语法分析程序实验报告

17341092 梁萍佳

一、实验目的

通过扩展已有的样例语言TINY的语法分析程序，为扩展TINY语言TINY＋构造语法分析程序，从而掌握语法分析程序的构造方法

二、实验内容

用EBNF描述TINY＋的语法，用C语言扩展TINY的语法分析程序，构造TINY＋的递归下降语法分析器

三、实验要求

将TOKEN序列转换成语法分析树，并能检查一定的语法错误

四、实验过程

下面我们首先找到TINY+的EBNF描述，然后根据描述来完成分析函数，从而完成构造分析树的语法分析程序。

1.语法定义

下面是TINY+的EBNF描述，红色字体的是TINY+语言扩充的语法规定

1 program → declarations stmt-sequence

2 declarations → decl ; declarations | ε

3 decl → type-specifier varlist

4 type-specifier → ***int*** | ***bool*** | ***char***

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 → ***while*** bool-exp ***do*** stmt-sequence ***end***

9 if-stmt → ***if*** bool-exp ***then*** stmt-sequence [***else*** stmt-sequence] ***end***

10 repeat-stmt → ***repeat*** stmt-sequence ***until*** bool-exp

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

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

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

14 exp -> arithmetic-exp | bool-exp | string-exp

15 arithmetic-exp -> term { addop term }

16 addop -> + | -

17 term -> factor { mulop factor }

18 mulop -> * | /

19 factor -> ***(***arithmetic-exp***)*** | ***number*** | ***identifier***

20 bool-exp -> bterm { ***or*** bterm }

21 bterm -> bfactor { ***and*** bfactor}

22 bfactor -> comparison-exp

23 comparison-exp -> arithmetic-exp comparison-op arithmetic-exp

24 comparison-op -> < | = | > | >= | <=

25 string-exp -> string

2.翻译程序

下面首先将介绍程序中的数据结构定义，然后给出几个辅助函数，最后再详细介绍部分比较有代表性的分析函数。

(1)数据定义

首先，先用枚举类型定义非终端文法符号：

```
enum syntax_symbols{program,declar,decl,type,varlist,
stmt_seq,stmt,while_stmt,if_stmt,repeat_stmt,assign_stmt,read_stmt,write_stmt,
exp,arith_exp,addop,term,mulop,factor,
bool_exp,bterm,bfactor,com_exp,com_op,str_exp};
```

然后再定义语法分析树的节点，这里`symbol`就是节点所代表的非终端文法符号，如果是叶子节点则无定义；`subtree`为该节点的子树集合，其顺序与产生式顺序是一致的；`token`是在当该节点为叶子节点时用来记录对应token的；`void print(int)`方法用来展示整个语法分析树结构：

```
struct Treenode{
    syntax_symbols symbol;
    vector<Treenode> subtree;
    Token token;//当这个是叶子节点时就是token了
    void print(int dep);
};
```

然后还有几个全局变量，这里用`token_cnt`记录当前处理到的位置而不是直接逐个获取token的原因会在后面分析函数那里给出。

```
int token_cnt=0;//计下当前处理到的token号
vector<int> tokens_line;//记一下这个token对应行数
int parsing_depth;//当前分析到的深度
```

(2)辅助函数

在正式开始写分析函数之前，还需要完成一些辅助函数来帮助后面的工作。

首先是输出分析树的方法，这里`dep`为输出深度，用来调节缩进，从而使得在控制台输出更方便看出结构的语法分析树：

```
void Treenode::print(int dep){
    for(int i=0;i<dep;i++){
        if(i%4==3)printf("|");
        else printf(" ");
    }
    if(subtree.size()==0){//叶子节点
        token.print();
        printf("\n");
    }else{
        cout<<syntax_symbols_strs[symbol]<<endl;
        for(int i=0;i<subtree.size();i++){
            subtree[i].print(dep+4);
        }
    }
}
```

