深圳大学实验报告

课程名称：　　　编译原理

实验项目名称：　　　自顶向下的语法分析程序设计

学院：　　　计算机与软件学院

专业：　　　计算机科学与技术

指导教师：　　　罗成文

报告人：　黎浩然　学号：2018112061　班级：　01

实验时间：　　　2022/5/21

实验报告提交时间：

教务部制

实验目的与要求：

实验目的

任务一：运行 TINY 语言的语法分析程序 TINYParser，理解 TINY 语言语法分析器的实现。

其中，TINY 语言的词法与实验二相同，TINY 语言的文法描述如下:

program -> stmt-seq

stmt-seq -> stmt-seq;stmt | stmt

stmt -> if-stmt|repeat-stmt|assign-stmt|read-stmt | write-stmt

if-stmt -> if exp then stmt-seq end | if exp then stmt-seq else stmt-seq end

repeat-stmt -> repeat stmt-seq until exp

assign-stmt -> id:= exp

read-stmt -> read id

write-stmt -> write exp

exp -> simp-exp cop simp-exp | simp-exp

cop -> < | =

simp-exp-> simp-exp addop term |term

term -> term mulop factor | factor

factor -> (exp) |num |id

addop -> + | -

mulop -> * | /

任务一要求:根据 TINY 语法，自己编写至少一个另外的 TINY 测试程序，运行 TINYParser 语法分析器，观察程序运行流程，得到正确的运行结果。

任务二:基于 TinyParser 语法分析器，实现拓展语言 TINY+的语法分析器。

其中，TINY+语言的词法与实验二相同，TINY+语言的文法描述如下(注：此处为了描述方便，对上下文无关文法的产生式表示进行了扩充，允许在产生式右部使用类似正则表达式的表示，例如第 5 条产生式右部花括号{, identifier}代表*闭包。其中红色部分为 TINY+文法更新的部分，其余部分为 TINY 文法原有的产生式:

1 program -> declarations stmt-sequence

2 declarations -> decl ***;*** declarations | ε

3 decl -> type-specifier varlist

4 type-specifier-> *int* | *bool* | *string* | *float* | *double*

5 varlist -> ***identifier { , identifier }***

6 stmt-sequence -> statement { **;** statement }

7 statement -> if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt | while-stmt

8 while-stmt -> ***do*** stmt-sequence ***while*** bool-exp

9 if-stmt -> if exp then stmt-seq end | if exp then stmt-seq else stmt-seq end

10 repeat-stmt -> ***repeat*** stmt-sequence ***until*** exp

11 assign-stmt -> ***identifier:=***exp

12 read-stmt -> ***read identifier***

13 write-stmt -> ***write*** exp

14 exp -> simp-exp cop simp-exp | simp-exp

15 cop -> < | =
16 simp-exp -> simp-exp addop term |term
17 term -> term mulop factor | factor
18 factor -> (exp) |num |id
19 addop -> + | -
20 mulop -> * | /

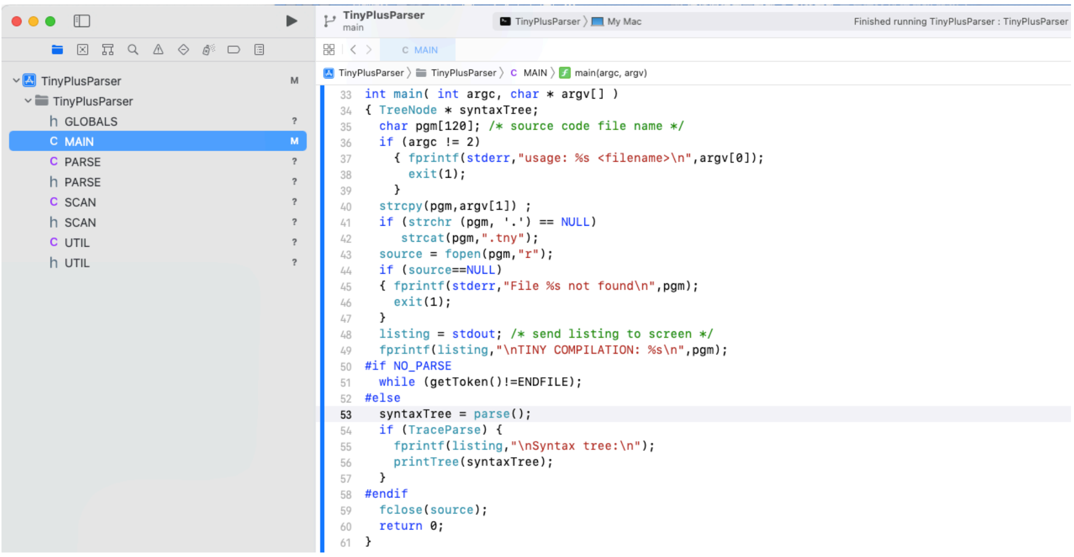
任务二要求:根据 TINY+语法，修改给定的 TINY 语法分析器，实现更新的 TINY+语法分析器，成功实现对上述示例程序的语法分析。并根据 TINY+文法的定义，编写至少一个另外的TINY+测试程序，对该测试程序完成语法分析，得到正确的语法分析结果。

实验要求

- 完成任务一及任务二的要求;
- 使用实验所提供的模板撰写实验报告，要求内容详实，有具体的设计描述、关键的代码片段、及实验结果屏幕截图;
- 在截止日期前将代码、实验报告、测试文件(如有)等所有实验相关文件压缩到一个压缩包姓名_学号_实验三.rar 上传至 Blackboard。

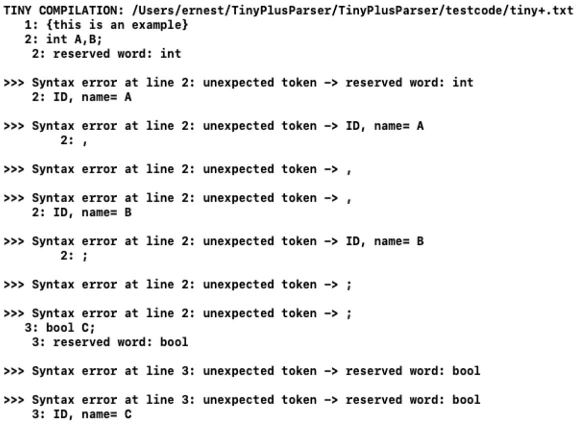
实验内容、方法及步骤：

任务一/任务二：直接通过 tiny+.txt 的运行结果改写代码：

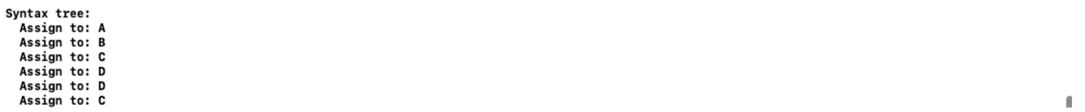


```
int main( int argc, char * argv[] )
{ TreeNode * syntaxTree;
  char pgm[120]; /* source code file name */
  if (argc != 2)
    { fprintf(stderr,"usage: %s <filename>\n",argv[0]);
      exit(1);
    }
  strcpy(pgm,argv[1]) ;
  if (strchr (pgm, '.') == NULL)
     strcat(pgm,".tny");
  source = fopen(pgm,"r");
  if (source==NULL)
  { fprintf(stderr,"File %s not found\n",pgm);
    exit(1);
  }
  listing = stdout; /* send listing to screen */
  fprintf(listing,"\nTINY COMPILATION: %s\n",pgm);
#if NO_PARSE
  while (getToken()!=ENDFILE);
#else
  syntaxTree = parse();
  if (TraceParse) {
    fprintf(listing,"\nSyntax tree:\n");
    printTree(syntaxTree);
  }
#endif
  fclose(source);
  return 0;
}
```

将 Tiny Parser 的所有源文件和头文件导入 Xcode 集成开发环境中进行开发。对原始代码文件作适当的修改，使其符合 ISO C++11 标准。运行测试 tiny+.txt 文件：

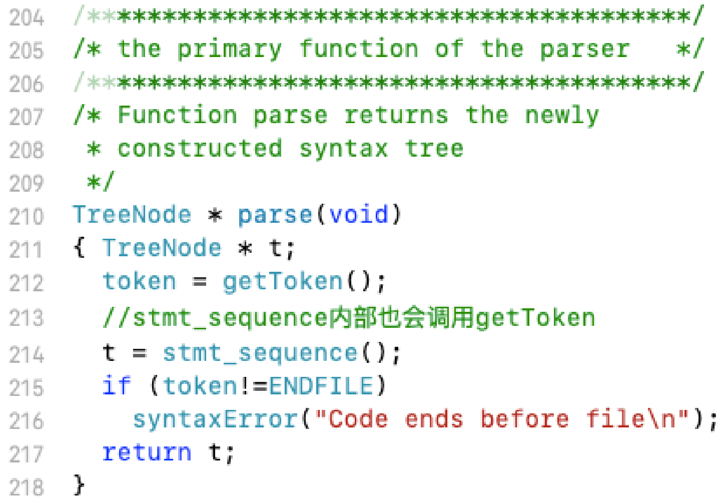


```
TINY COMPILATION: /Users/ernest/TinyPlusParser/TinyPlusParser/testcode/tiny+.txt
   1: {this is an example}
   2: int A,B;
    2: reserved word: int

>>> Syntax error at line 2: unexpected token -> reserved word: int
    2: ID, name= A

>>> Syntax error at line 2: unexpected token -> ID, name= A
        2: ,

>>> Syntax error at line 2: unexpected token -> ,

>>> Syntax error at line 2: unexpected token -> ,
    2: ID, name= B

>>> Syntax error at line 2: unexpected token -> ID, name= B
        2: ;

>>> Syntax error at line 2: unexpected token -> ;

>>> Syntax error at line 2: unexpected token -> ;
   3: bool C;
    3: reserved word: bool

>>> Syntax error at line 3: unexpected token -> reserved word: bool

>>> Syntax error at line 3: unexpected token -> reserved word: bool
    3: ID, name= C
```



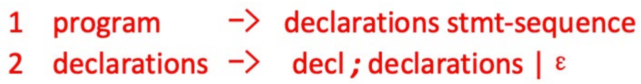
```
Syntax tree:
  Assign to: A
  Assign to: B
  Assign to: C
  Assign to: D
  Assign to: D
  Assign to: C
```

发现 Tiny Parser 还不能正确分析声明语句！找到 Tiny Parser 的入口函数为 parse():



```
/****************************************/
/* the primary function of the parser   */
/****************************************/
/* Function parse returns the newly
 * constructed syntax tree
 */
TreeNode * parse(void)
{ TreeNode * t;
  token = getToken();
  //stmt_sequence内部也会调用getToken
  t = stmt_sequence();
  if (token!=ENDFILE)
    syntaxError("Code ends before file\n");
  return t;
}
```

parse 函数直接进入 stmt_sequence 的识别，但是函数的声明（declarations）不属于 stmt_sequence 过程处理的范围内。

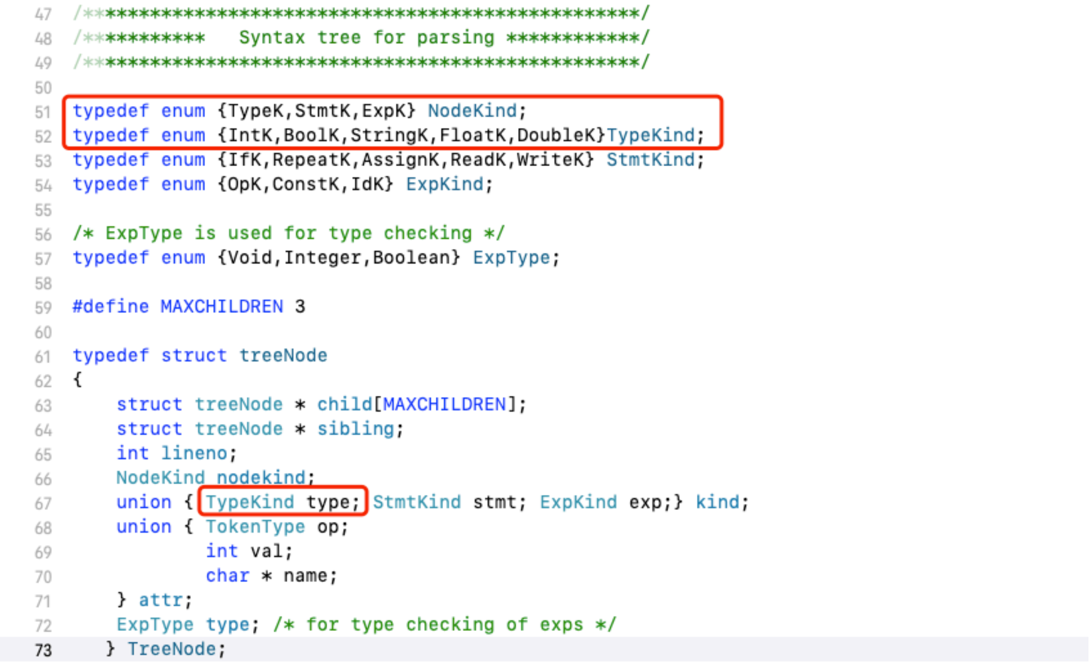


1 program -> declarations stmt-sequence

2 declarations -> decl ; declarations | ε

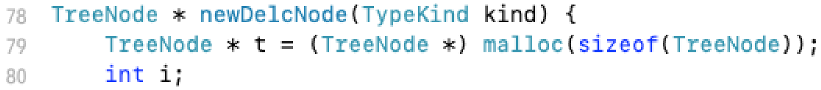
因此 declarations 的分析需要另外实现：

为了能够创建对应类型（declarations）的 TreeNode 结点，首先需要对 TreeNode 及相关枚举（如 NodeKind）的定义作修改，并添加 TypeKind 枚举类型：

