

四川大学计算机学院学院

实 验 报 告

学号： 2017141461179 姓名：王兆基 专业： 计算机科学与技术 班级： 173040105 班

| | | | |
|------|---|------|-------------------------------|
| 课程名称 | 编译原理课程设计 | 实验课时 | 4 |
| 实验项目 | 手工构造 Tiny 语言的词法分析器 | 实验时间 | 2019 年 4 月 2 日、2019 年 4 月 9 日 |
| 实验目的 | <ol style="list-style-type: none">1. 熟悉 Tiny 语言词法的特点。2. 根据 Tiny 语言词法构造 DFA，使其可以识别 Tiny 语言 Token。3. 用 C 语言实现 Tiny 语言的词法分析器，设计数据类型、数据结构与其他代码，并且可以正常编译、运行、调试。 | | |
| 实验环境 | <ul style="list-style-type: none">➤ 设备：Microsoft Surface Pro 5 i5 8+256G➤ 操作系统：Windows 10 64 位➤ Linux 模拟环境：MSYS2 MinGW 64 (mintty) (和在 Linux 下操作是完全一样的)➤ Shell：zsh➤ 编程语言：C Language➤ 编译器：gcc.exe (Rev2, Built by MSYS2 project) 8.3.0➤ Make：GNU Make 4.2.1 (为 x86_64-pc-msys 编译)➤ 文本编辑器：VSCode 作为编辑器，文本文件采用 utf-8 编码 | | |

实验内容

一、Tiny 语言词法的特点

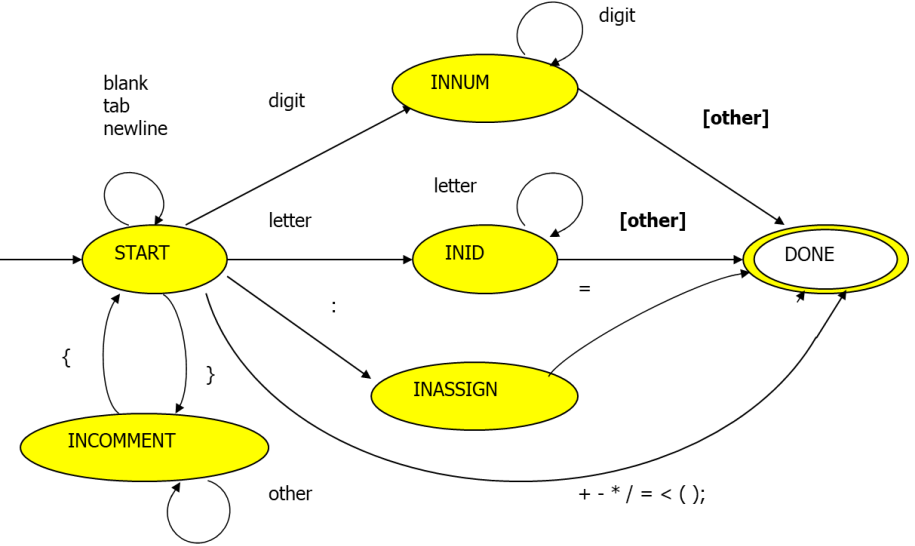
- Tiny 语言仅是一个由分号分隔开的语句序列，它既无过程也无声明。
- 所有的变量都是整型变量，通过对其赋值可声明变量。
- 只有两个控制语句：if 语句和 repeat 语句，这两个控制语句本身也可包含语句序列。
 - if 语句有一个可选的 else 部分且必须由关键字 end 结束。
 - repeat 语句需以 until 语句结束。
- read 语句和 write 语句完成输入/输出。
- TINY 的表达式局限于布尔表达式和整型算术表达式。
 - 布尔表达式由对两个算术表达式的比较组成，使用<与=比较算符。布尔表达式只作为测试出现在控制语句中。
 - 算术表达式可以包括整型常数、变量、参数以及 4 个整型算符 +、-、*、/，支持小括号改变运算优先级。
- 代码格式自由，空格、tab、换行都可以用来作为分隔符。
- 在花括号中可以有注释，但注释不能嵌套。
- 遵循最长子串原则。
- 标识符要求由字母组成，大小写敏感。
- 数支持无符号整数。

Tiny 语言的 Token 共有 22 种，如下表：

| 保留字 | 运算符 | 其他 |
|------|-----|--------------------------|
| if | + | number identifier |
| then | - | |
| else | * | |
| end | / | |

| | | |
|--------|----|--|
| repeat | = | |
| until | < | |
| read | (| |
| write |) | |
| | ; | |
| | := | |

二、识别 Tiny 语言 Token 的 DFA 设计



三、重要的数据类型、数据结构设计

1. 首先，为了合理利用计算机资源，为数组分配合适的大小，因此需要限定文件名最大长度、**token** 最大长度等，这就需要定义一些常量：

```

// 定义布尔值
#define TRUE (1)
#define FALSE (0)

// 文件名最大长度 120 个字符
#define MAX_FILENAME_LENGTH 120

// 一个 token 的长度最大为 40
#define MAX_TOKEN_LENGTH 40

// Tiny 源代码一行最多 255 个字符

```

```
#define MAX_BUFFER_LENGTH 256

// Tiny 语言一共有 8 个保留字

#define MAX_RESERVED_NUMBER 8
```

2. Tiny 语言共有 22 个 Token 类型，可以用 C 语言中的枚举数据类型实现。因此，定义一个 `enum TokenType`，并用 `typedef` 将其定义为类型 `TokenType`，方便之后将枚举 `TokenType` 作为函数返回值使用：

```
typedef enum TokenType {
    // 标注特殊状态的 Type，
    // 它们不代表实际的 Token，只是代表遇到文件尾和错误时，
    // getToken() 函数的返回值
    ENDFILE,
    ERROR,
    // Tiny 语言的 8 个保留关键字
    IF,
    THEN,
    ELSE,
    END,
    REPEAT,
    UNTIL,
    READ,
    WRITE,
    // 标识符
    ID,
    // 数字
    NUM,
    // 运算符
    ASSIGN,
    EQ,
```

```
LT,  
PLUS,  
MINUS,  
TIMES,  
OVER,  
LPAREN,  
RPAREN,  
SEMI  
} TokenType;
```

