

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 C-语言的语法分析器	实验时间	2019 年 5 月 21 日、2019 年 5 月 28 日
实验目的	<ol style="list-style-type: none">1. 熟悉 C-语言语法, 选择实现方法2. 设计数据类型、数据结构3. 用 C 或 C++实现 C-语言的语法分析器4. 调试、运行5. 提交实验报告		
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

本次实验使用递归下降法构造 C-语言的语法分析器，对词法分析器生成的单词序列进行语法分析，产生一颗抽象语法树作为语法分析器的输出，并将抽象语法树按照约定的格式打印出来。如果语法分析中发生错误，需要提供有一定意义的错误信息，并将错误信息在打印语法树之前输出。

一、C-语言语法的特点

在之前的词法分析实验中，我们了解到了 C-语言的词法特点，并编写了可以依次获取 Token 的函数。将这些 Token 组合在一起，就形成了 C-语言的语句，而描述这些语句的文法，就是 C-语言的语法。首先要把 C-语言的语法从 BNF 转换为 EBNF（扩展巴科斯-诺尔范式）表示。

BNF 语法为（红色的是需要修改的）：

	BNF
1	program \rightarrow declaration_list
2	declaration_list \rightarrow declaration_list declaration declaration
3	declaration \rightarrow var_declaration fun_declaration
4	var_declaration \rightarrow type_specifier ID; type_specifier ID [NUM];
5	type_specifier \rightarrow int void
6	fun_declaration \rightarrow type_specifier ID (params) compound_stmt
7	params \rightarrow param_list void
8	param_list \rightarrow param_list , param param
9	param \rightarrow type_specifier ID type_specifier ID []
10	compound_stmt \rightarrow { local_declarations statement_list }
11	local_declarations \rightarrow local_declarations var_declaration ϵ
12	statement_list \rightarrow statement_list statement ϵ

13	statement \rightarrow expression_stmt compound_stmt selection_stmt iteration_stmt return_stmt
14	expression_stmt \rightarrow expression ; ;
15	selection_stmt \rightarrow if (expression) statement if (expression) statement else statement
16	iteration_stmt \rightarrow while (expression) statement
17	return_stmt \rightarrow return; return expression;
18	expression \rightarrow var = expression simple_expression
19	var \rightarrow ID ID [expression]
20	simple_expression \rightarrow additive_expression relop additive_expression additive_expression
21	relop \rightarrow <= < > >= == !=
22	additive_expression \rightarrow additive_expression addop term term
23	addop \rightarrow + -
24	term \rightarrow term mulop factor factor
25	mulop \rightarrow * /
26	factor \rightarrow (expression) var call NUM
27	call \rightarrow ID (args)
28	args \rightarrow arg_list ϵ
29	arg_list \rightarrow arg_list , expression expression

将其去除左递归、左因子后，修改后的 EBNF 表示的语法为：

	EBNF
1	program \rightarrow declaration_list
2	declaration_list \rightarrow declaration{ declaration }
3	declaration \rightarrow var_declaration fun_declaration
4	var_declaration \rightarrow type_specifier ID; type_specifier ID [NUM];

5	$\text{type_specifier} \rightarrow \text{int} \mid \text{void}$
6	$\text{fun_declatation} \rightarrow \text{type_specifier ID (params) compound_stmt}$
7	$\text{params} \rightarrow \text{param_list} \mid \text{void}$
8	$\text{param_list} \rightarrow \text{param}\{, \text{param}\}$
9	$\text{param} \rightarrow \text{type_specifier ID}\{[]\}$
10	$\text{compound_stmt} \rightarrow \{ \text{local_declaration statement_list} \}$
11	$\text{local_declarations} \rightarrow \varepsilon \{ \text{var_declaration} \}$
12	$\text{statement_list} \rightarrow \{ \text{statement} \}$
13	$\text{statement} \rightarrow \text{expression_stmt} \mid \text{compound_stmt} \mid \text{selection_stmt} \mid \text{iteration_stmt} \mid \text{return_stmt}$
14	$\text{expression_stmt} \rightarrow [\text{expression}] ;$
15	$\text{selection_stmt} \rightarrow \text{if (expression) statement [else statement]}$
16	$\text{iteration_stmt} \rightarrow \text{while (expression) statement}$
17	$\text{return_stmt} \rightarrow \text{return [expression] ;}$
18	$\text{expression} \rightarrow \text{var} = \text{expression} \mid \text{simple_expression}$
19	$\text{var} \rightarrow \text{ID} \mid \text{ID [expression]}$
20	$\text{simple_expression} \rightarrow \text{additive_expression}\{ \text{relop additive_expression} \}$
21	$\text{relop} \rightarrow \leq \mid < \mid > \mid \geq \mid == \mid !=$
22	$\text{additive_expression} \rightarrow \text{term}\{ \text{addop term} \}$
23	$\text{addop} \rightarrow + \mid -$
24	$\text{term} \rightarrow \text{factor}\{ \text{mulop factor} \}$
25	$\text{mulop} \rightarrow * \mid /$
26	$\text{factor} \rightarrow (\text{expression}) \mid \text{var} \mid \text{call} \mid \text{NUM}$
27	$\text{call} \rightarrow \text{ID}(\text{args})$
28	$\text{args} \rightarrow \text{arg_list} \mid \varepsilon$

29	<code>arg_list → expression{ , expression }</code>
----	--

二、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

// C-Minus 源代码一行最多 255 个字符
#define MAX_BUFFER_LENGTH 256

// C-Minus 语言一共有 6 个保留字
#define MAX_RESERVED_NUMBER 6

// 定义语法树的最大子节点数量
#define MAX_CHILDREN 4
```

2. C-语言共有 29 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：

```
// 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,  
    // C-Minus 语言的 6 个保留关键字  
    IF,  
    ELSE,  
    INT,  
    RETURN,  
    VOID,  
    WHILE,  
    // 标识符  
    ID,  
    // 数字  
    NUM,  
    // 运算符  
    ASSIGN,  
    EQ,  
    LT,  
    LE,  
    GT,  
    GE,  
    NEQ,  
    PLUS,  
    MINUS,  
    TIMES,  
    OVER,  
    LPAREN,
```

```
RPAREN,  
LBRACKET,  
RBRACKET,  
LBRACE,  
RBRACE,  
COMMA,  
SEMI  
} TokenType;
```

3. 如上图所示，在识别 C-语言 Token 的 DFA 中，一共有 11 个状态，分别为 START、INCOMMENT、INNUM、INID、INEQ、INLE、INGE、INNEQ、LBUFFER、RBUFFER、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：

```
// 定义词法扫描 DFA 的状态  
typedef enum StateType {  
    START,  
    INCOMMENT,  
    INNUM,  
    INID,  
    INEQ,  
    INLE,  
    INGE,  
    INNEQ,  
    LBUFFER,  
    RBUFFER,  
    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 6 的 `ReservedWord[]` 数组，存放 C-语言的 6 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},  
    {"else", ELSE},  
    {"int", INT},  
    {"return", RETURN},  
    {"void", VOID},  
    {"while", WHILE}};
```

5. 在语法树中，我们将节点分为 21 类，有两种不同的返回值（`Void` 和 `Integer`）。这 21 种节点类型同样用一个枚举 `NodeKind` 表示：