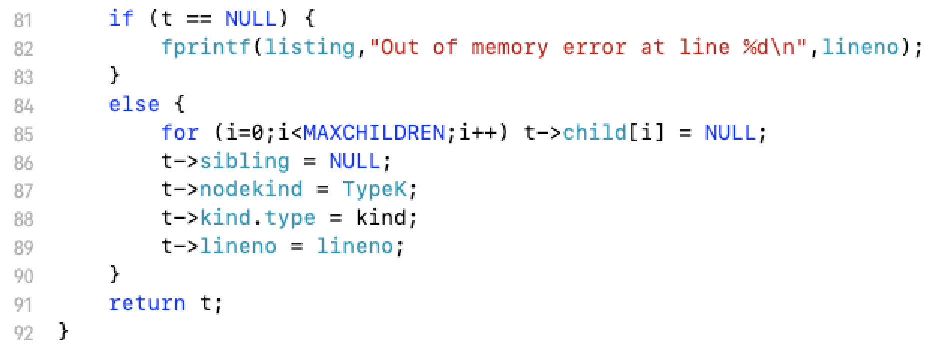


```
/****************************************************/
/**********   Syntax tree for parsing ************/
/****************************************************/

typedef enum {TypeK,StmtK,ExpK} NodeKind;
typedef enum {IntK,BoolK,StringK,FloatK,DoubleK}TypeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
{
    struct treeNode * child[MAXCHILDREN];
    struct treeNode * sibling;
    int lineno;
    NodeKind nodekind;
    union { TypeKind type; StmtKind stmt; ExpKind exp;} kind;
    union { TokenType op;
            int val;
            char * name;
    } attr;
    ExpType type; /* for type checking of exps */
  } TreeNode;
```

然后仿照 newStmtNode 和 newExpNode 函数，编写对应的 newDelcNode 函数：



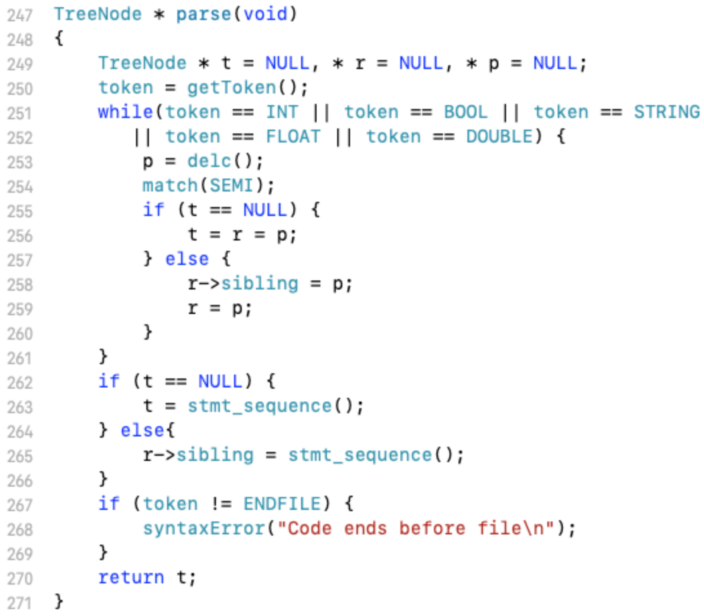
```
TreeNode * newDelcNode(TypeKind kind) {
    TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
    int i;
```



```
    if (t == NULL) {
        fprintf(listing,"Out of memory error at line %d\n",lineno);
    }
    else {
        for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
        t->sibling = NULL;
        t->nodekind = TypeK;
        t->kind.type = kind;
        t->lineno = lineno;
    }
    return t;
}
```

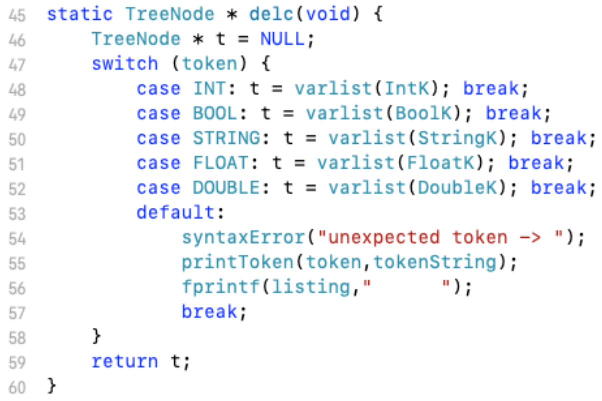
这样就可以为每一个 delc decl -> type-specifier varlist 创建一个 TreeNode。

在 parse 函数里面的开始添加解析 delcarations 的代码。根据行头的 token 是否为数据类型 INT/BOOL/STRING/FLOAT/DOUBLE 中一种来决定是否进入 stmt_sequence 阶段的解析！



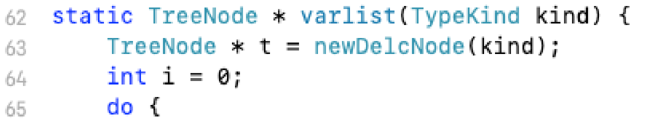
```
TreeNode * parse(void)
{
    TreeNode * t = NULL, * r = NULL, * p = NULL;
    token = getToken();
    while(token == INT || token == BOOL || token == STRING
       || token == FLOAT || token == DOUBLE) {
        p = delc();
        match(SEMI);
        if (t == NULL) {
            t = r = p;
        } else {
            r->sibling = p;
            r = p;
        }
    }
    if (t == NULL) {
        t = stmt_sequence();
    } else{
        r->sibling = stmt_sequence();
    }
    if (token != ENDFILE) {
        syntaxError("Code ends before file\n");
    }
    return t;
}
```

declarations 中的 delc 可能会有多个，因此需要用 while 循环来遍历：

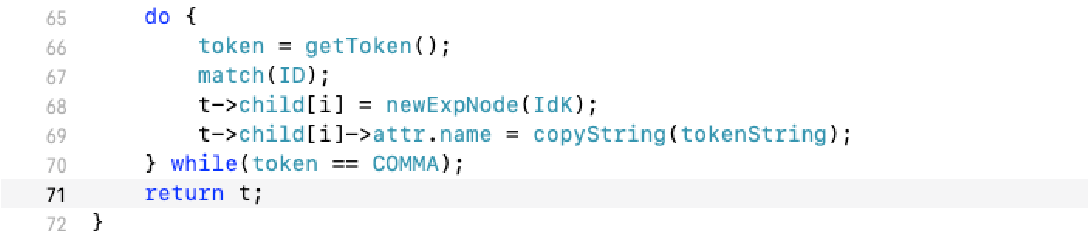


```
static TreeNode * delc(void) {
    TreeNode * t = NULL;
    switch (token) {
        case INT: t = varlist(IntK); break;
        case BOOL: t = varlist(BoolK); break;
        case STRING: t = varlist(StringK); break;
        case FLOAT: t = varlist(FloatK); break;
        case DOUBLE: t = varlist(DoubleK); break;
        default:
            syntaxError("unexpected token -> ");
            printToken(token,tokenString);
            fprintf(listing,"      ");
            break;
    }
    return t;
}
```

delc 首先会识别第一个字符以决定数据类型，然后识别出后面的一个或者多个 ID(entifier)。通过逗号（COMMA）和分号（SEMI）标记当前 delc 是否结束。

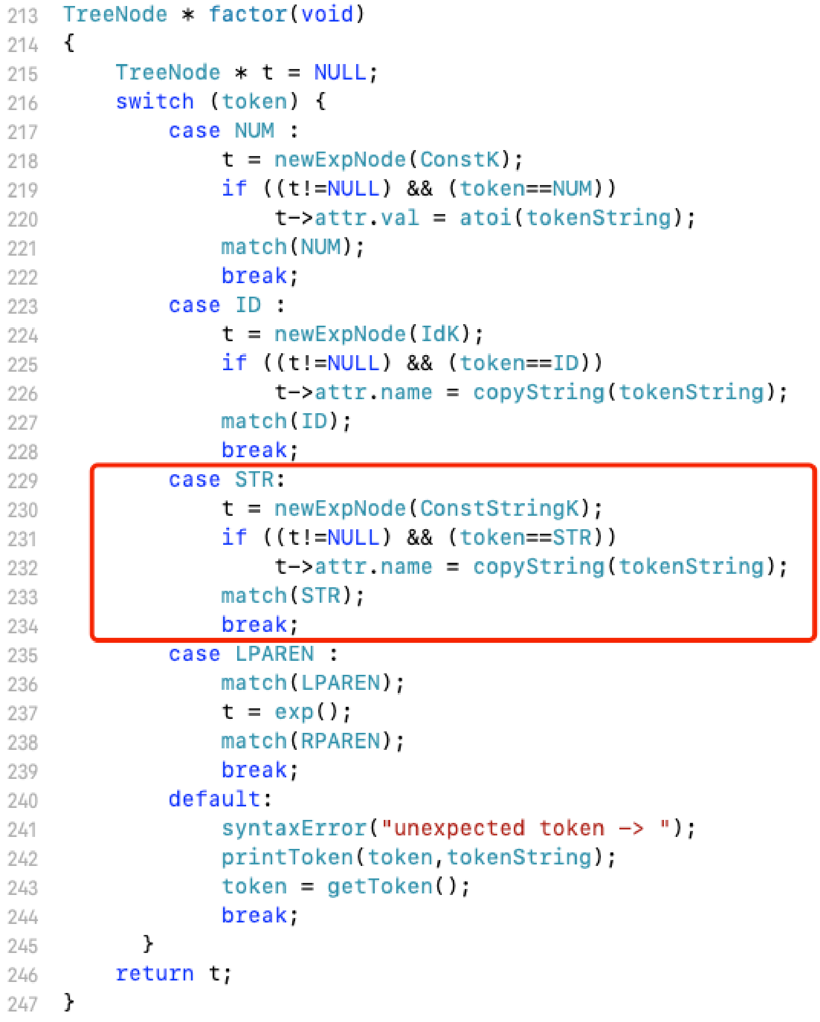


```
static TreeNode * varlist(TypeKind kind) {
    TreeNode * t = newDelcNode(kind);
    int i = 0;
    do {
```



```
    do {
        token = getToken();
        match(ID);
        t->child[i] = newExpNode(IdK);
        t->child[i]->attr.name = copyString(tokenString);
    } while(token == COMMA);
    return t;
}
```

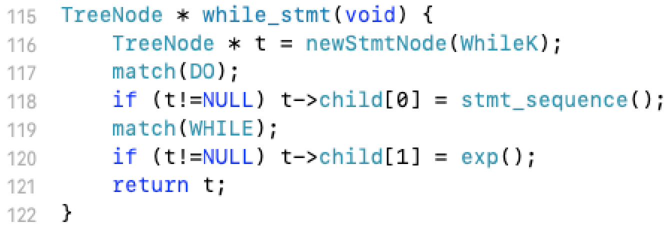
varlist 函数就是读取当前 delc 的一个或者多个 ID(entifier)，然后构建成为语法树中当前 Type 节点的子节点。