3. 如上图所示，在识别 Tiny 语言 Token 的 DFA 中，一共有六个状态，分别为 **START**、**INASSIGN**、**INCOMMENT**、**INNUM**、**INID**、**DONE**。同样可以将其定义为枚举并用 **typedef** 定义为类型方便后续调用：

```
typedef enum StateType {  
    START,  
    INASSIGN,  
    INCOMMENT,  
    INNUM,  
    INID,  
    DONE  
} StateType;
```

4. 由于保留关键字和标识符同为字符序列，在特征上没有差异，因此需要定义保留关键字表，将保留关键字的字符序列映射到它们对应的 **Token** 类型上，这样一一对应的映射关系可以采用 C 语言中的结构体实现。如下所示，在结构体类型 **ReservedWord** 中，成员 **string** 存储了保留关键字的字符序列，而成员 **token** 则存储了保留关键字对应的 **Token** 类型。

```
typedef struct ReservedWord {
```

```
char* string;

TokenType token;

} ReservedWord;
```

然后，就可以声明一个长度为 8 的 ReservedWord[] 数组，存放 Tiny 语言的 8 个保留关键字映射方式所对应的结构体：

```
ReservedWord reserved_words[MAX_RESERVED_NUMBER] = {

    {"if", IF},

    {"then", THEN},

    {"else", ELSE},

    {"end", END},

    {"repeat", REPEAT},

    {"until", UNTIL},

    {"read", READ},

    {"write", WRITE}

};
```

四、实现的关键代码设计

1. 全局变量设计

首先，按照要求，程序需要从 tny 文本文件中读取 Tiny 语言源代码，将其进行词法分析后，将分析的结果保存到[源文件名].txt 文件中。所以，首先需要两个保存文件名的 char[] 类型变量和两个控制文件 I/O 的 FILE 类型变量：

```
char source_file_name[MAX_FILENAME_LENGTH + 1];

char result_file_name[MAX_FILENAME_LENGTH + 1];

FILE* source_file;

FILE* result_file;
```

其次，在进行词法扫描时，程序是逐行读取源代码的，且每行代码不得超过 255 个字符。这就需要创建一个缓存区每次按行缓存源代码内容。同

时，也需要记录行号，方便后续输出。因此，需要声明下面五个变量：

```
// 当前行号
int line_number = 0;
// 缓存当前行的内容
char line_buffer[MAX_BUFFER_LENGTH];
// 当前行读取到的下标
int line_buffer_index = 0;
// 当前行的实际字符数
int line_buffer_size = 0;
// 确保当前行还没有结束，为了预防 ungetNextChar() 在遇到 EOF
时出错
int is_EOF = FALSE;
```

2. 命令行调用与输入输出设计

在 C 语言程序中，main 函数的定位常为：

```
int main(int argc, char* argv[]);
```

其中，argc 表示参数的数目，argv 是传入的所有参数的值的集合。在执行程序时，参数至少会有一个，即当前程序的文件名（argv[0]）。考虑到该程序只能接收一个 tny 源代码文件名作为参数，需要在 main 函数的开始做如下判断，当参数格式不正确时，拒绝执行并且提示用户正确的使用格式：

```
if (argc != 2) {
    printf("Usage: %s <filename>\n", argv[0]);
    exit(1);
}
```

并且，在人们的使用习惯中，输入文件名时往往会省略掉扩展名。这也需要在程序中进行检测，如果输入的文件名中不包含扩展名，则为其补上：

```
strcpy(source_file_name, argv[1]);
if (strchr(source_file_name, '.') == NULL) {
```

```
strcat(source_file_name, ".tny");  
}
```

然后就可以用 `fopen()` 函数读取文件了，需要注意的是，用户指定的文件不一定存在。如果文件无法读取，同样需要显式报错并以非零状态码退出程序：

```
source_file = fopen(source_file_name, "r");  
if (source_file == NULL) {  
    printf("File %s not found\n", source_file_name);  
    exit(1);  
}
```

接下来需要为后续输出做准备，用 `fprintf` 输出词法扫描的结果即可。根据要求，需要结合源文件的名称，将分析的结果保存到 [源文件名].txt 文件中，所以需要先把 `source_file_name` 的内容复制到 `result_file_name` 中，再将后半部分替换为「.txt」。然后用 `fopen` 以写模式打开文件，如果无法写入的话也要显式报错并以非零状态码退出程序：

```
strcpy(result_file_name, source_file_name);  
*strstr(result_file_name, ".tny") = '\0';  
strcat(result_file_name, ".txt");  
result_file = fopen(result_file_name, "w");  
if (result_file == NULL) {  
    printf("File %s cannot be written\n",  
source_file_name);  
    exit(1);  
}
```

最后，按照格式要求，往文件里输入「TINY COMPILATION:」作为开头，再循环运行 `getToken()` 函数，不断截取 Token 并输出即可。在源代码文件全部处理完后，再在 Shell 里显示提示，告知用户词法分析的结果保存在了 [源文件名].txt 文件中。


```

fprintf(result_file, "TINY COMPILATION:\n");

while (getToken() != ENDFILE) {
}

printf("The lexical analysis result is saved to
file %s\n", result_file_name);

```

3. getToken()函数设计

这是整个词法扫描程序最重要的一个函数，因为实际上做完前面的准备工作后，后面的时间程序都是在不断的运行 `getToken()` 函数。因为我们是逐行读取 `tny` 源文件内容的，所以首先要初始化一些变量，如下所示：

```

// 存放当前 token 的字符串
// 注意这里必须初始化为空串，否则内存中的垃圾可能会随机填充
到这里，影响后面的运行

char token_string[MAX_TOKEN_LENGTH + 1] = "";
// 标记在 token_string 中存到哪个下标了
int token_string_index = 0;
// 作为返回值的 token
TokenType current_token;

```

然后，就是一个经典的双层 `case` 处理关键字了。根据 Tiny 语言的 DFA 图，设计处理代码如下：

```

// 当前处于 DFA 中的哪个状态，一共有 START, INASSIGN,
INCOMMENT, INNUM,
// INID, DONE 六种
StateType state = START;

// 跑 DFA，只要没到终态就一直跑
while (state != DONE) {