```
// 节点返回值类型  
typedef enum ExpType { Void, Integer } ExpType;  
  
// 节点类型  
typedef enum NodeKind {
```



```

    StmtK,
    ExpK,
    IntK,
    VoidK,
    Var_DeclK,
    Arry_DeclK,
    Funk,
    ParamsK,
    ParamK,
    CompK,
    Selection_StmtK,
    Iteration_StmtK,
    Return_StmtK,
    AssignK,
    Arry_ElemK,
    CallK,
    ArgsK,
    UnkownK,
    OpK,
    ConstK,
    IdK
} NodeKind;

```

他们的说明分别为：

节点类型	描述	子节点组成
IntK	变量或返回值 类型（ int ）	无
VoidK	变量或返回值	无

		类型 (void)	
	IdK	标示符 (id)	无
	ConstK	数值	无
	Var_DeclK	变量声明	变量类型+变量名
	Var_DeclK	数组声明	数组名+数组大小
	Funk	函数声明	返回类型+函数名+参数列表+函数体
	ParamsK	Funk 的参数列表	参数（如果有多个参数，则之间为兄弟节点）
	ParamK	Funk 的参数	参数类型+参数名
	CompK	复合语句体	变量声明列表+语句列表
	Selection_StmtK	if	条件表达式+IF 体+[ELSE 体]
	Iteration_StmtK	while	条件表达式+循环体
	Return_StmtK	return	[表达式]
	AssignK	赋值	被赋值变量+赋值变量
	OpK	运算	运算符左值+运算符右值
	Arry_ElemK	数组元素	数组名+下标
	CallK	函数调用	函数名+参数列表
	ArgsK	CallK 的参数列表	[表达式]（如果有多个表达式，则之间为兄弟节点）
	UnkownK	未知节点	无
	6. 每个语法树节点包括子节点、同属连接产生的的兄弟节点、节点类型、节点属性，为了方便输出错误信息提示，还需要记录当前语句的行号。因此，		

采用 C 语言的结构体设计如下：

```
// 语法树节点
typedef struct treeNode {
    struct treeNode* child[MAX_CHILDREN];
    struct treeNode* sibling;
    int line_number;
    NodeKind node_kind;
    union {
        TokenType operation;
        int value;
        const char* name;
    } attribute;
    ExpType type;
} TreeNode;
```

三、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 C-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。并且，还需要记录当前的 **token** 字符串和前一个 **token** 字符串。因此，需要声明下面七个变量：

```
// 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;
// 存放当前 token 的字符串
char token_string[MAX_TOKEN_LENGTH + 1] = "";
// 存放前一个 token 的字符串
char token_string_prev[MAX_TOKEN_LENGTH + 1] = "";
```

在语法分析时，需要记录下当前的 **Token** 是什么，也需要记录下前一个 **Token** 是什么。而在打印语法树时，则需要递归打印不断变化的缩进。因此需要三个全局变量处理：

```
// 记录当前的 Token
TokenType token;
// 记录前一个 Token
TokenType token_prev;
// 在 printTree() 中缩进的数量
int indent_number = 0;
```

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, ".c-");  
}
```

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 输出语法分析的结果即可。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, ".c-") = '\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「CMINUS PARSING:」作为开头，再调用 `parse` 函数不断读取 Token，按照 EBNF 文法递归下降处理，以生成语法树，并在生成语法树成功后将其打印输出到结果文件中。此外，在分析中如果遇到了错误，也需要将错误输出，并给予具有意义的错误信息。在源代码文件全部处理完后，再在 Shell 里显示提示，告知用户语法分析的结果保存在了「源文件名」.txt 文件中。

```
fprintf(result_file, "CMINUS PARSING:\n");

syntaxTree = parse();
fprintf(result_file, "\nSyntax tree:\n");
printTree(syntaxTree);

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

在一切都结束后，还需要做一点清理工作，用 `fclose` 将先前用 `fopen` 打开的 `source_file` 关闭，最后令 `main` 函数返回 0，程序正常结束：

```
fclose(source_file);  
return 0;
```

3. 词法扫描部分函数设计的变化

这是整个语法分析程序所依赖的函数，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，获取 Token 再生成语法树。在先前词法分析的实验中，我的 `token_string` 是局部变量，因为现在 `token_string` 需要给后面的语法分析程序访问，`token_string` 我已经设置成了全局变量。因为我们是逐行读取 c-源文件内容的，所以首先要初始化一些变量，如下所示：

```
// 保存前一个 Token  
memset(token_string_prev, 0,  
sizeof(token_string_prev));  
strcpy(token_string_prev, token_string);  
// 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  
memset(token_string, 0, sizeof(token_string));  
// 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, INCOMMENT,  
INNUM, INID, INEQ, INLE,  
// INGE, INNEQ, LBUFFER, RBUFFER, DONE 十一种  
StateType state = START;
```

然后，就是一个经典的双层 case 处理关键字了，由于这不是本次实验重点，**略过**。因为在后面的语法分析报错时，需要按照规定的格式将 token

打印出来，所以我编写了一个方法叫做 `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 ELSE:
        case INT:
        case RETURN:
        case VOID:
        case WHILE:
            fprintf(result_file, "reserved word: %s\n",
string);
            break;
        case ASSIGN:
            fprintf(result_file, "=\n");
            break;
        case EQ:
            fprintf(result_file, "==\n");
            break;
        case LT:
            fprintf(result_file, "<\n");
            break;
        case LE:
            fprintf(result_file, "<=\n");
            break;
        case GT:
            fprintf(result_file, ">\n");
            break;
        case GE:
```



```
    fprintf(result_file, ">=\n");
    break;
case NEQ:
    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 LPAREN:
    fprintf(result_file, "(\n");
    break;
case RPAREN:
    fprintf(result_file, ")\n");
    break;
case LBRACKET:
    fprintf(result_file, "[\n");
    break;
case RBRACKET:
    fprintf(result_file, "]\n");
    break;
case LBRACE:
```

```

        fprintf(result_file, "{\n");
        break;
    case RBRACE:
        fprintf(result_file, "}\n");
        break;
    case COMMA:
        fprintf(result_file, ",\n");
        break;
    case SEMI:
        fprintf(result_file, ";\n");
        break;
    case ENDFILE:
        fprintf(result_file, "EOF\n");
        break;
    case NUM:
        fprintf(result_file, "NUM, value= %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);
    }
}

