CS F363 Compiler Construction
Second Semester, 2024-25
Lab-2: Building Lexer for a Toy Language

1 Objectives

The objectives of this lab are the following.

1. To build a lexical analyzer (lexer) for a simplified toy programming language based on the given specifications.
2. To understand and implement Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA) for recognizing various tokens such as operators, constants, identifiers, and keywords.
3. To write C programs capable of recognizing and classifying tokens according to the language rules and reporting lexical errors where applicable.
4. To integrate all components into a complete lexer that can process an entire program and generate a sequence of tokens with their types and lexemes.

2 Language Specifications

The toy language is made up of the following alphabet.

Alphabet

- **Lowercase alphabets:**

  ```
  a, b, c, d, ..., z
  ```

- **Digits** :

  ```
  0, 1, 2, ..., 9
  ```

- **Special symbols:**

  ```
  + - % / * < > = _ ( ) ; , (comma) : { }
  ```

Operators

- **Arithmetic Operators :**

```
+, -, *, /, %
```

- **Relational Operators**

```
= (equal to), >, <, >=, <=, <> (not equal to)
```

- **Assignment Operators**

```
:=, +=, -=, *=, /=, %=
```

- **Separators**

```
( ) , ; { } "
```

Constants

Constants in C represent fixed values that do not change during the execution of a program. Types of constants include the following.

- Integer Constants: Whole numbers without a decimal point, for example, `42, -17, 0`, etc.
- Floating-Point Constants: Numbers with a decimal point, for example, `3.14, -0.001, 12.4565`, etc.

Variables and Identifiers

An identifier in the toy language must begin with a lowercase letter $(a - z)$ and contain only lowercase letters, digits $(0 - 9)$, and underscores (_). However, at most one underscore (_) is allowed.

Example:

```
Valid variable names: age, count, tax_12, net_income, is_ready;

Invalid variable names: _sum, sum_curr_total, 1sum;
```

Keywords

The toy language provides the following keywords:

```
int, char, float, if, else, while, for, main
```

Keywords cannot be used as variable names.

3 DFA/NFA to recognize tokens and C implementation

3.1 Relational operator

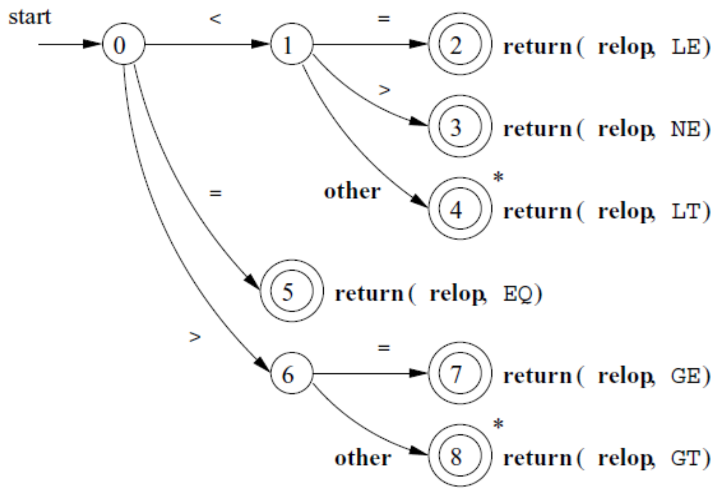


Figure 1: NFA to recognize the relational operators

A snippet of the implementation is below (you must complete the code).

```
#include <stdio.h>
#include <ctype.h>

#define YES 1
#define NO 0

// Token structure definition
typedef struct {
    char type[10];
    char value[3];
} token;

token newToken(const char *type, const char *value) {
    token t;
    snprintf(t.type, sizeof(t.type), "%s", type);
    snprintf(t.value, sizeof(t.value), "%s", value);
    return t;
}

void retract() {
    ungetc(getchar(), stdin);
}

void fail() {
    printf("Lexical error: invalid relational operator.\n");
    exit(1);
}
```

```c
token getRelop() {
    int state = 0;
    char c;

    while (YES) {
        switch (state) {
            case 0:
                c = getchar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fail();
                break;
            case 1:
                .
                .
                .
            case 2:
                return newToken("relop", "LE");


            case 4:
                retract();
                return newToken("relop", "LT");


            default:
                state = 0;
                break;
        }
    }
}
int main() {
    token result = getRelop();
    printf("Token Type: %s, Token Value: %s\n", result.type, result.value);
    return 0;
}
```