```



```

        current_token = LT;
        break;
    case '+':
        current_token = PLUS;
        break;
    case '-':
        current_token = MINUS;
        break;
    case '*':
        current_token = TIMES;
        break;
    case '/':
        current_token = OVER;
        break;
    case '(':
        current_token = LPAREN;
        break;
    case ')':
        current_token = RPAREN;
        break;
    case ';':
        current_token = SEMI;
        break;
    default:
        current_token = ERROR;
        break;
    }
}

break;

case INCOMMENT:

```

```
can_be_saved = FALSE;
if (c == EOF) {
    state = DONE;
    current_token = ENDFILE;
} else if (c == '}') {
    state = START;
}
break;
case INASSIGN:
    state = DONE;
    if (c == '=') {
        current_token = ASSIGN;
    } else {
        ungetNextChar();
        can_be_saved = FALSE;
        current_token = ERROR;
    }
    break;
case INNUM:
    if (!isdigit(c)) {
        ungetNextChar();
        can_be_saved = FALSE;
        state = DONE;
        current_token = NUM;
    }
    break;
case INID:
    if (!isalpha(c)) {
        ungetNextChar();
        can_be_saved = FALSE;
```

```

        state = DONE;
        current_token = ID;
    }
    break;
case DONE:
default:
    // 如果已经进行到了 DONE 状态，那应该已经跳出了，不可能进入到 switch 中
    // 所有的状态在上面都涉及到了，不可能出现 default 响应的情况
    // 如果出现了上述两种情况，那一定是出现了不可思议的问题，直接跳出就好了
    fprintf(result_file, "Scanner Bug: state= %d\n", state);
    state = DONE;
    current_token = ERROR;
    break;
}

if (can_be_saved && token_string_index <= MAX_TOKEN_LENGTH) {
    // 如果当前处理的字符是多字符 token 的一部分，而且 token 没有过长的话，就保存下来
    token_string[token_string_index++] = (char)c;
}

if (state == DONE) {
    token_string[token_string_index] == '\0';
    if (current_token == ID) {

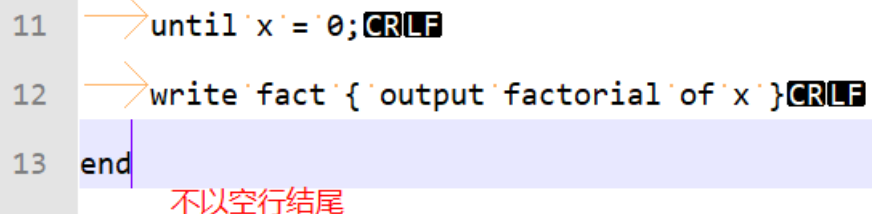
```

```
        current_token =  
        distinguishReservedWordAndIdentifier(token_string);  
    }  
}  
}
```

在双层 `case` 处理的过程中，我创建了一个 `can_be_saved` 变量作为 `flag`，还调用了编写的 `getNextChar()` 函数、`ungetNextChar()` 函数和 `distinguishReservedWordAndIdentifier()` 函数，接下来对这一个变量与三个函数逐一说明。

`can_be_saved` 变量，顾名思义，是判断当前的字符需不需要被保存到 `token_string` 内。因为有一些元素，比如标识符，关键字还有数字，是可能由多个字符构成的。如果遇到这种情况，就需要在下一次循环中继续往 `token_string` 填充字符，直到结束这个状态（数字或标识符）。

`getNextChar()` 函数，顾名思义，是取出下一个字符判断的。思路是每次取一行源码，然后挨个读取字符，如果这一行读取完了，再读下一行，并且输出这一行的源码和行号。在这里我发现了原版 `tiny` 编译器的一个 `bug`，就是如果 `tny` 文件的末尾不是空行，那么输出的行号格式就会错乱，如下图：



```
11  → until x = 0;CRLF  
12  → write fact { output factorial of x }CRLF  
13  end  
    不以空行结尾
```

```

10: :=
10: ID, name= x
10: -
10: NUM, val= 1 (line_buffer);
10: ;
11: until x = 0; // 如果最后一行字符是没有 \n 的，需要补一个 \n
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved word: write
12: ID, name= fact
13: end      14: reserved word: end
15: EOF

```

明明只有13行，这里却错乱了，还出现了14行和15行的行号

D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\TINY 编译器源码

我觉得这是这个程序鲁棒性不强的地方，怎么能保证源码文件一定以空行结尾呢？因此，我引入了 `currentLineHasLF()` 方法，它只有一行代码：

```

int currentLineHasLF(void) { return
line_buffer[line_buffer_size - 1] == '\n'; }

```

但是却可以检测到行尾是不是以回车结束的。如果行尾不是以回车结束，这样的情况只有一种，就是它不仅是行尾，也是文件尾。这时候就需要做特殊处理。我写的 `getNextChar()` 代码如下，处理的地方已经用注释作了说明：

```

int getNextChar(void) {
    if (line_buffer_index < line_buffer_size) {
        // 如果当前读取的内容在缓存区边界内，就直接返回下标对应的
        // 字符，并将下标加一
        return line_buffer[line_buffer_index++];
    } else {
        // 如果当前读取的内容不在当前行缓存区边界内，说明这一行已
        // 经读取完了

        // C 库函数 char *fgets(char *str, int n, FILE
        *stream) 从指定的流 stream 读取一行，并把它存储在 str 所指向
    }
}

```

的字符串内。当读取 (n-1) 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。

// 这里读取的是 (MAX_BUFFER_LENGTH - 1) 个字符，是为了给 \0 腾出一个字节

```
if (fgets(line_buffer, MAX_BUFFER_LENGTH - 1,
source_file)) {
```

// 如果 source_file 中的内容还有，打印这一行的内容，附带行号

// 行号加一

```
line_number++;
```

// 记录这一行的实际字符数

```
line_buffer_size = strlen(line_buffer);
```

// 如果 EOF 没有独立成行的话，最后一行字符是没有 \n 的，需要补一个 \n 让其输出换行

```
if (!currentLineHasLF()) {
```

```
    line_buffer[line_buffer_size++] = '\n';
```

```
    line_buffer[line_buffer_size] = '\0';
```

```
    fprintf(result_file, "%4d: %s", line_number,
line_buffer);
```

```
    line_buffer[--line_buffer_size] = '\0';
```

```
} else {
```

```
    fprintf(result_file, "%4d: %s", line_number,
line_buffer);
```

```
}
```

