实验三：语法分析实验

1.TINY 语言的语法

TINY 语言的语法已经由文档给出：

Program -> MethodDecl MethodDecl*
MethodDecl -> Type [MAIN] Id '(' FormalParams ')' Block
FormalParams -> [FormalParam (',' FormalParam)*]
FormalParam -> Type Id
Type -> INT | REAL
Block -> BEGIN Statement* END
Statement -> Block| LocalVarDecl | AssignStmt | ReturnStmt | IfStmt| WriteStmt| ReadStmt
LocalVarDecl -> INT Id ';' | REAL Id ';'
AssignStmt -> Id := Expression ';'
ReturnStmt -> RETURN Expression ';'
IfStmt -> IF '(' BoolExpression ')' Statement | IF '(' BoolExpression ')' Statement ELSE Statement
WriteStmt -> WRITE '(' Expression ',' QString ')' ';'
ReadStmt -> READ '(' Id ',' QString ')' ';'
Expression -> MultiplicativeExpr (('+' | '-') MultiplicativeExpr)*
MultiplicativeExpr -> PrimaryExpr (('*' | '/') PrimaryExpr)*
PrimaryExpr -> Num| Id | '(' Expression ')' | Id '(' ActualParams ')'
BoolExpression -> Expression '==' Expression|Expression '!=' Expression
ActualParams -> [Expression (',' Expression)*]

2. 语法分析程序的构造

该程序是基于 YACC 构造的，LALR 文法分析的部分由 YACC 生成。

为了构造出语法树，我在本实验中将对语法树的操作作为语义规则的一部分，使得语法树在翻译的过程中就被构造出来。

2.1 语法树的结构

考虑到语法树中一个节点的子节点数是不定的，该程序所构造的语法树是一棵 LCRS 树，即一棵节点的左指针指向子节点，右指针指向自己最大的兄弟节点的树。该树节点的结构体定义如下：

typedef struct synTreenode{

```
    treenodecontent* nodeContent;
    struct synTreenode* son;
    struct synTreenode* brother;
}synTreenode;
```

其中树节点的内容在指针 nodeContent 所指向的实例中，treenodecontent 定义如下：

```
typedef struct {
    int dataType;
}exprAttr;
typedef union {
    exprAttr exprA;
}attr;
typedef struct {
    int lexType;
    char* tokenStr;
    char* nonTerminalStr;
    attr a;
}treenodecontent;
```

其中可以存储终结符或者非终结符的属性和字符串。在后期只需要向联合体 attr 中增加成员便可完成属性的扩充。

2.2 语法树的构造

YACC 所构造的是一个 LALR 语法分析器，在语法分析过程中有一个栈结构，其在每调用一次函数 yylex()（由实验一中 flex 构造，返回一个 Token 类型）后就会将当前的全局变量 yylval 压入栈中，每次规约时会按照语义规则中的语句更新栈顶并最终弹出对应数量 yylval 元素。特别的，yylval 的类型可由编程者定义，我将其定义为：

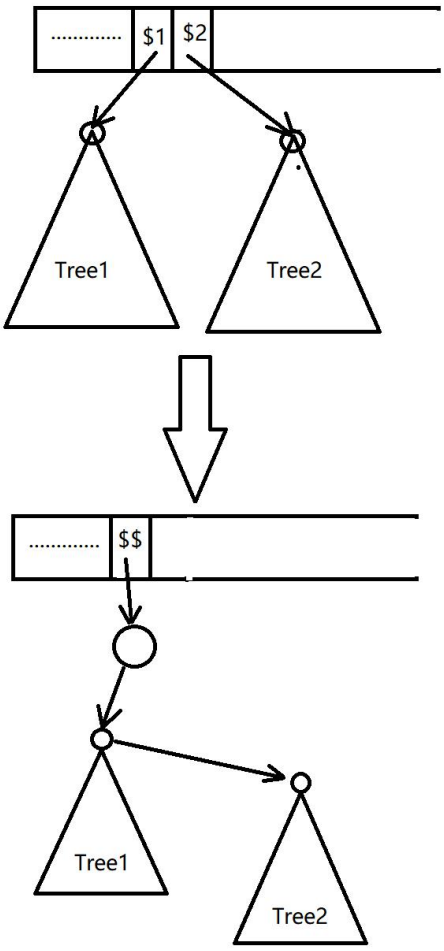
```
#define YYSTYPE synTreenode*
```

即一个指向树节点的指针。

该语法分析程序将利用这一动作进行语法树的构造，如下面这个例子：

```
Statements : Statements Statement {
    $$ = mkTreenode(mkTreeNodeContent(statements));
    insertSonNode($$, $1);
    insertSonNode($$, $2);
}
```

在该例子中，当一个 Statements 和 Statement 规约成一个 Statements 时，新的栈顶元素将会是一个指向树节点的指针，前两者在栈中指针所指向的树节点会成为新树节点的子节点并最终被弹出。示意图如下：



由于在词法分析程序中，每匹配一个词法单元，就会调用函数：

```
void loadYylval(int typenum) {
    treenodecontent* lexUnit = (treenodecontent*)malloc(sizeof(treenodecontent));
    lexUnit->lexType = typenum;
    lexUnit->tokenStr = (char*)malloc(sizeof(char) * (strlen(yytext) + 1));
    strcpy(lexUnit->tokenStr, yytext);
    lexUnit->a.exprA.dataType = -1;
    yylval = mkTreenode(lexUnit);
}
```