```

此外，因为按照样例，我不仅需要按照规定格式输出出错的 Token，还

需要输出预期的 (Expected) Token 的类型, 这就需要我们编写一个函数, 可以将 Token 的类型名写入到给出的 char 指针位置:

```
// 获取 Token 的类型标识文字
void getTokenString(TokenType token, char* string) {
    switch (token) {
        case IF:
            strcpy(string, "IF");
            break;
        case ELSE:
            strcpy(string, "ELSE");
            break;
        case INT:
            strcpy(string, "INT");
            break;
        case RETURN:
            strcpy(string, "RETURN");
            break;
        case VOID:
            strcpy(string, "VOID");
            break;
        case WHILE:
            strcpy(string, "WHILE");
            break;
        case ASSIGN:
            strcpy(string, "ASSIGN");
            break;
        case EQ:
            strcpy(string, "EQ");
            break;
    }
}
```

```
case LT:
    strcpy(string, "LT");
    break;
case LE:
    strcpy(string, "LE");
    break;
case GT:
    strcpy(string, "GT");
    break;
case GE:
    strcpy(string, "GE");
    break;
case NEQ:
    strcpy(string, "NEQ");
    break;
case PLUS:
    strcpy(string, "PLUS");
    break;
case MINUS:
    strcpy(string, "MINUS");
    break;
case TIMES:
    strcpy(string, "TIMES");
    break;
case OVER:
    strcpy(string, "OVER");
    break;
case LPAREN:
    strcpy(string, "LPAREN");
    break;
```

```
case RPAREN:
    strcpy(string, "RPAREN");
    break;

case LBRACKET:
    strcpy(string, "LBRACKET");
    break;

case RBRACKET:
    strcpy(string, "RBRACKET");
    break;

case LBRACE:
    strcpy(string, "LBRACE");
    break;

case RBRACE:
    strcpy(string, "RBRACE");
    break;

case COMMA:
    strcpy(string, "COMMA");
    break;

case SEMI:
    strcpy(string, "SEMI");
    break;

case ENDFILE:
    strcpy(string, "ENDFILE");
    break;

case NUM:
    strcpy(string, "NUM");
    break;

case ID:
    strcpy(string, "ID");
    break;
```

```

    case ERROR:
        strcpy(string, "ERROR");
        break;
    default:
        // 永不发生
        strcpy(string, "Unknown token");
    }
}

```

4. 语法树生成

按照前文所说,我们的语法树由节点进行子连接与同属连接产生,因此,需要编写返回节点的生成函数,他们可以返回指向节点内存存储位置的指针,实际上是链表结构:

```

// 创建新节点
TreeNode* newNode(NodeKind kind) {
    TreeNode* p = (TreeNode*)malloc(sizeof(TreeNode));
    int k;
    if (p == NULL) {
        fprintf(result_file, "Out of memory error at
line %d\n", line_number);
    } else {
        for (k = 0; k < MAX_CHILDREN; k++) {
            p->child[k] = NULL;
        }
        p->sibling = NULL;
        p->node_kind = kind;
        p->line_number = line_number;
    }
}

```

```

    if (kind == OpK || kind == IntK || kind == IdK) {
        p->type = Integer;
    }
    if (kind == IdK) {
        p->attribute.name = "";
    }
    if (kind == ConstK) {
        p->attribute.value = 0;
    }
}
return p;
}

```

在得到生成两种节点的函数后，就可以开始编写生成语法树的 `parse` 函数了。语句通过同属域而不是子域来排序，即由父亲到他的孩子的唯一物理连接是到最左孩子的。孩子则在一个标准连接表中自左向右连接到一起，这种连接称作同属连接（左），用于区别父子连接（右）。



这个函数实际上非常简单，就和链表的编写方法一致，首元素获取到 `Token` 之后生成首个节点（一定是语句节点），然后程序就会自动按照 `EBNF` 规则生成整个语法树。这里需要注意的一点是，如果在程序运行结束后，剩下的最后一个 `Token` 不是 `ENDFILE`（文件尾）的话，那说明语法分析过程应该是出错了，导致源代码未能正常分析完。这部分代码为：

```

// 生成语法树
TreeNode* parse(void) {

```

```

TreeNode* t;
token = getToken();
t = declaration_list();
if (token != ENDFILE) {
    printSyntaxError("Code ends before file\n");
}
return t;
}

```

5. EBNF

EBNF 部分由 21 个相互递归的函数组成，他们与前文给出的 C-语言的 EBNF 文法直接对应：

```

// 下面的函数全部对应 EBNF

TreeNode* declaration_list(void);
TreeNode* declaration(void);
TreeNode* params(void);
TreeNode* param_list(TreeNode* k);
TreeNode* param(TreeNode* k);
TreeNode* compound_stmt(void);
TreeNode* local_declaration(void);
TreeNode* statement_list(void);
TreeNode* statement(void);
TreeNode* expression_stmt(void);
TreeNode* selection_stmt(void);
TreeNode* iteration_stmt(void);
TreeNode* return_stmt(void);

```



```
TreeNode* expression(void);
TreeNode* var(void);
TreeNode* simple_expression(TreeNode* k);
TreeNode* additive_expression(TreeNode* k);
TreeNode* term(TreeNode* k);
TreeNode* factor(TreeNode* k);
TreeNode* call(TreeNode* k);
TreeNode* args(void);
```

下面将一一介绍：

`TreeNode* declaration_list()`

使用递归向下分析方法直接调用 `declaration()` 函数，并返回树节点

```
TreeNode* declaration_list() {
    TreeNode* t = declaration();
    TreeNode* p = t;

    //在开始语法分析出错的情况下找到 int 和 void 型，过滤掉 int 和
    void 之前的所有 Token，防止雪崩式出错

    while ((token != INT) && (token != VOID) && (token !=
    ENDFILE)) {
        printSyntaxError("");
        getToken();
        if (token == ENDFILE) {
            break;
        }
    }

    //寻找语法分析的入口，即找到 int 和 void

    while ((token == INT) || (token == VOID)) {
```

```

    TreeNode* q;
    q = declaration();
    if (q != NULL) {
        if (t == NULL) {
            t = p = q;
        } else {
            p->sibling = q;
            p = q;
        }
    }
}
match(ENDFILE);
return t;
}

```

TreeNode* declaration(void)

C-语言的声明分为变量声明和函数声明。declaration(void)函数并不是直接调用 var-declaration 或 fun-declaration 文法所对应的函数，令其返回节点，实际上程序并没有为 var-declaration 和 fun-declaration 文法定义函数，而是在 declaration(void)函数中，通过求 First 集合的方式先确定是变量定义还是函数定义，然后分别根据探测的结果生成变量声明节点 (Var_DeclK) 或函数声明节点 (FunK)。所以 var-declaration 和 fun-declaration 文法在 declaration → var-declaration|fun-declaration 文法中就都已经分析了

```

TreeNode* declaration(void) {
    TreeNode* t = NULL;
    TreeNode* p = NULL;
    TreeNode* q = NULL;

```

```

TreeNode* s = NULL;

if (token == INT) {
    p = newNode(IntK);
    match(INT);
} else if (token == VOID) {
    p = newNode(VoidK);
    match(VOID);
} else {
    printSyntaxError("Type Error");
}

if ((p != NULL) && (token == ID)) {
    q = newNode(IdK);
    q->attribute.name = copyString(token_string);
    match(ID);

    if (token == LPAREN) {
        //'(': 函数情况
        t = newNode(Funk);
        t->child[0] = p;
        t->child[1] = q;
        match(LPAREN);
        t->child[2] = params();
        match(RPAREN);
        t->child[3] = compound_stmt();
    } else if (token == LBRACKET) {
        //'[': 数组声明
        t = newNode(Var_DeclK);
        TreeNode* m = newNode(Arry_DeclK);