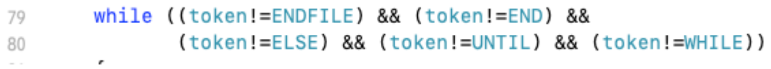
```
TreeNode * factor(void)
{
    TreeNode * t = NULL;
    switch (token) {
        case NUM :
            t = newExpNode(ConstK);
            if ((t!=NULL) && (token==NUM))
                t->attr.val = atoi(tokenString);
            match(NUM);
            break;
        case ID :
            t = newExpNode(IdK);
            if ((t!=NULL) && (token==ID))
                t->attr.name = copyString(tokenString);
            match(ID);
            break;
        case STR:
            t = newExpNode(ConstStringK);
            if ((t!=NULL) && (token==STR))
                t->attr.name = copyString(tokenString);
            match(STR);
            break;
        case LPAREN :
            match(LPAREN);
            t = exp();
            match(RPAREN);
            break;
        default:
            syntaxError("unexpected token -> ");
            printToken(token,tokenString);
            token = getToken();
            break;
      }
    return t;
}
```

测试发现 Tiny Parser 未能匹配赋值语句中的字符串常量，因此需要在对应的 factor 函数中添加对字符串常量的匹配。

增加对 do while 语句的匹配：



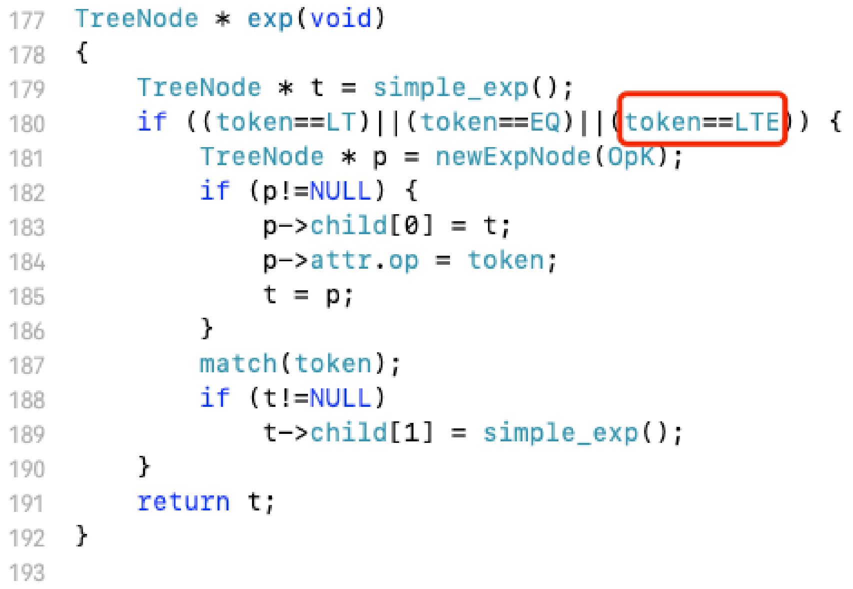
```
TreeNode * while_stmt(void) {
    TreeNode * t = newStmtNode(WhileK);
    match(DO);
    if (t!=NULL) t->child[0] = stmt_sequence();
    match(WHILE);
    if (t!=NULL) t->child[1] = exp();
    return t;
}
```



```
    while ((token!=ENDFILE) && (token!=END) &&
           (token!=ELSE) && (token!=UNTIL) && (token!=WHILE))
```

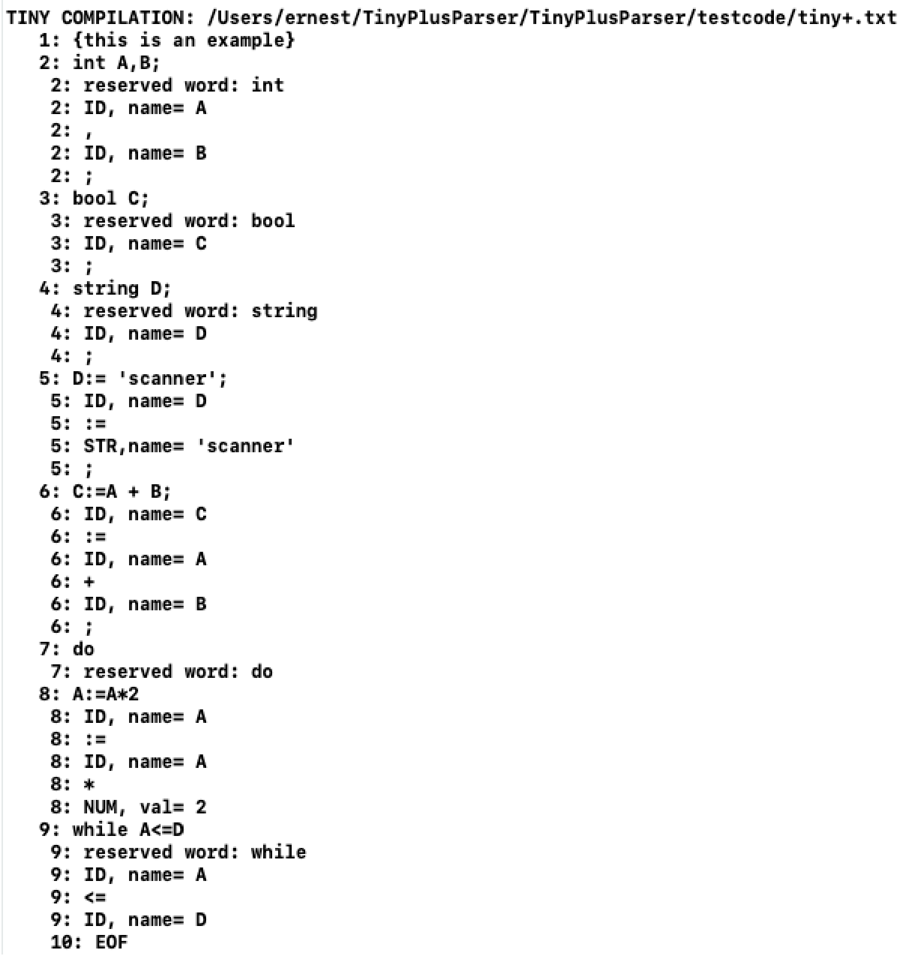
当匹配到 DO 到时候，进入 do while statement 的匹配。while-stmt -> do stmt-sequence while bool-exp 来调用对应的函数匹配，并且构建树对应的节点！

最后增加对其它不等号的识别：



```
TreeNode * exp(void)
{
    TreeNode * t = simple_exp();
    if ((token==LT)||(token==EQ)||(token==LTE)) {
        TreeNode * p = newExpNode(OpK);
        if (p!=NULL) {
            p->child[0] = t;
            p->attr.op = token;
            t = p;
        }
        match(token);
        if (t!=NULL)
            t->child[1] = simple_exp();
    }
    return t;
}
```

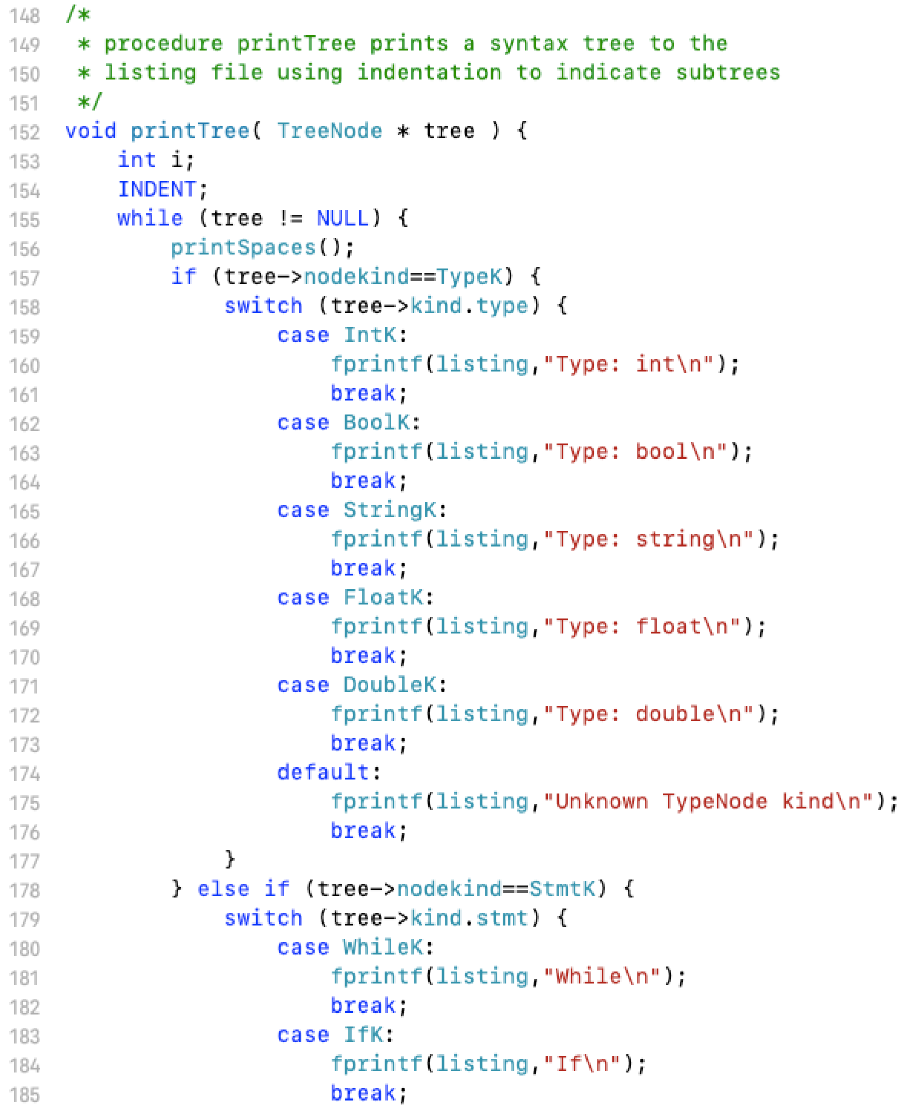
至此，词法分析已经不再报错：



```
TINY COMPILATION: /Users/ernest/TinyPlusParser/TinyPlusParser/testcode/tiny+.txt
   1: {this is an example}
   2: int A,B;
    2: reserved word: int
    2: ID, name= A
    2: ,
    2: ID, name= B
    2: ;
   3: bool C;
    3: reserved word: bool
    3: ID, name= C
    3: ;
   4: string D;
    4: reserved word: string
    4: ID, name= D
    4: ;
   5: D:= 'scanner';
    5: ID, name= D
    5: :=
    5: STR,name= 'scanner'
    5: ;
   6: C:=A + B;
    6: ID, name= C
    6: :=
    6: ID, name= A
    6: +
    6: ID, name= B
    6: ;
   7: do
    7: reserved word: do
   8: A:=A*2
    8: ID, name= A
    8: :=
    8: ID, name= A
    8: *
    8: NUM, val= 2
   9: while A<=D
    9: reserved word: while
    9: ID, name= A
    9: <=
    9: ID, name= D
    10: EOF
```

因为一开始的 Tiny Parser 在词法分析阶段会报很多 unexpected token 的错误。通过修改代码在对应的时候识别合适的 token 消除错误。此时的语法分析树也已经构建完成，需