所以当进行移入时，也会有对应的指向树的指针压入栈中。

最终，经过上述移入/规约动作，语法树将被构造出来。

2.3 错误检测

在 YACC 构造出的程序中，其会在移入项与预测表不匹配时调用 yyerror 输出错误信息”syntax error”，通过实现 yyerror 函数：

```
void yyerror (char const *s) {
    printf ("%s at line %d:", s, yylineno);
```

}

来打印并定位错误位置（行）。

考虑到一个程序中，语句的占比比较大，所以在产生式

```
Statements : error Statement {
        $$ = mkTreenode(mkTreeNodeContent(statements));
        insertSonNode($$, $2);
    }
```

的定义中，错误将会被保留直至匹配到一个 Statement，该 Statement 和上一个被规约成的文法单元之间的文法单元将会被丢弃，最后语法分析从错误中恢复。

同时在.y 文件中加入`%error-verbose`，yacc 所产生的程序会自动打印出缺少的文法单元：

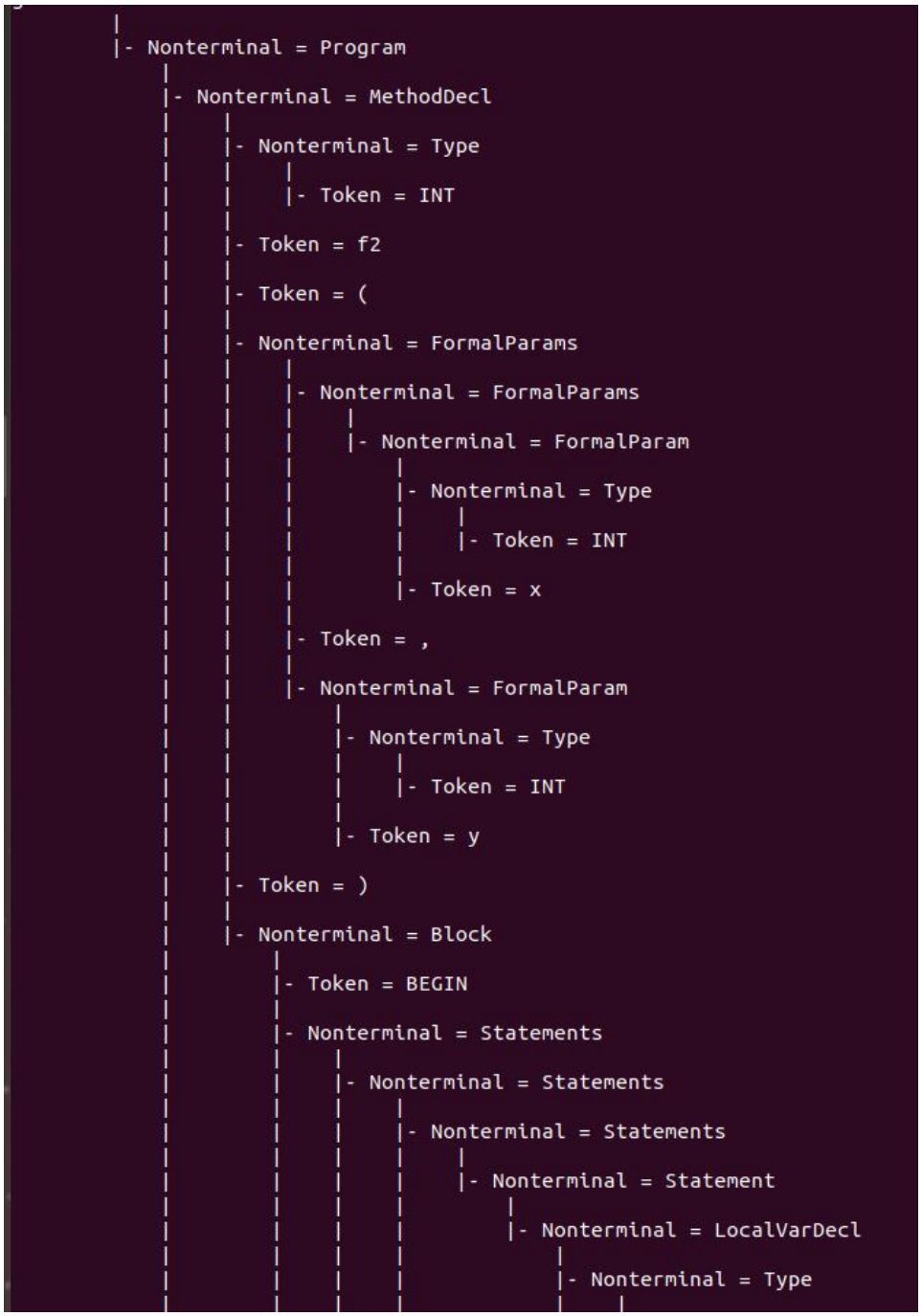
```
syntax error, unexpected INT, expecting SEMICOLON at line 13:
```

3. 实验结果

3.1 测试用例

测试用例为实验一中的程序，也是文档中给出的样例程序。

3.2 测试结果



在上面的打印的部分结果截图中，属于同一竖线的元素为兄弟，某文法单元下属竖线上的文法单元为其子节点，特别的，所有终结符都为叶子结点。

综上，该程序实现了将 Token 序列转换为语法树的功能。