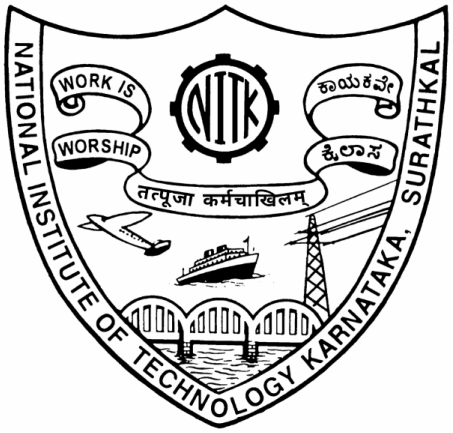
A Report on
Compiler Design Lab (CS304): Mini Project Phase 1

Abhiram Suresh (Roll No: 231CS202)
Advaith Nair (Roll No: 231CS205)
Arjun Rijesh (Roll No: 231CS212)



Department of Computer Science and Engineering
National Institute of Technology Karnataka
Surathkal, Mangaluru - 575025

15-August-2025

1 Introduction

This report presents the implementation of a lexical analyzer for the C programming language using Flex. A lexical analyzer forms the first phase of a compiler, where the source program is read as a stream of characters and transformed into a sequence of tokens.

1.1 Lexical Analysis

Lexical analysis is the process of converting a sequence of raw characters into a sequence of structured tokens. A lexical analyzer performs this transformation by scanning the input character by character, grouping them into meaningful units (tokens), and categorizing them according to their role in the programming language.
The lexical analysis phase is crucial because it provides the foundation for the subsequent compilation stages, including syntax analysis, semantic analysis, and code generation. By filtering out irrelevant details (such as whitespace and comments), building symbol and constant tables, and detecting invalid inputs early, the lexical analyzer ensures that later phases, like semantic analysis, receive a stream of tokens accompanied by valuable contextual metadata.
Our implementation not only performs token recognition, but also integrates the construction of a **symbol table** to record identifiers and their frequencies of use, as well as a **constant table** to store details about constants, including their type, value, and line number of occurrence. In addition, it provides mechanisms for handling nested comments, reporting lexical errors such as invalid tokens and unterminated comments, and maintaining accurate line numbers for error reporting.

1.2 Tokens & Lexemes

Token: A pair consisting of a token name and an optional attribute value that uniquely identifies a sequence of characters.

Lexeme: The actual sequence of characters in the source program that matches the pattern for a token.

In our mini-project, we have implemented a lexical analyzer for the C language using **Flex**. The scanner can recognize the following:

- Identifiers - Variable names, function names, and other user-defined names; these are stored in the symbol table along with their frequency of occurrence
- Preprocessor Directives - Lines beginning with the # symbol, such as #include and #define
- Keywords - Reserved words in C (e.g., int, float, if, while, etc.)
- Constants - Numeric constants (integer, floating-point, hexadecimal, octal), character constants, and string literals; these are recorded in the constant table with their type, value, and line number
- Operators - Arithmetic, relational, logical, bitwise, assignment, increment/decrement, pointer, address-of, and member operators

- Punctuation - Parentheses, braces, brackets, semicolons, and commas
- Comments - single-line and multi-line including nested multi-line comments

For example, in:

```
int x = 10;
```

“int” is a keyword token, “x” is an identifier token, and “10” is a constant token.

2 Code Listing

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* ---------------- Utilities ---------------- */

static char* xstrdup(const char* s) {
    if (!s) {
        return NULL;
    }
    size_t n = strlen(s) + 1;
    char* p = (char*)malloc(n);
    if (p) {
        memcpy(p, s, n);
    }
    return p;
}

static char* xsubstr(const char* s, size_t a, size_t b) {
    if (!s || b < a) {
        return xstrdup("");
    }
    size_t n = b - a;
    char* p = (char*)malloc(n + 1);
    if (!p) {
        return NULL;
    }
    memcpy(p, s + a, n);
    p[n] = '\0';
    return p;
}

static char* trim(char* s) {
    if (!s) {
        return s;
    }
```

```
    size_t n = strlen(s);
    size_t i = 0;
    while (i < n && isspace((unsigned char)s[i])) {
        i++;
    }
    size_t j = n;
    while (j > i && isspace((unsigned char)s[j - 1])) {
        j--;
    }
    size_t m = j - i;
    memmove(s, s + i, m);
    s[m] = '\0';
    return s;
}

