

# 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 语言的语法分析器	实验时间	2019 年 5 月 7 日、2019 年 5 月 14 日
实验目的	<ol style="list-style-type: none"><li>1. 熟悉 <b>tiny</b> 语言语法, 选择实现方法</li><li>2. 设计数据类型、数据结构</li><li>3. 用 C 或 C++实现 <b>tiny</b> 语言的语法分析器</li><li>4. 调试、运行</li><li>5. 提交实验报告</li></ol>		
实验环境	<ul style="list-style-type: none"><li>➤ <b>设备:</b> Microsoft Surface Pro 5 i5 8+256G</li><li>➤ <b>操作系统:</b> Windows 10 64 位</li><li>➤ <b>Linux 模拟环境:</b> MSYS2 MinGW 64 (mintty) (和在 Linux 下操作是完全一样的)</li><li>➤ <b>Shell:</b> zsh</li><li>➤ <b>编程语言:</b> C Language</li><li>➤ <b>编译器:</b> gcc.exe (Rev2, Built by MSYS2 project) 8.3.0</li><li>➤ <b>Make:</b> GNU Make 4.2.1 (为 x86_64-pc-msys 编译)</li><li>➤ <b>文本编辑器:</b> VSCode 作为编辑器, 文本文件采用 utf-8 编码</li></ul>		

实验内容

本次实验使用递归下降法构造 Tiny 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、Tiny 语言语法的特点

在之前的词法分析实验中，我们了解到了 Tiny 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 Tiny 语言的语句，而描述这些语句的文法，就是 Tiny 语言的语法。Tiny 的语法特点可以用 EBNF（扩展巴科斯-诺尔范式）表示如下：

---

```
program → stmt-sequence
stmt-sequence → statement { ; statement }
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt → if exp then stmt-sequence [ else stmt-sequence ] end
repeat-stmt → repeat stmt-sequence until exp
assign-stmt → identifier := exp
read-stmt → read identifier
write-stmt → write exp
exp → simple-exp [ comparison-op simple-exp ]
comparison-op → < | =
simple-exp → term { addop term }
addop → + | -
term → factor { mulop factor }
mulop → * | /
factor → ( exp ) | number | identifier
```

---

### 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

// 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

// 定义语法树的最大子节点数量
#define MAX_CHILDREN 3
```

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}  
};
```

5. 在语法树中，节点从整体上可以分为两种，一种是语句类型的节点，另外一种为表达式类型的节点。而继续往下细分，语句类型节点又可以分为 **If** 语句、**Repeat** 语句、**Assign** 语句、**Read** 语句和 **Write** 语句五种，表达式类型节点则可以分为操作符（Op）表达式、常量（Const）表达式和标识符（Id）表达式三种。因此，为了描述这个两层的分类结构，需要创建三种枚举：**NodeKind**、**StatementKind** 和 **ExpressionKind**，它们的具体定义如下：

```
// 节点类型
typedef enum NodeKind { StmtK, ExpK } NodeKind;

// 语句节点的小类型
typedef enum StatementKind {
    IfK,
    RepeatK,
    AssignK,
    ReadK,
    WriteK
} StatementKind;

// 表达式节点的小类型
typedef enum ExpressionKind { OpK, ConstK, IdK }
ExpressionKind;
```

6. 每个语法树节点包括子节点、同属连接产生的兄弟节点、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 kind {
        StatementKind statement;
        ExpressionKind expression;
    } kind;
    union attribute {
        TokenType operation;
        int value;
        char* name;
    } attribute;
} 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 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;
// 存放当前 token 的字符串
char token_string[MAX_TOKEN_LENGTH + 1] = "";
```

在语法分析时，需要记录下当前的 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。因此需要两个全局变量处理：

```
// 记录当前的 Token
TokenType token;
// 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, ".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, ".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 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, "TINY 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 `tny` 源文件内容的，所以首先要初始化一些变量，如下所示：

```
// 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, INASSIGN, INCOMMENT, INNUM,
// INID, DONE 六种
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 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);
}
}

```

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 THEN:
    strcpy(string, "THEN");
    break;
case ELSE:
    strcpy(string, "ELSE");
    break;
case END:
    strcpy(string, "END");
    break;
case REPEAT:
    strcpy(string, "REPEAT");
    break;
case UNTIL:
    strcpy(string, "UNTIL");
    break;
case READ:
    strcpy(string, "READ");
    break;
case WRITE:
    strcpy(string, "WRITE");
    break;
case ASSIGN:
    strcpy(string, "ASSIGN");
    break;
case LT:
    strcpy(string, "LT");
    break;
```

