

# TINY+ 语法分析程序实验

---

## TINY+ 语法分析程序实验

实验目的

实验内容

实验要求

实验环境

TINY+语言

实验过程

实验测试结果展示

心得体会

## 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。

## 实验内容

---

用EBNF描述TINY + 的语法，用C/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。

## 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

## 实验环境

---

- Linux(Ubuntu)
- g++
- make

## TINY+语言

---

1. 语法EBNF定义如下

## EBNF定义

1 program -> declarations stmt-sequence

2 declarations -> decl ; declarations |  $\epsilon$

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

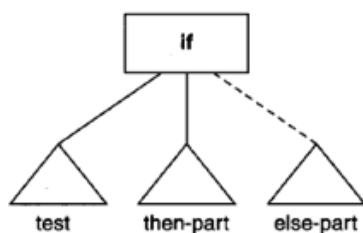
24 comparison-op -> < | = | > | >= | <=

25 string-exp -> string

其中Tiny+新增定义为2, 3, 4, 5, 8, 14, 20-25。

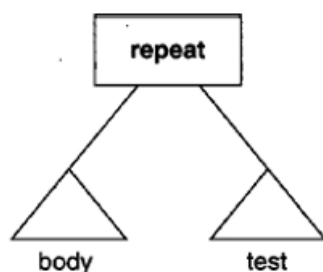
## 2. 语法树基本结构：

### ○ If语句



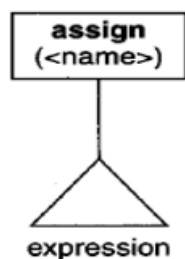
test位于 If 语句的child[0], then-part位于child[1], else-part位于child[2]。

### ○ Repeat语句



body位于repeat语句中的child[0], test位于child[1]

### ○ Assign语句



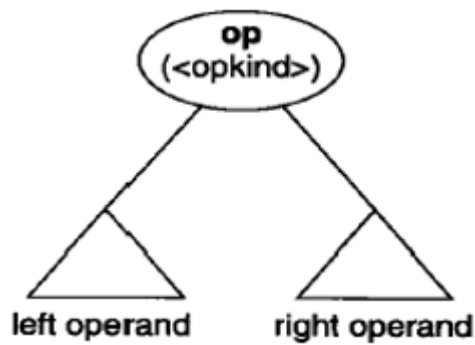
Expression 位于assign语句child[0]

### ○ write语句



Expression位于write的child[0]

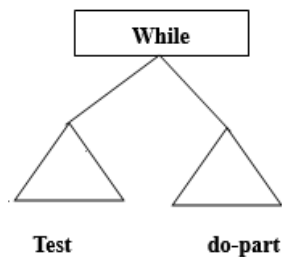
### ○ Op逻辑运算



Left-operand 位于op语句的child[0],

Right-operand 位于语句的child[1];

- while语句



Test位于while语句的child[0]

Do-part位于child[1]

- .....

### 3. 语法分析

- 语法错误检查

- 语法结构的开始符号以及跟随符号错误;
- 标识符错误, 例如在程序的变量声明中, 关键字**int** 后没有跟随标识符;
- 括号不匹配的的错误, 例如左括号(和右括号 )不匹配。
- 符号错误, 例如赋值语句中要求使用的正确符号是‘=’, 而在关系比较表达式要求使用的正确符号是‘==’。
- .....

