions. The balanced parentheses program effectively utilized LIFO stack mechanics to handle nested structures, pushing opening brackets and popping for closing ones. The even-length string validator employed elegant state-flipping logic to determine length parity without explicit counting. These implementations showcased how PDAs extend finite automata capabilities through auxiliary memory, enabling recognition of more complex language patterns essential for parsing, compiler design, and syntactic analysis in programming languages.

Lab 5

Title: Design and Implementation of Turing Machine.

Theory:

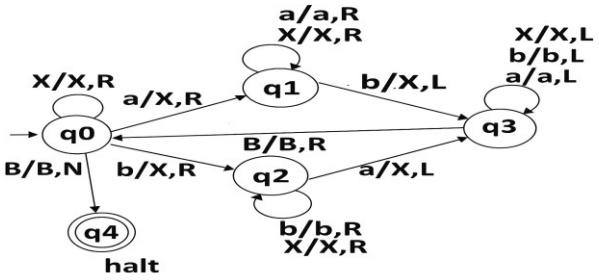
A Turing Machine (TM) is a theoretical model of computation. It's considered the most powerful and fundamental model for a general-purpose computer. Unlike DFAs and PDAs, which have limited memory, a TM has an infinite tape that serves as its memory. This makes it capable of solving any problem that a modern computer can solve, provided there is enough time and memory. The Church-Turing thesis states that a Turing Machine can compute any function computable by an algorithm.

Components of a Turing Machine

A TM is formally defined as a 7-tuple: $(Q,\Sigma,\Gamma,\delta,q0,B,F)$, where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (the alphabet).
- Γ is a finite set of tape symbols. This includes all symbols from the input alphabet plus a special blank symbol (B).
- δ is the transition function. This is the core of the TM's logic. It maps a state and a tape symbol to a new state, a new tape symbol to write, and a direction to move the tape head (left or right). Formally, $\delta: Q\times\Gamma \rightarrow Q\times\Gamma\times\{L,R\}$.
- q0 is the start state.
- B is the blank symbol, used for empty cells on the tape. $B\in\Gamma$.
- F is a set of final (or accepting) states.

Program: Write a C program to simulate a Turing Machine for $L=\{w\in\{a,b\}*|w$ has equal number of a's and b's$\}$



Source code:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define MAX 1000  int
main() {
    char tape[MAX]; int
    len;
    bool changed;
    printf("Program by: Sulav\n");
     printf("Enter string over {a,b}: ");
```

```
    scanf("%s",  tape);
    len  =  strlen(tape);
    do {
        changed = false;
        int i_a = -1, i_b = -1;
        for (int i = 0; i < len; i++) {
            if (tape[i] == 'a' && i_a == -1) i_a = i; if
            (tape[i] == 'b' && i_b == -1) i_b = i;
        }
        if (i_a != -1 && i_b != -1) {
            tape[i_a] = 'x';
            tape[i_b] = 'x'; changed
            = true;
        }
    } while (changed);
    bool accept = true;
    for (int i = 0; i < len; i++) {
        if (tape[i] == 'a' || tape[i] == 'b') { accept =
            false;
            break;
        }
    }
    if (accept)
        printf("Accepted: Equal number of a's and b's\n"); else
        printf("Rejected: Unequal number of a's and b's\n");
    return 0;
}
```

Output and discussion

```
Program by: Sulav
Enter string over {a,b}: aaabba
Rejected: Unequal number of a's and b's
```

```
Program by: Sulav
Enter string over {a,b}: ababba
Accepted: Equal number of a's and b's
```

This C program checks whether a string over {a, b} contains an equal number of a's and b's. It repeatedly finds and replaces one 'a' and one 'b' with 'x' until no more pairs remain. If no unmatched 'a' or 'b' is left, the string is accepted. Otherwise, it is rejected. It simulates a basic Turing machine.

Conclusion:

This lab explored Turing Machine implementation for recognizing strings with equal numbers of 'a's and 'b's over the alphabet {a,b}. The program simulates a TM by repeatedly pairing and marking one 'a' with one 'b' using 'x' symbols until no more pairs exist. The algorithm validates string acceptance by checking if any unmarked characters remain after processing. This implementation demonstrates fundamental TM concepts including tape manipulation, iterative processing, and decision logic. The program effectively highlights how Turing Machines can solve computational problems that require unlimited memory and complex state transitions, illustrating the theoretical foundation of modern computing and algorithmic problem-solving capabilities in formal language recognition.