/* ---------------- Symbol Table ---------------- */

typedef struct Symbol {
    char* name;
    char* type;
    char* dimensions;
    int   frequency;
    char* return_type;
    char* param_lists;
    int   is_function;
    struct Symbol* next;
} Symbol;

#define SYM_HASH_SIZE 211
static Symbol* symtab[SYM_HASH_SIZE];

static unsigned long djb2(const char* str) {
    unsigned long hash = 5381;
    int c;
    while ((c = (unsigned char) *str++)) {
        hash = ((hash << 5) + hash) + c;
    }
    return hash;
}

static Symbol* sym_lookup(const char* name) {
    unsigned long h = djb2(name) % SYM_HASH_SIZE;
    Symbol* s = symtab[h];
    while (s) {
        if (strcmp(s->name, name) == 0) {
            return s;
        }
        s = s->next;
    }
    return NULL;
}
```

```
static Symbol* sym_insert(const char* name) {
    unsigned long h = djb2(name) % SYM_HASH_SIZE;
    Symbol* s = (Symbol*)calloc(1, sizeof(Symbol));
    s->name = xstrdup(name);
    s->frequency = 0;
    s->next = symtab[h];
    symtab[h] = s;
    return s;
}

static Symbol* sym_touch(const char* name) {
    Symbol* s = sym_lookup(name);
    if (!s) {
        s = sym_insert(name);
    }
    s->frequency++;
    return s;
}

static void sym_set_type(Symbol* s, const char* type) {
    if (!s) return;
    if (s->type) free(s->type);
    s->type = xstrdup(type);
}

static void sym_set_return(Symbol* s, const char* r) {
    if (!s) return;
    if (s->return_type) free(s->return_type);
    s->return_type = xstrdup(r);
}

static void sym_append_dims(Symbol* s, const char* dims) {
    if (!s || !dims) return;
    size_t old = s->dimensions ? strlen(s->dimensions) : 0;
    size_t add = strlen(dims);
    char* p = (char*)malloc(old + add + 1);
    if (!p) return;
    if (old) {
        memcpy(p, s->dimensions, old);
        free(s->dimensions);
    }
    memcpy(p + old, dims, add);
    p[old + add] = '\0';
    s->dimensions = p;
}

static void sym_append_params(Symbol* s, const char* params) {
    if (!s || !params) return;
    const char* sep = s->param_lists ? " ; " : "";
    size_t old = s->param_lists ? strlen(s->param_lists) : 0;
```

```
    size_t add = strlen(sep) + strlen(params);
    char* p = (char*)malloc(old + add + 1);
    if (!p) return;
    if (old) {
        memcpy(p, s->param_lists, old);
        free(s->param_lists);
    }
    memcpy(p + old, sep, strlen(sep));
    memcpy(p + old + strlen(sep), params, strlen(params));
    p[old + add] = '\0';
    s->param_lists = p;
}

/* ---------------- Constant Table ---------------- */

typedef struct Constant {
    char* var_name;
    int   line;
    char* value;
    char* type;
} Constant;

static Constant* consts = NULL;
static size_t nconsts = 0, capconsts = 0;

static void const_add(const char* var, int line, const char* val,
    const char* type) {
    if (nconsts == capconsts) {
        capconsts = capconsts ? capconsts * 2 : 64;
        consts = (Constant*)realloc(consts, capconsts * sizeof(
           Constant));
    }
    consts[nconsts].var_name = xstrdup(var ? var : "-");
    consts[nconsts].line = line;
    consts[nconsts].value = xstrdup(val ? val : "");
    consts[nconsts].type = xstrdup(type ? type : "");
    nconsts++;
}

static int in_declaration = 0;
static char last_type[256] = {0};
static char last_ident[256] = {0};
static int last_was_ident = 0;
static int array_capture_for_ident = 0;

static void start_declaration(const char* t) {
    in_declaration = 1;
    last_type[0] = '\0';
    if (t) {
        strncat(last_type, t, sizeof(last_type) - 1);
    }
```