```
case EQ:
    strcpy(string, "EQ");
    break;
case LPAREN:
    strcpy(string, "LPAREN");
    break;
case RPAREN:
    strcpy(string, "RPAREN");
    break;
case SEMI:
    strcpy(string, "SEMI");
    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 ENDFILE:
    strcpy(string, "EOF");
    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，而节点分为语句节点和表达式节点两大类。因此，需要编写返回这两大类节点的生成函数，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：

```

// 创建新的语句节点
TreeNode* newStmtNode(StatementKind kind) {
    TreeNode* t = (TreeNode*)malloc(sizeof(TreeNode));
    int i;
    if (t == NULL)
        fprintf(result_file, "Out of memory error at
line %d\n", line_number);
    else {
        for (i = 0; i < MAX_CHILDREN; i++) {
            t->child[i] = NULL;
        }
    }
}

```



```

    t->sibling = NULL;
    t->node_kind = StmtK;
    t->kind.statement = kind;
    t->line_number = line_number;
}
return t;
}

// 创建新的表达式节点
TreeNode* newExpNode(ExpressionKind kind) {
    TreeNode* t = (TreeNode*)malloc(sizeof(TreeNode));
    int i;
    if (t == NULL)
        fprintf(result_file, "Out of memory error at
line %d\n", line_number);
    else {
        for (i = 0; i < MAX_CHILDREN; i++) {
            t->child[i] = NULL;
        }
        t->sibling = NULL;
        t->node_kind = ExpK;
        t->kind.expression = kind;
        t->line_number = line_number;
    }
    return t;
}

```

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 = stmt_sequence();
    if (token != ENDFILE) {
        printSyntaxError("Code ends before file\n");
    }
    return t;
}
```

## 5. EBNF

EBNF 部分由 11 个相互递归的函数组成，他们与前文给出的 Tiny 语言的 EBNF 文法直接对应：

- 一个对应于 `stmt-sequence`，为 `stmt_sequence(void)`
- 一个对应于 `satement`，为 `statement(void)`
- 5 个分别对应于 5 种不同的语句
  - If 为 `if_stmt(void)`
  - Repeat 为 `repeat_stmt(void)`
  - Assign 为 `assign_stmt(void)`
  - Read 为 `read_stmt(void)`
  - Write 为 `write_stmt(void)`
- 4 个对应于表达式的不同优先层次。操作符非终结符并未包括到过程之中，但却作为与它们相关的表达式被识别。
  - Exp 为 `expression(void)`
  - Simple\_exp 为 `simple_expression(void)`
  - Term 为 `term(void)`
  - Factor 为 `factor(void)`

注意，这里没有函数与 `program` 相对应，这是因为一个程序就是一个语句序列，所以 `parse` 函数仅调用 `stmt-sequence`。因为只是 EBNF 到 C 语言的翻译，因此不做过多赘述，仅举例说明：

例如，在 `stmt_sequence` 中，会将代码按照语句切割，分隔符是分号（SEMI）。ENDFILE 和 END 表示文件已经完毕，不用再读取。而 ELSE 和 UNTIL 这两个 Token，一定在 `if_stmt` 和 `repeat_stmt` 中会被完整匹配，不可能独立出现，因此如果 ELSE 或 UNTIL 独立出现了，也说明程序出错了。这样异常中断会被 `parse` 函数后面的语句检测到。之后就是同属连接各个节点的过程，在注释中已经说明。

再比如说，If 语句的 `exp` 位于 `child[0]` 位置，`then` 的那一部分放在 `child[1]`，是一个新的语句节点。而 `else` 的那一部分放在 `child[1]` 中，也是一个新的语句节点。整个匹配的过程就是用 `match` 匹配 IF，看看能不能匹配到，然后给 `child[0]` 赋值，再看看后面有没有 THEN，如果有的话就给 `child[1]` 再赋值。最后这个地方是关键，因为 `else` 部分本身是可选的，

所以这里先检测下一个 **token** 是不是 **ELSE**，再判断用不用去用 **match** 匹配 **ELSE**（否则就直接匹配 **END** 了）。

这里存在的一个问题是“悬挂 **else**”问题，Tiny 语言的 **then** 部分和 **else** 部分都是同一个 **stmt-sequence**，没有区分的方法。因此，就需要补充 **if** 的语法解决悬挂 **else** 的问题。Tiny 语言采用的解决方案是给 **if** 加一个 **END Token** 做结束标志。这样每一次 **if-stmt** 的匹配都一定会以 **END** 结束，否则就报错了，这样就会让 **if** 语句即时结束，不会与和自己不配套的 **else** 发生联系（二义性），避免了悬挂 **else** 问题。

与之形成对比的是类似的 **repeat-stmt** 语句。他们的 **body** 实际上只有 **repeat** 和 **until** 两个 **token** 的中间的语句，也不存在可选的情况，这让每一个 **repeat** 之后，属于它的 **until** 一定会被即时匹配，所以也不存在“悬挂 **until**”的情况。

将节点对应的内存地址返回，该节点构建完成。之后打印语法树结构时可以顺着子节点和同属连接的兄弟节点将完整的语法书全部访问到。

11 个函数的全部代码如下：