然后是调用词法分析程序，获得整个token序列的函数，其中全局变量`tokens`数据类型为`vector<Token>`，定义在语法分析程序中lexical_dev.cpp：

```
//先直接得到整个token序列再说
void prepare(FILE* source){
    // FILE* source = fopen("src.txt", "r");
    while (cursym != -1) {
        Token a=getNextToken(source);
        // a.print();
        // cout<<" ";
        tokens.push_back(a);
        tokens_line.push_back(line_cnt);
    }
    if(tokens[tokens.size()-1].type!=-1){
        tokens.push_back(Token());
        tokens_line.push_back(line_cnt);
    }
}
```

接着是用来在语法分析时遇到语法错误时输出提示信息的函数，`expect`为预期需要的输入符号，`t`为当前实际得到的token：

```
void syntax_error(string expect,Token t){
    cout<<"syntax error at line "<<tokens_line[token_cnt]<<": expect "
<<expect<<", but got a ";
    t.print();
    cout<<endl;
}
```

用来在分析过程中追踪输出分析过程的两个函数，分别在分析函数进入和退出时调用

```
void parsing_mes_in(string mes){//用来在分析过程输出，方便知道分析过程
    for(int i=0;i<parsing_depth;i++){
        if(i%4==3)printf("|");
        else printf(" ");
    }
    cout<<mes<<" in ";
    tokens[token_cnt].print();
    cout<<endl;
    parsing_depth+=4;
}
void parsing_mes_out(string mes){//用来在分析过程输出，方便知道分析过程
    parsing_depth-=4;
    for(int i=0;i<parsing_depth;i++){
        if(i%4==3)printf("|");
        else printf(" ");
    }
    cout<<mes<<" out ";
    tokens[token_cnt].print();
    cout<<endl;
}
```

最后是用来匹配终结符号和token的函数

```
//终结符号与token的匹配
//终结符号有 各类符号，关键字，ID，string ，NUM
bool match(Token t,string t_sym){
    if(t_sym=="identifier"){
        return t.type==ID;
    }
    if(t_sym=="string"){
        return t.type==STR;
    }
    if(t_sym=="number"){
        return t.type==NUM;
    }
    if(t_sym=="$"){
        return t.type==EMPTY;
    }
    for(int i=0;i<15;i++){//看是否要匹配关键字
        if(t_sym==keywords[i]){
            if(t.type==KEY && t.value==t_sym){
                return true;
            }else{
                return false;
            }
        }
    }
    for(int i=0;i<15;i++){//看是否要匹配符号
        if(t_sym==symbols[i]){
            if(t.type==SYM && t.value==t_sym){
                return true;
            }else{
                return false;
            }
        }
```

```
    }
    //正常程序不管输入源码如何都应该是不会到这的
    cout<<"program bug:there is no "<<t_sym<<endl;
    exit(0);
}
```

(3)分析函数

所有分析函数开头都会调用`parsing_mes_in("自身函数名")`来告知分析进入到了该阶段，并设置好返回节点代表的非终端符号；结尾还会再调用`parsing_mes_out("自身函数名")`来告知退出该分析函数了。

①

下面首先介绍的是最简单的不需要考虑选择哪类产生式的分析函数，例如规则1，3，8，9，10，11，12，13，22，23，25

1 program -> declarations stmt-sequence

3 decl -> type-specifier varlist

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

9 if-stmt -> ***if*** bool-exp ***then*** stmt-sequence [***else*** stmt-sequence] ***end***

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

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

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

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

22 bfactor -> comparison-exp

23 comparison-exp -> arithmetic-exp comparison-op arithmetic-exp

25 string-exp -> string

其中一个代表为`parse_program()`，这里只需要获得其产生式右端分析来的分析树，然后加入到自己子树中，再返回即可。

```
Treenode parse_program(){//程序 对应规则1
    parsing_mes_in("parse_program");
    Treenode retnode;
    retnode.symbol=program;

    Treenode declarations=parse_declar();
    retnode.subtree.push_back(declarations);

    Treenode stmt_sequence=parse_stmt_seq();
    retnode.subtree.push_back(stmt_sequence);

    parsing_mes_out("parse_program");
    return retnode;
```

```
}
```

