

Lab1: 词法分析

学号	姓名
19335109	李雪堃

Lab1: 词法分析

- (一) 实验环境
- (二) 实验内容
 - (1) 数据结构
 - (2) 词法分析
- (三) 实验结果
- (四) 实验总结

(一) 实验环境

- openSUSE Tumbleweed (should also work on other linux distributions)
- gcc (SUSE Linux) 11.2.1 20220103 [revision d4a1d3c4b377f1d4acb34fe1b55b5088a3f293f6]
- GNU Make 4.3

(二) 实验内容

(1) 数据结构

数据结构的声明在 `include/token.h` 和 `include/lexer.h` 中。

Token 分为下面 6 种类型：

- `T_EOF`：'`\0`'
- `T_NUM`：数值类型，包括整数、浮点数
 - 具体类型是 `int`、`float` 还是 `double` 等，由 parser 确定
- `T_CHAR`：字符类型，即用单引号括起来的字符字面量
- `T_STRING`：字符串类型，即用双引号括起来的字符串字面量
- `T_ID`：标识符类型，包括变量名、函数名
 - 代表变量还是函数，由 parser 确定
- `T_KEYWORD`：关键字类型，C 保留的单词，如 `if`、`else` 等
- `T_DELIMITER`：分隔符类型，包括括号、运算符、分号、冒号、逗号等

```

1  typedef enum token_type_t token_type_t;
2  enum token_type_t
3  {
4      T_EOF,
5
6      T_NUM, // numeric literals
7      T_CHAR, // character literals
8      T_STR, // string literals
9
10     T_ID, // identifiers
11     T_KEYWORD, // reserved words by C standard
12     T_DELIMITER, // everything including operators, parens, brackets, braces, commas,
        colons, semicolons
13 };

```

Token 中包含该 token 所在的位置 `loc` 和它的长度 `len`，这样就可以确定一个 token 的原始字符串。另外，还包含了该 token 的类型，我打算在 token 阶段，将字面值常量由原始字符串转换为对应的值，`T_NUM` 应转换为对应的整型值或浮点数，`T_CHAR` 转换为字符，`T_STR` 转换为字符串并在末尾自动补 `'\0'`。这个过程我觉得不应该由 parser 完成。

在我的设计中，使用一个简单的单向链表作为 lexer 输出的结果，parser 接收这个 token list，做语法分析。`next` 指向下一个 token。

```

1  typedef struct token_t token_t;
2  struct token_t
3  {
4      char *loc;
5      size_t len;
6      token_type_t type;
7
8      union // if type is T_NUM, it stores the value
9      {
10         int64_t ival;
11         long double fval;
12     };
13
14     char cval; // if type is T_CHAR_LITERAL, it stores the character
15     char *sval; // if type is T_STR_LITERAL, it stores the string with null terminated
16
17     token_t *next; // tokens are stored as a single linked list
18 };

```

Lexer 中 `buf` 是读取到的源代码缓冲区，因为我目前不打算支持多个文件、超大文件的情况，所以没有保存文件路径、使用 double buffering，我计划先实现单个文件编译、链接运行时库、生成可执行代码，再考虑对多个文件的支持。指针 `p` 是 lexer 当前指向的字符，`src_size` 是源代码的大小。

```

1  typedef struct lexer_t
2  {
3      char *buf;
4      char *p;
5      size_t src_size;
6  } lexer_t;

```

(2) 词法分析

做词法分析的代码主要在 `src/lexer.c` 的 `lex()` 函数中。

在我的实现中，我没有严格根据 DFA 进行状态转换和分词，实现的是 Hand-Coded Scanner，主要有两点原因：

- Lexer 需要有错误处理，并指出错误的地方，如果严格按照 "优雅" 的 DFA 进行分词，很难处理错误情况，并给出错误信息 (在实验结果中有说明这部分)。
- 经过调查，gcc 和 clang 的前端貌似都是 hand-coded，所以我打算采取同样的策略。