```

```

        match(LBRACKET);
        match(NUM);
        s = newNode(ConstK);
        s->attribute.value =
atoi(copyString(token_string_prev));
        m->child[0] = q;
        m->child[1] = s;
        t->child[0] = p;
        t->child[1] = m;
        match(RBRACKET);
        match(SEMI);
    } else if (token == SEMI)  //'';'结尾: 普通变量声明
    {
        t = newNode(Var_DeclK);
        t->child[0] = p;
        t->child[1] = q;
        match(SEMI);
    } else {
        printSyntaxError("");
    }
} else {
    printSyntaxError("");
}
return t;
}

```

TreeNode* params(void)

函数参数列表要么是 void, 要么是多个参数组成。params(void)函数先判

断第一个是 `void` 还是 `int`，如果是 `int` 说明是由 `param_list` 组成，则调用 `param_list(TreeNode * k)` 函数递归向下分析；如果是 `void` 说明可能是 `void` 型的参数，也可能参数就是 `void`，所以再求其 Follow 集合，如果集合求出来是右括号，则说明参数就是 `void`，于是新建一个 `VoidK` 节点就行；如果集合求出来不是右括号则说明是 `void` 型的参数，然后再调用 `param_list(TreeNode * k)` 函数递归向下分析，并将已经取出 `VoidK` 节点传递给 `param_list(TreeNode * k)` 函数

```
TreeNode* params(void) {
    TreeNode* t = newNode(ParamsK);
    TreeNode* p = NULL;
    if (token == VOID) {
        //开头为 void, 参数列表可能是(void)和(void id,[.....])两种情况
        p = newNode(VoidK);
        match(VOID);
        if (token == RPAREN) {
            //参数列表为(void)
            if (t != NULL) {
                t->child[0] = p;
            }
        } else {
            //参数列表为(void id,[.....]) ->void 类型的变量
            t->child[0] = param_list(p);
        }
    } else if (token == INT) {
        //参数列表为(int id,[.....])
        t->child[0] = param_list(p);
    } else {
        printSyntaxError("");
    }
}
```

```

    }

    return t;
}

```

`TreeNode* param_list(TreeNode* k)`

参数列表由 `param` 序列组成，并由逗号隔开。`param_list(TreeNode * k)` 函数使用递归向下分析方法直接调用 `param(TreeNode * k)` 函数，并返回树节点

```

TreeNode* param_list(TreeNode* k) {
    // k 可能是已经被取出来的 VoidK，但又不是(void)类型的参数列表，所以一直传到 param 中去，作为其一个子节点

    TreeNode* t = param(k);
    TreeNode* p = t;
    k = NULL; //没有要传给 param 的 VoidK，所以将 k 设为 NULL
    while (token == COMMA) {
        TreeNode* q = NULL;
        match(COMMA);
        q = param(k);
        if (q != NULL) {
            if (t == NULL) {
                t = p = q;
            } else {
                p->sibling = q;
                p = q;
            }
        }
    }
    return t;
}

```

```
}
```

```
TreeNode* param(TreeNode* k)
```

参数由 `int` 或 `void`、标示符组成，最后可能有中括号表示数组。

`param(TreeNode * k)`函数根据探测先行 Token 是 `int` 还是 `void` 而新建 `IntK` 或 `VoidK` 节点作为其第一个子节点，然后新建 `IdK` 节点作为其第二个子节点，最后探测 `Follow` 集合，是否是中括号，而确定是否再新建第三个子节点表示数组类型

```
TreeNode* param(TreeNode* k) {
    TreeNode* t = newNode(ParamK);
    TreeNode* p = NULL; // ParamK 的第一个子节点
    TreeNode* q = NULL; // ParamK 的第二个子节点

    if (k == NULL && token == INT) {
        p = newNode(IntK);
        match(INT);
    } else if (k != NULL) {
        p = k;
    }

    if (p != NULL) {
        t->child[0] = p;
        if (token == ID) {
            q = newNode(IdK);
            q->attribute.name = copyString(token_string);
            t->child[1] = q;
            match(ID);
        } else {
            printSyntaxError("");
        }
    }
}
```

```

    }

    if ((token == LBRACKET) && (t->child[1] != NULL)) {
        match(LBRACKET);
        t->child[2] = newNode(IdK);
        match(RBRACKET);
    } else {
        return t;
    }
} else {
    printSyntaxError("");
}
return t;
}

```

TreeNode* compound_stmt(void)

复合语句由用花括号围起来的一组声明和语句组成。

compound_stmt(void) 函数使用递归向下分析方法直接调用 **local_declaration()**函数和 **statement_list()**函数，并根据返回的树节点作为其第一个子节点和第二个子节点

```

TreeNode* compound_stmt(void) {
    TreeNode* t = newNode(CompK);
    match(LBRACE);
    t->child[0] = local_declaration();
    t->child[1] = statement_list();
    match(RBRACE);
    return t;
}

```


TreeNode* local_declaration(void)

局部变量声明要么是空，要么由许多变量声明序列组成。

local_declaration(void)函数通过判断先行的 Token 是否是 int 或 void 便可知道局部变量声明是否为空，如果不为空，则新建一个变量定义节点 (Var_DeclK)

```
TreeNode* local_declaration(void) {
    TreeNode* t = NULL;
    TreeNode* q = NULL;
    TreeNode* p = NULL;
    while (token == INT || token == VOID) {
        p = newNode(Var_DeclK);
        if (token == INT) {
            TreeNode* q1 = newNode(IntK);
            p->child[0] = q1;
            match(INT);
        } else if (token == VOID) {
            TreeNode* q1 = newNode(VoidK);
            p->child[0] = q1;
            match(INT);
        }
        if ((p != NULL) && (token == ID)) {
            TreeNode* q2 = newNode(IdK);
            q2->attribute.name = copyString(token_string);
            p->child[1] = q2;
            match(ID);
        }
        if (token == LBRACKET) {
```

```

        TreeNode* q3 = newNode(Var_DeclK);
        p->child[3] = q3;
        match(LBRACKET);
        match(RBRACKET);
        match(SEMI);
    } else if (token == SEMI) {
        match(SEMI);
    } else {
        match(SEMI);
    }
} else {
    printSyntaxError("");
}
if (p != NULL) {
    if (t == NULL)
        t = q = p;
    else {
        q->sibling = p;
        q = p;
    }
}
return t;
}

```

TreeNode* statement_list(void)

语句列表由 0 到多个 statement 组成。statement_list(void) 函数通过判断先行 Token 类型是否为 statement 的 First 集合确定后面是否是一