Sample input / output:

```
$ ./a.out
<=
Token Type: relop, Token Value: LE
$ ./a.out
=>
Token Type: relop, Token Value: EQ // the longest prefix matched is =
$ ./a.out
<>
Token Type: relop, Token Value: NE
$ ./a.out
_<
Lexical error: invalid relational operator. // - is not a relational operator
```

Task : 1

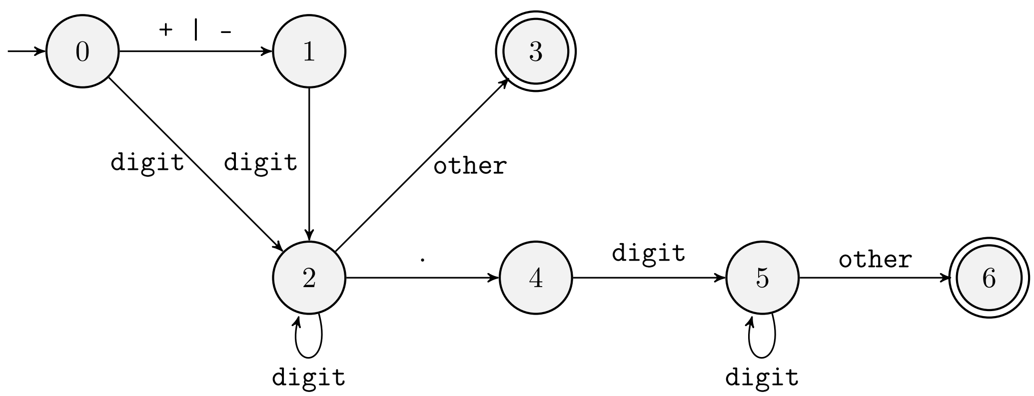
(a) Complete the above code that recognizes the relational operators at the prefix of the input string.

(b) Give a DFA-based C implementation to identify the arithmetic operators (the list is given above) at the prefix of the input string.

(c) Give a DFA-based C implementation to identify the assignment operators (the list is given above) at the prefix of the input string.

(d) Give a DFA-based C implementation to identify all three types of operators: relational operators, arithmetic operators, and assignment operators. In addition, your code should also identify the separators at the prefix of the input string.

3.2 Constants



Here, `digit` is any numeric character of the set $\{0, 1, \ldots, 9\}$ and `other` is a character other than a `digit` or a `dot (.)`. Furthermore, nodes `3` and `6` are retracted states.

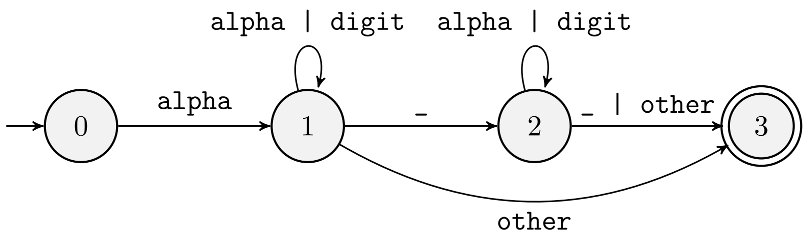
Task: 2 Complete the following code so that it recognizes signed integers and real numbers, which is the (longest) prefix of the given string.

Sample input / output:

```
$ ./a.out
12
Token Type: Integer, Token Value: IN
$ ./a.out
-11
Token Type: Integer, Token Value: IN
$./a.out
-56.78q
Token Type: Real num, Token Value: FL // the longest prefix is -58.76
$./a.out
--12
Lexical error: invalid constant. // the first - is not part of the number
```

```
token getNum() {
    int state = 0;
    char c;

    while (YES) {
        switch (state) {

            case 0:
                c = getchar();
                if (c == '+' || c == '-') state = 1;
                else if (isdigit(c)) state = 2;
                else



            case 3:
               retract();
                return newToken("Integer", "INT");

            case 4:




            case 6:
               retract();
               return newToken("Real num", "FLOAT");


            default:
                fail();
        }
    }
}
```

3.3 Identifiers



Here, `alpha` is the set of all lowercase alphabets, `digit` is the set $\{0, 1, \ldots, 9\}$. Further, `other` is the set of all characters other than lowercase letters and digits. Here, 3 is a retracted state.

