Lexical Analyzer Report

Ahmed Badr 900202868

Youssef Khaled 900213467

October 18, 2024

Course Information

Course Name: CSCE4101 - Compiler Design (Fall 2024)
Instructor: Dr. Ahmed Rafea

Introduction

This report presents the development of a lexical analyzer using **Lex/Flex**. The analyzer identifies keywords, identifiers, numbers, operators, and comments while detecting and reporting specific errors. The errors include unclosed comments, invalid identifiers, and malformed numbers. This document discusses the decisions made during development, test cases used, and the program output.

Decisions Made

The following decisions were made during the implementation of the lexical analyzer:

1. **Handling Unclosed Comments:** If the input file contains an unclosed comment (i.e., ‘/ without ‘), the program detects this at EOF and reports the error with the relevant line number and position.

2. **Handling Invalid Characters:** If a character not belonging to the language’s alphabet appears, the scanner reports it as an invalid character, indicating its position in the input.

3. **Premature Token Termination:** The program detects if a token is prematurely interrupted. For example:

- In identifiers: If an invalid character appears (e.g., ‘invalid_*id‘), it reports "Invalid identifier format."
- In numbers: If an invalid letter follows an exponential notation (e.g., ‘12.3eX‘), it reports "Invalid number format."

4. **Number-Identifier Handling:** The scanner treats ‘12ab‘ as two tokens: ‘NUM‘ and ‘ID‘. Such sequences are not flagged by the lexer but will be handled by the parser.

Instructions to Run the Program

1. Save the code in a file named `scanner.l`.
2. Generate the scanner using the command:

```
flex scanner.l
```

3. Compile the generated C code:

```
gcc lex.yy.c -o scanner -lfl
```

4. Create a test input file named `input.txt` with the content shown below.
5. Run the scanner:

```
./scanner input.txt
```

Test Input

```
/* Test identifiers */
valid_id
another@id
third.id
invalid_*id
x_1
y.2
z@3

/* Test numbers */
123
45.67
```

```
2.3E-4
12.3eX
89.1e+2

/* Test keywords */
if then else
IF THEN ELSE
If Then Else

/* Test invalid cases *
abc_*
123abc
```

Program Output

```
salieri@salieri:~/Downloads/Compiler Design/scanner$ flex scanner.l
salieri@salieri:~/Downloads/Compiler Design/scanner$ gcc lex.yy.c -o scanner -ll
salieri@salieri:~/Downloads/Compiler Design/scanner$ ./scanner input.txt
TOKEN: ID (valid_id) at line 2, position 1
TOKEN: ID (another@id) at line 3, position 1
TOKEN: ID (third.id) at line 4, position 1
Error: Invalid identifier format at line 5, position 1: 'invalid_*id'
TOKEN: ID (x_1) at line 6, position 1
TOKEN: ID (y.2) at line 7, position 1
TOKEN: ID (z@3) at line 8, position 1
TOKEN: NUM (123) at line 11, position 1
TOKEN: NUM (45.67) at line 12, position 1
TOKEN: NUM (2.3E-4) at line 13, position 1
Error: Invalid number format at line 14, position 1: '12.3eX'
TOKEN: NUM (89.1e+2) at line 15, position 1
TOKEN: IF at line 18, position 1
TOKEN: THEN at line 18, position 4
TOKEN: ELSE at line 18, position 9
TOKEN: IF at line 19, position 1
TOKEN: THEN at line 19, position 4
TOKEN: ELSE at line 19, position 9
TOKEN: IF at line 20, position 1
TOKEN: THEN at line 20, position 4
TOKEN: ELSE at line 20, position 9
Error: Unclosed comment at end of file at line 25, position 1: ''
```

Appendix A: Code

Listing 1: Lexical Analyzer Code: `scanner.l`

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

/* Token definitions */
#define ID_TOKEN 1
#define NUM_TOKEN 2
#define IF_TOKEN 3
#define THEN_TOKEN 4
#define ELSE_TOKEN 5

int line_number = 1;
int char_position = 1;
int token_start_pos = 1;
void error_message(char* msg);
%}