## 实验过程

1. 在global.h文件中定义TreeNode类来定义语法树的节点。类里面主要定义了节点的类型以及节点的值类型以及其具体的值, 其次就是一些createNode新建节点的函数, 根据不同的参数, 不同的节点类型需要不同的函数, 以及setchild设置它的子节点。部分代码如下:

```

class TreeNode
{
public:
    TreeNode *child[MAXCHILDREN];
    NodeType nodeType; // 节点的类型
    ValType valType; // 节点值的类型
    /**tk属性**/
    /**该属性随NodeType的不同而意义不同*/
    /**FACTOR—tk存储的是对应的token**/
    /**READ_STMT—tk存储的是待读的对象对应的token**/
    /**ASSIGN_STMT—tk存储的是待赋值的对象对应的token**/
    Token *tk;
    void setChild(TreeNode *c1 = nullptr, TreeNode *c2 = nullptr, TreeNode *c3 = nullptr)
    {
        /**父认子**/
        child[0] = c1;
        child[1] = c2;
        child[2] = c3;
    }
    TreeNode(NodeType nodeType, ValType valType = VT_INT, Token *tk = nullptr,
            TreeNode *c1 = nullptr,
            TreeNode *c2 = nullptr,
            TreeNode *c3 = nullptr)
        : nodeType(nodeType), valType(valType), tk(tk)
    {
        setChild(c1, c2, c3);
    }
    static TreeNode *
    create_Node(NodeType nodeType, ValType valType, TreeNode *c1 = nullptr, TreeNode *c2 = nullptr,
            TreeNode *c3 = nullptr)
    {
        return new TreeNode(nodeType, valType, nullptr, c1, c2, c3);
    }
    static TreeNode *
    create_Node(NodeType nodeType, TreeNode *c1 = nullptr, TreeNode *c2 = nullptr, TreeNode *c3 = nullptr)
    {
        return new TreeNode(nodeType, VT_INT, nullptr, c1, c2, c3);
    }
    static TreeNode *
    create_Node(NodeType nodeType, Token *tk)
    {
        return new TreeNode(nodeType, VT_INT, tk);
    }
};

```

其中，节点类型和节点值类型做以下枚举表示：

```

enum NodeType
{
    PROGRAM = 0,
    STMT_SEQUENCE,
    IF_STMT,
    REPEAT_STMT,
    ASSIGN_STMT,
    READ_STMT,
    WRITE_STMT,
    WHILE_STMT,

    GTR_EXP, // 大于
    GEQ_EXP, // 大于等于
    LEQ_EXP, // 小于等于
    LSS_EXP, // 小于
    EQU_EXP, // 等于
    LOG_OR_EXP,
    LOG_AND_EXP,
    LOG_NOT_EXP,
    ADD_EXP,
    SUB_EXP,
    MUL_EXP,
    DIV_EXP,
    FACTOR
};

```

```

typedef enum
{
    /**函数**/
    OT_FUNC,
    /**变量**/
    OT_VAR,
    /**常量**/
    OT_CONST
} ObjType;

typedef enum
{
    /**整型**/
    VT_INT,
    /**布尔**/
    VT_BOOL,
    /**字符串**/
    VT_STR
} ValType;

```

2. 在parser.h文件中声明函数，针对不同的语句，采用不同的创造节点的创操作，代码截图如下：

```
#ifndef TINY_PARSER_H
#define TINY_PARSER_H

#include "global.h"

TreeNode *parse();

TreeNode *mul_exp();
TreeNode *add_exp();
TreeNode *stmt_sequence();
TreeNode *read_stmt();
TreeNode *logical_and_exp();
TreeNode *program();
TreeNode *assign_stmt(Token id_token);
TreeNode *factor();
TreeNode *logical_or_exp();
TreeNode *if_stmt();
TreeNode *write_stmt();
TreeNode *repeat_stmt();
TreeNode *declarations();
TreeNode *comparison_exp();
TreeNode *while_stmt();

#endif //TINY_LUS_PARSER_H
```

3. 在parser.cpp实现以上函数以供使用。其中还定义了许多其他的辅助函数以方便parser的实现。

定义nextToken函数，其中使用了之前的getToken函数来获取Token序列。

```
void nextToken()
{
    auto tmp = getToken(lineno);
    token.kind = tmp.first;
    if (token.kind == TK_INT)
        token.i_val = atoi(tmp.second.c_str());
    else if (token.kind == TK_STRING || token.kind == ID)
        token.s_val = tmp.second;
}
```

下面对于部分语句的语法树构造做介绍：

对于声明语句我选择直接不展示在语法树中，感觉不是很有必要，所以在declarations中如果token.kind为声明的三种类型，直接跳过调用nextToken。

```
TreeNode *declarations()
{
    while (token.kind == TK_INT || token.kind == TK_BOOL || token.kind == TK_STRING)
    {
        Token type = token; //保存类型的符号
        // nextToken(); //跳过类型声明
        do
        {
            nextToken(); //跳过类型声明
        }
    }
}
```