下面是在 lexer 中用到的辅助函数。

`start_with()` 判断 lexer 当前指向的位置是否与 `prefix` 匹配，比如以 `//` 或者 `/*` 开头就被识别为注释的开始。

```
1 static bool start_with(char *buf, const char *prefix)
2 {
3     return strncmp(buf, prefix, strlen(prefix)) == 0 ? true : false;
4 }
```

`is_keyword()` 用于进一步判断标识符是否是关键字，这里将 token 与 keyword list 中的关键字逐一匹配。

```
1 static bool is_keyword(token_t *token)
2 {
3     // ENHANCE:
4     // * support more keywords :^)
5     static char *keyword_list[] =
6     {
7         "if", "else", "do", "while", "goto", "continue", "break", "for",
8         "switch", "case", "default", "return", "void", "struct", "enum",
9         "signed", "unsigned", "short", "int", "long", "char", "float", "double",
10        "const", "typedef", "sizeof", "typeof"
11    };
12
13    for (size_t i = 0; i < sizeof(keyword_list) / sizeof(*keyword_list); i++)
14    {
15        if (strlen(keyword_list[i]) == token->len && \
16            !memcmp(keyword_list[i], token->loc, strlen(keyword_list[i])))
17            return true;
18    }
19    return false;
20 }
```

`is_delimiter()` 用于判断 lexer 当前指向的位置是否是分隔符。我们首先判断是否与多个字符的分隔符匹配，再判断是否与单个字符的分隔符匹配，因为有些单个字符的分隔符是多个字符的前缀，先匹配多个字符的就不会出问题，注意到多个字符的分隔符中并没有谁是谁的前缀，所以我将它们放到一起。

`is_delimiter()` 会返回匹配到的分隔符的长度，用于生成 token。

```
1 static int is_delimiter(char *p)
2 {
3     // delimiters with one character
4     static char delimiters_1[] =
5     {
6         '(', ')', '[', ']', '{', '}', ':', ';', ',', '.',
```

```

7     '?', '!', '<', '>', '=',
8     '+', '-', '*', '/', '%',
9     '~', '&', '|'
10 };
11
12 // delimiters with multiple characters
13 static char *delimiters_2[] =
14 {
15     "==", "!=", "<=", ">=", "->",
16     "+=", "-=", "*=", "/=", "%=",
17     "&&", "||",
18     "++", "--",
19     "&=", "|=", "^=", "<<", ">>", "<=>", ">=>"
20 };
21
22 for (size_t i = 0; i < sizeof(delimiters_2) / sizeof(*delimiters_2); i++)
23     if (start_with(p, delimiters_2[i]))
24         return strlen(delimiters_2[i]);
25
26 for (size_t i = 0; i < sizeof(delimiters_1) / sizeof(*delimiters_1); i++)
27     if (*p == delimiters_1[i])
28         return 1;
29 return 0;
30 }

```

下面是 `lex()` 函数，其中 `NEXT_CHAR`、`NEXT_NCHAR`、`PEEK_CHAR` 是我定义的函数宏，在 `include/lexer.h` 中。`NEXT_CHAR` 会让 `lexer` 的当前指针 `p` 前进一个字符，`NEXT_NCHAR` 会让 `p` 前进 `n` 个字符，`PEEK_CHAR` 选择 `p + offset` 位置的字符。

首先跳过所有的空格、换行等空白字符，再跳过行间注释和块注释，然后再识别字符常量、字符串常量、标识符和关键字、分隔符。关键字的确定在标识符中，如果首先被识别为标识符 `token`，再调用 `is_keyword()` 判断是否是关键字。

最后，返回生成的 `token` 链表。

```

1 token_t *lex(lexer_t *lexer)
2 {
3     token_t head = {};
4     token_t *curr = &head;
5
6     while (*(lexer->p) != '\0')
7     {
8         // skip whitespaces, line feeds, carriage returns, tabs
9         if (*(lexer->p) == '\n' || *(lexer->p) == '\r' || *(lexer->p) == ' ' || *
(lexer->p) == '\t')
10         {
11             NEXT_CHAR(lexer);
12             continue;
13         }
14
15         if (start_with(lexer->p, "//")) // skip inline comment
16         {
17             NEXT_NCHAR(lexer, 2);
18             while (*(lexer->p) != '\n')
19                 NEXT_CHAR(lexer);