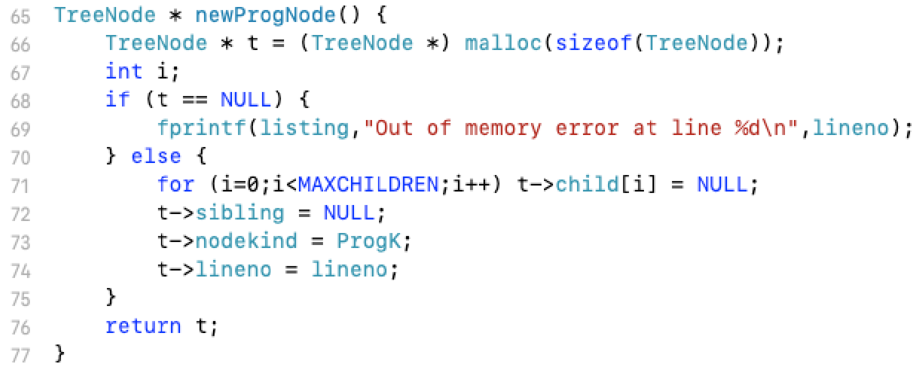
要做的只是打印语法树。



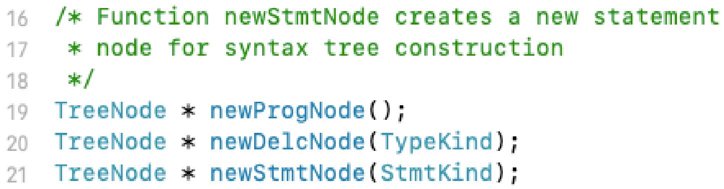
```
/*
 * procedure printTree prints a syntax tree to the
 * listing file using indentation to indicate subtrees
 */
void printTree( TreeNode * tree ) {
    int i;
    INDENT;
    while (tree != NULL) {
        printSpaces();
        if (tree->nodekind==TypeK) {
            switch (tree->kind.type) {
                case IntK:
                    fprintf(listing,"Type: int\n");
                    break;
                case BoolK:
                    fprintf(listing,"Type: bool\n");
                    break;
                case StringK:
                    fprintf(listing,"Type: string\n");
                    break;
                case FloatK:
                    fprintf(listing,"Type: float\n");
                    break;
                case DoubleK:
                    fprintf(listing,"Type: double\n");
                    break;
                default:
                    fprintf(listing,"Unknown TypeNode kind\n");
                    break;
            }
        } else if (tree->nodekind==StmtK) {
            switch (tree->kind.stmt) {
                case WhileK:
                    fprintf(listing,"While\n");
                    break;
                case IfK:
                    fprintf(listing,"If\n");
                    break;
```

添加对 delc 节点及相关子节点的打印。添加对 while 节点及其子节点的打印。

至此，代码的主要修改就完成了！接下来需要添加一个 Program 主节点

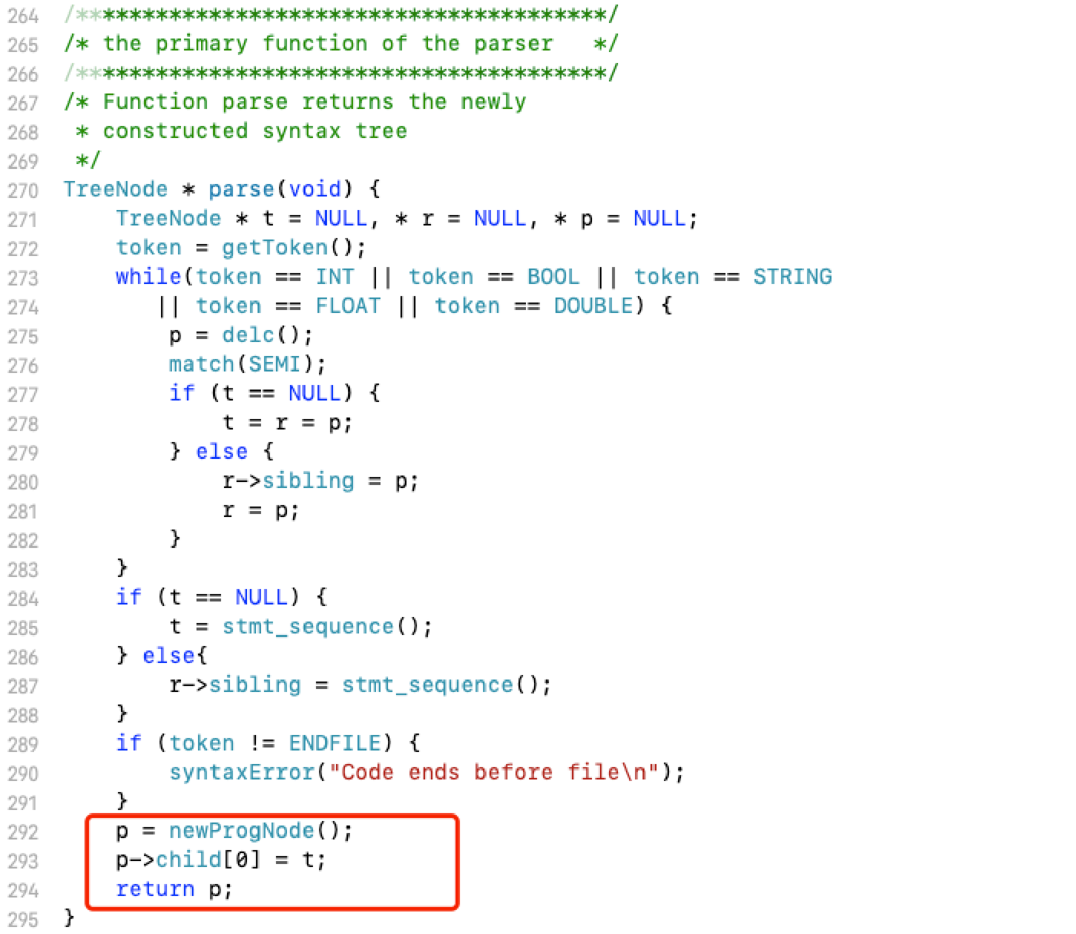


```
TreeNode * newProgNode() {
    TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
    int i;
    if (t == NULL) {
        fprintf(listing,"Out of memory error at line %d\n",lineno);
    } else {
        for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
        t->sibling = NULL;
        t->nodekind = ProgK;
        t->lineno = lineno;
    }
    return t;
}
```



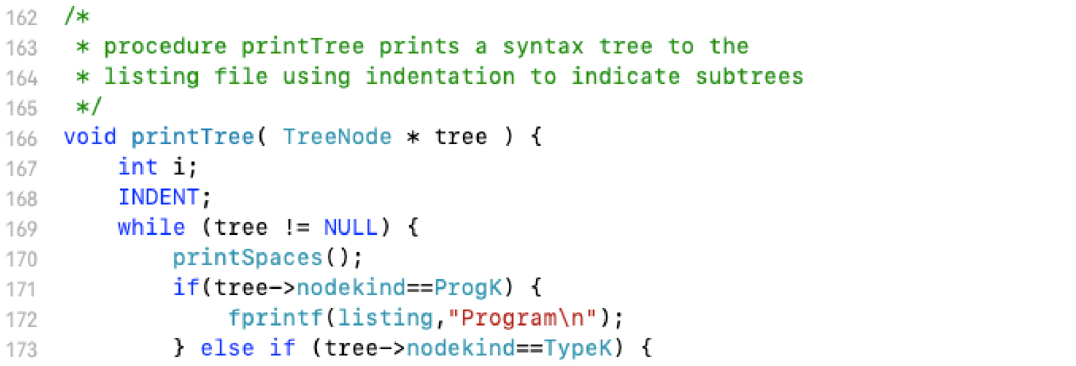
```
/* Function newStmtNode creates a new statement
 * node for syntax tree construction
 */
TreeNode * newProgNode();
TreeNode * newDelcNode(TypeKind);
TreeNode * newStmtNode(StmtKind);
```

以上表示在 util.c 中定义 newProgNode 创建 Program 节点，然后在头文件中声明函数。



```
/****************************************/
/* the primary function of the parser   */
/****************************************/
/* Function parse returns the newly
 * constructed syntax tree
 */
TreeNode * parse(void) {
    TreeNode * t = NULL, * r = NULL, * p = NULL;
    token = getToken();
    while(token == INT || token == BOOL || token == STRING
       || token == FLOAT || token == DOUBLE) {
        p = delc();
        match(SEMI);
        if (t == NULL) {
            t = r = p;
        } else {
            r->sibling = p;
            r = p;
        }
    }
    if (t == NULL) {
        t = stmt_sequence();
    } else{
        r->sibling = stmt_sequence();
    }
    if (token != ENDFILE) {
        syntaxError("Code ends before file\n");
    }
    p = newProgNode();
    p->child[0] = t;
    return p;
}
```

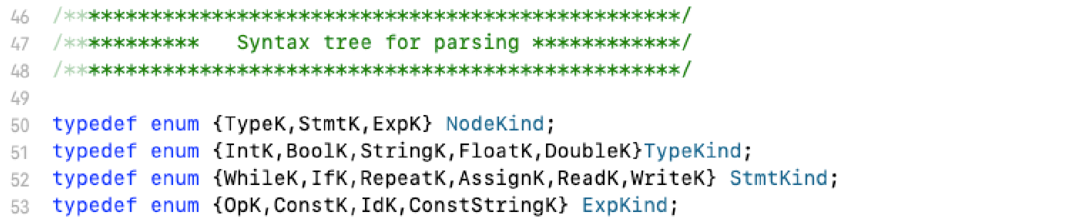
在 parse 函数中加入创建 Program 节点的代码。再添加对应的打印 Progra 节点的代码。



```
/*
 * procedure printTree prints a syntax tree to the
 * listing file using indentation to indicate subtrees
 */
void printTree( TreeNode * tree ) {
    int i;
    INDENT;
    while (tree != NULL) {
        printSpaces();
        if(tree->nodekind==ProgK) {
            fprintf(listing,"Program\n");
        } else if (tree->nodekind==TypeK) {
```

Dirty Work && Complement：

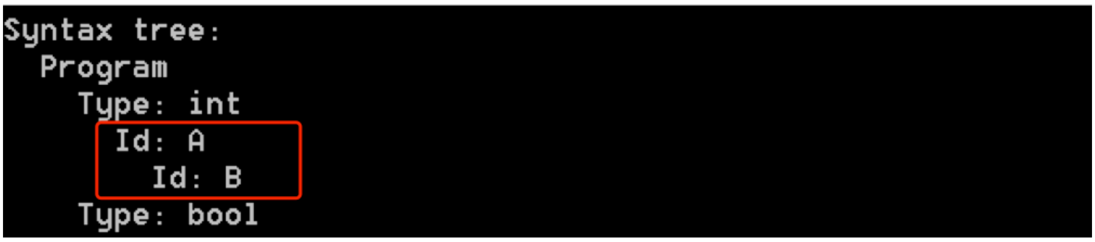
这里主要介绍除了基本逻辑以外的一些改动。这些改动都可以通过参考原来的代码来快速确定需要作出的改变，因此比较简单。此处挑一些地方描述：



```
/**************************************************/
/**********   Syntax tree for parsing ************/
/**************************************************/

typedef enum {TypeK,StmtK,ExpK} NodeKind;
typedef enum {IntK,BoolK,StringK,FloatK,DoubleK}TypeKind;
typedef enum {WhileK,IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK,ConstStringK} ExpKind;
```

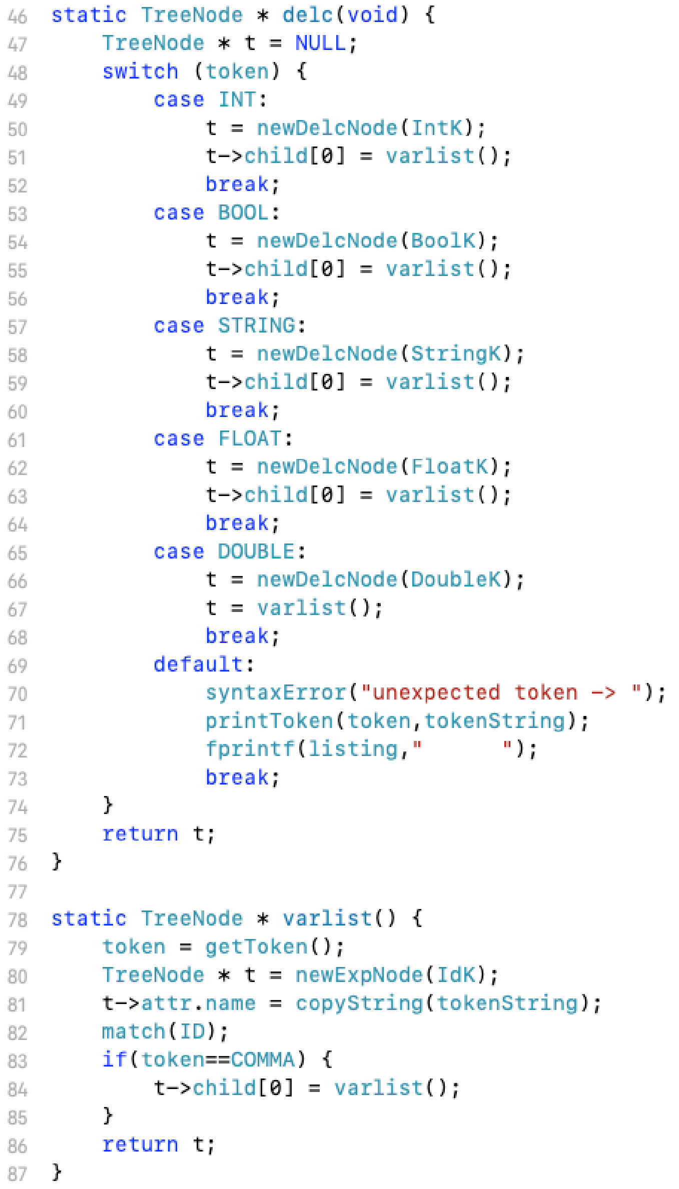
首先向 TokenType 添加 FLOAT 和 DOUBLE 两个枚举实例。其它的类型实例也要修改！