剩下的规则对应分析函数基本一样，除了规则22较为多余所以分析函数直接简单返回`parse_com_exp()`，以及规则25直接用`match`函数代替了分析函数。

②

一些分析函数要处理的产生式中带有自身递归或是产生式右端有某些部分会有重复的，例如规则2，5，6，15，17，20，21

其中一个代表为规则2对应的`parse_declar()`，可以看到这里分析函数直接将递归拆解成了一个while循环来做。注意到first(declarations)={int,bool,char}而follow(declarations)=first(stmt-sequence)=first(statement)={while,if,repeat,identifier,read,write}两者没有交集，因此只需要确认当前token属于哪个集合就可以判断是继续运用产生式declarations->decl;declarations，还是运用产生式declarations->ε完成分析。

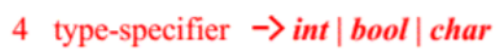
```
Treenode parse_declar(){//声明部分语句  对应规则2
    parsing_mes_in("parse_declar");
    Treenode retnode;
    retnode.symbol=declar;

 while(match(tokens[token_cnt],"int")||match(tokens[token_cnt],"bool")||match(to
kens[token_cnt],"char")){
        retnode.subtree.push_back(parse_decl());
        if(!match(tokens[token_cnt],";")){//声明语句完了居然没有; 语法错
            syntax_error(";",tokens[token_cnt]);
        }else{//有那加入然后就下一个
            Treenode leaf;
            leaf.token=tokens[token_cnt];
            token_cnt++;
            retnode.subtree.push_back(leaf);
        }
    }
    parsing_mes_out("parse_declar");
    return retnode;
}
```

其余类似函数不再赘述。

③

一些分析函数处理的规则只需要简单进行匹配即可，例如规则4，7，16，18，19，24



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

16 addop -> + | -

18 mulop -> * | /

19 factor -> (arithmetic-exp) | *number* | *identifier*

24 comparison-op -> < | = | > | >= | <=

```
Treenode parse_type(){//类型标识符 对应规则4
    parsing_mes_in("parse_type");
    Treenode retnode;
    retnode.symbol=type;
    string strs[]={"int","bool","char"};
    for(int i=0;i<3;i++){
        if(match(tokens[token_cnt],strs[i])){
            Treenode leaf;
            leaf.token=tokens[token_cnt];
            token_cnt++;
            retnode.subtree.push_back(leaf);
            parsing_mes_out("parse_type");
            return retnode;
        }
    }
    //没匹配上，理论上也不会到这的
    syntax_error("type-specifier",tokens[token_cnt]);
    exit(0);
}
```

剩下的规则对应分析函数基本一样，除了规则16、18、24较为简单所以直接用`match`函数进行或运算来代替了。

④

最后剩下一个比较特殊的，是需要特别处理的分析函数，对应规则14

14 exp -> arithmetic-exp | bool-exp | string-exp

这个特殊在于其需要选择产生式，而first(bool-exp)与first(arithmetic-exp)是一样的，从这个推导就可以看出来：`bool-exp -> bterm -> bfactor -> comparison-exp -> arithmetic-exp comparison-op arithmetic-exp`

为了处理这个产生式的选择，只向前看一个token是不够的，这也是为什么上面定义那里说要先获取整个token序列而不是逐个获取token的原因。要确定exp是bool-exp还是arithmetic-exp，关键在于看整个exp有无comparison-op，而这就需要知道exp对应的token有哪些了。为此，我们先计算follow(exp):

```
follow(exp) =follow(write-stmt)+follow(assign-stmt)
            =follow(statement)
            = {';'} + follow(stmt-sequence)
            = {';' + '$' + "end" + "else" + "until"}
```