```

```

20     continue;
21 }
22
23 if (start_with(lexer->p, "/*")) // skip block comment
24 {
25     char *q = strstr(lexer->p + 2, "*/");
26     if (!q) // unclosed block comment
27     {
28         error_at(lexer->buf, lexer->p, "unterminated block comment");
29     }
30     lexer->p = q + 2;
31     continue;
32 }
33
34 // numeric literals
35 // ENHANCE:
36 // read and convert binary, octal and hexadecimal number
37 // currently we only read and convert decimal number
38 // FIXME:
39 // floating number format check
40 if (isdigit(CURR_CHAR(lexer)) || (CURR_CHAR(lexer) == '.' &&
    isdigit(PEEK_CHAR(lexer, 1))))
41 {
42     char *q = NEXT_CHAR(lexer);
43     loop
44     {
45         if (CURR_CHAR(lexer) && PEEK_CHAR(lexer, 1) && \
46             strchr("eE", CURR_CHAR(lexer)) && strchr("+-", PEEK_CHAR(lexer, 1)))
47             NEXT_NCHAR(lexer, 2);
48         else if (isalnum(CURR_CHAR(lexer)) || CURR_CHAR(lexer) == '.')
49             NEXT_CHAR(lexer);
50         else
51             break;
52     }
53     curr->next = make_token(q, lexer->p - 1, T_NUM);
54     curr = curr->next;
55     continue;
56 }
57
58 // character literal
59 if (CURR_CHAR(lexer) == '\\')
60 {
61     if (PEEK_CHAR(lexer, 1) == '\\0') // unclosed char literal
62     {
63         error_at(lexer->buf, lexer->p, "unterminated character literal");
64     }
65
66     // ENHANCE:
67     // escape character and utf-8 support
68     char c = PEEK_CHAR(lexer, 1);
69
70     char *q = strchr(lexer->p + 1, '\\');
71     if (!q) // unclosed char literal
72     {
73         error_at(lexer->buf, lexer->p, "unterminated character literal");
74     }

```

```

75
76     curr->next = make_token(lexer->p, q, T_CHAR);
77     curr->next->cval = c;
78     curr = curr->next;
79     NEXT_NCHAR(lexer, curr->len);
80     continue;
81 }
82
83 // string literal
84 if (CURR_CHAR(lexer) == '"')
85 {
86     if (PEEK_CHAR(lexer, 1) == '\\0') // unclosed string literal
87     {
88         error_at(lexer->buf, lexer->p, "unterminated string literal");
89     }
90
91     // ENHANCE:
92     // * escape character and utf-8 support
93     // * ignore single '\'
94
95     char *q = strchr(lexer->p + 1, '"');
96     if (!q) // unclosed string literal
97     {
98         error_at(lexer->buf, lexer->p, "unterminated string literal");
99     }
100
101     curr->next = make_token(lexer->p, q, T_STR);
102     curr = curr->next;
103     NEXT_NCHAR(lexer, curr->len);
104     continue;
105 }
106
107 // identifiers or keywords
108 // letter_ -> [A-Za-z_]
109 // numbers -> [0-9]
110 // id -> letter_ (letter_ | numbers)*
111 if (isalpha(CURR_CHAR(lexer)) || CURR_CHAR(lexer) == '_')
112 {
113     char *q = lexer->p;
114     loop
115     {
116         NEXT_CHAR(lexer);
117         if (isalnum(CURR_CHAR(lexer)) || CURR_CHAR(lexer) == '_')
118             continue;
119         else
120             break;
121     }
122     curr->next = make_token(q, lexer->p - 1, T_ID);
123     if (is_keyword(curr->next))
124         curr->next->type = T_KEYWORD;
125     curr = curr->next;
126     continue;
127 }
128
129 // delimiters
130 int delim_len = is_delimiter(lexer->p);