// 将当前行读取到的下标重置为 0

```
line_buffer_index = 0;
```

// 返回当前行的第一个字符，并将下标加一

```
return line_buffer[line_buffer_index++];
```

```
} else {
```



```

// 如果 source_file 中的内容已经全部读取完了，就返回
EOF

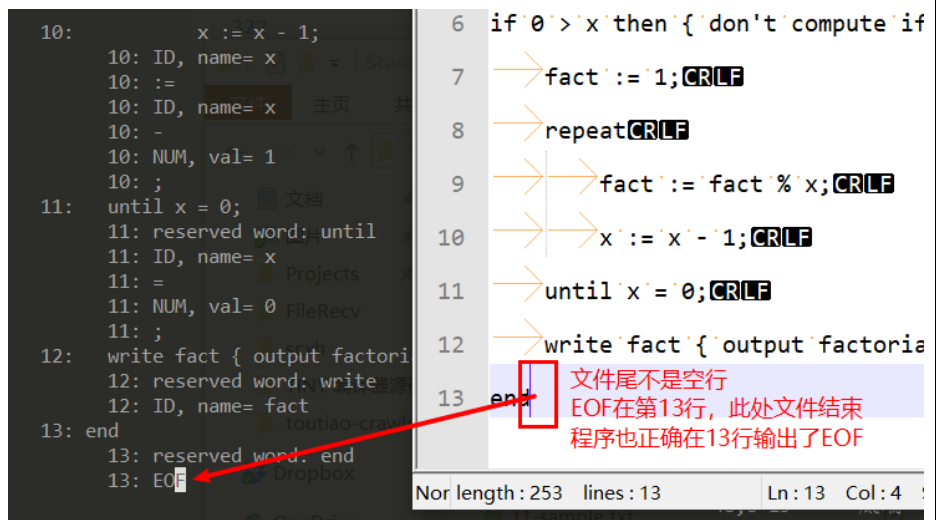
is_EOF = TRUE;

if (currentLineHasLF()) {
    line_number++;
}

return EOF;
}
}
}

```

而我的 getNextChar()函数即使遇到文件尾没有空行的情况，也能正常处理，鲁棒性更强，如下图所示：



```

10:      x := x - 1;
10: ID, name= x
10: :=
10: ID, name= x
10: -
10: NUM, val= 1
10: ;
11: until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: Projects
11: NUM, val= 0
11: FileRecv
11: ;
12: write fact { output factori
12: reserved word: write
12: ID, name= fact
13: end
13: reserved word: end
13: EOF

```

```

6 if '0' > x then { 'don't compute if
7   fact := 1; CRLF
8   repeat CRLF
9     fact := fact * x; CRLF
10    x := x - 1; CRLF
11    until x = 0; CRLF
12    write fact { output factoria
13 end

```

文件尾不是空行
EOF在第13行，此处文件结束
程序也正确在13行输出了EOF

Nor length: 253 lines: 13 Ln: 13 Col: 4

ungetNextChar()函数，顾名思义，是处理状态转换图中「回退一个字符」那一步的操作，将刚刚读取到的下一个字符，退回缓存区中。当然，如果「下一个字符」就是文件尾的话，那也就没有回退的必要了。代码如下：

```

void ungetNextChar(void) {
    // 如果下一个字符就是文件尾，那就没有回退的必要了。

    if (!is_EOF) {
        line_buffer_index--;
    }
}

```

```
}  
}
```

distinguishReservedWordAndIdentifier()函数, 顾名思义, 作用是判断一个标识符是不是关键字。方法是查询关键字表, 看当前识别出的这个标识符是否对应。因为关键字很少, 只有 8 个, 所以只需要线性查找一遍关键字表即可。代码如下:

```
TokenType distinguishReservedWordAndIdentifier(char*  
string) {  
    for (int i = 0; i < MAX_RESERVED_NUMBER; i++) {  
        if (!strcmp(string, reserved_words[i].string)) {  
            return reserved_words[i].token;  
        }  
    }  
    return ID;  
}
```

关键字表的定义在前文已经给出, 不再赘述。

在双层 case 循环结束之后, 整个源码文件就已经全部被遍历完了。而这时, 原版的编译器又出现了一个问题, 明明 EOF 是在下一行了, 行号也输出的是下一行的行号。但是, 缩进居然和上一行的关键字是同一个缩进, 这就很尴尬了, 成了四不像, 如下图所示:

```

10: ID, name= x
10: -
10: NUM, val= 1
10: ;
11: until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved word: write
12: ID, name= fact
12: ;
13: end
13: reserved word: end
14: EOF

```

14行的缩进不正确，应该和前面的13行源码的输出对齐

D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\TINY 编译器源码

λ

因此我在这里又做了一个检测。如果文件尾无空行时，那么 EOF 作为上一行的一份子输出（前文有截图）；如果文件尾有空行时，那么 EOF 独立成行输出，缩进和其他行的源代码输出对齐，代码如下：

```

// 当 EOF 独立成行时，这里需要用行号的方式输出
if (current_token == ENDFILE && currentLineHasLF()) {
    fprintf(result_file, "%4d: ", line_number);
} else {
    fprintf(result_file, "\t%d: ", line_number);
}

```

做了这个调整后，输出就正确了：

```

10:      x := x - 1;
10: ID, name= x
10: :=
10: ID, name= x
10: -
10: NUM, val= 1
10: ;
11: until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved word: write
12: ID, name= fact
12: ;
13: end
13: reserved word: end
14: EOF

```

输出正确，对齐了

最后，需要按照规定的格式将 `token` 打印出来。为此编写了一个方法，`printfToken()`，只是简单地用 `switch` 判断了一下 `token` 类型，然后输出就好了：

```
void printfToken(TokenType token, char* string) {
    switch (token) {
        case IF:
        case THEN:
        case ELSE:
        case END:
        case REPEAT:
        case UNTIL:
        case READ:
        case WRITE:
            fprintf(result_file, "reserved word: %s\n",
string);
            break;
        case ASSIGN:
            fprintf(result_file, ":=\n");
            break;
        case LT:
            fprintf(result_file, "<\n");
            break;
        case EQ:
            fprintf(result_file, "=\n");
            break;
        case LPAREN:
            fprintf(result_file, "(\n");
            break;
        case RPAREN:
```

```
        fprintf(result_file, ")\n");
        break;
    case SEMI:
        fprintf(result_file, ";\n");
        break;
    case PLUS:
        fprintf(result_file, "+\n");
        break;
    case MINUS:
        fprintf(result_file, "-\n");
        break;
    case TIMES:
        fprintf(result_file, "*\n");
        break;
    case OVER:
        fprintf(result_file, "/\n");
        break;
    case ENDFILE:
        fprintf(result_file, "EOF\n");
        break;
    case NUM:
        fprintf(result_file, "NUM, val= %s\n", string);
        break;
    case ID:
        fprintf(result_file, "ID, name= %s\n", string);
        break;
    case ERROR:
        fprintf(result_file, "ERROR: %s\n", string);
        break;
    default:
```

```

        // 永不发生

        fprintf(result_file, "Unknown token: %d\n", token);
    }
}