至于If语句，就需要进行多种判断了。先判断是否有错误，再根据EBNF定义 *if-stmt -> if bool-exp then stmt-sequence [else stmt-sequence] end* 匹配一下是否存在then和else以及end。最后调用之前定义的Create\_Node函数创建节点。

```
TreeNode *if_stmt()
{
    TreeNode *cond_exp = nullptr, *then_stmt = nullptr, *else_stmt = nullptr;
    /** 语义检查 **/
    cond_exp = logical_or_exp();

    if (cond_exp->valType != VT_BOOL)
        throw_syntax_error(SEMANTIC_COND_BOOL_ERROR, lineno);
    match(TK_THEN);
    then_stmt = stmt_sequence();
    if (token.kind == TK_ELSE)
    { // 如果还有else部分...
        match(TK_ELSE);
        else_stmt = stmt_sequence();
    }
    match(TK_END);

    return TreeNode::create_Node(IF_STMT, cond_exp, then_stmt, else_stmt);
}
```

而一些不再需要匹配的语句就容易实现的多。如：

```
TreeNode *write_stmt()
{
    //TK_WRITE 不再需要匹配
    TreeNode *cond_exp;
    cond_exp = logical_or_exp();
    return TreeNode::create_Node(WRITE_STMT, cond_exp);
}

TreeNode *repeat_stmt()
{
    //TK_REPEAT 不再需要匹配
    TreeNode *rep_stmt = nullptr, *cond_exp = nullptr;
    rep_stmt = stmt_sequence();
    match(TK_UNTIL);
    cond_exp = logical_or_exp();
    return TreeNode::create_Node(REPEAT_STMT, rep_stmt, cond_exp);
}
```

还有如 *comparison-op* -> < | = | > | >= | <= 这种的，我定义了一个vector来存放右边的各种TokenType，并实现了一个token\_is\_in函数来判断Token是否在其中，方便实现。最后仍然是调用create\_Node创建节点。部分代码如下：

```
TreeNode *comparison_exp()
{
    TreeNode *arith_exp = nullptr, *comp_exp = nullptr;
    arith_exp = add_exp();
    std::vector<TokenType> comp_op_list{TK_GTR, TK_EQU, TK_GEQ, TK_LEQ, TK_LSS};
    NodeType type;
    if (token_is_in(comp_op_list))
    {
        switch (token.kind)
        {
            case TK_GTR:
                type = GTR_EXP;
                break;
            case TK_GEQ:
                type = GEQ_EXP;
                break;
            case TK_LEQ:
                type = LEQ_EXP;
                break;
            case TK_LSS:
                type = LSS_EXP;
                break;
            case TK_EQU:
                type = EQU_EXP;
                break;
        }
        match_one(comp_op_list);
        comp_exp = comparison_exp();
    }
    if (!comp_exp) // 减少语法树的高度，直接返回
        return arith_exp;
    TreeNode *ret_node = create_Node(type, arith_exp, comp_exp);
}
```

语句还有很多，这里就不一一进行其语法树的创建代码展示了，详见src/pasrser.cpp。

4. 在main函数中调用parser()进行语法分析，返回其root节点，调用printTree函数进行打印。

```
if ((argc >= 3 && !strcmp(argv[2], "tree")) || argc >= 0)
{ // 仅输出tree
    listing = stdout;
    has_error = false;
    TreeNode *root = parse();
    if (has_error)
    {
        fprintf(listing, "error");
    }
    std::cout << "\n语法树如下: " << std::endl;
    printTree(root);
    return 0;
}
```



5. 在print.h和print.cpp中分别声明且定义printTree函数用来打印语法树。  
首先，解决打印空格的问题，需要根据节点的遍历情况改变空格打印的数量。  
代码如下：

```
/* Variable indentno is used by printTree to
 * store current number of spaces to indent
 */
static int indentno = 0;

/* macros to increase/decrease indentation */
#define INDENT indentno += 2
#define UNINDENT indentno -= 2

/* printSpaces indents by printing spaces */
static void printSpaces(void)
{
    for (int i = 0; i < indentno; i++)
        std::cout<<" ";
}
```