递归的 delc。观察实验要求发现：



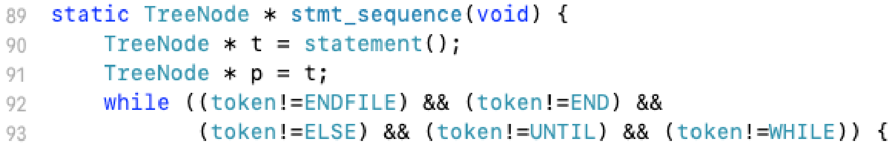
```
Syntax tree:
  Program
    Type: int
      Id: A
        Id: B
    Type: bool
```

当一个 delc 中有多个 ID 时，后面的 ID 作为前面的 ID 的子节点。因此 delc 函数：



```c
static TreeNode * delc(void) {
    TreeNode * t = NULL;
    switch (token) {
        case INT:
            t = newDelcNode(IntK);
            t->child[0] = varlist();
            break;
        case BOOL:
            t = newDelcNode(BoolK);
            t->child[0] = varlist();
            break;
        case STRING:
            t = newDelcNode(StringK);
            t->child[0] = varlist();
            break;
        case FLOAT:
            t = newDelcNode(FloatK);
            t->child[0] = varlist();
            break;
        case DOUBLE:
            t = newDelcNode(DoubleK);
            t = varlist();
            break;
        default:
            syntaxError("unexpected token -> ");
            printToken(token,tokenString);
            fprintf(listing,"      ");
            break;
    }
    return t;
}

static TreeNode * varlist() {
    token = getToken();
    TreeNode * t = newExpNode(IdK);
    t->attr.name = copyString(tokenString);
    match(ID);
    if(token==COMMA) {
        t->child[0] = varlist();
    }
    return t;
}
```

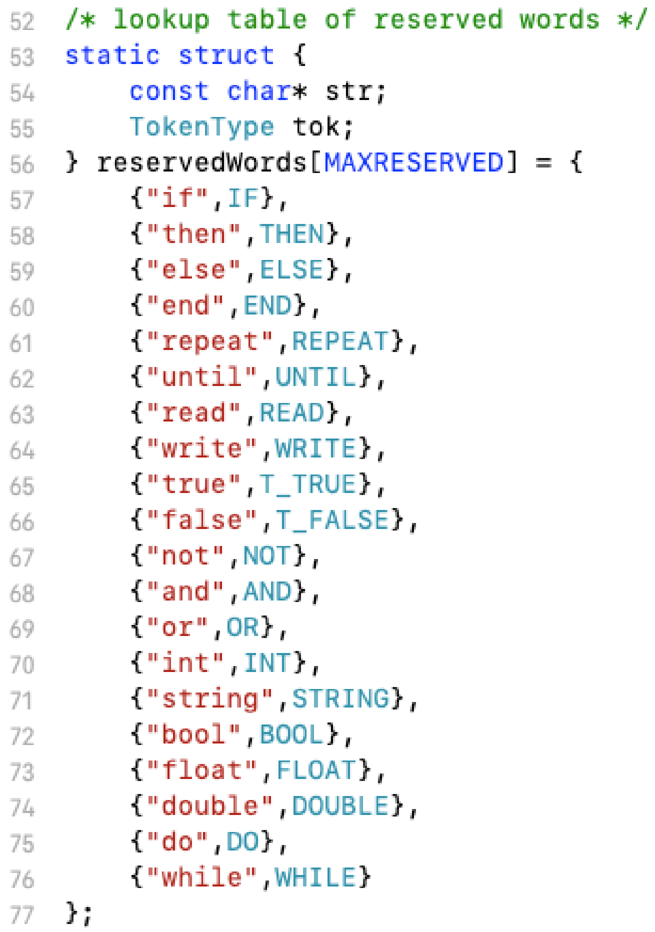
delc 函数调用 varlist 函数。而 varlist 是一个递归的函数。递归终止的条件是 COMMA 和 SEMI 两个符号判断的。从而使得后面的 ID 作为前面的 ID 的 child[0]。



```c
static TreeNode * stmt_sequence(void) {
    TreeNode * t = statement();
    TreeNode * p = t;
    while ((token!=ENDFILE) && (token!=END) &&
           (token!=ELSE) && (token!=UNTIL) && (token!=WHILE)) {
```

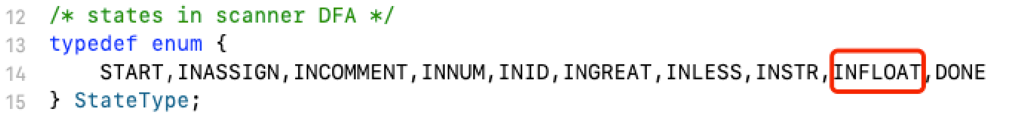
Stmt_sequence 需要增加 WHILE 作为终止条件之一。

Robust：

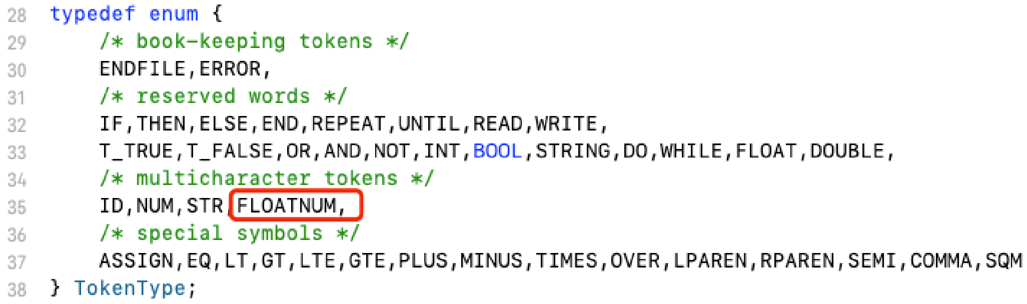


```
/* lookup table of reserved words */
static struct {
    const char* str;
    TokenType tok;
} reservedWords[MAXRESERVED] = {
    {"if",IF},
    {"then",THEN},
    {"else",ELSE},
    {"end",END},
    {"repeat",REPEAT},
    {"until",UNTIL},
    {"read",READ},
    {"write",WRITE},
    {"true",T_TRUE},
    {"false",T_FALSE},
    {"not",NOT},
    {"and",AND},
    {"or",OR},
    {"int",INT},
    {"string",STRING},
    {"bool",BOOL},
    {"float",FLOAT},
    {"double",DOUBLE},
    {"do",DO},
    {"while",WHILE}
};
```

增加对 float 和 double 关键字对识别。

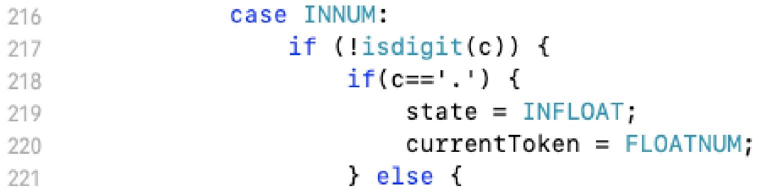


```
/* states in scanner DFA */
typedef enum {
    START,INASSIGN,INCOMMENT,INNUM,INID,INGREAT,INLESS,INSTR,INFLOAT,DONE
} StateType;
```



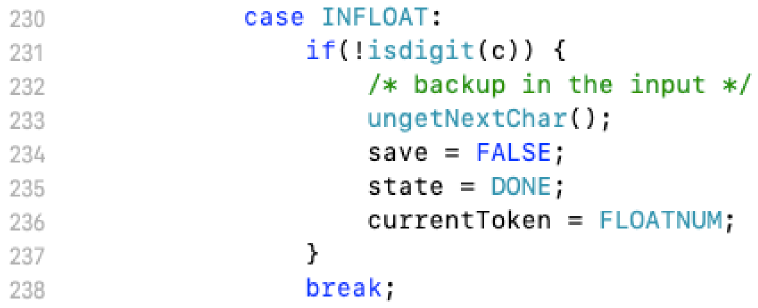
```
typedef enum {
    /* book-keeping tokens */
    ENDFILE,ERROR,
    /* reserved words */
    IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,
    T_TRUE,T_FALSE,OR,AND,NOT,INT,BOOL,STRING,DO,WHILE,FLOAT,DOUBLE,
    /* multicharacter tokens */
    ID,NUM,STR,FLOATNUM,
    /* special symbols */
    ASSIGN,EQ,LT,GT,LTE,GTE,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI,COMMA,SQM
} TokenType;
```

1. 在识别常整数时，遇到第一个小数点进入常浮点数对识别状态，然后识别余下的数字。因此 getToken 函数能够识别带有一个小数点的小数为常浮点数，默认精度为 float：

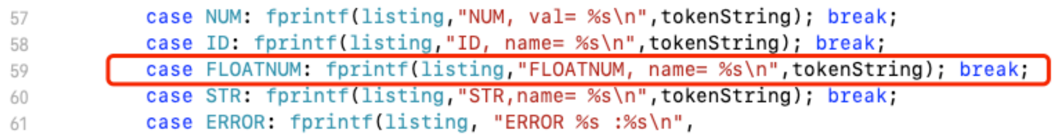


```
            case INNUM:
                if (!isdigit(c)) {
                    if(c=='.') {
                        state = INFLOAT;
                        currentToken = FLOATNUM;
                    } else {
```

当第一个小数点后，FA 进入 INFLOAT 状态：

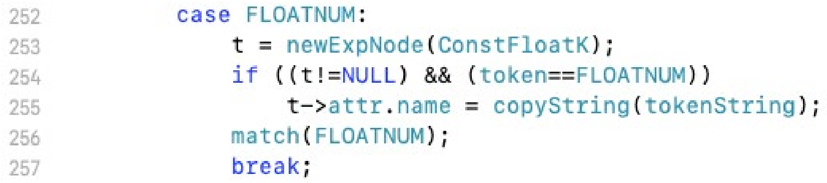


```
        case INFLOAT:
            if(!isdigit(c)) {
                /* backup in the input */
                ungetNextChar();
                save = FALSE;
                state = DONE;
                currentToken = FLOATNUM;
            }
            break;
```



```
    case NUM: fprintf(listing,"NUM, val= %s\n",tokenString); break;
    case ID: fprintf(listing,"ID, name= %s\n",tokenString); break;
    case FLOATNUM: fprintf(listing,"FLOATNUM, name= %s\n",tokenString); break;
    case STR: fprintf(listing,"STR,name= %s\n",tokenString); break;
    case ERROR: fprintf(listing, "ERROR %s :%s\n",
```

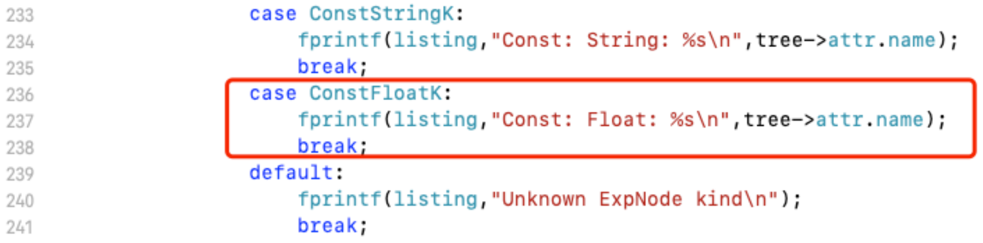
2. 然后在 printToken 函数添加打印 Const float 的 case。



```
    case FLOATNUM:
        t = newExpNode(ConstFloatK);
        if ((t!=NULL) && (token==FLOATNUM))
            t->attr.name = copyString(tokenString);
        match(FLOATNUM);
        break;
```