```
}

static void add_type_token(const char* t) {
    if (last_type[0]) {
        strncat(last_type, " ", sizeof(last_type) - 1);
    }
    strncat(last_type, t, sizeof(last_type) - 1);
}

static void end_declaration() {
    in_declaration = 0;
    last_type[0] = '\0';
}

static int paren_depth = 0;
static int capturing_args = 0;
static char* arg_buffer = NULL;
static size_t arg_cap = 0, arg_len = 0;
static Symbol* current_func_sym = NULL;

static void args_begin(Symbol* s) {
    capturing_args = 1;
    paren_depth = 1;
    arg_len = 0;
    if (!arg_buffer) {
        arg_cap = 256;
        arg_buffer = (char*)malloc(arg_cap);
    }
    current_func_sym = s;
}

static void args_push_char(int c) {
    if (!capturing_args) return;
    if (arg_len + 2 > arg_cap) {
        arg_cap *= 2;
        arg_buffer = (char*)realloc(arg_buffer, arg_cap);
    }
    arg_buffer[arg_len++] = (char)c;
    arg_buffer[arg_len] = '\0';
}

static void args_end() {
    if (capturing_args && current_func_sym) {
        if (arg_len && arg_buffer[arg_len - 1] == ')') {
            arg_buffer[arg_len - 1] = '\0';
        }
        char* s = xstrdup(arg_buffer);
        trim(s);
        sym_append_params(current_func_sym, s);
        free(s);
    }
```

```
    capturing_args = 0;
    paren_depth = 0;
    arg_len = 0;
    current_func_sym = NULL;
}

/* ---------------- Token printing ---------------- */

static void print_token(const char* kind, const char* lexeme) {
    printf("[line %d] %-12s : %s\n", yylineno, kind, lexeme);
}

/* ---------------- classify numbers ---------------- */

static const char* classify_int(const char* s) {
    if (!s) return "int";
    if (strlen(s) > 2 && s[0] == '0' && (s[1] == 'x' || s[1] == '
       X')) return "hex";
    if (strlen(s) > 2 && s[0] == '0' && (s[1] == 'b' || s[1] == '
       B')) return "bin";
    if (s[0] == '0' && strlen(s) > 1) return "oct";
    return "int";
}

%}

%option noyywrap
%option yylineno

/* --------------- Definitions ------------------ */
%x COMMENT
%x STRING
%x CHARLIT
%x PP
%x FUNCARGS

DIGIT       [0-9]
LETTER      [A-Za-z_]
ID          {LETTER}[A-Za-z0-9_]*
WS          [ \t\r]+

HEX         0[xX][0-9A-Fa-f]+
BIN         0[bB][01]+
OCT         0[0-7]+
INTSUFFIX   ([uU][lL]?|[lL][uU]?|[uU][lL][lL]|[lL][lL][uU]?)
EXP         ([eE][+-]?{DIGIT}+)
FLOAT1      {DIGIT}+"."{DIGIT}+({EXP})?
FLOAT2      {DIGIT}+"."({EXP})?
FLOAT3      "."{DIGIT}+({EXP})?
FLOAT4      {DIGIT}+{EXP}
FLOATSUF    [fFlL]?
```

```
ESC         (\\[abfnrtv\\'"?]|\\x[0-9A-Fa-f]+|\\[0-7]{1,3})

%%

^[ \t]*\#[^\n\\]*(\\\n[^\n\\]*)* {
    print_token("PREPROC", yytext);
    char* tmp = xstrdup(yytext);
    char* p = tmp;

    while (*p == ' ' || *p == '\t') p++;
    if (*p == '#') p++;
    while (*p == ' ') p++;

    if (strncmp(p, "define", 6) == 0 && isspace((unsigned char)p
       [6])) {
        p += 6;
        while (*p == ' ') p++;

        char namebuf[256] = {0};
        int i = 0;
        while (*p && (isalnum((unsigned char)*p) || *p == '_')) {
            if (i < 255) namebuf[i++] = *p;
            p++;
        }
        namebuf[i] = '\0';
        while (*p == ' ' || *p == '\t') p++;

        char* val = trim(p);
        if (val && namebuf[0]) {
            const_add(namebuf, yylineno, val, "macro");
        }
    }
    free(tmp);
}

{WS}        { /* ignore */ }
\n          { /*ignore*/ }

"//".* { /* ignore */ }

"/*"        {
    BEGIN(COMMENT);
    int depth = 1;
    int c1, c2;
    while (depth > 0) {
        c1 = input();
        if (c1 == EOF) {
            fprintf(stderr, "[line %d] ERROR: Unterminated
               comment\n", yylineno);
            BEGIN(INITIAL);
```

```
            break;
        }
        if (c1 == '\n') {
        }
        if (c1 == '/') {
            c2 = input();
            if (c2 == '*') {
                depth++;
            } else if (c2 != EOF) {
                unput(c2);
            }
        } else if (c1 == '*') {
            c2 = input();
            if (c2 == '/') {
                depth--;
            } else if (c2 != EOF) {
                unput(c2);
            }
        }
    }
    BEGIN(INITIAL);
}