%option noyywrap
%option caseless

/* States */
%x COMMENT

/* Regular Definitions */
LETTER          [A-Za-z]
DIGIT           [0-9]
WHITESPACE      [ \t\r]
SPECIAL         [.@_]
EXPONENT        [Ee][+-]?{DIGIT}+

ID              {LETTER}({LETTER}|{DIGIT})*({SPECIAL}({LETTER}|{
   DIGIT})+)?
INVALID_ID      {LETTER}({LETTER}|{DIGIT})*{SPECIAL}?[^A-Za-z0-9.
   @_\n \t]+({LETTER}|{DIGIT})*
NUM             {DIGIT}+(\.{DIGIT}+)?({EXPONENT})?
INVALID_NUM     {DIGIT}+(\.{DIGIT}+)?({EXPONENT})?[A-Za-z]+

```

```
%%

"/*"            {
    token_start_pos = char_position;
    char_position += yyleng;
    BEGIN(COMMENT);
}

<COMMENT>"*/"   {
    char_position += yyleng;
    BEGIN(INITIAL);
}

<COMMENT>\n     {
    line_number++;
    char_position = 1;
}

<COMMENT>.      {
    char_position += yyleng;
}

<COMMENT><<EOF>> {
    error_message("Error: Unclosed comment at end of file");
    return 0;
}

{WHITESPACE}    {
    char_position += yyleng;
}

\n              {
    line_number++;
    char_position = 1;
}

if              {
    token_start_pos = char_position;
    char_position += yyleng;
    printf("TOKEN: IF at line %d, position %d\n", line_number,
       token_start_pos);
```

```
77      return IF_TOKEN;
78  }
79
80  then            {
81      token_start_pos = char_position;
82      char_position += yyleng;
83      printf("TOKEN: THEN at line %d, position %d\n", line_number,
            token_start_pos);
84      return THEN_TOKEN;
85  }
86
87  else            {
88      token_start_pos = char_position;
89      char_position += yyleng;
90      printf("TOKEN: ELSE at line %d, position %d\n", line_number,
            token_start_pos);
91      return ELSE_TOKEN;
92  }
93
94  {ID}            {
95      token_start_pos = char_position;
96      char_position += yyleng;
97      printf("TOKEN: ID (%s) at line %d, position %d\n", yytext,
            line_number, token_start_pos);
98      return ID_TOKEN;
99  }
100
101 {INVALID_ID}    {
102     token_start_pos = char_position;
103     error_message("Error: Invalid identifier format");
104     char_position += yyleng;
105     /* Continue scanning after the invalid identifier */
106 }
107
108 {NUM}           {
109     token_start_pos = char_position;
110     char_position += yyleng;
111     printf("TOKEN: NUM (%s) at line %d, position %d\n", yytext,
            line_number, token_start_pos);
112     return NUM_TOKEN;
113 }
```

```
114
115 {INVALID_NUM}   {
116     token_start_pos = char_position;
117     error_message("Error: Invalid number format");
118     char_position += yyleng;
119     /* Continue scanning after the invalid number */
120 }
121
122 [+\-*/=<>!]     {
123     token_start_pos = char_position;
124     char_position += yyleng;
125     printf("TOKEN: OPERATOR (%s) at line %d, position %d\n",
        yytext, line_number, token_start_pos);
126     return yytext[0];
127 }
128
129 .               {
130     token_start_pos = char_position;
131     error_message("Error: Invalid character");
132     char_position += yyleng;
133     /* Consume the invalid character */
134 }
135
136 %%
137
138 void error_message(char* msg) {
139     fprintf(stderr, "%s at line %d, position %d: '%s'\n",
140             msg, line_number, token_start_pos, yytext);
141 }
142
143 int main(int argc, char** argv) {
144     if (argc < 2) {
145         printf("Usage: %s input_file\n", argv[0]);
146         return 1;
147     }
148
149     FILE* input = fopen(argv[1], "r");
150     if (!input) {
151         fprintf(stderr, "Error: Cannot open input file '%s'\n",
            argv[1]);
152         return 1;
```

```
153     }
154
155     yyin = input;
156
157     while (yylex()) {
158         // Token processing is handled in the rules above
159     }
160
161     fclose(input);
162     return 0;
163 }
```