在打印的循环中调用INDENT和UNINDENT就可以控制printSpaces的数量。  
然后，就可以开始前序遍历整棵语法树了。函数太长，以下仅展示部分代码，  
详见src/print.cpp中。

```
void printTree(TreeNode *root)
{
    if (root->nodeType != STMT_SEQUENCE)
        INDENT;
    if (root)
    {
        printSpaces();
        switch (root->nodeType)
        {
            case STMT_SEQUENCE:
                // 声明语句就算了，不输出了，占地！
                /*if (root->valType == VT_INT)
                    fprintf(listing, "Int Declaration\n");
                else if (root->valType == VT_BOOL)
                    fprintf(listing, "Bool Declaration\n");
                else
                    fprintf(listing, "Str Declaration\n");
                */
                break;
            case IF_STMT:
                fprintf(listing, "If\n");
                if (root->tk)
                    std::cout << root->tk->s_val << std::endl;
                break;
            case REPEAT_STMT:
                std::cout << "Repeat\n";
                break;
            case ASSIGN_STMT:
                if (root->tk)
                    std::cout << "Assign to " + root->tk->s_val << std::endl;
```

```

case LEQ_EXP:
    std::cout << "Op: (SYM,<=)\n";
    break;
case LSS_EXP:
    std::cout << "Op: (SYM,<)\n";
    break;
case EQU_EXP:
    std::cout << "Op: (SYM,==)\n";
    break;
case LOG_OR_EXP:
    std::cout << "LogicOp: (KEY,or)\n";
    break;
case LOG_AND_EXP:
    std::cout << "LogicOp: (KEY,and)\n";
    break;
case LOG_NOT_EXP:
    std::cout << "LogicOp: (KEY,not)\n";
    break;
case ADD_EXP:
    std::cout << "Op: (SYM,+)\n";
    break;
case SUB_EXP:
    std::cout << "Op: (SYM,-)\n";
    break;
case MUL_EXP:
    std::cout << "Op: (SYM,*)\n";
    break;
case DIV_EXP:
    std::cout << "Op: (SYM,/)\n";
    break;

```

将不同的TokenType用switch case进行判断，输出对应的值及类型。

```

        break;
    case ID:
        std::cout << "ID: " + root->tk->s_val << std::endl;
        break;
    case STRING:
        std::cout << "Str: '\" + root->tk->s_val + '\" << std::endl;
        break;
    }
}
else
{
    //tk为空的唯一一种情况是括号嵌套的结构
    /*s1_str = analyzingStack.pop().val;
    t_str = get_t_num_str();
    emit(C_ASSIGN, t_str, s1_str);
    analyzingStack.push(root, t_str);*/
    //fixed 应该不要做任何操作，因为嵌套的后一次都已经搞定好中间变量了，而且只有一
}
break;
default:
    fprintf(listing, "UNKOWN NODE\n");
    break;
}

for (int i = 0; i < MAXCHILDREN; i++)
    if (root->child[i])
        printTree(root->child[i]);
}
if (root->nodeType != STMT_SEQUENCE)
    UNINDENT;
}

```

以上实验过程部分关键代码展示完毕，详见tiny/src或tiny/include中。

## 实验测试结果展示

- 输入make得到可执行程序main，并在命令行输入参数运行，测试Tiny源程序以及输出语法树如下图（测试test.tny）：

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# make
g++ -lpthread -lm -Iinclude -g -w src/generate.cpp src/main.cpp src/parser.cpp src/print.cpp src/scan.cpp -o bin/main
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny
```

语法树如下：