个 `statement`，如果是，则使用递归向下分析方法直接调用 `statement()` 函数

```
TreeNode* statement_list(void) {
    TreeNode* t = statement();
    TreeNode* p = t;
    while (IF == token || LBRACE == token || ID == token ||
WHILE == token ||
        RETURN == token || SEMI == token || LPAREN ==
token || NUM == token) {
        TreeNode* q;
        q = statement();
        if (q != NULL) {
            if (t == NULL) {
                t = p = q;
            } else {
                p->sibling = q;
                p = q;
            }
        }
    }
    return t;
}
```

`TreeNode* statement(void)`

`statement` 由表达式或复合语句或 `if` 语句或 `while` 语句或 `return` 语句构成。`statement(void)` 函数通过判断先行 `Token` 类型确定到底是哪一种类型。如果是 `IF` 则是 `if` 语句类型，如果是 `WHILE`，则是 `while` 语句类型，如果是 `RETURN`，则是 `return` 语句类型，如果是左大括号，则是复合语句

类型，如果是 ID、SEMI、LPAREN、NUM，则是表达式语句类型

```
TreeNode* statement(void) {
    TreeNode* t = NULL;
    switch (token) {
        case IF:
            t = selection_stmt();
            break;
        case WHILE:
            t = iteration_stmt();
            break;
        case RETURN:
            t = return_stmt();
            break;
        case LBRACE:
            t = compound_stmt();
            break;
        case ID:
        case SEMI:
        case LPAREN:
        case NUM:
            t = expression_stmt();
            break;
        default:
            printSyntaxError("");
            token = getToken();
            break;
    }
    return t;
}
```

TreeNode* selection_stmt(void)

selection_stmt(void) 函数直接调用 expression() 函数和 statement() 函数分别得到其第一个和第二个子节点，然后通过判断先行 Token 类型是否为 ELSE，如果是则直接调用 statement() 函数得到其第三个子节点

```
TreeNode* selection_stmt(void) {
    TreeNode* t = newNode(Selection StmtK);
    match(IF);
    match(LPAREN);
    if (t != NULL) {
        t->child[0] = expression();
    }
    match(RPAREN);
    t->child[1] = statement();
    if (token == ELSE) {
        match(ELSE);
        if (t != NULL) {
            t->child[2] = statement();
        }
    }
    return t;
}
```

TreeNode* expression_stmt(void)

表达式语句有一个可选的且后面跟着分号的表达式。

expression_stmt(void) 函数通过判断先行 Token 类型是否为分号，如果

不是则直接调用函数 `expression()`

```
TreeNode* expression_stmt(void) {  
    TreeNode* t = NULL;  
    if (token == SEMI) {  
        match(SEMI);  
        return t;  
    } else {  
        t = expression();  
        match(SEMI);  
    }  
    return t;  
}
```

`TreeNode* iteration_stmt(void)`

`iteration_stmt(void)` 函数直接调用 `expression()` 函数和 `statement()` 函数分别得到其第一个和第二个子节点

```
TreeNode* iteration_stmt(void) {  
    TreeNode* t = newNode(Iteration_StmtK);  
    match(WHILE);  
    match(LPAREN);  
    if (t != NULL) {  
        t->child[0] = expression();  
    }  
    match(RPAREN);  
    if (t != NULL) {  
        t->child[1] = statement();  
    }  
}
```

```
    return t;
}
```

TreeNode* return_stmt(void)

return_stmt(void)函数通过判断先行 Token 类型是否为分号，如果不是则直接调用函数 expression()得到其子节点

```
TreeNode* return_stmt(void) {
    TreeNode* t = newNode(Return_StmtK);
    match(RETURN);
    if (token == SEMI) {
        match(SEMI);
        return t;
    } else {
        if (t != NULL) {
            t->child[0] = expression();
        }
    }
    match(SEMI);
    return t;
}
```

TreeNode* expression(void)

expression(void)函数通过判断先行 Token 类型是否为 ID，如果不是说明只能是 simple_expression 情况，则调用 simple_expression(TreeNode * k)函数递归向下分析；如果是 ID 说明可能是赋值语句，或 simple_expression 中的 var 和 call 类型的情况，所以再求其 Follow 集合，如果集合求出来是赋值类型的 Token，则说明是

赋值语句，于是新建一个 `AssignK` 节点就行；如果集合求出来不是赋值类型的 `Token` 则说明是 `simple_expression` 中的 `var` 和 `call` 类型的情况，然后再调用 `simple_expression(TreeNode * k)` 函数递归向下分析，并将已经取出 `IdK` 节点传递给 `simple_expression(TreeNode * k)` 函数

```
TreeNode* expression(void) {
    TreeNode* t = var();
    if (t == NULL) //不是以 ID 开头，只能是 simple_expression
情况
    {
        t = simple_expression(t);
    } else {
        //以 ID 开头，可能是赋值语句，或 simple_expression 中的 var
和 call 类型的情况

        TreeNode* p = NULL;
        if (token == ASSIGN) {
            //赋值语句
            p = newNode(AssignK);
            p->attribute.name = copyString(token_string_prev);
            match(ASSIGN);
            p->child[0] = t;
            p->child[1] = expression();
            return p;
        } else {
            // simple_expression 中的 var 和 call 类型的情况
            t = simple_expression(t);
        }
    }
    return t;
}
```


TreeNode* simple_expression(TreeNode* k)
simple_expression(TreeNode * k) 函数先调用 additive_expression(TreeNode * k)函数返回一个节点，然后再一直判断后面的 Token 是否为<=、<、>、>=、==、!=，如果是则新建 OpK 节点，然后令之前的节点为其第一个子节点，然后继续调用 additive_expression(TreeNode * k)函数返回一个节点，作为 OpK 节点的第二个节点

```
TreeNode* simple_expression(TreeNode* k) {  
    TreeNode* t = additive_expression(k);  
    k = NULL;  
    if (EQ == token || GT == token || GE == token || LT ==  
token || LE == token ||  
        NEQ == token) {  
        TreeNode* q = newNode(OpK);  
        q->attribute.operation = token;  
        q->child[0] = t;  
        t = q;  
        match(token);  
        t->child[1] = additive_expression(k);  
        return t;  
    }  
    return t;  
}
```

TreeNode* additive_expression(TreeNode* k)
additive_expression(TreeNode * k)函数先调用 term(TreeNode *

k)函数返回一个节点，然后再一直判断后面的 Token 是否为+或-，如果是则新建 OpK 节点，然后令之前的节点为其第一个子节点，然后继续调用 term(TreeNode * k)函数返回一个节点，作为 OpK 节点的第二个节点

```
TreeNode* additive_expression(TreeNode* k) {
    TreeNode* t = term(k);
    k = NULL;
    while ((token == PLUS) || (token == MINUS)) {
        TreeNode* q = newNode(OpK);
        q->attribute.operation = token;
        q->child[0] = t;
        match(token);
        q->child[1] = term(k);
        t = q;
    }
    return t;
}
```