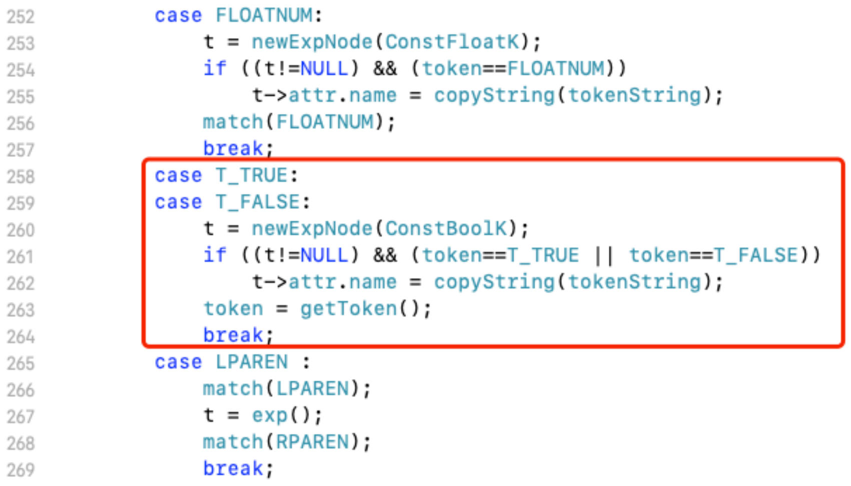
3. 接着修改 factor 函数增加常浮点数。

4. 最后修改 printTree：



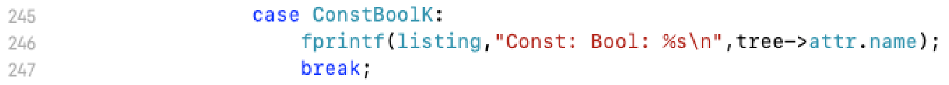
```
        case ConstStringK:
            fprintf(listing,"Const: String: %s\n",tree->attr.name);
            break;
        case ConstFloatK:
            fprintf(listing,"Const: Float: %s\n",tree->attr.name);
            break;
        default:
            fprintf(listing,"Unknown ExpNode kind\n");
            break;
```

增加对其它类型的语法分析，大概也为上面四步不差。



```
    case FLOATNUM:
        t = newExpNode(ConstFloatK);
        if ((t!=NULL) && (token==FLOATNUM))
            t->attr.name = copyString(tokenString);
        match(FLOATNUM);
        break;
    case T_TRUE:
    case T_FALSE:
        t = newExpNode(ConstBoolK);
        if ((t!=NULL) && (token==T_TRUE || token==T_FALSE))
            t->attr.name = copyString(tokenString);
        token = getToken();
        break;
    case LPAREN :
        match(LPAREN);
        t = exp();
        match(RPAREN);
        break;
```

最后添加对 bool 常量的分析

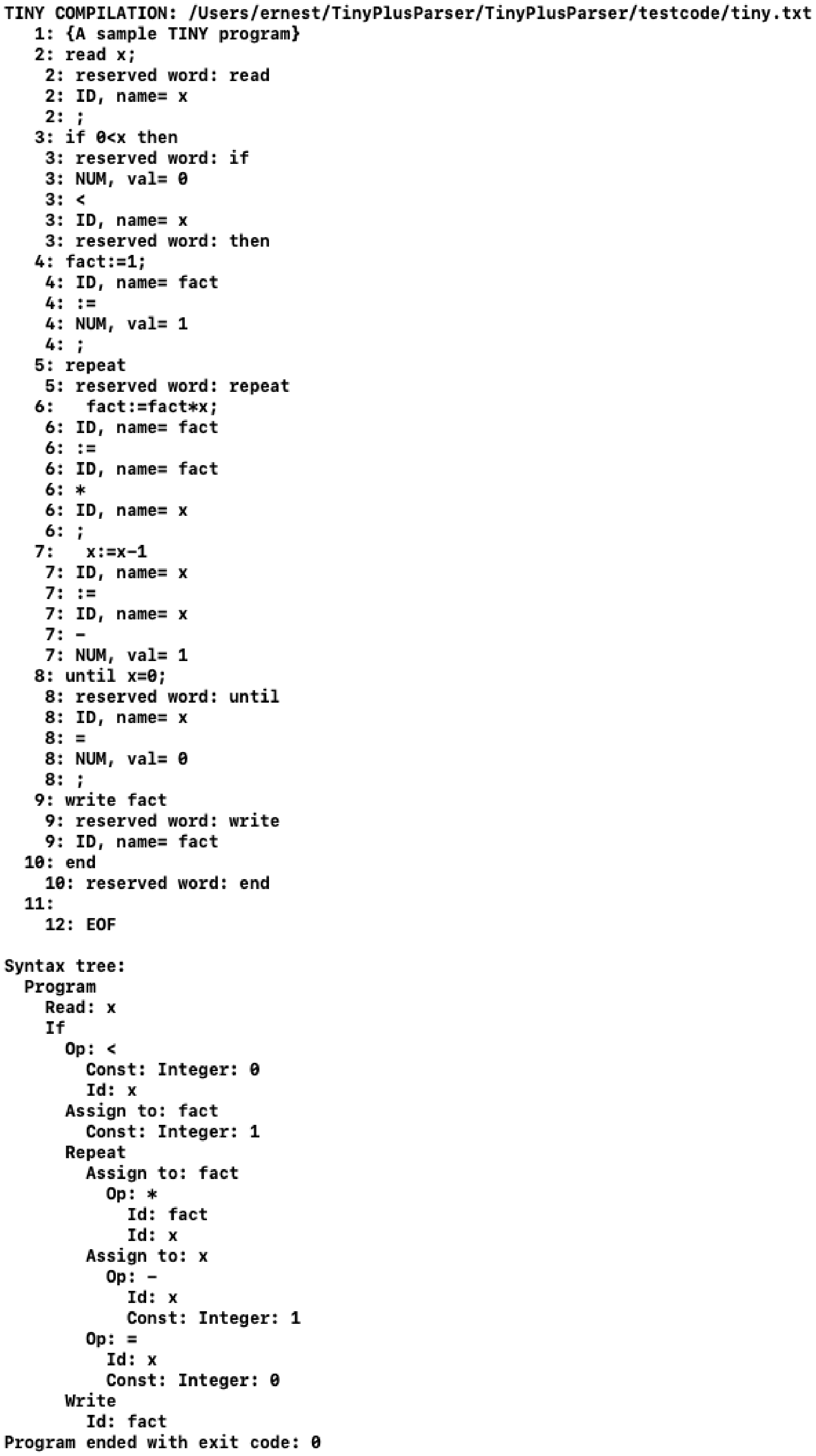


```
        case ConstBoolK:
            fprintf(listing,"Const: Bool: %s\n",tree->attr.name);
            break;
```

以及输出！

实验结果与分析：

首先来看看 tiny 的运行结果：



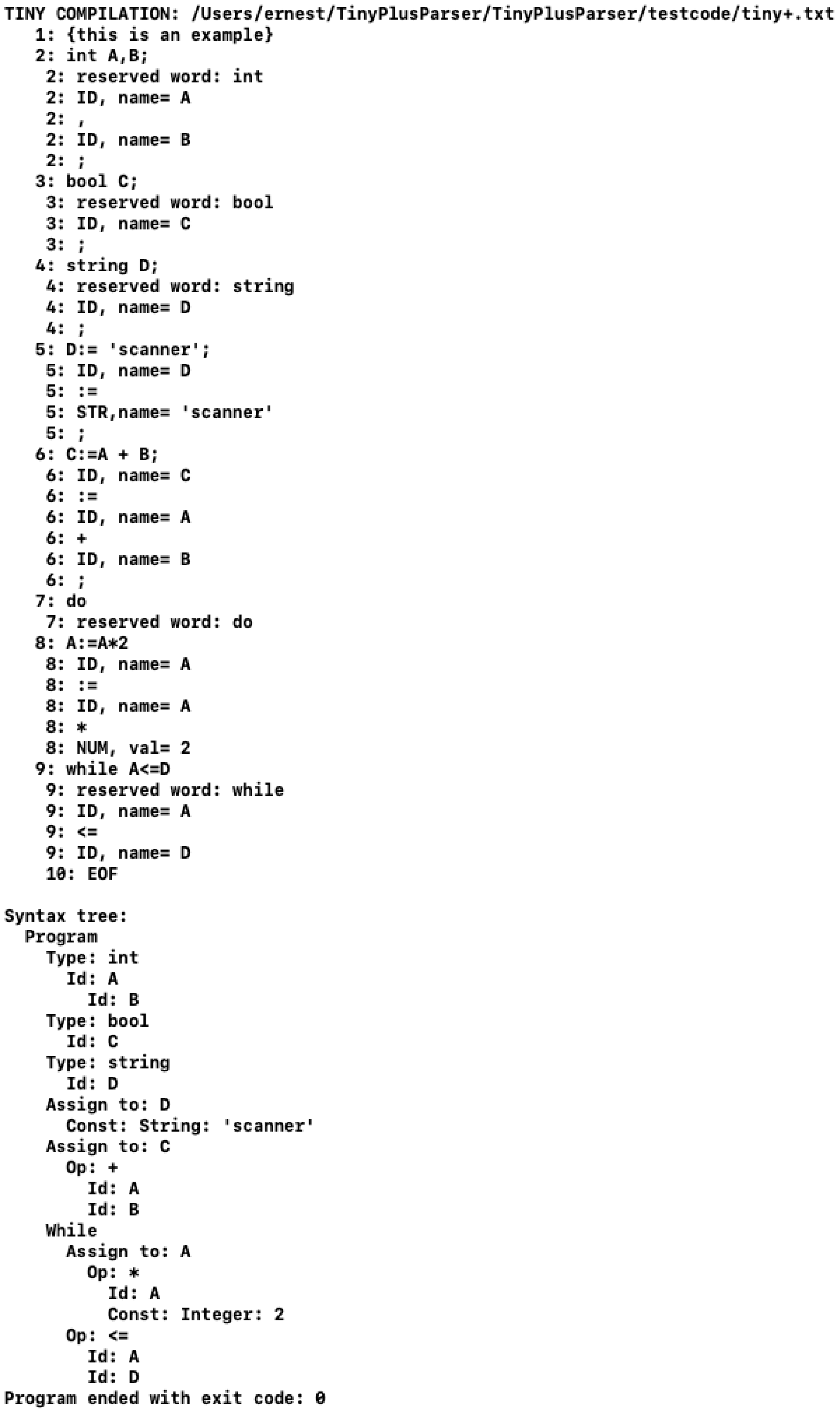
```
TINY COMPILATION: /Users/ernest/TinyPlusParser/TinyPlusParser/testcode/tiny.txt
   1: {A sample TINY program}
   2: read x;
    2: reserved word: read
    2: ID, name= x
    2: ;
   3: if 0<x then
    3: reserved word: if
    3: NUM, val= 0
    3: <
    3: ID, name= x
    3: reserved word: then
   4: fact:=1;
    4: ID, name= fact
    4: :=
    4: NUM, val= 1
    4: ;
   5: repeat
    5: reserved word: repeat
   6:   fact:=fact*x;
    6: ID, name= fact
    6: :=
    6: ID, name= fact
    6: *
    6: ID, name= x
    6: ;
   7:   x:=x-1
    7: ID, name= x
    7: :=
    7: ID, name= x
    7: -
    7: NUM, val= 1
   8: until x=0;
    8: reserved word: until
    8: ID, name= x
    8: =
    8: NUM, val= 0
    8: ;
   9: write fact
    9: reserved word: write
    9: ID, name= fact
  10: end
    10: reserved word: end
  11:
    12: EOF

Syntax tree:
  Program
    Read: x
    If
      Op: <
        Const: Integer: 0
        Id: x
      Assign to: fact
        Const: Integer: 1
      Repeat
        Assign to: fact
          Op: *
            Id: fact
            Id: x
        Assign to: x
          Op: -
            Id: x
            Const: Integer: 1
        Op: =
          Id: x
          Const: Integer: 0
      Write
        Id: fact
Program ended with exit code: 0
```

