语法分析实验报告

- 姓名：胡泽钊
- 学号：18327034
- 专业：软件工程
- 年级：2019 级

---

0. 前言

在进行这一个阶段的工作的时候，发现之前写的词法分析的接口完全用不了，比如之前做词法分析的时候，是把所有 `token` 一次性扫描出来，然后写入一个文件中。但是写语法分析的时候，需要一个个将 `token` 扫描出来，进行语法解析，而不能一次性将 `token` 扫描出来，这也就为这个阶段的工作带来很大的不方便。

所以就需要将前面的词法分析从头开始写了，这也耗费了我很多的时间，为了能够进一步提高了效率，我找到了 `编译原理及其实践` 这本书，从这本书上面找到 `TINY` 词法和语法的定义，书中也给了很多写词法分析器和语法分析器的思路，接下来将先描述 `TINY+` 的 `EBNF` 语法，然后先展示一下程序结果，再描述程序思路。

一开始写词法分析的时候，做了一个 `GUI` 的程序页面，但是后面电脑重装了，重新配置 `GTK` 库浪费了大量的时间，而且使用 `GTK3` 也很不方便，所以本次语法分析舍去了 `GUI` 页面，专注于代码逻辑。

1. TINY+ 语言的 EBNF 语法

1.0 符号描述

一开始也找了很多 `TINY` 的 `EBNF` 语法的描述，看的也是云里雾里，后面才发现，大多数 `TINY` 的语法描述和我们在《龙书》中描述的语法不太一样，那么这里就专门用一小节来解释其中的语法定义：

```
stmt_sequence -> statement {; statement}
```

上述语法表示**语句序列(stmt_sequence)**，由多条**语句(statement)**构成，其中大括号里面的内容表示可以**多次添加**，即你可以写成如下形式：

```
statement1;
statement2;
statement3
```

注意，在这个定义下**最后一条语句不用带分号**，如只需要一条语句的时候，应该写成下面的形式：

```
x := 1
```

而不能写成下面的形式：

```
x := 1;
```

然后，让我们来看 `if` 语句的语法：

```
if-stmt -> if exp then stmt_sequence [else stmt_sequence] end
```

注意，和大括号不同，中括号里面的内容表示，只能添加一次或者不添加，也就是说对于 `if` 语句只有以下两种形式：

```
if exp then stmt_sequence end
或
if exp then stmt_sequence else stmt_sequence end
```

而没有以下形式：

```
if exp then stmt_sequence else stmt_sequence else stmt_sequence end
```

可能大家会觉得上面说的是废话，大家都明白 `if` 只能接 `else` ，但是明确这个符号定义，有助于我们后面程序的编写，所以这里才需要专门拿出来说。

1.1 顶层语法

修改后的产生式，都会在后面加上注释，如下面的 `program` 产生式就有所不同，没有加注释的产生式，就是没有修改的：

```
program -> stmt_sequence (TINY 语法)
修改为
program -> declarations stmt_sequence (TINY+ 语法)

添加 declaration -> decl {;, declarations} (TINY+ 语法)
添加 decl -> type-specifier varlist (TINY+ 语法)
添加 type-specifier -> int | bool | char (TINY+ 语法)
添加 varlist -> identifier {,identifier} (TINY+ 语法)
stmt_sequence -> statement {; statement}
```

1.2 语句语法

```
statement -> if-stmt|repeat-stmt|assign-stmt|read-stmt|write-stmt(TINY 语法)
修改为
statement -> if-stmt|repeat-stmt|assign-stmt|read-stmt|write-stmt|while-
stmt(TINY+ 语法)

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
while_stmt -> while bool_exp do stmt_sequence end (TINY+ 语法)
```

1.3 表达式语法

```
exp -> simple_exp [comparison-op simple-exp]

comparison-op -> < | = (TINY 语法)
修改为
comparison-op -> < | = | <= | > | >=  (TINY+ 语法)

simple_exp -> term {addop term}
addop -> + | -
term -> factor {mulop factor}
mulop -> * | /
factor -> (exp) | number | identifier
```

1.4 TINY+ 语法总览

可以看到我们修改的地方有以下几个部分：

- 在使用变量前必须要声明变量。
- 增加了声明部分的产生式。
- 增加了变量类型部分的产生式。
- 增加了变量序列部分的产生式。
- 增加了 `while` 运算的产生式。
- 完善了比较运算的运算符号集。

完整的 `TINY+` 的 `EBNF` 产生式如下所示：

```
program -> declarations stmt_sequence
declaration -> decl {;, declarations}
decl -> type-specifier varlist
type-specifier -> int | bool | char
varlist -> identifier {,identifier}
stmt_sequence -> statement {; statement}

statement -> if-stmt|repeat-stmt|assign-stmt|read-stmt|write-stmt|while-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
while_stmt -> while bool_exp do stmt_sequence end

exp -> simple_exp [comparison-op simple-exp]
comparison-op -> < | = | <= | > | >=
simple_exp -> term {addop term}
addop -> + | -
term -> factor {mulop factor}
mulop -> * | /
factor -> (exp) | number | identifier
```