TreeNode* term(TreeNode* k)

term(TreeNode * k)函数先调用 factor(TreeNode * k)函数返回一个节点，然后再一直判断后面的 Token 是否为*或/，如果是则新建 OpK 节点，然后令之前的节点为其第一个子节点，然后继续调用 actor(TreeNode * k)函数返回一个节点，作为 OpK 节点的第二个节点

```
TreeNode* term(TreeNode* k) {
    TreeNode* t = factor(k);
    k = NULL;
    while ((token == TIMES) || (token == OVER)) {
        TreeNode* q = newNode(OpK);
        q->attribute.operation = token;
```

```

    q->child[0] = t;
    t = q;
    match(token);
    t->child[1] = factor(k);
}
return t;
}

```

`TreeNode* factor(TreeNode* k)`

`factor(TreeNode * k)` 函数首先判断 `k` 是否为空，如果不为空，则 `k` 为上面传下来的已经解析出来的以 `ID` 开头的 `var`，此时可能为 `call` 或 `var` 的情况，然后判断后面的 `Token` 是否为左括号，如果是则说明是 `call` 的情形，如果不是则为 `var` 的情形。如果 `k` 为空，再根据先行的 `Token` 类型判断是 4 中推导中的哪一种，然后直接调用相关的函数返回一个节点

```

TreeNode* factor(TreeNode* k) {
    TreeNode* t = NULL;
    if (k != NULL) {
        // k 为上面传下来的已经解析出来的以 ID 开头的 var, 可能为
        // call 或 var
        if (token == LPAREN && k->node_kind != Array_ElemK) {
            // call
            t = call(k);
        } else {
            t = k;
        }
    } else {
        //没有从上面传下来的 var
        switch (token) {

```

```

    case LPAREN:
        match(LPAREN);
        t = expression();
        match(RPAREN);
        break;
    case ID:
        k = var();
        if (LPAREN == token && k->node_kind != Arry_ElemK)
        {
            t = call(k);
        } else {
            t = k;
        }
        break;
    case NUM:
        t = newNode(ConstK);
        if ((t != NULL) && (token == NUM)) {
            t->attribute.value =
atoi(copyString(token_string));
        }
        match(NUM);
        break;
    default:
        printSyntaxError("");
        token = getToken();
        break;
    }
}

return t;
}

```

```
TreeNode* var(void)
```

var(void)函数先新建一个 IdK 节点，再通过判断先行 Token 类型是否为左中括号，如果是则新建一个数组节点 Arry_ElemK，将 IdK 作为 Arry_ElemK 节点的第一个子节点，再直接调用函数 expression()得到 Arry_ElemK 的第二个子节点，最后返回的节点时 Arry_ElemK；如果先行 Token 类型不是左中括号，则将之前的 IdK 返回

```
TreeNode* var(void) {
    TreeNode* t = NULL;
    TreeNode* p = NULL;
    TreeNode* q = NULL;
    if (token == ID) {
        p = newNode(IdK);
        p->attribute.name = copyString(token_string);
        match(ID);
        if (token == LBRACKET) {
            match(LBRACKET);
            q = expression();
            match(RBRACKET);

            t = newNode(Arry_ElemK);
            t->child[0] = p;
            t->child[1] = q;
        } else {
            t = p;
        }
    }
    return t;
}
```

```
}
```

```
TreeNode* call(TreeNode* k)
```

call(TreeNode * k)函数新建一个 call 语句的节点 CallK, 如果 k 不为空, 则将 k 设为 CallK 的第一个子节点, 然后通过调用 args(void)函数获得其第二个节点, 最后返回 CallK 节点

```
TreeNode* call(TreeNode* k) {  
    TreeNode* t = newNode(CallK);  
    if (k != NULL) t->child[0] = k;  
    match(LPAREN);  
    if (token == RPAREN) {  
        match(RPAREN);  
        return t;  
    } else if (k != NULL) {  
        t->child[1] = args();  
        match(RPAREN);  
    }  
    return t;  
}
```

```
TreeNode* args(void)
```

args(void)函数首先判断后面的 Token 是否为右括号, 如果是则说明是 empty 的情形, 如果不是则为至少有一个 expression 的情形, 然后调用 expression()函数返回节点, 然后一直判断后面的 Token 是否为逗号, 如果是则说明后面还有一个 expression, 则再调用 expression()函数, 使各 expression 返回的节点为兄弟节点, 然后再将第一个 expression 返回

的节点作为函数调用语句参数节点 **ArgsK** 的子节点

```
TreeNode* args(void) {
    TreeNode* t = newNode(ArgsK);
    TreeNode* s = NULL;
    TreeNode* p = NULL;
    if (token != RPAREN) {
        s = expression();
        p = s;
        while (token == COMMA) {
            TreeNode* q;
            match(COMMA);
            q = expression();
            if (q != NULL) {
                if (s == NULL) {
                    s = p = q;
                } else {
                    p->sibling = q;
                    p = q;
                }
            }
        }
    }
    if (s != NULL) {
        t->child[0] = s;
    }
    return t;
}
```

值得一提的是，在为动态创建的节点分配字符串类型成员的值时，由于 C 语言不为字符串自动分配空间，且扫描程序会为它所识别的记号的字符串值（或词法）重复使用相同的空间，所以编写了工具函数 `copyString`，用其取一个字符串参数，然后为拷贝结果分配足够的空间，再拷贝字符串，同时返回一个指向新分配的拷贝的指针。

6. 打印语法树

打印语法树的关键思路在于递归调用打印每一行的函数。而样例上打印的语法树的缩进十分严格，如何控制缩进是个问题。在本次实验中，我将缩进数作为一个全局变量，在搜索树的每个节点的过程中动态增加，并且在结束当前节点访问之后将其减少，这样就可以让每一层函数调用时，全局缩进数变量的值与它的缩进层级一致。

按照之前的分析，按照样例规定的格式遍历每一个子节点与兄弟节点并且判断读取到的 `Token` 是否符合文法，给出对应输出即可。

```
// 递归打印语法树
void printTree(TreeNode* tree) {
    indent_number += 2;
    while (tree != NULL) {
        printSpaces();
        switch (tree->node_kind) {
            case VoidK:
                fprintf(result_file, "VoidK\n");
                break;
            case IntK:
                fprintf(result_file, "IntK\n");
                break;
            case Var_DeclK:
```



```
    fprintf(result_file, "Var_DeclK\n");
    break;
case Arry_DeclK:
    fprintf(result_file, "Arry_DeclK\n");
    break;
case FunK:
    fprintf(result_file, "FuncK\n");
    break;
case ParamsK:
    fprintf(result_file, "ParamsK\n");
    break;
case ParamK:
    fprintf(result_file, "ParamK\n");
    break;
case CompK:
    fprintf(result_file, "CompK\n");
    break;
case Selection_StmtK:
    fprintf(result_file, "If\n");
    break;
case Iteration_StmtK:
    fprintf(result_file, "While\n");
    break;
case Return_StmtK:
    fprintf(result_file, "Return\n");
    break;
case Arry_ElemK:
    fprintf(result_file, "Arry_ElemK\n");
    break;
case CallK:
```

```

        fprintf(result_file, "CallK\n");
        break;
    case ArgsK:
        fprintf(result_file, "ArgsK\n");
        break;
    case AssignK:
        fprintf(result_file, "Assign\n");
        break;
    case OpK:
        fprintf(result_file, "Op: ");
        printToken(tree->attribute.operation, "\0");
        break;
    case ConstK:
        fprintf(result_file, "ConstK: %d\n",
tree->attribute.value);
        break;
    case IdK:
        fprintf(result_file, "IdK: %s\n",
tree->attribute.name);
        break;
    default:
        fprintf(result_file, "Unknown ExpNode kind\n");
        break;
}
for (int i = 0; i < MAX_CHILDREN; i++) {
    printTree(tree->child[i]);
}
tree = tree->sibling;
}

indent_number -= 2;