```

READ x
Repeat
  Assign to A
    Op: (SYM,*)
    ID: A
    Const: 2
  Op: (SYM,<)
    Op: (SYM,+)
    ID: A
    ID: C
    Op: (SYM,+)
    ID: B
    ID: D
While
  Op: (SYM,<)
    Op: (SYM,+)
    ID: A
    Op: (SYM,+)
    ID: B
    ID: C
  Const: 10
  Assign to B
    Op: (SYM,+)
    ID: B
    Const: 3
If
  LogicOp: (KEY,or)
  LogicOp: (KEY,and)
  Op: (SYM,<)
    ID: x
    Const: 10
  Op: (SYM,>)
    ID: x
    Const: 5
  Op: (SYM,<)
    ID: x
    Const: 9
  Assign to fact
    Const: 4
  Assign to fact
    Const: 6

```

其中测试的test.tny如下：

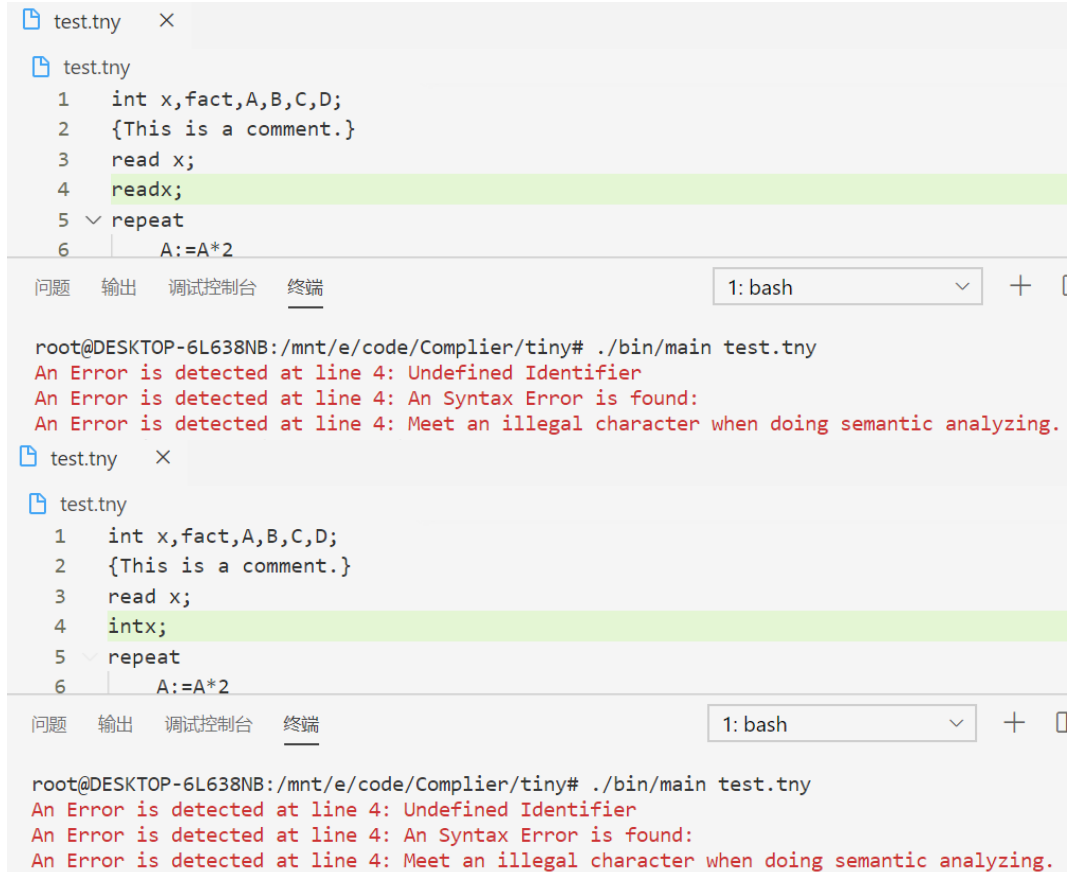
```

test.tny  X
test.tny
1  int x,fact,A,B,C,D;
2  {This is a comment.}
3  read x;
4
5  repeat
6  |   A:=A*2
7  |   until (A+C) < (B+D);
8
9  while (A+B+C) < 10 do
10 |   B := B + 3
11 end;
12
13 if x < 10 and x > 5 or x < 9 then
14 |   fact := 4
15 else
16 |   fact := 6
17 end;

```

- 测试语法错误

○ 关键字后未接空格：



```
test.tny x
test.tny
1  int x,fact,A,B,C,D;
2  {This is a comment.}
3  read x;
4  readx;
5  repeat
6  A:=A*2