2. 程序使用方法

在终端中输入以下指令编译 (请勿使用文件夹中的 makefile 文件，那个是后续完善使用的)：

```
gcc main.c scan.c util.c parse.c
```

之后即可得到编译后的软件 `a.exe`，之后在终端使用以下命令：

```
a.exe [你的 tiny 文件的路径]
```

即可看到最后的程序运行结果。

---

3. 运行效果

下述是我用来测试 `tiny` 源代码：

```
{ Sample program
  in TINY language
}
char x,y,z;
int a,b,c;
bool d,e
read x;
if 0 < x then { don't compute if x <= 0 }
  while 0 <= x do
    x := 1
  end;
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact  { output factorial of x }
end
```

产生的语法树：

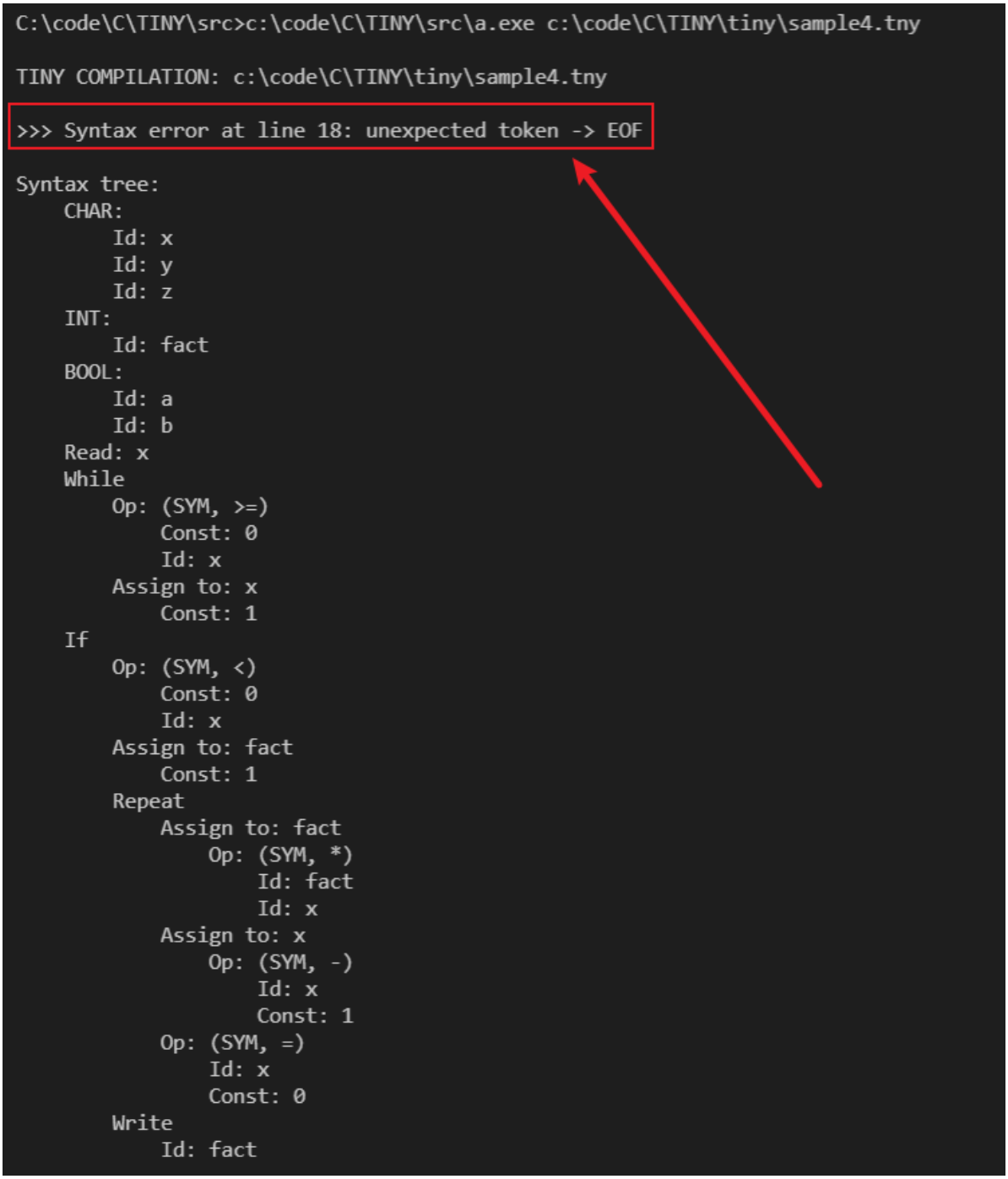
```
C:\code\C\TINY\src>c:\code\C\TINY\src\a.exe c:\code\C\TINY\tiny\sample4.tny

Syntax tree:
    CHAR:
        Id: x
        Id: y
        Id: z
    INT:
        Id: fact
    BOOL:
        Id: a
        Id: b
    Read: x
    While
        Op: (SYM, >=)
            Const: 0
            Id: x
        Assign to: x
            Const: 1
    If
        Op: (SYM, <)
            Const: 0
            Id: x
        Assign to: fact
            Const: 1
        Repeat
            Assign to: fact
                Op: (SYM, *)
                    Id: fact
                    Id: x
            Assign to: x
                Op: (SYM, -)
                    Id: x
                    Const: 1
            Op: (SYM, =)
                Id: x
                Const: 0
        Write
            Id: fact

C:\code\C\TINY\src>
```