有了follow(exp)之后，我们就只需要向前一直扫描token，直到出现follow(exp)为止，如果中间出现了comparison-op，则可以知道应该用exp->bool-exp，否则就要用exp->arithmetic-exp

对应代码如下

```
Treenode parse_exp(){//规则14
    parsing_mes_in("parse_exp");
    Treenode retnode;
    retnode.symbol=exp;
    Treenode t1;
    if(match(tokens[token_cnt],"string")){//是str那就直接好判断
        t1.token=tokens[token_cnt];
        token_cnt += 1;
    }else{//由于bool-exp也可能以arithmetic-exp开头，所以这里还得先判断
        //也可以通过更改文法定义的方式来做，但这里就直接简单做法好了
        //先找到follow(exp)=follow(write-stmt)+follow(assign-
stmt)=follow(statement)
        // follow(statement)= ';' + follow(stmt-sequence)
        //  = ';' + '$' + "end" + "else" + "until"
        string follow_exp[]={";","$","end","else","until"};
        string comops[]={"<","=",">","<=",">="};
        int i=1;//从当前之后第一个开始找，因为arithmetic-exp的first不为空
        int bool_mark=0;

        while(1){
            int end_mark=0;
            for(int j=0;j<5;j++){//是否已经遇到exo之后了
                if(match(tokens[token_cnt+i],follow_exp[j])){
                    end_mark=1;
                }
            }
            if(end_mark==1)break;
            for(int j=0;j<5;j++){//找下有没有比较运算符
                if(match(tokens[token_cnt+i],comops[j])){
                    bool_mark=1;
                    end_mark=1;
                    break;
                }
            }
            if(end_mark==1)break;
            i++;
            if(i>=tokens.size()){//正常程序最后一个token是结束符应该不会执行到这里的
                cout<<"program bug,exceed tokens' size"<<endl;
                exit(0);
            }
        }
        //现在已经知道该选择哪个表达式了
        if(bool_mark==1){
            t1=parse_bool_exp();
        }else{
            t1=parse_arith_exp();
        }
    }
    retnode.subtree.push_back(t1);
    parsing_mes_out("parse_exp");
    return retnode;
```

```
}
```

至此，各类分析函数的代表都介绍完了。

五、验证结果

为了验证得到的程序的正确性，下面编写了调用语法分析的程序，以及几段样例TINY+代码，然后观察输出是否符合预期。程序将接收待分析的源代码文件名，然后先词法分析并输出token序列，然后进行语法分析，最后输出生成的语法树。

```
int main(){
    char fname[100];
    scanf("%s",fname);
    FILE* source = fopen(fname, "r");
    prepare(source);
    printf("token list:\n");
    for (int i = 0; i < tokens.size(); i++) {
        tokens[i].print();
        cout <<"  ";
        if(i%5==4)cout<<endl;
    }
    cout<<endl<<endl;

    Treenode root=parse_program();

    cout<<endl<<"parsing result:"<<endl;
    root.print(0);
    return 0;
}
```

下面将用两个正确的样例程序和四个有语法错误的样例程序来进行验证。

正确样例1

src1.txt

```
{this is an example source code}
int A,B,C,D;
while A<C and B>D do
    if A=1 then A:=B*C+37
           else repeat A:=A*2
        until A+C<=B+D
       end
end
```

