语法分析

17341197 张富瑞

一.实验目的

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

二.实验思路

1.定义的tiny+的EBNF的文法

1 program **->** declarations stmt-sequence

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

3 decl **->** type-specifier varlist

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

5 varlist **-> identifier [,** varlist **]**

6 stmt-sequence **->** statement [**;** stmt-sequence]

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

8 while-stmt **-> while** logical-or-exp **do** stmt-sequence **end**

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

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

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

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

13 write-stmt **-> write** logical-or-exp

14 logical-or-exp **->** logical-and-exp [**or** logical-or-exp]

15 logical-and-exp **->** comparison-exp [**and** logical-and-exp]

16 comparison-exp **->** add-exp [comparison-op comparison-exp]

17 comparison-op **-> <** | **= | > | >= | <=**

18 add-exp **->** mul-exp [addop add-exp]

19 addop **-> +** | **-**

20 mul-exp **->** factor [mulop mul-exp]

21 mulop -> * | /

22 factor -> number | string | identifier | true | false| (logical-or-exp)

2.实验思路

根据定义的文法，将上一个实验中获得的token序列取出，来进行文法的匹配，建立好语法树，如果出现token类型和文法的类型不匹配的情况就进行语法的错误提示

三.实验过程

```
enum class NodeType {
    PROGRAM, STMT_SEQUENCE, IF_STMT, REPEAT_STMT, ASSIGN_STMT,
    READ_STMT, WRITE_STMT, WHILE_STMT,
    GT_EXP, GE_EXP, LT_EXP, LE_EXP, EQ_EXP, // > >= < <= =
    OR_EXP, AND_EXP, NOT_EXP,
    PLUS_EXP, SUB_EXP, MUL_EXP, DIV_EXP,
    FACTOR,
    NONE
};
```

根据上面定义的EBNF的文法来进行语法树节点的类型的定义。

```
enum class VarType {
    VT_VOID, VT_INT, VT_BOOL, VT_STRING
};
```

定义变量的type。

```
void Parser::parse() {
    //构建语法树
    root = program();
    // 检查节点的类型类型是否正确
    checkType(root);

    //check statement
    checkStatementType(root);

    //如果有错throw
    if (log.hasError()) {
        std::stringstream msg;
        msg << "You have " << log.getErrorCount() << " errors.";
        throw msg.str();
    }
}
```

文法搭建语法树的过程。首先通过program来建好整颗语法树，之后先检查树的节点的类型，再检查statement的类型，如果有错误就抛出错误，不进行语法树的输出。其中program就对应上面文法的第一条的左边的非终结符。

```
TreeNode* Parser::program() {
    //进入的时候拿到一个token
    token = Stream.nextToken();

    //变量声明
    declarations();

    //进行语法树的构建
    TreeNode* body = stmt_sequence();
    if (token.type != TokenType::NO_MORE_TOKEN)
        log.parseError("无效的符号，之后的输入将被忽略", token.line, token.offset);

    //返回整颗语法树
    return body;
}
```

根据第一条文法进行匹配，program -> declarations stmt-sequence 。

```
//变量声明
void Parser::declarations() {
    //临时的type
    VarType type = VarType::VT_VOID;
    while (match(TokenType::KEY_INT) || match(TokenType::KEY_BOOL) ||
match(TokenType::KEY_STRING)) {
        switch (last_token.type) {
            case TokenType::KEY_INT:
                type = VarType::VT_INT;
                break;
            case TokenType::KEY_BOOL:
                type = VarType::VT_BOOL;
                break;
            case TokenType::KEY_STRING:
                type = VarType::VT_STRING;
                break;
            default:
                log.parseError("the token can not be parsed to a type: " +
last_token.token, last_token.line, last_token.offset);
                break;
        }
        do {
            // 期望获取一个标识符  int A 判断是不是A这种id
            if (!match(TokenType::ID, true))
                break;

            // 插入符号表
            //判重 是否有重复声明的变量

            if(table.count(last_token.token)){
                stringstream msg;
```

```
                msg << "the variable " << last_token.token << " has already
declared in line "<<table[last_token.token].lines[0];
                log.parseError(msg.str(), last_token.line, last_token.offset);
            }else{
                Symbol symbol;
                symbol.type = type;
                symbol.lines.push_back(last_token.line);
                table[last_token.token] = symbol;
            }

        } while(match(TokenType::OP_COMMA));
        match(TokenType::OP_SEMICOLON); // 分号可有可无  当前变量声明结束
    }
}
```