```

至此，我的关键代码全部都编写完了。

五、附录-源代码

types.h

```

#ifndef __TYPES_H__
#define __TYPES_H__

// 定义布尔值
#define TRUE (1)
#define FALSE (0)

// 文件名最大长度 120 个字符
#define MAX_FILENAME_LENGTH 120

// 一个 token 的长度最大为 40
#define MAX_TOKEN_LENGTH 40

// Tiny 源代码一行最多 255 个字符
#define MAX_BUFFER_LENGTH 256

// Tiny 语言一共有 8 个保留字
#define MAX_RESERVED_NUMBER 8

// 定义 Token 的类型
typedef enum TokenType {
    // 标注特殊状态的 Type,

```

```
// 它们不代表实际的 Token，只是代表遇到文件尾和错误时，  
getToken() 函数的返回值
```

```
ENDFILE,
```

```
ERROR,
```

```
// Tiny 语言的 8 个保留关键字
```

```
IF,
```

```
THEN,
```

```
ELSE,
```

```
END,
```

```
REPEAT,
```

```
UNTIL,
```

```
READ,
```

```
WRITE,
```

```
// 标识符
```

```
ID,
```

```
// 数字
```

```
NUM,
```

```
// 运算符
```

```
ASSIGN,
```

```
EQ,
```

```
LT,
```

```
PLUS,
```

```
MINUS,
```

```
TIMES,
```

```
OVER,
```

```
LPAREN,
```

```
RPAREN,
```

```
SEMI
```

```
} TokenType;
```

```
// 定义词法扫描 DFA 的状态
```

```
typedef enum StateType {
```

```
    START,
```

```
    INASSIGN,
```

```
    INCOMMENT,
```

```
    INNUM,
```

```
    INID,
```

```
    DONE
```

```
} StateType;
```

```
typedef struct ReservedWord {
```

```
    char* string;
```

```
    TokenType token;
```

```
} ReservedWord;
```

```
// 8 个保留字对应的 Token 类型
```

```
ReservedWord reserved_words[MAX_RESERVED_NUMBER] = {
```

```
    {"if", IF},          {"then", THEN},    {"else", ELSE},
```

```
    {"end", END},
```

```
    {"repeat", REPEAT}, {"until", UNTIL}, {"read", READ},
```

```
    {"write", WRITE}};
```

```
#endif
```

```
main.h
```

```
#ifndef __MAIN_H__
```

```
#define __MAIN_H__
```

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

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



```
#include <stdlib.h>
#include <string.h>

#include "types.h"

// 主程序的变量
extern char source_file_name[MAX_FILENAME_LENGTH + 1];
extern char result_file_name[MAX_FILENAME_LENGTH + 1];
extern FILE* source_file;
extern FILE* result_file;

// 词法扫描的变量

// 当前行号
extern int line_number;
// 缓存当前行的内容
extern char line_buffer[MAX_BUFFER_LENGTH];
// 当前行读取到的下标
extern int line_buffer_index;
// 当前行的实际字符数
extern int line_buffer_size;
// 确保当前行还没有结束，为了预防 ungetNextChar() 在遇到 EOF
// 时出错
extern int is_EOF;

int main(int argc, char* argv[]);
// 不断的读取 source 中的字符，凑成一个 Token 就返回
TokenType getToken(void);
// 取出下一个字符
int getNextChar(void);
```

```

// 将下一个字符退回缓存区
void ungetNextChar(void);
// 区分「保留字」和「普通的标识符」
TokenType distinguishReservedWordAndIdentifier(char*
string);
// 按照特定格式打印当前 Token
void printToken(TokenType, char* string);
// 检测当前行是否以\n结尾
int currentLineHasLF(void);

#endif

```

main.c

```

#include "main.h"

// 主程序的变量
char source_file_name[MAX_FILENAME_LENGTH + 1];
char result_file_name[MAX_FILENAME_LENGTH + 1];
FILE* source_file;
FILE* result_file;

// 词法扫描的变量

// 当前行号
int line_number = 0;
// 缓存当前行的内容
char line_buffer[MAX_BUFFER_LENGTH];
// 当前行读取到的下标
int line_buffer_index = 0;

```

```

// 当前行的实际字符数
int line_buffer_size = 0;
// 确保当前行还没有结束，为了预防 ungetNextChar() 在遇到 EOF
时出错
int is_EOF = FALSE;

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        exit(1);
    }

    strcpy(source_file_name, argv[1]);
    if (strchr(source_file_name, '.') == NULL) {
        strcat(source_file_name, ".tny");
    }

    source_file = fopen(source_file_name, "r");
    if (source_file == NULL) {
        printf("File %s not found\n", source_file_name);
        exit(1);
    }

    strcpy(result_file_name, source_file_name);
    *strstr(result_file_name, ".tny") = '\0';
    strcat(result_file_name, ".txt");
    result_file = fopen(result_file_name, "w");
    if (result_file == NULL) {
        printf("File %s cannot be written\n",
source_file_name);
    }
}

```

```

        exit(1);
    }

    fprintf(result_file, "TINY COMPILATION:\n");

    while (getToken() != ENDFILE) {
    }

    printf("The lexical analysis result is saved to
file %s\n", result_file_name);

    return 0;
}

TokenType getToken(void) {
    // 存放当前 token 的字符串
    // 注意这里必须初始化为空串，否则内存中的垃圾可能会随机填充
    到这里，影响后面的运行
    char token_string[MAX_TOKEN_LENGTH + 1] = "";
    // 标记在 token_string 中存到哪个下标了
    int token_string_index = 0;
    // 作为返回值的 token
    TokenType current_token;
    // 当前处于 DFA 中的哪个状态，一共有 START, INASSIGN,
    INCOMMENT, INNUM,
    // INID, DONE 六种
    StateType state = START;

    // 跑 DFA，只要没到终态就一直跑
    while (state != DONE) {