\"          {
    BEGIN(STRING);
    yyless(0);
}
<STRING>\"([^"\\\n]|{ESC})*\" {
    print_token("STRING", yytext);
    const_add("-", yylineno, yytext, "string");
    BEGIN(INITIAL);
}
<STRING>\"([^"\\\n]|{ESC})*\\\n {
    /* continued line inside string - treat as part of string */
}
<STRING>\n  {
    fprintf(stderr, "[line %d] ERROR: Unterminated string literal
       \n", yylineno - 1);
    BEGIN(INITIAL);
}
<STRING>.   { /* consume */ }


\'          {
    BEGIN(CHARLIT);
    yyless(0);
}
<CHARLIT>\'([^'\\\n]|{ESC})+\' {
    print_token("CHAR", yytext);
    const_add("-", yylineno, yytext, "char");
    BEGIN(INITIAL);
```

```
}
<CHARLIT>\n {
    fprintf(stderr, "[line %d] ERROR: Unterminated char literal\n
       ", yylineno - 1);
    BEGIN(INITIAL);
}
<CHARLIT>.  { /* consume */ }

"auto"|"break"|"case"|"const"|"continue"|"default"|"do"|"else"|"
   enum"|"extern"|"for"|"goto"|"if"|"register"|"return"|"sizeof
   "|"static"|"struct"|"switch"|"typedef"|"union"|"volatile"|"
   while"|"inline"|"restrict"|"_Alignas"|"_Alignof"|"_Atomic"|"
   _Bool"|"_Complex"|"_Generic"|"_Imaginary"|"_Noreturn"|"
   _Static_assert"|"_Thread_local" {
    print_token("KEYWORD", yytext);
    last_was_ident = 0;
}

"void"|"char"|"short"|"int"|"long"|"float"|"double"|"signed"|"
   unsigned" {
    print_token("TYPE", yytext);
    if (!in_declaration) {
        start_declaration(yytext);
    } else {
        add_type_token(yytext);
    }
    last_was_ident = 0;
}

{ID}        {
    print_token("IDENT", yytext);
    Symbol* s = sym_touch(yytext);
    strncpy(last_ident, yytext, sizeof(last_ident) - 1);
    last_ident[sizeof(last_ident) - 1] = '\0';
    last_was_ident = 1;
    array_capture_for_ident = 1;
    if (in_declaration) {
        if (!s->type) {
            sym_set_type(s, last_type);
        }
    }
}

{HEX}{INTSUFFIX}?   { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, "hex");   last_was_ident = 0; }
{BIN}{INTSUFFIX}?   { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, "bin");   last_was_ident = 0; }
{FLOAT1}{FLOATSUF}? { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, "float"); last_was_ident = 0; }
{FLOAT2}{FLOATSUF}? { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, "float"); last_was_ident = 0; }
```

```
{FLOAT3}{FLOATSUF}? { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, "float"); last_was_ident = 0; }
{FLOAT4}{FLOATSUF}? { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, "float"); last_was_ident = 0; }
{OCT}{INTSUFFIX}?   { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, "oct");   last_was_ident = 0; }
{DIGIT}+{INTSUFFIX}? { print_token("NUMBER", yytext); const_add
   ("-", yylineno, yytext, classify_int(yytext)); last_was_ident
   = 0; }

"("         {
    print_token("PUNCT", yytext);
    if (last_was_ident) {
        Symbol* s = sym_lookup(last_ident);
        if (!s) {
            s = sym_touch(last_ident);
        }
        if (in_declaration) {
            s->is_function = 1;
            if (last_type[0]) {
                sym_set_return(s, last_type);
            }
        }
        args_begin(s);
        BEGIN(FUNCARGS);
    }
    last_was_ident = 0;
}
<FUNCARGS>[^()\n]+ {
    /* collect raw text inside args */
    for (size_t i = 0; i < yyleng; i++) {
        args_push_char(yytext[i]);
    }
}
<FUNCARGS>"("       { args_push_char('('); paren_depth++; }
<FUNCARGS>")"       {
    args_push_char(')');
    paren_depth--;
    if (paren_depth == 0) {
        args_end();
        BEGIN(INITIAL);
    }
}
<FUNCARGS>\n        { args_push_char('\n'); }

")"         { print_token("PUNCT", yytext); last_was_ident = 0; }

"["{WS}*{DIGIT}+{WS}*"]" {
    print_token("PUNCT", yytext);
    if (array_capture_for_ident) {
        Symbol* s = sym_lookup(last_ident);
```