```
// 下面的函数全部对应 EBNF

TreeNode* stmt_sequence(void) {
    TreeNode* t = statement();
    TreeNode* p = t;
    // 将代码按照语句切割，分隔符是分号（SEMI）
    // ENDFILE 和 END 表示文件已经完毕，不用再读取
    // ELSE 和 UNTIL 这两个 Token，一定在 if_stmt 和 repeat_stmt
    中会被完整匹配，不可能独立出现
    // 因此如果 ELSE 或 UNTIL 独立出现了，也说明程序出错了
    while ((token != ENDFILE) && (token != END) &&
        (token != ELSE) &&
        (token != UNTIL)) {
```

```

    TreeNode* q;
    // 分号分割
    match(SEMI);
    q = statement();
    if (q != NULL) {
        if (t == NULL) {
            // 如果前一个节点不是语句类节点，报错了，没关系，从下
            一个 Token 重新开始

            t = p = q;
        } else {
            // 同属连接，不断循环
            p->sibling = q;
            p = q;
        }
    }
}

return t;
}

TreeNode* statement(void) {
    TreeNode* t = NULL;
    switch (token) {
        case IF:
            t = if_stmt();
            break;
        case REPEAT:
            t = repeat_stmt();
            break;
        case ID:
            t = assign_stmt();

```

```

        break;
    case READ:
        t = read_stmt();
        break;
    case WRITE:
        t = write_stmt();
        break;
    default:
        printSyntaxError("unexpected token -> ");
        printToken(token, token_string);
        token = getToken();
        break;
    }
    return t;
}

```

```

TreeNode* if_stmt(void) {
    TreeNode* t = newStmtNode(IfK);
    match(IF);
    if (t != NULL) {
        t->child[0] = expression();
    }
    match(THEN);
    if (t != NULL) {
        t->child[1] = stmt_sequence();
    }
    if (token == ELSE) {
        match(ELSE);
        if (t != NULL) {
            t->child[2] = stmt_sequence();
        }
    }
}

```

```

    }
}
match(END);
return t;
}

TreeNode* repeat_stmt(void) {
    TreeNode* t = newStmtNode(RepeatK);
    match(REPEAT);
    if (t != NULL) {
        t->child[0] = stmt_sequence();
    }
    match(UNTIL);
    if (t != NULL) {
        t->child[1] = expression();
    }
    return t;
}

TreeNode* assign_stmt(void) {
    TreeNode* t = newStmtNode(AssignK);
    if ((t != NULL) && (token == ID)) {
        t->attribute.name = copyString(token_string);
    }
    match(ID);
    match(ASSIGN);
    if (t != NULL) {
        t->child[0] = expression();
    }
    return t;
}

```

```
}
```

```
TreeNode* read_stmt(void) {  
    TreeNode* t = newStmtNode(ReadK);  
    match(READ);  
    if ((t != NULL) && (token == ID)) {  
        t->attribute.name = copyString(token_string);  
    }  
    match(ID);  
    return t;  
}
```

```
TreeNode* write_stmt(void) {  
    TreeNode* t = newStmtNode(WriteK);  
    match(WRITE);  
    if (t != NULL) {  
        t->child[0] = expression();  
    }  
    return t;  
}
```

```
TreeNode* expression(void) {  
    TreeNode* t = simple_expression();  
    if ((token == LT) || (token == EQ)) {  
        TreeNode* p = newExpNode(OpK);  
        if (p != NULL) {  
            p->child[0] = t;  
            p->attribute.operation = token;  
            t = p;  
        }  
    }
```



```

        match(token);
        if (t != NULL) {
            t->child[1] = simple_expression();
        }
    }
    return t;
}

TreeNode* simple_expression(void) {
    TreeNode* t = term();
    while ((token == PLUS) || (token == MINUS)) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->child[0] = t;
            p->attribute.operation = token;
            t = p;
            match(token);
            t->child[1] = term();
        }
    }
    return t;
}