```



```

        current_token = LT;
        break;
    case '+':
        current_token = PLUS;
        break;
    case '-':
        current_token = MINUS;
        break;
    case '*':
        current_token = TIMES;
        break;
    case '/':
        current_token = OVER;
        break;
    case '(':
        current_token = LPAREN;
        break;
    case ')':
        current_token = RPAREN;
        break;
    case ';':
        current_token = SEMI;
        break;
    default:
        current_token = ERROR;
        break;
    }
}

break;

case INCOMMENT:

```

```
can_be_saved = FALSE;
if (c == EOF) {
    state = DONE;
    current_token = ENDFILE;
} else if (c == '}') {
    state = START;
}
break;
case INASSIGN:
    state = DONE;
    if (c == '=') {
        current_token = ASSIGN;
    } else {
        ungetNextChar();
        can_be_saved = FALSE;
        current_token = ERROR;
    }
    break;
case INNUM:
    if (!isdigit(c)) {
        ungetNextChar();
        can_be_saved = FALSE;
        state = DONE;
        current_token = NUM;
    }
    break;
case INID:
    if (!isalpha(c)) {
        ungetNextChar();
        can_be_saved = FALSE;
```

```

        state = DONE;
        current_token = ID;
    }
    break;
case DONE:
default:
    // 如果已经进行到了 DONE 状态，那应该已经跳出了，不可能进入到 switch 中
    // 所有的状态在上面都涉及到了，不可能出现 default 响应的情况
    // 如果出现了上述两种情况，那一定是出现了不可思议的问题，直接跳出就好了
    fprintf(result_file, "Scanner Bug: state= %d\n", state);
    state = DONE;
    current_token = ERROR;
    break;
}

if (can_be_saved && token_string_index <= MAX_TOKEN_LENGTH) {
    // 如果当前处理的字符是多字符 token 的一部分，而且 token 没有过长的话，就保存下来
    token_string[token_string_index++] = (char)c;
}

if (state == DONE) {
    token_string[token_string_index] == '\0';
    if (current_token == ID) {

```



```

        current_token =
distinguishReservedWordAndIdentifier(token_string);
    }
}

// 当 EOF 独立成行时，这里需要用行号的方式输出
if (current_token == ENDFILE && currentLineHasLF()) {
    fprintf(result_file, "%4d: ", line_number);
} else {
    fprintf(result_file, "\t%d: ", line_number);
}

printToken(current_token, token_string);

return current_token;
}

int getNextChar(void) {
    if (line_buffer_index < line_buffer_size) {
        // 如果当前读取的内容在缓存区边界内，就直接返回下标对应的
        // 字符，并将下标加一
        return line_buffer[line_buffer_index++];
    } else {
        // 如果当前读取的内容不在当前行缓存区边界内，说明这一行已
        // 经读取完了

        // C 库函数 char *fgets(char *str, int n, FILE
        *stream) 从指定的流 stream

```

```
// 读取一行，并把它存储在 str 所指向的字符串内。当读取 (n-1)
// 个字符时，或者读取到换行符时，或者到达文件末尾时，它会
// 停止，具体视情况而定。
// 这里读取的是 (MAX_BUFFER_LENGTH - 1) 个字符，是为了给
// \0 腾出一个字节
if (fgets(line_buffer, MAX_BUFFER_LENGTH - 1,
source_file)) {
    // 如果 source_file 中的内容还有，
    // 打印这一行的内容，附带行号
    // 行号加一
    line_number++;
    // 记录这一行的实际字符数
    line_buffer_size = strlen(line_buffer);
    // 如果 EOF 没有独立成行的话，最后一行字符是没有 \n 的，
    // 需要补一个 \n
    // 让其输出换行
    if (!currentLineHasLF()) {
        line_buffer[line_buffer_size++] = '\n';
        line_buffer[line_buffer_size] = '\0';
        fprintf(result_file, "%4d: %s", line_number,
line_buffer);
        line_buffer[--line_buffer_size] = '\0';
    } else {
        fprintf(result_file, "%4d: %s", line_number,
line_buffer);
    }
    // 将当前行读取到的下标重置为 0
    line_buffer_index = 0;
    // 返回当前行的第一个字符，并将下标加一
```

```

        return line_buffer[line_buffer_index++];
    } else {
        // 如果 source_file 中的内容已经全部读取完了，就返回
EOF
        is_EOF = TRUE;
        if (currentLineHasLF()) {
            line_number++;
        }
        return EOF;
    }
}

void ungetNextChar(void) {
    // 如果下一个字符就是文件尾，那就没有回退的必要了。
    if (!is_EOF) {
        line_buffer_index--;
    }
}

TokenType distinguishReservedWordAndIdentifier(char*
string) {
    for (int i = 0; i < MAX_RESERVED_NUMBER; i++) {
        if (!strcmp(string, reserved_words[i].string)) {
            return reserved_words[i].token;
        }
    }
    return ID;
}

```

```
void printToken(TokenType token, char* string) {
    switch (token) {
        case IF:
        case THEN:
        case ELSE:
        case END:
        case REPEAT:
        case UNTIL:
        case READ:
        case WRITE:
            fprintf(result_file, "reserved word: %s\n",
string);
            break;
        case ASSIGN:
            fprintf(result_file, ":=\n");
            break;
        case LT:
            fprintf(result_file, "<\n");
            break;
        case EQ:
            fprintf(result_file, "=\n");
            break;
        case LPAREN:
            fprintf(result_file, "(\n");
            break;
        case RPAREN:
            fprintf(result_file, ")\n");
            break;
        case SEMI:
            fprintf(result_file, ";\n");
```

```
        break;
    case PLUS:
        fprintf(result_file, "+\n");
        break;
    case MINUS:
        fprintf(result_file, "-\n");
        break;
    case TIMES:
        fprintf(result_file, "*\n");
        break;
    case OVER:
        fprintf(result_file, "/\n");
        break;
    case ENDFILE:
        fprintf(result_file, "EOF\n");
        break;
    case NUM:
        fprintf(result_file, "NUM, val= %s\n", string);
        break;
    case ID:
        fprintf(result_file, "ID, name= %s\n", string);
        break;
    case ERROR:
        fprintf(result_file, "ERROR: %s\n", string);
        break;
    default:
        // 永不发生
        fprintf(result_file, "Unknown token: %d\n", token);
}
}
```