```
        if (s) {
            sym_append_dims(s, yytext);
        }
    }
    last_was_ident = 0;
}
"["|"]"     { print_token("PUNCT", yytext); last_was_ident = 0; }

";"         {
    print_token("PUNCT", yytext);
    array_capture_for_ident = 0;
    if (in_declaration) {
        end_declaration();
    }
    last_was_ident = 0;
}
","         { print_token("PUNCT", yytext); last_was_ident = 0; }
"{"         {
    print_token("PUNCT", yytext);
    if (in_declaration) {
        end_declaration();
    }
    last_was_ident = 0;
}
"}"         { print_token("PUNCT", yytext); last_was_ident = 0; }

"."|"->"    { print_token("OP", yytext); last_was_ident = 0; }
"++"|"--"   { print_token("OP", yytext); last_was_ident = 0; }
"+="|"-="|"*="|"/="|"%=" { print_token("OP", yytext);
   last_was_ident = 0; }
"&="|"^="|"|="          { print_token("OP", yytext);
   last_was_ident = 0; }
"=="|"!="|"<="|">="     { print_token("OP", yytext);
   last_was_ident = 0; }
"<<"|">>"   { print_token("OP", yytext); last_was_ident = 0; }
"&&"|"||"   { print_token("OP", yytext); last_was_ident = 0; }
"+"|"-"|"*"|"/"|"%"     { print_token("OP", yytext);
   last_was_ident = 0; }
"<"|">"|"=" { print_token("OP", yytext); last_was_ident = 0; }
"!"|"~"|"&"|"^"         { print_token("OP", yytext);
   last_was_ident = 0; }
"|"         { print_token("OP", yytext); last_was_ident = 0; }
"?"|":"     { print_token("OP", yytext); last_was_ident = 0; }

.           {
    fprintf(stderr, "[line %d] ERROR: Invalid token '%s'\n",
       yylineno, yytext);
}

<<EOF>>     {
    printf("\n==== SYMBOL TABLE ====\n");
```

```
    printf("%-20s %-15s %-12s %-10s %-15s %s\n", "Name", "Type",
       "Dimensions", "Frequency", "Return Type", "Parameters
       Lists in Function call");
    for (int i = 0; i < SYM_HASH_SIZE; i++) {
        for (Symbol* s = symtab[i]; s; s = s->next) {
            printf("%-20s %-15s %-12s %-10d %-15s %s\n",
                   s->name,
                   s->type ? s->type : "-",
                   s->dimensions ? s->dimensions : "-",
                   s->frequency,
                   s->return_type ? s->return_type : (s->
                      is_function ? "unknown" : "-"),
                   s->param_lists ? s->param_lists : "-"
                  );
        }
    }
    printf("\n==== CONSTANT TABLE ====\n");
    printf("%-20s %-10s %-30s %s\n", "Variable Name", "Line No.",
        "Value", "Type");
    for (size_t i = 0; i < nconsts; i++) {
        printf("%-20s %-10d %-30s %s\n",
               consts[i].var_name ? consts[i].var_name : "-",
               consts[i].line,
               consts[i].value ? consts[i].value : "",
               consts[i].type ? consts[i].type : ""
              );
    }
    return 0;
}