问题 输出 调试控制台 终端 1: bash

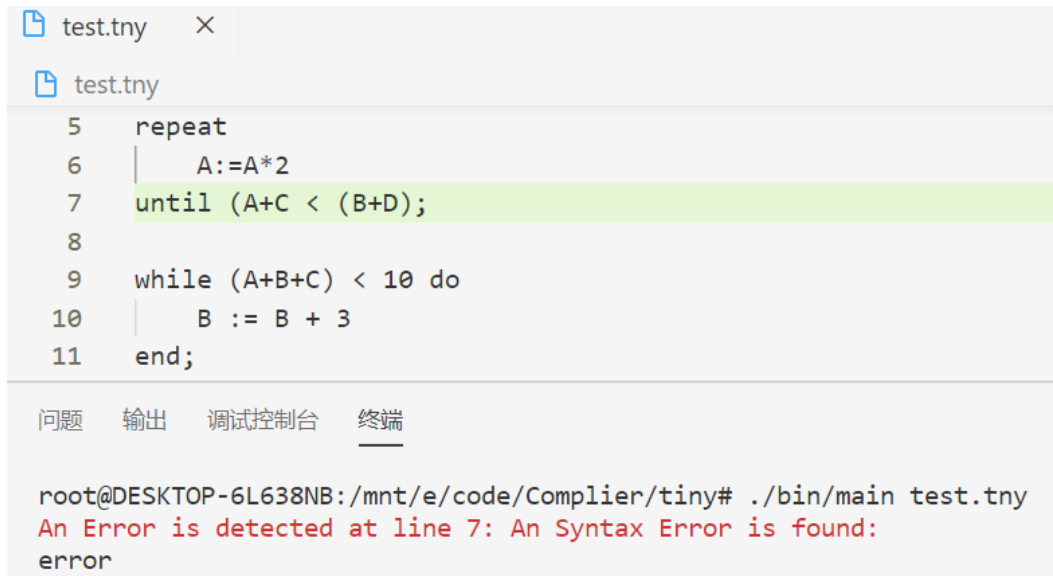
root@DESKTOP-6L638NB:/mnt/e/code/Complier/tiny# ./bin/main test.tny
An Error is detected at line 4: Undefined Identifier
An Error is detected at line 4: An Syntax Error is found:
An Error is detected at line 4: Meet an illegal character when doing semantic analyzing.

test.tny x
test.tny
1  int x,fact,A,B,C,D;
2  {This is a comment.}
3  read x;
4  intx;
5  repeat
6  A:=A*2

问题 输出 调试控制台 终端 1: bash

root@DESKTOP-6L638NB:/mnt/e/code/Complier/tiny# ./bin/main test.tny
An Error is detected at line 4: Undefined Identifier
An Error is detected at line 4: An Syntax Error is found:
An Error is detected at line 4: Meet an illegal character when doing semantic analyzing.
```

○ 左括号和右括号不匹配



```
test.tny x
test.tny
5  repeat
6  A:=A*2
7  until (A+C < (B+D);
8
9  while (A+B+C) < 10 do
10 B := B + 3
11 end;

问题 输出 调试控制台 终端

root@DESKTOP-6L638NB:/mnt/e/code/Complier/tiny# ./bin/main test.tny
An Error is detected at line 7: An Syntax Error is found:
error
```

- 关键字后未接标识符

```
test.tny  X
test.tny
1  int x,fact,A,B,C,D;
2  {This is a comment.}
3  read x;
4  int
5  repeat
6      A:=A*2
7  until (A+C) < (B+D);
```

问题 输出 调试控制台 终端

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny
Code ends before file
```

- :=赋值符号错误

```
test.tny
1  int x,fact,A,B,C,D;
2  {This is a comment.}
3  read x;
4
5  repeat
6      A:A*2
7  until (A+C) < (B+D);
```

问题 输出 调试控制台 终端

```
root@DESKTOP-6L638NB:/mnt/e/code/Compiler/tiny# ./bin/main test.tny
An Error is detected at line 6: An Syntax Error is found:
An Error is detected at line 6: Meet an illegal character when doing semantic analyzing.
```

- 本语法分析程序还能其他语法错误，此处就不一一展示，略占篇幅，欢迎尝试。

## 心得体会

词法分析程序实现的难点还是在于parser的实现，根据EBNF定义对各种语句各种情况进行分类讨论并创造节点。事实上在实现的时候，边根据TokenType进行判断边重载多种创建节点的函数，这样逻辑会清晰一些。经过这次实验，我对语法分析的实现有了更深的理解，并实现了从课堂理论知识到实际应用实现的映射，获益匪浅。