Tiny.txt 文件的运行结果一开始就没有问题。8 个代码文件中实际上也只是比上次实验

多了解析部分，通过阅读代码，很容易就分析出代码的逻辑。**因此，我的思路是先运行 tiny+.txt 的结果。根据 unexpected token 错误出现的地方逐渐增加对应 token 的识别分析。**

本质上，只要看懂代码的逻辑和 FA 之间的契合，按照类似的逻辑添加相应的文法代码，然后再作调试即可。

Tiny+.txt 的完整运行结果如下：

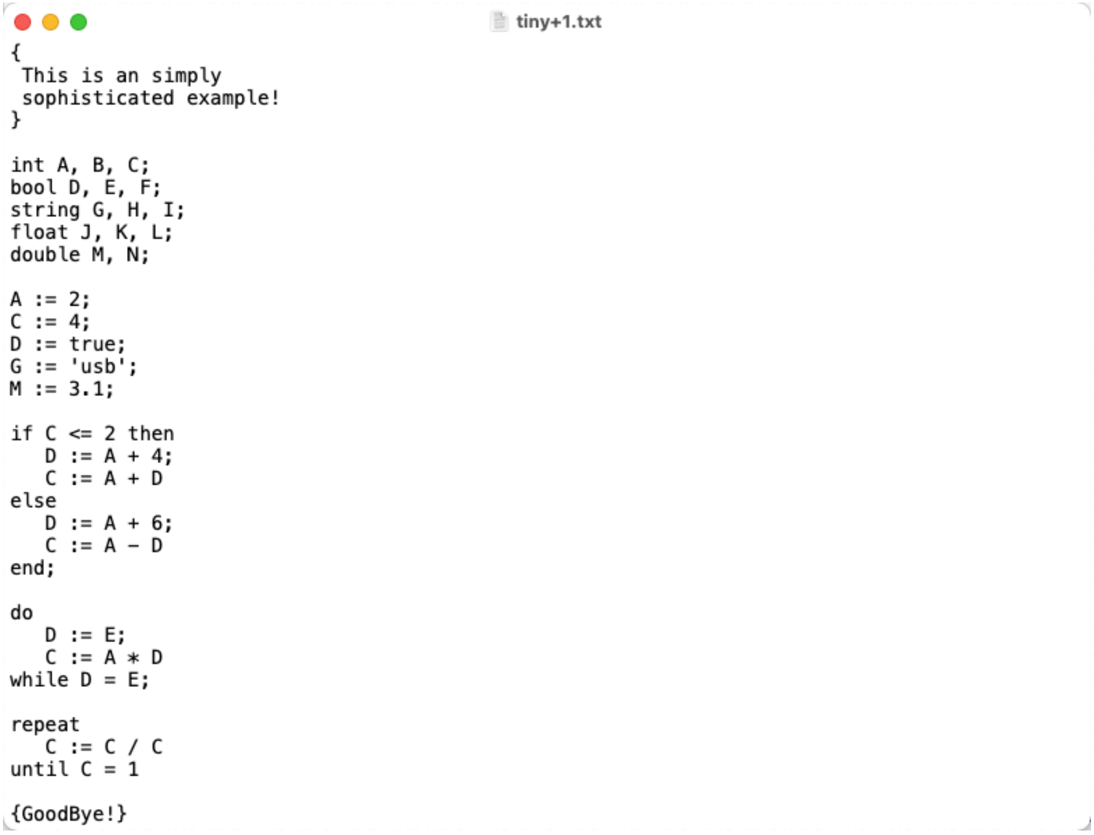


```
TINY COMPILATION: /Users/ernest/TinyPlusParser/TinyPlusParser/testcode/tiny+.txt
   1: {this is an example}
   2: int A,B;
    2: reserved word: int
    2: ID, name= A
    2: ,
    2: ID, name= B
    2: ;
   3: bool C;
    3: reserved word: bool
    3: ID, name= C
    3: ;
   4: string D;
    4: reserved word: string
    4: ID, name= D
    4: ;
   5: D:= 'scanner';
    5: ID, name= D
    5: :=
    5: STR,name= 'scanner'
    5: ;
   6: C:=A + B;
    6: ID, name= C
    6: :=
    6: ID, name= A
    6: +
    6: ID, name= B
    6: ;
   7: do
    7: reserved word: do
   8: A:=A*2
    8: ID, name= A
    8: :=
    8: ID, name= A
    8: *
    8: NUM, val= 2
   9: while A<=D
    9: reserved word: while
    9: ID, name= A
    9: <=
    9: ID, name= D
    10: EOF

Syntax tree:
  Program
    Type: int
      Id: A
        Id: B
    Type: bool
      Id: C
    Type: string
      Id: D
    Assign to: D
      Const: String: 'scanner'
    Assign to: C
      Op: +
        Id: A
        Id: B
    While
      Assign to: A
        Op: *
          Id: A
          Const: Integer: 2
      Op: <=
        Id: A
        Id: D
Program ended with exit code: 0
```

可以看到，程序已经能够正确解析所有的示例测试文件了。

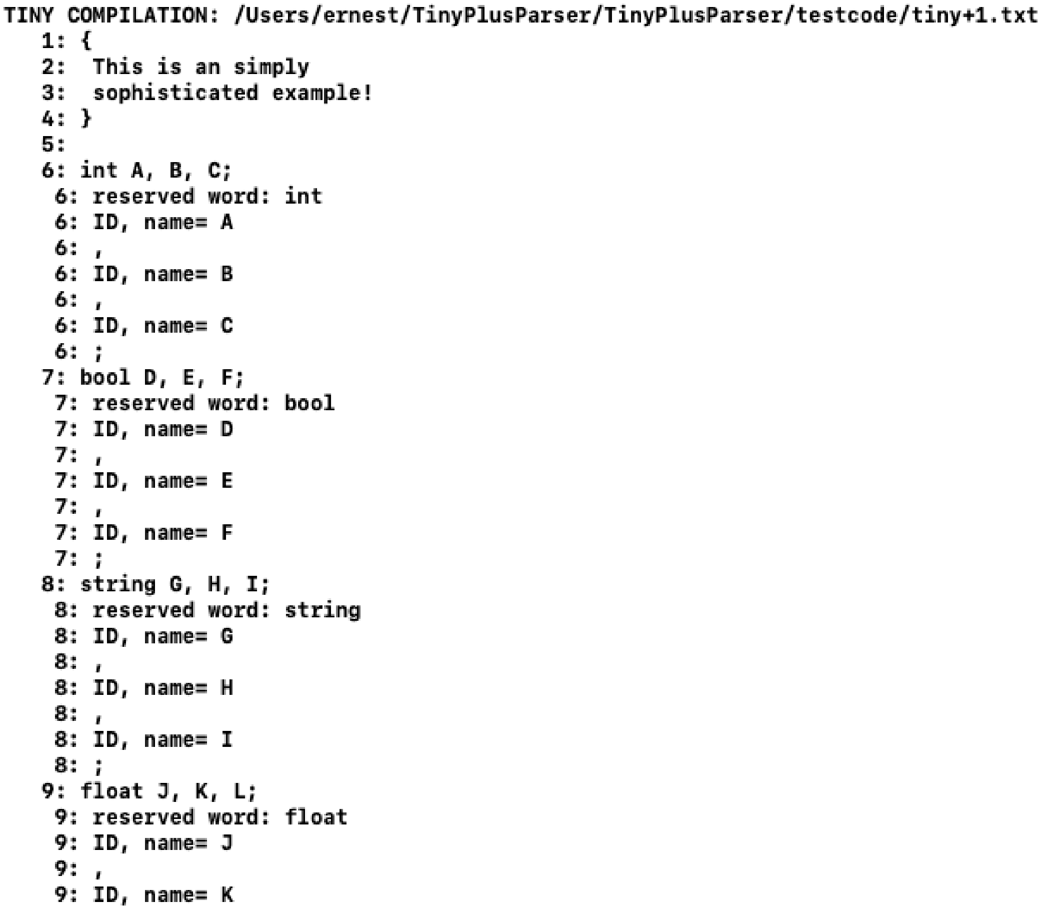
接下来就是编写另外的测试文件：

自己编写的 tiny+源文件

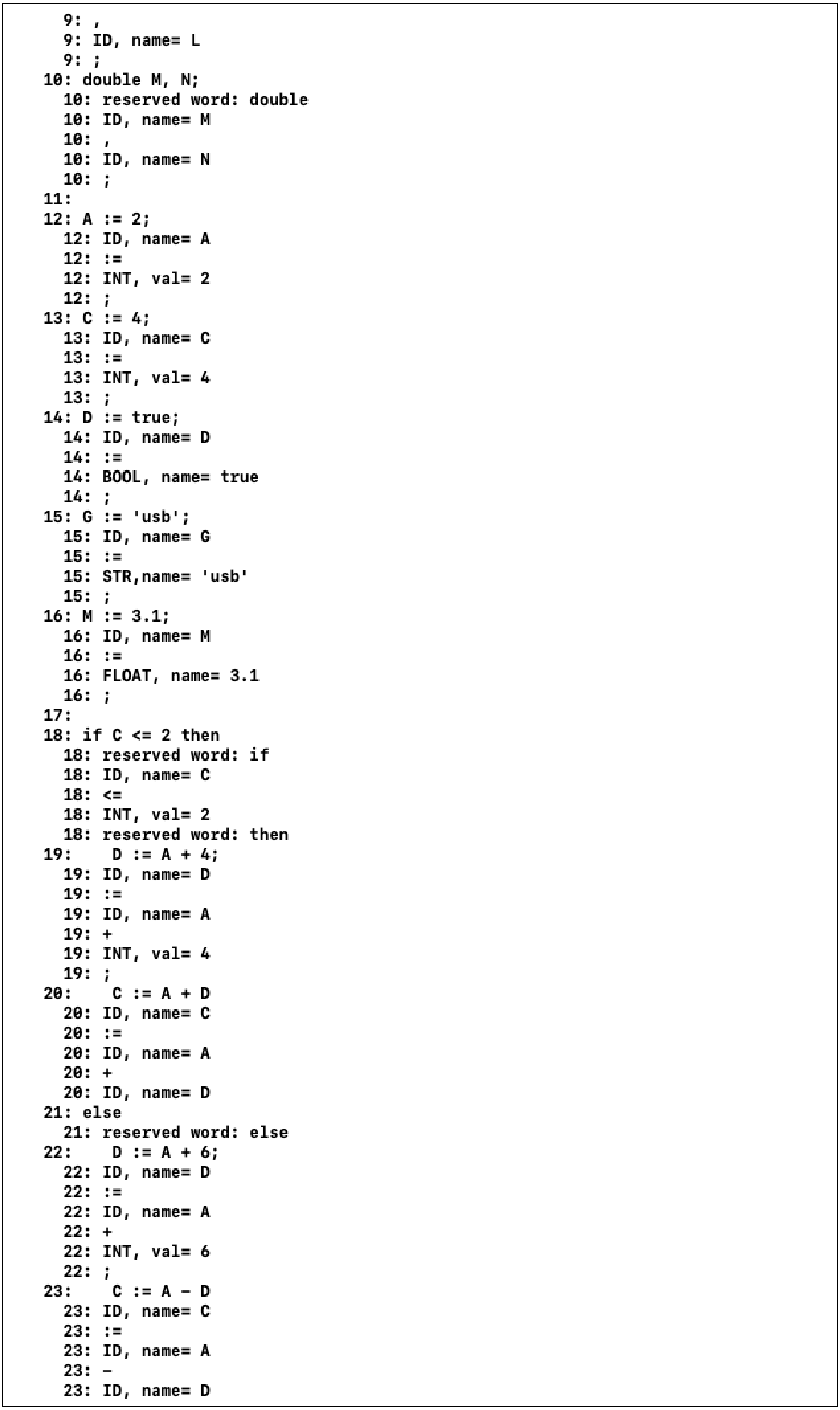


```
{
 This is an simply
 sophisticated example!
}

int A, B, C;
bool D, E, F;
string G, H, I;
float J, K, L;
double M, N;

A := 2;
C := 4;
D := true;
G := 'usb';
M := 3.1;

if C <= 2 then
   D := A + 4;
   C := A + D
else
   D := A + 6;
   C := A - D
end;

do
   D := E;
   C := A * D
while D = E;

repeat
   C := C / C
until C = 1

{GoodBye!}
```