| | |
|------|---|
| | <pre>int currentLineHasLF(void) { return line_buffer[line_buffer_size - 1] == '\n'; }</pre> <p>makefile</p> <pre>objects = main.o tiny: \$(objects) gcc \$(objects) -o tiny main.o: main.c main.h types.h gcc -c main.c -std=c11 .PHONY: clean clean: -rm tiny.exe -rm tiny_debug.exe -rm tiny.out -rm tiny_debug.out -rm main.o -rm main.obj all: tiny</pre> |
| 实验结果 | <p>首先，运行 <code>make clean</code> 做一点清理工作，把旧的中间文件删除（已经考虑到了 <code>.exe</code> 和 <code>.out</code>，以及 <code>.obj</code> 和 <code>.o</code> 文件可能在不同系统下生成的差异）：</p> |

问题 19 输出 调试控制台 终端

1: zsh

```
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [20:27:37]
$ make clean
rm tiny.exe
rm tiny.debug.exe
rm: 无法删除'tiny.debug.exe': No such file or directory
make: [makefile:12: clean] 错误 1 (已忽略)
rm tiny.out
rm: 无法删除'tiny.out': No such file or directory
make: [makefile:13: clean] 错误 1 (已忽略)
rm tiny.debug.out
rm: 无法删除'tiny.debug.out': No such file or directory
make: [makefile:14: clean] 错误 1 (已忽略)
rm main.o
rm main.obj
rm: 无法删除'main.obj': No such file or directory
make: [makefile:16: clean] 错误 1 (已忽略)
```

(Global Scope) 行 322, 列 1 (已选择8381) 空格: 2 UTF-8 CRLF C Win32

然后, 运行 `make` 命令编译:

问题 19 输出 调试控制台 终端

1: zsh

```
rm: 无法删除'tiny.out': No such file or directory
make: [makefile:13: clean] 错误 1 (已忽略)
rm tiny.debug.out
rm: 无法删除'tiny.debug.out': No such file or directory
make: [makefile:14: clean] 错误 1 (已忽略)
rm main.o
rm main.obj
rm: 无法删除'main.obj': No such file or directory
make: [makefile:16: clean] 错误 1 (已忽略)
```

```
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:09:20]
$ make
gcc -c main.c -std=c11
gcc main.o -o tiny
```

```
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:10:18]
$
```

(Global Scope) 行 322, 列 1 (已选择8381) 空格: 2 UTF-8 CRLF C Win32

编译成功后, 运行 `./tiny` (bash/zsh/PowerShell) 或者 `tiny` (Windows 的 `cmd`):

```
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:10:18]
$ ./tiny
Usage: D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\Scanner\tiny.exe <filename>
```

```
C:\WINDOWS\system32\cmd.exe

D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\Scanner>tiny
Usage: tiny <filename>

D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\Scanner>!
```

如图，如果终端是 `bash/zsh/PowerShell`，那么是上面第一张图显示的提示。如果是 `cmd`，那么是上面第二张图的提示。即提示了用户「正确使用方法」是「`tiny 文件 <filename>`」的格式调用：

然后输入 `./tiny t1.tny` 调用程序，如果终端不是 `bash/zsh/PowerShell` 而是 `cmd`，调用命令同上，不再赘述：

```
问题 19 输出 调试控制台 终端 1: zsh

er/Week 6/Scanner [21:09:20]
$ make
gcc -c main.c -std=c11
gcc main.o -o tiny

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:10:18]
$ ./tiny
Usage: D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\Scanner\tiny.exe <filename>

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:13:06] C:1
$ ./tiny t1.tny
The lexical analysis result is saved to file t1.txt

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:16:22]
$
```

如图，程序会提示把词法扫描的结果输出到了 `t1.txt` 文件当中。

打开 `t1.txt` 文件：


```
D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\Scanner\t1.txt - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
t1.txt
1 TINY COMPILATION:
2 1: { Sample program
3 2: in TINY language
4 3: computes factorial
5 4: }
6 5: read x; { input an integer }
7 5: reserved word: read
8 5: ID, name= x
9 5: ;
10 6: if 0 > x then { don't compute if x <= 0 }
11 6: reserved word: if
12 6: NUM, val= 0
13 6: ERROR: >
14 6: ID, name= x
Nor length: 904 lines: 49 Ln: 1 Col: 1 Sel: 0 | 0 Windows (CR LF) UTF-8 INS
```

和预期的输出格式一致，不过到底和样例有没有区别还是要比较一下。

我将样例命名为 `t1-sample.txt` 放在了同一目录下：

```
问题 19 输出 调试控制台 终端 1: zsh
er/Week 6/Scanner [21:10:18]
$ ./tiny
Usage: D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\Scanner\tiny.exe <filename>

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:13:06] C:1
$ ./tiny t1.tny
The lexical analysis result is saved to file t1.txt

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:16:22]
$ ls
main.c main.o t1.tny t1-sample.txt types.h
main.h makefile t1.txt tiny.exe
群文件里的样例输出

我的程序刚才生成的输出
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:19:15]
$
```

输入 `diff t1.txt t1-sample.txt` 比较：

| | |
|-----------|--|
| | <div data-bbox="329 196 1253 735"><div>问题 19 输出 调试控制台 终端 1: zsh</div><pre># hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:13:06] C:1 \$./tiny t1.tny The lexical analysis result is saved to file t1.txt # hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:16:22] \$ ls main.c main.o t1.tny t1-sample.txt types.h main.h makefile t1.txt tiny.exe diff 的输出为空，说明两个文件完全一致，实验成功 # hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:19:15] \$ diff t1.txt t1-sample.txt # hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 6/Scanner [21:20:18] \$</pre><div>(Global Scope) 行 322, 列 1 (已选择8381) 空格: 2 UTF-8 CRLF C Win32</div></div> <p>diff 的输出为空，说明两个文件完全一致，实验成功！</p> |
| <p>小结</p> | <p>1. 程序的鲁棒性</p> <p>我发现了原版 tiny 编译器的一个 bug，就是如果 tny 文件的末尾不是空行，那么输出的行号格式就会错乱，如下图：</p> <div data-bbox="329 999 1225 1342"><pre>11 —> until 'x' = '0';CRLF 12 —> write fact { 'output factorial of 'x'}CRLF 13 end</pre><p>不以空行结尾</p></div> |

```

10: :=
10: ID, name= x
10: -
10: NUM, val= 1 (line_buffer);
10: ;
11: until x = 0; // 如果最后一行字符是没有 \n 的，需要补一个 \n
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved word: write
12: ID, name= fact
13: end      14: reserved word: end
15: EOF