%%

int main(int argc, char** argv) {
    if (argc > 1) {
        FILE* f = fopen(argv[1], "r");
        if (!f) {
            perror("fopen");
            return 1;
        }
        yyin = f;
    }
    yylex();
    return 0;
}
```

3 DFA Diagram

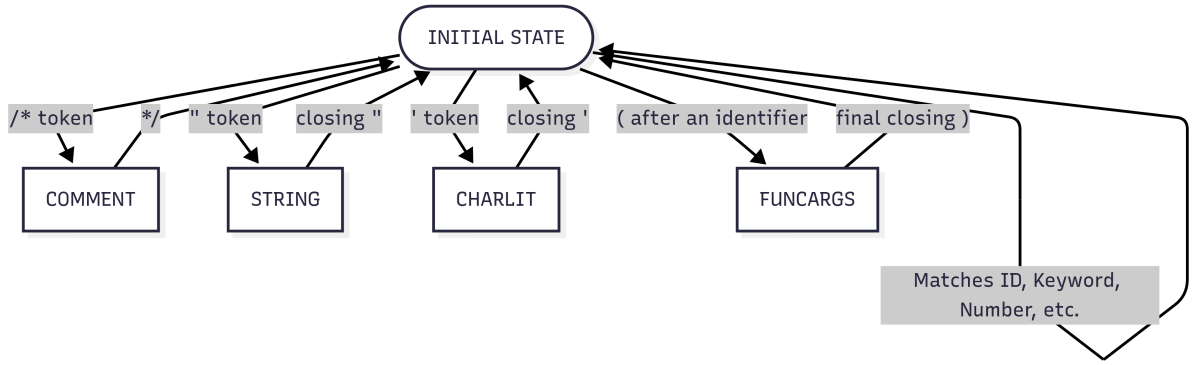


Figure 1: FULL DFA

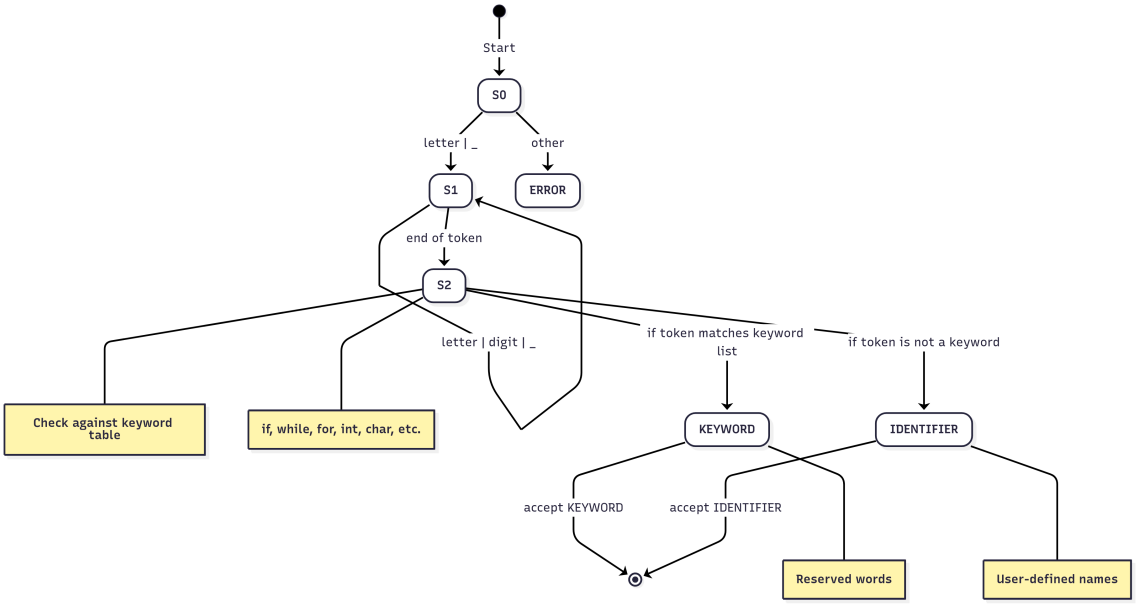


Figure 2: Identifier vs Keywords

The behavior of the DFA upon entering the `INITIAL STATE` can be summarized as follows:

- **Start of Tokenization:** Every time the analyzer looks for a new token, it begins in this state.
- **Specific Delimiters:** The initial character(s) route the DFA to a dedicated state for handling complex tokens:
  - If the input starts with '/*', the DFA transitions to the `COMMENT` state to process a multi-line comment.
  - A '"' (double quote) transitions to the `STRING` state to handle string literals.

  - A ‘'‘ (single quote) transitions to the `CHARLIT` state for character literals.

- **Context-Sensitive Tokens:** In some cases, the transition depends on context. For example, an opening parenthesis ‘(‘ after an identifier transitions the DFA to the `FUNCARGS` state to parse function arguments.

- **General Tokens:** For most other inputs, such as letters or digits that begin an identifier, keyword, or number, the DFA follows a general path to recognize these common tokens.

- **Return to Initial State:** After a token is fully recognized (e.g., upon reaching a closing ‘*/‘ for a comment or the final ‘"‘ for a string), the lexeme is processed, and the DFA returns to the `INITIAL STATE` to begin scanning for the next token. This is represented by the "closing" arrows returning to the top.

4 Assumptions

The following assumptions were made in addition to the C language specification:

1. Identifiers begin with a letter or underscore, followed by letters, digits, or underscores.
2. Numbers are recognized in decimal, octal (leading 0), and hexadecimal (prefix 0x) formats.
3. Floating point numbers must include a decimal point.
4. Strings are enclosed in double quotes and cannot span multiple lines.
5. Nested comments are supported.
6. Preprocessor directives are assumed to appear at the start of a line.
7. Cant differentiate between the deference operator and arithmetic operator *

5 Handling of Strings, Comments, and Errors

5.1 String and Character Literal Handling