输出语法分析树，里面同一层的节点会在同一列紧贴着同一条竖线，每个节点与它同层的下一个节点之间的节点为该节点的子树。

注意到由于有递归的规则对应分析函数那里将递归拆解成循环了，所以产生的分析树中例如varlist下面的各个ID，逗号，都会处于同一层。

```
program
   |declar
   |   |decl
   |   |   |type
```

```
|   |   |   |(KEY,int)
|   |   |varlist
|   |   |   |(ID,A)
|   |   |   |(SYM,,)
|   |   |   |(ID,B)
|   |   |   |(SYM,,)
|   |   |   |(ID,C)
|   |   |   |(SYM,,)
|   |   |   |(ID,D)
|   |(SYM,;)
|stmt_seq
|   |stmt
|   |   |while_stmt
|   |   |   |(KEY,while)
|   |   |   |bool_exp
|   |   |   |   |bterm
|   |   |   |   |   |com_exp
|   |   |   |   |   |   |arith_exp
|   |   |   |   |   |   |   |term
|   |   |   |   |   |   |   |   |factor
|   |   |   |   |   |   |   |   |   |(ID,A)
|   |   |   |   |   |   |(SYM,<)
|   |   |   |   |   |   |arith_exp
|   |   |   |   |   |   |   |term
|   |   |   |   |   |   |   |   |factor
|   |   |   |   |   |   |   |   |   |(ID,C)
|   |   |   |   |   |(KEY,and)
|   |   |   |   |   |com_exp
|   |   |   |   |   |   |arith_exp
|   |   |   |   |   |   |   |term
|   |   |   |   |   |   |   |   |factor
|   |   |   |   |   |   |   |   |   |(ID,B)
|   |   |   |   |   |   |(SYM,>)
|   |   |   |   |   |   |arith_exp
|   |   |   |   |   |   |   |term
|   |   |   |   |   |   |   |   |factor
|   |   |   |   |   |   |   |   |   |(ID,D)
|   |   |   |(KEY,do)
|   |   |   |stmt_seq
|   |   |   |   |stmt
|   |   |   |   |   |if_stmt
|   |   |   |   |   |   |(KEY,if)
|   |   |   |   |   |   |bool_exp
|   |   |   |   |   |   |   |bterm
|   |   |   |   |   |   |   |   |com_exp
|   |   |   |   |   |   |   |   |   |arith_exp
|   |   |   |   |   |   |   |   |   |   |term
|   |   |   |   |   |   |   |   |   |   |   |factor
|   |   |   |   |   |   |   |   |   |   |   |   |(ID,A)
|   |   |   |   |   |   |   |   |   |(SYM,=)
|   |   |   |   |   |   |   |   |   |arith_exp
|   |   |   |   |   |   |   |   |   |   |term
|   |   |   |   |   |   |   |   |   |   |   |factor
|   |   |   |   |   |   |   |   |   |   |   |   |(NUM,1)
|   |   |   |   |   |   |(KEY,then)
|   |   |   |   |   |   |stmt_seq
|   |   |   |   |   |   |   |stmt
|   |   |   |   |   |   |   |   |assign_stmt
```

```
    |   |   |   |   |   |   |   |   |   |(ID,A)
    |   |   |   |   |   |   |   |   |   |(SYM,:=)
    |   |   |   |   |   |   |   |   |   |exp
    |   |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,B)
    |   |   |   |   |   |   |   |   |   |   |   |   |(SYM,*)
    |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,C)
    |   |   |   |   |   |   |   |   |   |   |   |(SYM,+)
    |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(NUM,37)
    |   |   |   |   |   |   |(KEY,else)
    |   |   |   |   |   |   |stmt_seq
    |   |   |   |   |   |   |   |stmt
    |   |   |   |   |   |   |   |   |repeat_stmt
    |   |   |   |   |   |   |   |   |   |(KEY,repeat)
    |   |   |   |   |   |   |   |   |   |stmt_seq
    |   |   |   |   |   |   |   |   |   |   |stmt
    |   |   |   |   |   |   |   |   |   |   |   |assign_stmt
    |   |   |   |   |   |   |   |   |   |   |   |   |(ID,A)
    |   |   |   |   |   |   |   |   |   |   |   |   |(SYM,:=)
    |   |   |   |   |   |   |   |   |   |   |   |   |exp
    |   |   |   |   |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,A)
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |(SYM,*)
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |(NUM,2)
    |   |   |   |   |   |   |   |   |   |(KEY,until)
    |   |   |   |   |   |   |   |   |   |bool_exp
    |   |   |   |   |   |   |   |   |   |   |bterm
    |   |   |   |   |   |   |   |   |   |   |   |com_exp
    |   |   |   |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,A)
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(SYM,+)
    |   |   |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,C)
    |   |   |   |   |   |   |   |   |   |   |   |   |(SYM,<=)
    |   |   |   |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,B)
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(SYM,+)
    |   |   |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,D)
    |   |   |   |   |   |   |   |(KEY,end)
    |   |   |   |(KEY,end)
```