TreeNode* term(void) {
    TreeNode* t = factor();
    while ((token == TIMES) || (token == OVER)) {
        TreeNode* p = newExpNode(OpK);
        if (p != NULL) {
            p->child[0] = t;
            p->attribute.operation = token;

```

```

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

TreeNode* factor(void) {
    TreeNode* t = NULL;
    switch (token) {
        case NUM:
            t = newExpNode(ConstK);
            if ((t != NULL) && (token == NUM)) {
                t->attribute.value = atoi(token_string);
            }
            match(NUM);
            break;
        case ID:
            t = newExpNode(IdK);
            if ((t != NULL) && (token == ID)) {
                t->attribute.name = copyString(token_string);
            }
            match(ID);
            break;
        case LPAREN:
            match(LPAREN);
            t = expression();
            match(RPAREN);
            break;
    }
}

```

```
default:
    printSyntaxError("unexpected token -> ");
    printToken(token, token_string);
    token = getToken();
    break;
}
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，节点分为两大类，两大类下又有小类。因此，可以用 `if` 语句判断这种二元的大分类，然后在小分类中用 `switch` 处理。最终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) {
```

```
    int i;
```

```
    indent_number += 2;
```

```
    while (tree != NULL) {
```

```
        printSpaces();
```

```
        if (tree->node_kind == StmtK) {
```

```
            switch (tree->kind.statement) {
```

```
                case IfK:
```

```
                    fprintf(result_file, "If\n");
```

```
                    break;
```

```
                case RepeatK:
```

```
                    fprintf(result_file, "Repeat\n");
```

```
                    break;
```

```
                case AssignK:
```

```
                    fprintf(result_file, "Assign to: %s\n",  
tree->attribute.name);
```

```
                    break;
```

```
                case ReadK:
```

```
                    fprintf(result_file, "Read: %s\n",  
tree->attribute.name);
```

```
                    break;
```

```
                case WriteK:
```

```
                    fprintf(result_file, "Write\n");
```

```
                    break;
```

```
            default:
```

```
                fprintf(result_file, "Unknown ExpNode kind\n");
```

```
                break;
```

```
        }
```

```
    } else if (tree->node_kind == ExpK) {
```

```

switch (tree->kind.expression) {
    case OpK:
        fprintf(result_file, "Op: ");
        printToken(tree->attribute.operation, "\0");
        break;
    case ConstK:
        fprintf(result_file, "Const: %d\n",
tree->attribute.value);
        break;
    case IdK:
        fprintf(result_file, "Id: %s\n",
tree->attribute.name);
        break;
    default:
        fprintf(result_file, "Unknown ExpNode kind\n");
        break;
}
} else
    fprintf(result_file, "Unknown node kind\n");
for (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

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, "      ");
    }
}
```

## 五、附录-源代码

包括 types.h、main.h、main.c、makefile 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。

## 实验结果

首先, 运行 `make clean` 做一点清理工作, 把旧的中间文件删除 (已经考虑到了 `.exe` 和 `.out`, 以及 `.obj` 和 `.o` 文件可能在不同系统下生成的差异):

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

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

然后, 运行 `make` 命令编译:

```
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)

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
$
```

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

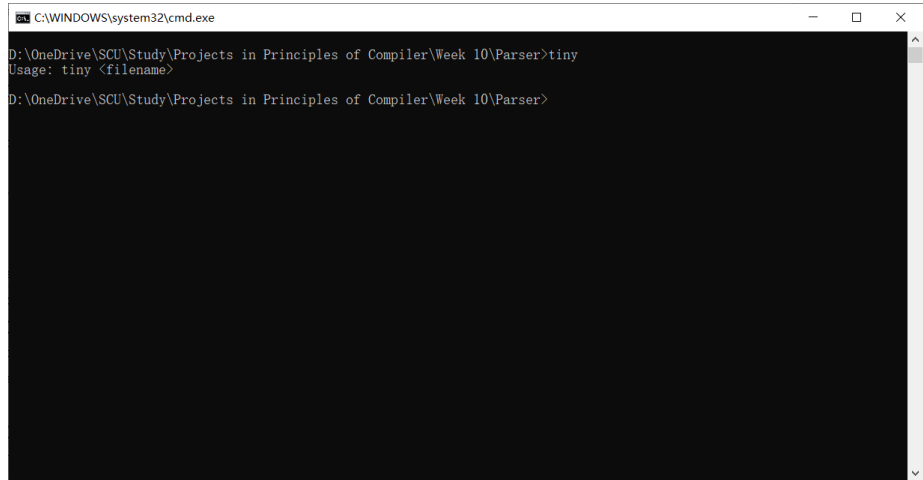
```
rm tiny_debug.out
rm: 无法删除 'tiny_debug.out': No such file or directory
make: [makefile:14: clean] 错误 1 (已忽略)
rm main.o
rm main.obj
rm: 无法删除 'main.obj': No such file or directory
make: [makefile:16: clean] 错误 1 (已忽略)