```

```
}
```

其中用到的 `printSpaces` 是辅助打印缩进的函数，代码很简单：

```
// 工具函数，打印空格缩进
void printSpaces(void) {
    for (int i = 0; i < indent_number; i++) {
        fprintf(result_file, " ");
    }
}
```

7. 错误处理

分析程序对于语法错误的反应通常是编译器使用中的主要问题。在最低限度之下，分析程序应能判断出一个程序在语句构成上是否正确。完成这项任务的分析程序被称作识别程序（`recognizer`），这是因为它的工作是在由正讨论的程序设计语言生成的上下文无关语言中识别串。值得一提的是：任何分析至少必须工作得像一个识别程序一样，即：若程序包括了一个语法错误，则分析必须指出某个错误的存在；反之若程序中没有语法错误，则分析程序不应声称有错误存在。

除了这个最低要求之外，分析程序可以显示出对于不同层次的错误的反应。通常地，分析程序会试图给出一个有意义的错误信息，这至少是针对于遇到的第 1 个错误，此外它还试图尽可能地判断出错误发生的位置。

我们的错误处理程序较为简单，但也需要给出语法错误发生的行号，预期出现的 `Token` 类型与造成错误的 `Token` 是什么。在我的程序中，`match` 函数用于分辨特殊的 `Token`，如果匹配就 `getToken` 继续进行下一个，否则

就代表发生了错误。

如何输出错误呢？这就需要之前编写的 `getTokenString` 函数，通过这个函数我们可以在知道当前预期（`expected`）的 Token 和当前 Token 的值是否一致后，将它的类别标识文字保存下来并输出。之后，再打印一些辅助信息，以及继续调用 `printToken` 函数输出 Token 的细节信息。代码如下所示：

```
// 工具函数，打印错误
void printSyntaxError(char* message) {
    fprintf(result_file, "\n>>> ");
    fprintf(result_file, "Syntax error at line %d: %s",
line_number, message);
}

// 查找特殊 Token 的 Match 过程
void match(TokenType expected) {
    if (token == expected)
        token = getToken();
    else {
        char msg[256] = "expected ";
        char token_tmp[MAX_TOKEN_LENGTH + 1];
        getTokenString(expected, token_tmp);
        strcat(msg, token_tmp);
        strcat(msg, " unexpected token -> ");
        printSyntaxError(msg);
        printToken(token, token_string);
        fprintf(result_file, "      ");
    }
}
```

	<p>五、附录-源代码</p> <p>包括 <code>types.h</code>、<code>main.h</code>、<code>main.c</code>、<code>makefile</code> 四个文件，因为篇幅不再赘述，可以在随实验报告一同提交的压缩文件中的代码文件夹中查看。</p>
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>

```
hwoam@Fred-HP-Laptop: /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:45:19]
$ make clean
rm cparser.exe
rm cparser_debug.exe
rm: 无法删除 'cparser_debug.exe': No such file or directory
make: [makefile:12: clean] 错误 1 (已忽略)
rm cparser.out
rm: 无法删除 'cparser.out': No such file or directory
make: [makefile:13: clean] 错误 1 (已忽略)
rm cparser_debug.out
rm: 无法删除 'cparser_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/Week12/Parser [22:45:24]
$
    printf(result_file, "%s");
}
}

1: bash
e clean
arser.exe
arser_debug.exe
rm: 无法删除 'cparser_debug.exe': No such file or directory
[makefile:12: clean] 错误 1 (已忽略)
```

然后，运行 make 命令编译：

```
hwoam@Fred-HP-Laptop: /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:45:19]
$ make clean
rm cparser.exe
rm cparser_debug.exe
rm: 无法删除 'cparser_debug.exe': No such file or directory
make: [makefile:12: clean] 错误 1 (已忽略)
rm cparser.out
rm: 无法删除 'cparser.out': No such file or directory
make: [makefile:13: clean] 错误 1 (已忽略)
rm cparser_debug.out
rm: 无法删除 'cparser_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/Week12/Parser [22:45:24]
$ make
gcc -c main.c -std=c11 -o file.o;
gcc main.o -o cparser

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:45:58]
$ ls
c1.c- c1.txt c1-sample.txt c2.c- c2-sample.txt cparser.exe main.c main.h main.o makefile types.h

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:46:08]
$ |
clean
arser.exe
arser_debug.exe
rm: 无法删除 'cparser_debug.exe': No such file or directory
[makefile:12: clean] 错误 1 (已忽略)
```

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

```
# hwoam@Fred-HP-Laptop: /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:45:19]
$ make clean
rm cparser.exe
rm cparser_debug.exe
rm: 无法删除 'cparser_debug.exe': No such file or directory
make: [makefile:12:clean] 错误 1 (已忽略)
rm cparser.out
rm: 无法删除 'cparser.out': No such file or directory
make: [makefile:13:clean] 错误 1 (已忽略)
rm cparser_debug.out
rm: 无法删除 'cparser_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/Week12/Parser [22:45:24]
$ make
gcc -c main.c -std=c11 -fno-PIE -fPIE;
gcc main.o -o cparser

# hwoam@Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:45:58]
$ ls
c1.c- c1.txt c1-sample.txt c2.c- c2-sample.txt cparser.exe main.c main.h main.o makefile types.h

# hwoam@Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:46:08]
$ ./cparser
Usage: D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week12\Parser\cparser.exe <filename>

# hwoam@Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:46:58]
C:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week12\Parser>rm cparser_debug.exe
rm: 无法删除 'cparser_debug.exe': No such file or directory
make: [makefile:12:clean] 错误 1 (已忽略)
```

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「**cparser 文件 <filename>**」的格式调用：

然后输入 `./cparser c1.c` 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：

```

rm cparser.exe
rm cparser_debug.exe
rm: 无法删除 'cparser_debug.exe': No such file or directory
make: [makefile:12: clean] 错误 1 (已忽略)
rm cparser.out
rm: 无法删除 'cparser.out': No such file or directory
make: [makefile:13: clean] 错误 1 (已忽略)
rm cparser_debug.out
rm: 无法删除 'cparser_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/Week12/Parser [22:45:24]
$ make
gcc -c main.c -std=c11
gcc main.o -o cparser

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:45:58]
$ ls
c1.c- c1.txt c1-sample.txt c2.c- c2-sample.txt cparser.exe main.c main.h main.o makefile types.h

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:46:08]
$ ./cparser
Usage: D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week12\Parser\cparsen.exe <filename>

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:46:58]
C:\
$ ./cparser c1.c-
The parse analysis result is saved to file c1.txt

# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:49:04]
$