Task: 3 Complete the following code so that it recognizes the identifier which is the (longest) prefix of the given input string.

```c
void fail() {
     printf("Lexical error: Not started with lower case alphabet\n");
   exit(1);
}

token getId() {
    int state = 0;
    char c;

    while (YES) {
        switch (state) {

            case 0:
                c = getchar();
                if (islower(c)) state = 1;
                else fail();
                break;

            case 1:


            case 2:


            case 3:
            retract();
            return newToken("Identifier", "ID");

            default:
                fail();

        }
    }
}
```

Sample input / output:

```
$ ./a.out
sum123
Token Type: Identifier, Token Value: ID
$ ./a.out
sum_123
Token Type: Identifier, Token Value: ID
$ ./a.out
sum12_45
Token Type: Identifier, Token Value: ID
$ ./a.out
sum_12_12
Token Type: Identifier, Token Value: ID // The valid lexeme is sum_12
$ ./a.out
_123
Lexical error: Not started with lowercase alphabet
$ ./a.out
12sum
Lexical error: Not started with lowercase alphabet
```

3.4 Key words

In this toy language, only the following words are reserved as keywords.

```
int, char, float, if, else, while, for, main
```

Task: 4 Construct a DFA that recognizes keywords from the above list and write a C implementation to identify a keyword that is the (longest) prefix of the given input string.

4 Bigger Task: Lexer for the toy language

In the preceding sections and tasks, you developed various lexers capable of recognizing operators, constants, identifiers, and keywords. Your objective is to write a C program that implements a lexer that will read a source code file named `input.txt`, and divide the program into a sequence of valid tokens.

A sample program can be seen below:

```
main( )
{
int sum, float_num;
float cgpa_sem1_1;

for(int i:=-5; i<=10; )
    sum *= i ;
if(sum <> 0)
}
```

The output of your code for the above program must be:

```
Lexeme          Token type
------------------------
main            Keyword
(               Separator
)               Separator
{               Separator
int             Keyword
sum             Identifier
,               Separator
float_num       Identifier
;               Separator
float           Keyword
cgpa_sem1       Identifier
_               Invalid operator
1               Integer
;               Separator
for             Keyword
(               Separator
int             Keyword
i               Identifier
:=               Assignment Operator
```

```
Lexeme          Token type
------------------------
-5              Integer
;               Separators
i               Identifier
<=              Relational Operator
10              Integer
;               Separator
)               Separator
sum             Identifier
*=              Assignment Operator
i               Identifier
;               Separator
if              Keyword
(               Separator
sun             Identifier
<>              Relational Operator
0               Integer
)               Separator
}               Separator
```

4.1 Adding more patterns

Modify your above lexer/scanner by adding logic to detect and process multi-line comments, single-line comments, and string literals. Ensure that unclosed comments or strings produce an error message. To handle this, you must consider that the toy language has a new constant type `String constant` (a string between " and ").

Sample program:

```
int a := 10; // This is a single-line comment
char str;
/* This is a
multi-line comment */
if (a < b) {
 str := "Value of a is less than b" // String literal
} else {
/* Unclosed multi-line comment
return 0;
}
```

The output of the above program must be:

```
Lexeme                         Token type
-------------------------------------------
int                            Keyword
a                              Identifier
:=                              Assignment operator
10                             Integer
;                              Separator
char                           Keyword
str                            Identifier
;                              Separator
if                             Keyword
(                              Separator
a                              Identifier
<                              Relational Operator
b                              Identifier
)                              Separator
{                              Separator
str                            Identifier
:=                              Assignement Operator
"Value of a sis less than b"   String constant
}                              Separator
else                           Keyword
{                              Separator
ERROR: Unclosed multi-line comment
```