```

```

131     if (delim_len > 0)
132     {
133         curr->next = make_token(lexer->p, lexer->p + delim_len - 1, T_DELIMITER);
134         curr = curr->next;
135         NEXT_NCHAR(lexer, curr->len);
136         continue;
137     }
138
139     fprintf(stderr, "oh shoot, an invalid token!\n");
140     exit(1);
141 }
142
143 return head.next;
144 }

```

(三) 实验结果

Lab1:

- Github 仓库地址: <https://github.com/lixk28/kric/tree/lab1>
- 完整提交历史: <https://github.com/lixk28/kric/commits/lab1>

Lab1 PPT 中给出的 demo 在 `demo/lab1_test.c` 中, 我自己给出的测试 demo 在 `demo/demo.c`。

`tokens.txt` 是对 `demo.c` 进行词法分析生成的结果, 格式为 `token type`, 每行是一个 token 和它对应的类型。

另外, kric 目前有个我自以为不错的 feature, kric 可以在 lex 阶段, 识别出某些位置的错误, kric 会直接 exit, 这不仅对用户友好, 而且可以为用户节省宝贵的时间和电费, 为环保作出贡献。报错信息的格式模仿的是 gcc, 给出错误位置的行、列数, 错误提示信息, 以及出错的那一行代码。这部分的实现在 `include/error.h` 和 `src/error.c`。

目前支持的错误识别有:

- 块注释缺失 `*/` 配对

去掉 `demo.c` 中 26 行的 `*/`, 报错块注释未终止。



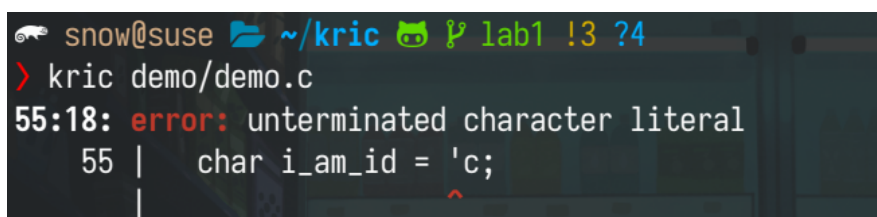
```

snow@suse ~/kric lab1 !3 ?4
> kric demo/demo.c
23:3: error: unterminated block comment
 23 |  /*
    |   ^

```

- 字符常量缺失 `'` 配对

去掉 `demo.c` 中 55 行的末尾的 `'`, 报错字符常量未终止。



```

snow@suse ~/kric lab1 !3 ?4
> kric demo/demo.c
55:18: error: unterminated character literal
 55 |  char i_am_id = 'c;
    |                  ^

```

- 字符串常量缺失 `"` 配对

去掉 `demo.c` 中 56 行的末尾 `"`, 报错字符串常量未终止。

```
snow@suse ~/kric lab1 !3 ?4
> kric demo/demo.c
56:26: error: unterminated string literal
    56 |     char *yet_another_id = "well, hello lexer :^);
       |                               ^
```

(四) 实验总结

这次实验基本完成了 lexer 的实现，但还有以下的不足和改进之处：

- 我的设计中，打算在 lex 阶段将字面量转换成对应的值并存储在 token 中，由于这一部分涉及到类型和后面的 parser 阶段 (比如字面量上 `0.4f` 是 `float`，但代码中声明为 `double`，编译器可能会 warning)，我目前的实现还有待改进，故在本地做了 commit 但没有 push 到 github 仓库中。
- 数据结构设计不完善，我写到后面发现用一个 `lexer_t` 里面存 `buf` 和 `p` 是非常愚蠢的行为，基于 KISS 的原则直接在 `lexer.c` 中声明两个 `static` 就行了，写代码的时候每次都需要写 `lexer->p` 而不是 `p`，而且 `error_at` 需要传 `lexer->buf` 这个本可以避免的参数，我觉得非常愚蠢。说明我还没有掌握面向对象的精髓，就是不要尝试用 C 模拟面向对象 :-)

另外，关于 kric 报错部分的设计我觉得十分合理，后面应当保留并延续，user-friendly 而且 eco-friendly :^)，可以节省电费。