```

如图，程序会提示把词法扫描的结果输出到了 `c1.txt` 文件当中。打开 `c1.txt` 文件：

```

1 CMINUS: PARSE: CRLF
2 CRLF
3 Syntax tree: CRLF
4 FuncK CRLF
5 IntK CRLF
6 IdK: gcd CRLF
7 ParamsK CRLF
8 ParamK CRLF
9 IntK CRLF
10 IdK: u CRLF
11 ParamK CRLF
12 IntK CRLF
13 IdK: v CRLF
14 CompK CRLF

```

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 `c1-sample.txt` 放在了同一目录下：


```
问题 输出 调试控制台 终端
1: bash
hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$ make
gcc -c main.c -std=c11
gcc main.o -o tiny

hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$ ./tiny
Usage: D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 10\Parser\tiny.exe <filename>

hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$ ./tiny t1.tny
The parse analysis result is saved to file t1.txt

hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$ ls
main.c main.h main.o makefile t1.tny t1.txt t1-sample.txt t2.tny t2.txt t2-sample.txt tiny.exe types.h

hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$
```

输入 diff c1.txt c1-sample.txt 比较:

```
hwoam@Fred-HP-Laptop: /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:50:20]
$ diff c1.txt c1-sample.txt
# hwoam @ Fred-HP-Laptop in /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week12/Parser [22:50:23]
$
```

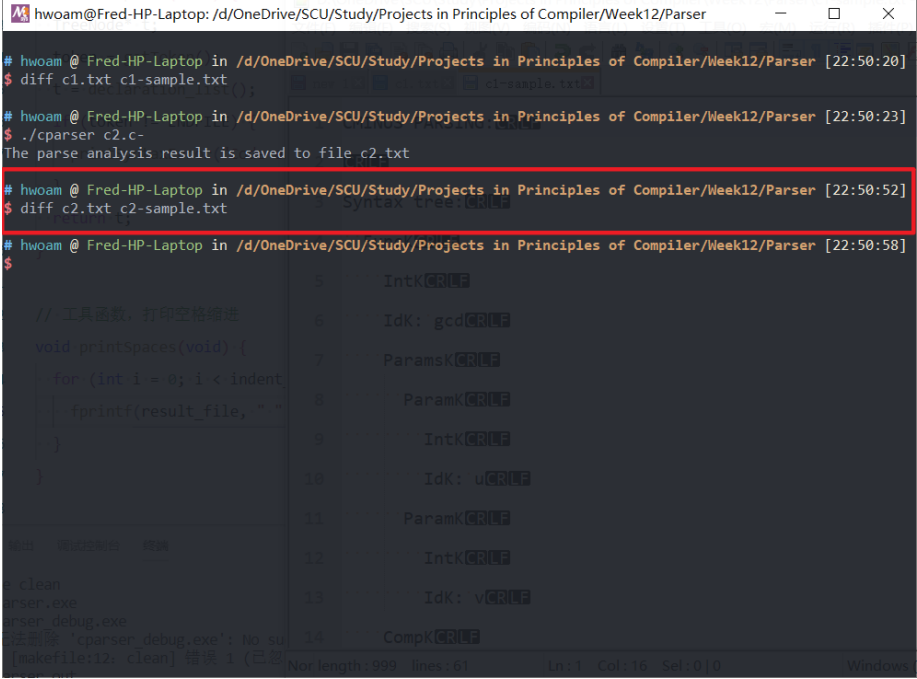
1	printSyntaxError("Code e	2	
2		3	Syntax tree:
3	return t;	4	FuncK
4		5	IntK
5		6	IdK: gcd
6	// 工具函数, 打印空格缩进	7	ParamsK
7	void printSpaces(void) {	8	ParamK
8	for (int i = 0; i < indent	9	IntK
9	fprintf(result_file, " "	10	IdK: u
10	}	11	ParamK
11		12	IntK
12		13	IdK: v
13		14	CompK

read_stmt(void) 行 589, 列 12 空格 2 UTF-8 CRLF C Win32

clean
arser.exe
arser_debug.exe
无法将 'c:\parser_debug.exe': No su
[makefile:12: clean] 错误 1 (已忽略)

Windows

diff 的输出为空, 说明两个文件完全一致。以此类推, 我们再测试 c2 的结果是否正确:

	<div data-bbox="326 211 1247 887"></div> <p>输出结果和样例完全一致，两个样例全部测试通过，实验成功！</p>
小 结	<p>1. EBNF 的转换与左递归、左因子的处理</p> <p>由于 C-语言给出的文法有左递归存在,于是自己将存在左递归的文法改写成 EBNF 的形式,并据此进行代码编写。由于在编写代码的过程中需要确定分析是否正确或选择多个文法中的某一个文法进行分析,有时必须探测需要的或下一个 Token 的类型,在这种情况下需要要求 First 集合,在推导中若存在 empty,又需要要求 Follow 集合,所以这样又需要我了解 First 集合和 Follow 集合,自己在程序中也根据求出的 First 集合和 Follow 集合进行判断,以确定程序的走向。在编写过程中,还有一类问题,就是存在公共左因子,如文法 $expression \rightarrow var = expression \mid simple-expression$,左因子为 ID,在分析过程中,由于已经取出了一个 ID 的 Token,且生成了一个 IdK 的节点,但是在当前状态无法确定是哪一个推导,然而 IdK 节点已经生成,又无法回退,并且是使用自顶向下的分析方法,已经生成的 IdK 在程序上方无法使用,自己通过查阅资料等途径的学习确定了在这种情形下的处理方式:</p>

	<p>将已经生成的 <code>IdK</code> 节点传到下方的处理程序，所以 <code>TreeNode * simple_expression(TreeNode * k)</code>、<code>TreeNode * additive_expression(TreeNode * k)</code>等函数都被设计成有节点类型参数的函数，目的就是为将已经生成的节点传到下面的分析函数中去。</p> <p>2. 错误提示</p> <p>常见的各种编译器在报错时会输出预期接收到的 <code>Token</code>。我利用 <code>strcat</code> 字符串拼接函数，在提取出 <code>Token</code> 对应的文字说明后将其与错误消息拼接在一起输出，实现了同样的功能。</p> <p>3. 完成了实验的任务，实现了编译器的词法分析和语法分析阶段的功能，词法分析主要能过滤注释、分析出语法分析阶段需要的 <code>Token</code> 并满足语法阶段的所有要求，能够判别词法分析阶段是否出错和出错类型和位置。语法分析主要能根据递归向下的分析思想和 C-文法对词法分析获取的 <code>Token</code> 进行语法分析，能够构造出语法树，能够判别语法分析过程中是否出错以及出错位置和错误类型。</p> <p>4. 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>5. 在实验过程中，我设计数据类型、数据结构与其他代码，努力让它们可以正常编译、运行、调试，这是过去没有尝试过的事情，让我感觉到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 师评 议	<div>成绩评定：</div> <div>指导教师签名：</div>