hwoam@Fred-HP-Laptop /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
$
```

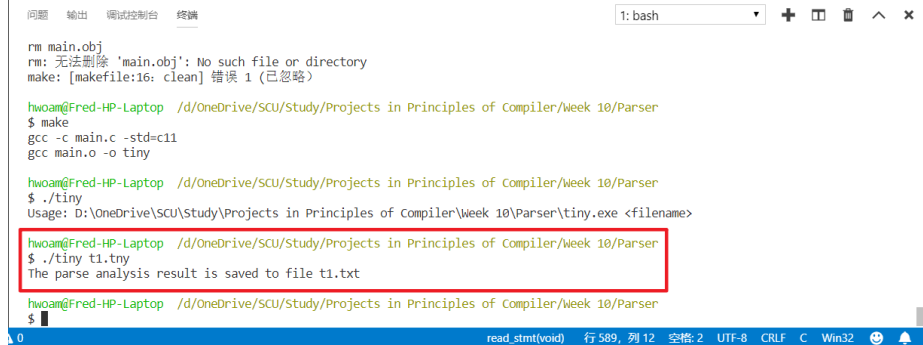




```
C:\WINDOWS\system32\cmd.exe
D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 10\Parser>tiny
Usage: tiny <filename>
D:\OneDrive\SCU\Study\Projects in Principles of Compiler\Week 10\Parser>
```

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 输出 调试控制台 终端
1: bash
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 /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
$
```

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 10\Parser\t1.txt - Notepad++
文件(E) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(I) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
mingw32.ini nsswitch.conf .zshrc titles.txt pos_result.txt t2.txt t2-sample.txt
1 TINY PARING: CRLF
2 CRLF
3 Syntax tree: CRLF
4 Read: x CRLF
5 If CRLF
6 Op: < CRLF
7 Const: 0 CRLF
8 Id: x CRLF
9 Assign to: fact CRLF
10 Const: 1 CRLF
11 Repeat CRLF
12 Assign to: fact CRLF
13 Op: * CRLF
14 Id: fact CRLF
Nor length: 357 lines: 25 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 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 t1.txt t1-sample.txt` 比较：

```
问题 输出 调试控制台 终端
1: bash
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
$ diff t1.txt t1-sample.txt
hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$
```

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 t2 的结果是否正确：

```
问题 输出 调试控制台 终端
1: bash
$ ./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
$ diff t1.txt t1-sample.txt
hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$ ./tiny t2
The parse analysis result is saved to file t2.txt
hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$ diff t2.txt t2-sample.txt
hwoam@Fred-HP-Laptop /d/OneDrive/SCU/Study/Projects in Principles of Compiler/Week 10/Parser
$
```

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！

## 小结

### 1. 函数与变量的命名

一开始，我将对应 `exp` 的函数直接命名为了 `exp`，结果报错了，说是 `exp` 本身就是 C 语言的常用内置函数，存在命名冲突。因此我将它改名为 `expression`，规避掉了这个问题。

### 2. 不输出多余的词法扫描内容

在原版的程序中，为了方便学习，打开对应开关变量后，会先输出词法扫描内容，再输出语法分析内容。本次实验只需要语法分析内容的输出。

### 3. 比原版增强的错误提示

原版的程序中，只会提示遇到的错误 `Token` 内容是什么，但没有像常见的各种编译器那样输出预期接收到的 `Token`。我利用 `strcat` 字符串拼接函数，在提取出 `Token` 对应的文字说明后将其与错误消息拼接在一起输出。

	<p>4. 学习了 Tiny 语言分割语句的方式，掌握了语句节点与语句节点之间同属连接的原则，也掌握了语句和语句内部的语句之间子连接的方式，增强了对程序转化为语句这个过程的理解。</p> <p>5. 学习了 Tiny 语言处理“悬挂 else”的方式，增强了对语句二义性处理的理解。</p> <p>6. 重温了 makefile 文件的编写，复习了 C 语言，回顾了 diff、cat 等命令的使用和 vim、gcc 等程序的使用，巩固了已有的知识。</p> <p>7. 熟悉了 Tiny 语言语法的特点，并根据 Tiny 语言语法的 EBNF 构造递归下降程序建立语法树，使其可以识别 Tiny 语言的语法。在这个过程中，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 师评 议	<p>成绩评定：</p> <p>指导教师签名：</p>