正确样例2

src2.txt

```
{this is an example source code}
int x,fact;
read x;
if x>0 and x<100 then {don't compute if x<=0}
    fact:=1;
    while x>0 do
        fact:=fact*x;
        x:=x-1
    end;
    write fact
end
```

输出语法分析树

```
program
   |declar
   |   |decl
   |   |   |type
   |   |   |   |(KEY,int)
   |   |   |varlist
   |   |   |   |(ID,x)
   |   |   |   |(SYM,,)
   |   |   |   |(ID,fact)
   |   |(SYM,;)
   |stmt_seq
   |   |stmt
   |   |   |read_stmt
   |   |   |   |(KEY,read)
   |   |   |   |(ID,x)
   |   |(SYM,;)
   |   |stmt
   |   |   |if_stmt
   |   |   |   |(KEY,if)
   |   |   |   |bool_exp
   |   |   |   |   |bterm
   |   |   |   |   |   |com_exp
   |   |   |   |   |   |   |arith_exp
   |   |   |   |   |   |   |   |term
   |   |   |   |   |   |   |   |   |factor
   |   |   |   |   |   |   |   |   |   |(ID,x)
   |   |   |   |   |   |   |(SYM,>)
   |   |   |   |   |   |   |arith_exp
   |   |   |   |   |   |   |   |term
   |   |   |   |   |   |   |   |   |factor
   |   |   |   |   |   |   |   |   |   |(NUM,0)
   |   |   |   |   |   |(KEY,and)
   |   |   |   |   |   |com_exp
   |   |   |   |   |   |   |arith_exp
   |   |   |   |   |   |   |   |term
   |   |   |   |   |   |   |   |   |factor
   |   |   |   |   |   |   |   |   |   |(ID,x)
   |   |   |   |   |   |(SYM,<)
   |   |   |   |   |   |arith_exp
   |   |   |   |   |   |   |term
   |   |   |   |   |   |   |   |factor
```

```
    |   |   |   |   |   |   |   |   |   |(NUM,100)
    |   |   |   |(KEY,then)
    |   |   |   |stmt_seq
    |   |   |   |   |stmt
    |   |   |   |   |   |assign_stmt
    |   |   |   |   |   |   |(ID,fact)
    |   |   |   |   |   |   |(SYM,:=)
    |   |   |   |   |   |   |exp
    |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |(NUM,1)
    |   |   |   |   |(SYM,;)
    |   |   |   |   |stmt
    |   |   |   |   |   |while_stmt
    |   |   |   |   |   |   |(KEY,while)
    |   |   |   |   |   |   |bool_exp
    |   |   |   |   |   |   |   |bterm
    |   |   |   |   |   |   |   |   |com_exp
    |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |(ID,x)
    |   |   |   |   |   |   |   |   |   |(SYM,>)
    |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |(NUM,0)
    |   |   |   |   |   |   |(KEY,do)
    |   |   |   |   |   |   |stmt_seq
    |   |   |   |   |   |   |   |stmt
    |   |   |   |   |   |   |   |   |assign_stmt
    |   |   |   |   |   |   |   |   |   |(ID,fact)
    |   |   |   |   |   |   |   |   |   |(SYM,:=)
    |   |   |   |   |   |   |   |   |   |exp
    |   |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,fact)
    |   |   |   |   |   |   |   |   |   |   |   |   |(SYM,*)
    |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,x)
    |   |   |   |   |   |   |   |(SYM,;)
    |   |   |   |   |   |   |   |stmt
    |   |   |   |   |   |   |   |   |assign_stmt
    |   |   |   |   |   |   |   |   |   |(ID,x)
    |   |   |   |   |   |   |   |   |   |(SYM,:=)
    |   |   |   |   |   |   |   |   |   |exp
    |   |   |   |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |   |(ID,x)
    |   |   |   |   |   |   |   |   |   |   |(SYM,-)
    |   |   |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |   |   |(NUM,1)
    |   |   |   |   |   |   |(KEY,end)
    |   |   |   |   |(SYM,;)
```

```
    |   |   |   |   |stmt
    |   |   |   |   |   |write_stmt
    |   |   |   |   |   |   |(KEY,write)
    |   |   |   |   |   |   |exp
    |   |   |   |   |   |   |   |arith_exp
    |   |   |   |   |   |   |   |   |term
    |   |   |   |   |   |   |   |   |   |factor
    |   |   |   |   |   |   |   |   |   |   |(ID,fact)
    |   |   |   |(KEY,end)
```

