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

1.1 Lexical Analysis

Lexical analysis is the first phase of the compiler where the source program is scanned from left to right, character by character, and divided into meaningful sequences called *lexemes*. The lexical analyzer produces tokens as output, which are then used by the syntax analyzer.

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

- Identifiers
- Preprocessor directives
- Keywords
- Constants (numeric, string, character)
- Operators
- Punctuation
- Comments (single-line and nested multi-line)

It also maintains a rudimentary symbol table and constant table.

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.

For example, in:

```
int x = 10;
```

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

2 DFA Diagram

Identifier DFA

Pattern recognized: `[A-Za-z_][A-Za-z0-9_]*`

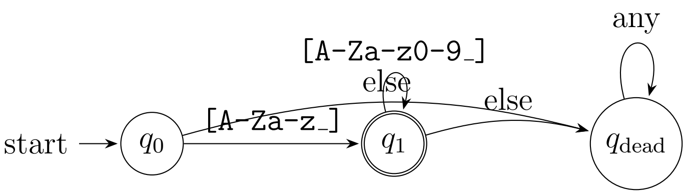

Figure 1: DFA for recognizing identifiers

Explanation

- q_0 (start, non-accepting): On a letter or underscore, move to q_1; otherwise go to dead.
- q_1 (accepting): Consume letters, digits, or underscores staying in q_1. On any other character, transition to dead (in practice, the scanner *stops* the lexeme before that character and leaves it to the next token).
- q_{dead}: Sink state for non-matching strings.

Notes

- This DFA intentionally treats keywords as *post-processing*: first match as an identifier, then check the lexeme against the keyword table to reclassify.
- This matches your scanner rule: `[A-Za-z_][A-Za-z0-9_]*`. (Unicode identifiers and universal character names are out of scope here.)

State	**[A-Za-z_]**	**[0-9]**	**else**
q_0	q_1	q_{dead}	q_{dead}
q_1	q_1	q_1	q_{dead}
q_{dead}	q_{dead}	q_{dead}	q_{dead}

Table 1: Transition table for the Identifier DFA

3 Results

Test Case 1

Input:

```
int main() {
    const int x = 10;
    const float y = 3.1415;
    return x + y;
}
```

Output:

- Keywords: `int`, `const`, `return`
- Identifiers: `main`, `x`, `y`
- Constants: `10`, `3.1415`

Symbol Table:

Name	Type	Line Number	Qualifier	
main	function	1	-	
x	int	2	const	
y	float	3	const	
10	int	2	-	
3.1415	float	3	-	

Test Case 2

Input:

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main() {
    int result = sum(5, 7);
    printf("Sum is %d\n", result);
    return 0;
}
```

Output:

- Preprocessor: `#include <stdio.h>`
- Keywords: `int`, `return`
- Identifiers: `sum`, `a`, `b`, `main`, `result`, `printf`
- Constants: `5`, `7`, `"Sum is %d\n"`

Symbol Table:

Name	Type	Line Number	Qualifier	
sum	function	3	-	
a	int	3	parameter	
b	int	3	parameter	
main	function	7	-	
result	int	8	-	
5	int	8	-	
7	int	8	-	
”Sum is %d\n”	string	9	-	

Appendix

Flex Source (scanner.l)

```
/*
 * C Language Scanner using Flex
 * Features:
 *  - Identifiers (variables, functions)
 *  - Preprocessor directives
 *  - Keywords
 *  - Constants (numeric/char/string) with table
 *  - Operators & punctuation
 *  - Ignores whitespace and comments (including nested /* */)
 *  - Reports invalid tokens with line numbers
 *  - Maintains a rudimentary Symbol Table (type inference from
    declarations)
 *  - Prints token stream + tables at program end
 *
 * Build: flex scanner.l && gcc lex.yy.c -lfl -o scanner
 */

%option noyywrap
%option yylineno
%option nounput
%option noinput

%{
#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;            /* e.g., int, float, char*, struct X
       */
    char* dimensions;      /* e.g., [10][20] */
    int   frequency;       /* number of appearances */
    char* return_type;     /* for functions */
    char* param_lists;     /* concatenated lists observed in
       function calls or declarations */
    int   is_function;     /* boolean */
    struct Symbol* next;   /* chaining in hash bucket */
} 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;  /* For #define NAME value, else "-" */
    int   line;
    char* value;     /* literal text */
    char* type;      /* "int", "float", "char", "string", "hex",
       "oct", "bin", "macro" */
} 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++;
}

/* ---------------- Declaration context tracking ----------------
    */
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';
}

/* Capture function arguments in calls/declarations */
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){
        /* remove trailing ) if present */
        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);
}

/* ---------------- Helper to 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";
}

/* Track whitespace/newline inside states manually when needed */
#define YY_USER_ACTION /* placeholder */

%}

/* -------------- 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})

%%

/* ---------- Preprocessor (line-start # with continuations)
   ---------- */
^[ \t]*\#[^\n\\]*(\\\n[^\n\\]*)*    {
    print_token("PREPROC", yytext);
    /* If it’s a #define NAME value, record in constant table */
    char* tmp = xstrdup(yytext);
    char* p = tmp;
    /* strip leading spaces and ’#’ */
    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++;
        /* NAME */
        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);
}
```

```
/* ---------- Whitespace & newlines ---------- */
{WS}       { /* ignore */ }
\n         { /* newline handled by %option yylineno */ }

/* ---------- Comments ---------- */
/* Single-line C++ style */
"//".*     { /* ignore */ }

/* Multi-line (with nesting) */
"/*"       { 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'){ /* yylineno auto increments via
                   flex */ }
                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);
           }

/* ---------- Strings & chars ---------- */
\"         { 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 */ }

/* ---------- Keywords / Types ---------- */
"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;
}
"*"  {
    print_token("OP", yytext);
    if(in_declaration) add_type_token("*");
    last_was_ident = 0;
}

/* ---------- Identifiers ---------- */
{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);
    }
}

/* ---------- Numbers ---------- */
{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; }

/* ---------- Punctuation & Operators ---------- */
/* Parentheses may indicate function declarator or call. We
   capture args on '(' after an identifier. */
"("       {
            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'); }
< FUNCARGS >.          { args_push_char(yytext[0]); }

")"       { 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;
}

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

<<EOF>>    {
             /* print tables */
             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;
}
```

Build Script

```
# Build the C language scanner with Flex
CC=gcc
LEX=flex
CFLAGS=
LIBS=-lfl

all: scanner

scanner: scanner.l
  $(LEX) scanner.l
  $(CC) lex.yy.c $(LIBS) -o scanner

clean:
  rm -f scanner lex.yy.c
```

Sample Inputs

```
#include <stdio.h>
#define MAX 100
int main(void){
    int a = 10;
    float b = 3.14;
    char c = ’x’;
    printf("Hello, world! %d %f %c\n", a, b, c);
    return 0;
}
```

```
#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);
}
```