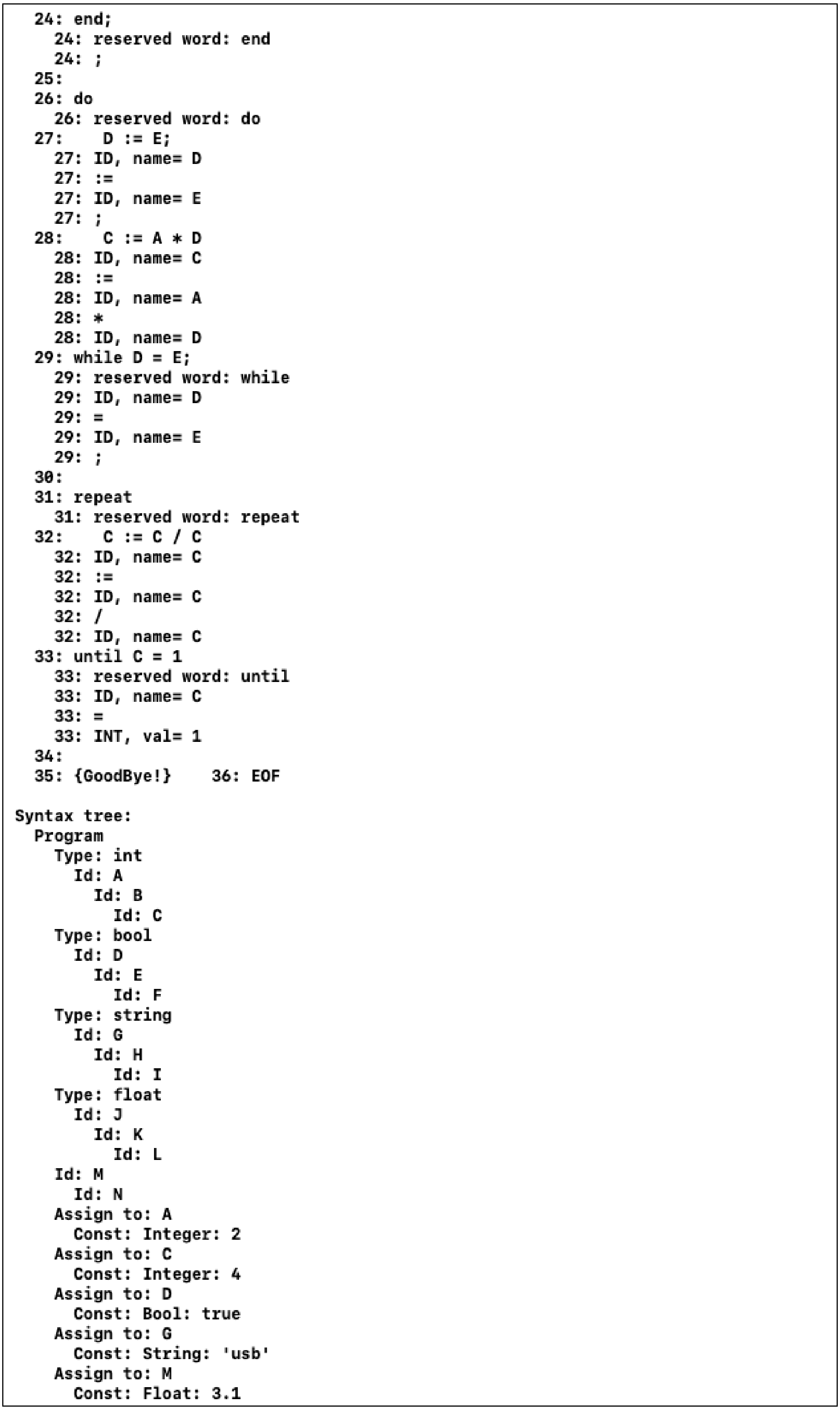
运行结果如下：



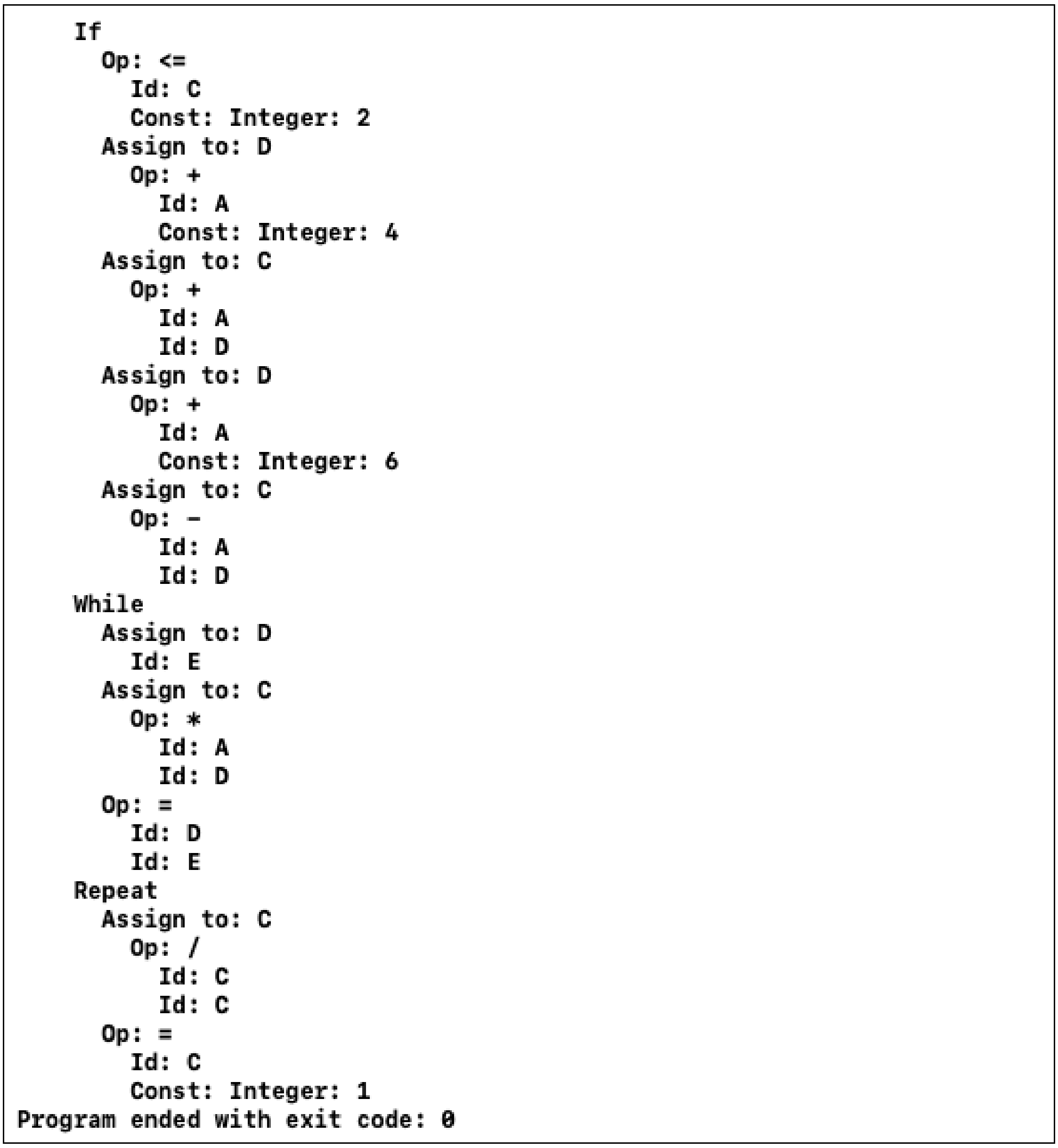
```
TINY COMPILATION: /Users/ernest/TinyPlusParser/TinyPlusParser/testcode/tiny+1.txt
   1: {
   2:  This is an simply
   3:  sophisticated example!
   4: }
   5:
   6: int A, B, C;
    6: reserved word: int
    6: ID, name= A
    6: ,
    6: ID, name= B
    6: ,
    6: ID, name= C
    6: ;
   7: bool D, E, F;
    7: reserved word: bool
    7: ID, name= D
    7: ,
    7: ID, name= E
    7: ,
    7: ID, name= F
    7: ;
   8: string G, H, I;
    8: reserved word: string
    8: ID, name= G
    8: ,
    8: ID, name= H
    8: ,
    8: ID, name= I
    8: ;
   9: float J, K, L;
    9: reserved word: float
    9: ID, name= J
    9: ,
    9: ID, name= K
```



```
  9: ,
  9: ID, name= L
  9: ;
10: double M, N;
  10: reserved word: double
  10: ID, name= M
  10: ,
  10: ID, name= N
  10: ;
11:
12: A := 2;
  12: ID, name= A
  12: :=
  12: INT, val= 2
  12: ;
13: C := 4;
  13: ID, name= C
  13: :=
  13: INT, val= 4
  13: ;
14: D := true;
  14: ID, name= D
  14: :=
  14: BOOL, name= true
  14: ;
15: G := 'usb';
  15: ID, name= G
  15: :=
  15: STR,name= 'usb'
  15: ;
16: M := 3.1;
  16: ID, name= M
  16: :=
  16: FLOAT, name= 3.1
  16: ;
17:
18: if C <= 2 then
  18: reserved word: if
  18: ID, name= C
  18: <=
  18: INT, val= 2
  18: reserved word: then
19:    D := A + 4;
  19: ID, name= D
  19: :=
  19: ID, name= A
  19: +
  19: INT, val= 4
  19: ;
20:    C := A + D
  20: ID, name= C
  20: :=
  20: ID, name= A
  20: +
  20: ID, name= D
21: else
  21: reserved word: else
22:    D := A + 6;
  22: ID, name= D
  22: :=
  22: ID, name= A
  22: +
  22: INT, val= 6
  22: ;
23:    C := A - D
  23: ID, name= C
  23: :=
  23: ID, name= A
  23: -
  23: ID, name= D
```



```
  24: end;
    24: reserved word: end
    24: ;
  25:
  26: do
    26: reserved word: do
  27:    D := E;
    27: ID, name= D
    27: :=
    27: ID, name= E
    27: ;
  28:    C := A * D
    28: ID, name= C
    28: :=
    28: ID, name= A
    28: *
    28: ID, name= D
  29: while D = E;
    29: reserved word: while
    29: ID, name= D
    29: =
    29: ID, name= E
    29: ;
  30:
  31: repeat
    31: reserved word: repeat
  32:    C := C / C
    32: ID, name= C
    32: :=
    32: ID, name= C
    32: /
    32: ID, name= C
  33: until C = 1
    33: reserved word: until
    33: ID, name= C
    33: =
    33: INT, val= 1
  34:
  35: {GoodBye!}    36: EOF

Syntax tree:
  Program
    Type: int
      Id: A
        Id: B
          Id: C
    Type: bool
      Id: D
        Id: E
          Id: F
    Type: string
      Id: G
        Id: H
          Id: I
    Type: float
      Id: J
        Id: K
          Id: L
    Id: M
      Id: N
    Assign to: A
      Const: Integer: 2
    Assign to: C
      Const: Integer: 4
    Assign to: D
      Const: Bool: true
    Assign to: G
      Const: String: 'usb'
    Assign to: M
      Const: Float: 3.1
```



```
    If
      Op: <=
        Id: C
        Const: Integer: 2
      Assign to: D
        Op: +
          Id: A
          Const: Integer: 4
      Assign to: C
        Op: +
          Id: A
          Id: D
      Assign to: D
        Op: +
          Id: A
          Const: Integer: 6
      Assign to: C
        Op: -
          Id: A
          Id: D
    While
      Assign to: D
        Id: E
      Assign to: C
        Op: *
          Id: A
          Id: D
      Op: =
        Id: D
        Id: E
    Repeat
      Assign to: C
        Op: /
          Id: C
          Id: C
      Op: =
        Id: C
        Const: Integer: 1
Program ended with exit code: 0
```

心得体会：

本次实验相比于上次实验主要是工作量比较大，但是读懂代码。按照自动机的逻辑来编写契合给定代码的补充，整体不难！

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：