```

明明只有13行，这里却错乱了，还出现了14行和15行的行号

D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\TINY 编译器源码

我觉得这是这个程序鲁棒性不强的地方，怎么能保证源码文件一定以空行结尾呢？因此，我引入了 `currentLineHasLF()` 方法，它只有一行代码：

```

int currentLineHasLF(void) { return
line_buffer[line_buffer_size - 1] == '\n'; }

```

但是却可以检测到行尾是不是以回车结束的。如果行尾不是以回车结束，这样的情况只有一种，就是它不仅是行尾，也是文件尾。这时候就需要做特殊处理。我写的 `getNextChar()` 代码如下，处理的地方已经用注释作了说明：

```

int getNextChar(void) {
    if (line_buffer_index < line_buffer_size) {
        // 如果当前读取的内容在缓存区边界内，就直接返回下标对应的
        // 字符，并将下标加一
        return line_buffer[line_buffer_index++];
    } else {
        // 如果当前读取的内容不在当前行缓存区边界内，说明这一行已
        // 经读取完了

        // C 库函数 char *fgets(char *str, int n, FILE
        *stream) 从指定的流 stream 读取一行，并把它存储在 str 所指向
    }
}

```

的字符串内。当读取 (n-1) 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。

// 这里读取的是 (MAX_BUFFER_LENGTH - 1) 个字符，是为了给 \0 腾出一个字节

```
if (fgets(line_buffer, MAX_BUFFER_LENGTH - 1,
source_file)) {
```

// 如果 source_file 中的内容还有，打印这一行的内容，附带行号

// 行号加一

```
line_number++;
```

// 记录这一行的实际字符数

```
line_buffer_size = strlen(line_buffer);
```

// 如果 EOF 没有独立成行的话，最后一行字符是没有 \n 的，需要补一个 \n 让其输出换行

```
if (!currentLineHasLF()) {
```

```
    line_buffer[line_buffer_size++] = '\n';
```

```
    line_buffer[line_buffer_size] = '\0';
```

```
    fprintf(result_file, "%4d: %s", line_number,
line_buffer);
```

```
    line_buffer[--line_buffer_size] = '\0';
```

```
} else {
```

```
    fprintf(result_file, "%4d: %s", line_number,
line_buffer);
```

```
}
```

// 将当前行读取到的下标重置为 0

```
line_buffer_index = 0;
```

// 返回当前行的第一个字符，并将下标加一

```
return line_buffer[line_buffer_index++];
```

```
} else {
```

```

// 如果 source_file 中的内容已经全部读取完了，就返回
EOF

is_EOF = TRUE;

if (currentLineHasLF()) {
    line_number++;
}

return EOF;
}
}
}

```

而我的 getNextChar()函数即使遇到文件尾没有空行的情况，也能正常处理，鲁棒性更强，如下图所示：

```

10:      x := x - 1;
10: ID, name= x
10: :=
10: ID, name= x
10: -
10: NUM, val= 1
10: ;
11: until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factori
12: reserved word: write
12: ID, name= fact
13: end
13: reserved word: end
13: EOF

```

```

6 if '0' > x then { 'don't compute if
7   fact := 1; CRLF
8   repeat CRLF
9     fact := fact * x; CRLF
10    x := x - 1; CRLF
11    until x = 0; CRLF
12    write fact { output factoria
13 end

```

文件尾不是空行
EOF在第13行，此处文件结束
程序也正确在13行输出了EOF

Nor length: 253 lines: 13 Ln: 13 Col: 4

2. 注意缩进的区别

原版的编译器又出现了一个问题，明明 EOF 是在下一行了，行号也输出的是下一行的行号。但是，缩进居然和上一行的关键字是同一个缩进，这就很尴尬了，成了四不像，如下图所示：

```

10: ID, name= x
10: -
10: NUM, val= 1
10: ;
11: until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved word: write
12: ID, name= fact
12: ;
13: end
13: reserved word: end
14: EOF

```

14行的缩进不正确，应该和前面的13行源码的输出对齐

D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 6\TINY 编译器源码

λ

因此我在这里又做了一个检测。如果文件尾无空行时，那么 EOF 作为上一行的一份子输出（前文有截图）；如果文件尾有空行时，那么 EOF 独立成行输出，缩进和其他行的源代码输出对齐，代码如下：

```

// 当 EOF 独立成行时，这里需要用行号的方式输出
if (current_token == ENDFILE && currentLineHasLF()) {
    fprintf(result_file, "%4d: ", line_number);
} else {
    fprintf(result_file, "\t%d: ", line_number);
}

```

做了这个调整后，输出就正确了：

```

10:      x := x - 1;
10: ID, name= x
10: :=
10: ID, name= x
10: -
10: NUM, val= 1
10: ;
11: until x = 0;
11: reserved word: until
11: ID, name= x
11: =
11: NUM, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved word: write
12: ID, name= fact
12: ;
13: end
13: reserved word: end
14: EOF

```

输出正确，对齐了

| | |
|----------------|--|
| | <p>3. 重温了 <code>makefile</code> 文件的编写，复习了 C 语言，回顾了 <code>diff</code>、<code>cat</code> 等命令的使用和 <code>vim</code>、<code>gcc</code> 等程序的使用，巩固了已有的知识。</p> <p>4. 熟悉了 Tiny 语言词法的特点，并根据 Tiny 语言词法构造 DFA，使其可以识别 Tiny 语言 Token，又用 C 语言实现 Tiny 语言的词法分析器，设计数据类型、数据结构与其他代码，并且可以正常编译、运行、调试。这是我过去没有尝试过的事情，感觉学习到了很多新知识，加深了我对编译原理这门课程的理解，在成为一名优秀的 985 大学计算机专业本科生的道路上迈出了坚实的一步。</p> |
| 指导老 师评 议 | <p>成绩评定：</p> <p>指导教师签名：</p> |