可以看到我们程序正确的识别了预定义的变量 `x`，`y`，`z`， `fact` ， `a` ， `b`，并且能够正确地解析出 `while` 语句，可以看出我们的 `TINY+` 语法是正确的。

如果我们特意删除一个 `end` ，解析出来的结果如下：



可以看到，我们的程序敏锐的指出我们程序的错误，它指出我们的错误在程序的第 18 行，那正是我们删除 `EOF` 的位置。可以看出我们的程序是有**一定的纠错能力的。**

---

4. 程序思路

其实整个程序一旦确定好思路，写起来就很简单，我们设计的语法树的结构如下：

```
typedef struct treeNode {
    /* 每一个节点都有若干个孩子节点 */
    struct treeNode * child[MAXCHILDREN];
    /* 每一个节点还会记录兄弟节点 */
    struct treeNode * sibling;

    /* 行号，该节点对应源代码中的哪一行 */
    int lineno;

    /* 节点类型 */
```

```
    NodeKind nodekind;
    /* 详细的节点类型 */
    union {
        StmtKind stmt;
        ExpKind exp;
        DeclKind decl;
    } kind;

    /* 属性 */
    union {
        TokenType op;
        /* 假如要输出 Const: 0，就必须要用 val 记录下 0 */
        int val;
        /* 假如要输出 Assign to: fact，就必须要用 name 来记录下 fact */
        char * name;
    } attr;

    /* 用于表达式的类型检查 */
    ExpType type;
} TreeNode;
```

我们设计的语法树是一个多叉树的结构，它有指针指向它的子节点，也有指针指向它的兄弟节点，指向兄弟节点的指针能够让它输出完自己的子节点以后，就开始输出兄弟节点的数据。

在明确好数据结构以后，后面的工作就很简单了，我们这里以 `if` 产生式为例：

```
/* if_stmt -> if exp then stmt_sequence [else stmt_sequence] end */
TreeNode *if_stmt(void) {
    /* 产生一个新的节点 */
    TreeNode *t = newStmtNode(IfK);

    /* 进行匹配，并移动 token */
    match(IF);

    if (t != NULL) {
        /* if 语句第一个孩子节点必须是表达式 */
        t->child[0] = expr();
    }

    /* 进行匹配，并移动 token */
    match(THEN);

    if (t != NULL) {
        /* if 语句第二个孩子节点必须是语句 */
        t->child[1] = stmt_sequence();
    }

    /* 检测是否有 ELSE */
    if (token==ELSE) {
        match(ELSE);
        if (t!=NULL){
            /* 在有 ELSE 的情况下， if 语句第三个孩子节点必须是语句 */
            t->child[2] = stmt_sequence();
        }
    }

    /* 进行匹配，并移动 token */
```

```
    match(END);
    return t;
}
```

可以看到 `if` 产生式对应的函数工作过程如下所示：

- 先通过 `newStmtNode` 函数创建一个新的语句节点 `t`。
- 使用 `match` 函数检查当前的 `token` 是否匹配 `IF`
  - 若不匹配，则会报错。
  - 若匹配，则程序继续，`token` 被赋值为下一个 `tokenType`。
- 设置 `if` 节点的第一个孩子节点为表达式节点(`expr()`)。
- 使用 `match` 函数检查当前的 `token` 是否匹配 `THEN`
  - 若不匹配，则会报错。
  - 若匹配，则程序继续，`token` 被赋值为下一个 `tokenType`。
- 设置 `if` 节点的第二个孩子节点为语句序列节点(`stmt_sequence()`)。
- 检查现在的 `token` 是否为 `ELSE`
  - 若为 `ELSE`，则匹配 `ELSE`，并将 `if` 节点的第三个节点设置为语句序列节点(`stmt_sequence()`)。
  - 若不为 `ELSE`，则继续执行。
- 使用 `match` 函数检查当前的 `token` 是否匹配 `END`
  - 若不匹配，则会报错。
  - 若匹配，则程序继续，`token` 被赋值为下一个 `tokenType`。
- 返回我们最初创建好的 `if` 节点 `t`。

其他的产生式实现方法和 `if` 类似，这里就不再赘了。