The scanner utilizes exclusive start conditions to parse string and character literals, ensuring that special characters are not misinterpreted.

- **State Transition:** An opening double-quote (`"`) transitions the scanner into the `STRING` state. Similarly, an opening single-quote (`'`) enters the `CHARLIT` state. The `yyless(0)` function is used to ensure the quote itself is included in the token text.

- **Tokenization:** Once in the respective state, a pattern matches the entire literal content until the closing quote. The matched literal is then printed as a token and added to the constant table.

- **Error Handling:** If a newline character is encountered before the literal is terminated by its closing quote, it is considered an error. An "Unterminated string literal" or "Unterminated char literal" message is printed to standard error, and the scanner returns to the `INITIAL` state.

5.2 Comment Handling

The scanner is capable of processing both single-line and multi-line C-style comments.

- **Single-Line Comments:** Any text beginning with `//` until the end of the line is matched and ignored.
- **Multi-Line Comments:** An opening `/*` transitions the scanner into the `COMMENT` start condition.
- **Nested Comment Support:** The scanner correctly handles nested multi-line comments. It initializes a `depth` counter to 1 and increments it for each subsequent `/*` and decrements it for each `*/`. The scanner only returns to the `INITIAL` state when the depth counter reaches zero.
- **Error Handling:** If the end of the file is reached while the scanner is still in the `COMMENT` state (i.e., the depth is non-zero), an "Unterminated comment" error is reported.

5.3 General Error Handling

In addition to specific cases for literals and comments, the scanner has general mechanisms for error reporting.

- **Invalid Tokens:** A catch-all rule, `.`, matches any single character that does not conform to any of the other defined rules. This triggers an "Invalid token" error message, which includes the offending character(s).
- **File Handling Errors:** In the `main` function, if a source file is provided as a command-line argument but cannot be opened, the `fopen` call will fail. This condition is checked, and the system error is reported via `perror` before the program exits with a status of 1.

6 Results

Test Case 1

Input:

```c
#include <stdio.h>
unsigned long fact(int n);
int add(int x, int y) { return x + y; }
int arr[10][20];
int main()
{
    int r = add(3, 5);
    printf("%d\n", r);
    return 0;
}
unsigned long fact(int n)
{
    if (n <= 1)
        return 1;
    return n * fact(n - 1);
}
```

```
==== SYMBOL TABLE ====
Name                Type            Dimensions   Frequency  Return Type     Parameters Lists in Function call
n                   -               -            2          -               -
r                   int             -            1          -               -
x                   -               -            1          -               -
y                   -               -            1          -               -
add                 int             -            2          int             int x, int y ; 3, 5
printf              -               -            1          -               "%d\n", r
arr                 int             [10][20]     1          -               -
fact                unsigned long   -            3          unsigned long   int n ; int n ; n - 1
main                int             -            1          int

==== CONSTANT TABLE ====
Variable Name       Line No.   Value                    Type
-                   9          0                        int
-                   13         1                        int
-                   14         1                        int
```

Symbol Table (Test Case 1)

Identifier	Type (if known)	Scope	First Line	Last Line	Freq
main	int (function)	global	5	10	1
add	int (function)	global	3	3	1
fact	unsigned long (function)	global	2	16	1
arr	int[10][20]	global	4	4	1
r	int	local	7	9	2
x	int (param)	local	3	3	1
y	int (param)	local	3	3	1
n	int (param)	local	11	15	3
printf	identifier (extern)	extern	8	8	1

Constant Table (Test Case 1)

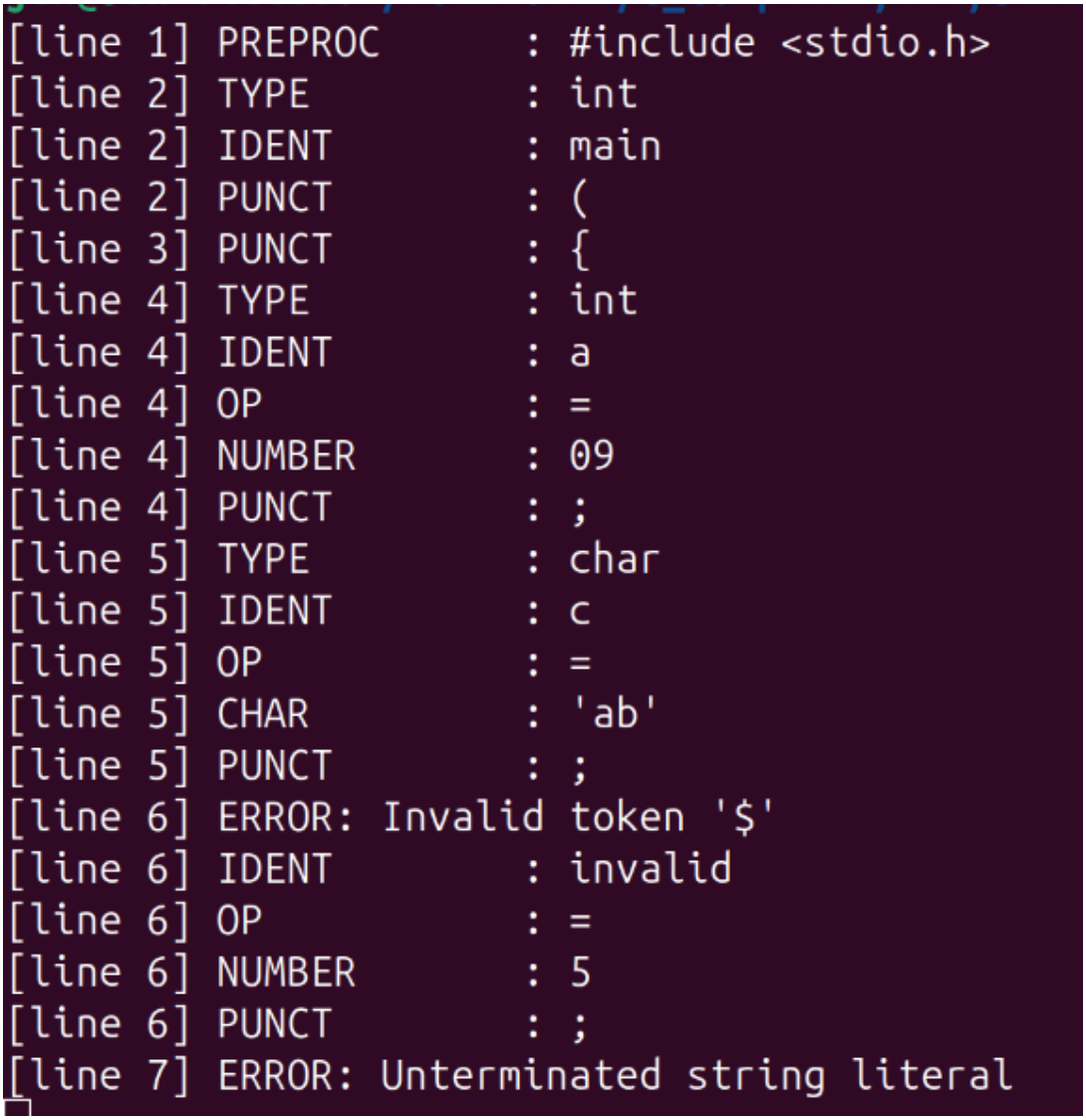
Kind	Lexeme/Value	Datatype	Base	Line
integer	10	int	dec	4
integer	20	int	dec	4
integer	3	int	dec	7
integer	5	int	dec	7
string	%d\n	char[]	N/A	8
integer	0	int	dec	9
integer	1	int	dec	14

Test Case 2

Input:

```c
#include <stdio.h>
int main()
{
    int a = 09;    // octal invalid in C (but lexer will just see
        a number token)
    char c = 'ab'; // invalid char literal (lexer will catch as
       char literal or error depending on quotes)
    $invalid = 5;  // invalid token starting with $
    "unterminated string
    /* unterminated comment
    return 0;
}
```

Output:



Symbol Table (Test Case 2)

Identifier	Type (if known)	Scope	First Line	Last Line	Freq
main	int (function)	global	2	10	1
a	int	local(main)	4	4	1
c	char	local(main)	5	5	1

Constant Table (Test Case 2)

Kind	Lexeme/Value	Datatype	Base	Line
integer	09	int	oct? / dec form	4

(continued on next page)

Kind	Lexeme/Value	Datatype	Base	Line
integer	5	int	dec	6

7 Conclusion

In this phase of the mini project, we successfully implemented a lexical analyzer using Flex. The analyzer correctly identifies tokens, generates symbol and constant tables, and reports lexical errors with line numbers. The test cases demonstrate the correctness of token recognition and the robustness of error handling. Future work will focus on integrating this lexical analyzer with parsing and se- mantic analysis modules, eventually leading to a complete compiler front-end imple- mentation.