错误样例1

src_error1.txt

```
{wrong example source code 1:expect keyword 'do'}
int A,B;
while A<B
    if A=1 then A:=B
```

这里面while-exp少了个do关键字，程序正确的发现了这个错误：

```
   |   |   |   |parse_bterm out (KEY,if)
   |   |   |parse_bool_exp out (KEY,if)
syntax error at line 4: expect keyword 'do', but got a (KEY,if)
```

错误样例2

src_error2.txt

```
{wrong example source code 2:expect identifier}
int A,B;
char ;
A:=B
```

这里面声明语句后面缺了标识符，程序正确的发现了这个错误：

```
   |   |   |parse_type out (SYM,;)
   |   |   |parse_varlist in (SYM,;)
syntax error at line 3: expect identifier, but got a (SYM,;)

--------------------------------
```

错误样例3

src_error3.txt

```
{wrong example source code 3:expect symbol ')' }
int A,B;
A:= ((A+B)+(A*B)
```

这里面算术表达式的左右括号不匹配，缺了一个右括号，程序正确的发现了这个错误：

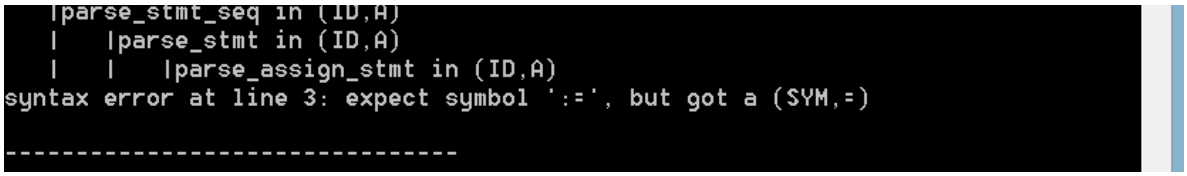
```
   |   |   |   |   |   |   |   |parse_arith_exp out (-1,)
syntax error at line 3: expect symbol ')' , but got a (-1,)
```

错误样例4

src_error4.txt

```
{wrong example source code 4:expect symbol ':='}
int A;
A=5
```

这里面赋值语句错误地使用了布尔表达式里面的'='而不是':='，程序正确的发现了这个错误：



六、实验总结

这次实验我找到了关于TINY+语法定义的资料，然后根据其定义以及自顶向下的递归下降分析法来实现了分析生成语法分析树的算法。这个递归分析过程中大部分函数都是比较简单容易能够实现的，不太需要选择使用哪个产生式，只有少数是需要稍加考虑的。这个语法分析程序还能检查一定的语法错误，给出所在行数和预期要得到的词，从而方便定位。最后我还给出了几段程序，包括正确的样例和有语法错误的样例，通过实验验证了程序的正确性。