变量声明的过程，变量声明首先会有三种int，bool，string的token出现，如果token没有匹配到任意一种类型，就会进行之后stmt_sequence的匹配。如果匹配到相应的类型，就进行持续扫描变量的token，因为可能出现int A,B,C；这样的变量声明。同时这里需要进行变量符号的记录，因为不能够出现相同的变量名。这里使用unordered_map进行判重。

```
// stmt-sequence -> statement {; stmt-sequence }
TreeNode* Parser::stmt_sequence() {
    TreeNode* node = TreeNode::createNode(NodeType::STMT_SEQUENCE);
    node->children[0] = statement();  //左节点
    if (node->children[0] == nullptr) {
        delete node;
        return nullptr;
    }
    match(TokenType::OP_SEMICOLON); // 分号可有可无
    node->children[1] = stmt_sequence();
    if (node->children[1] == nullptr) {
        TreeNode* tmp = node->children[0];
        node->children[0] = nullptr;
        delete node;
        node = tmp;
    }
    return node;
}
```

根据文法进行递归下降的匹配，同时建立语法树。之后的递归下降的文法类似于此。

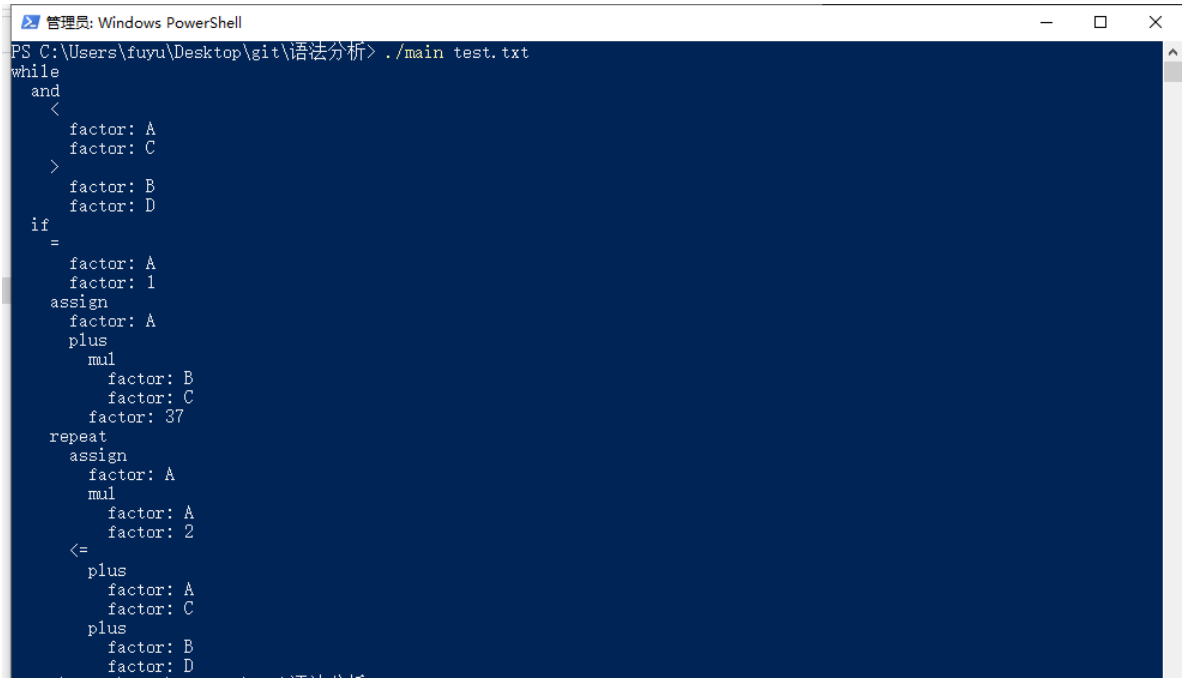
最后遍历的时候利用先序遍历的方式，从根节点进行遍历。

四.实验结果

tiny语言程序

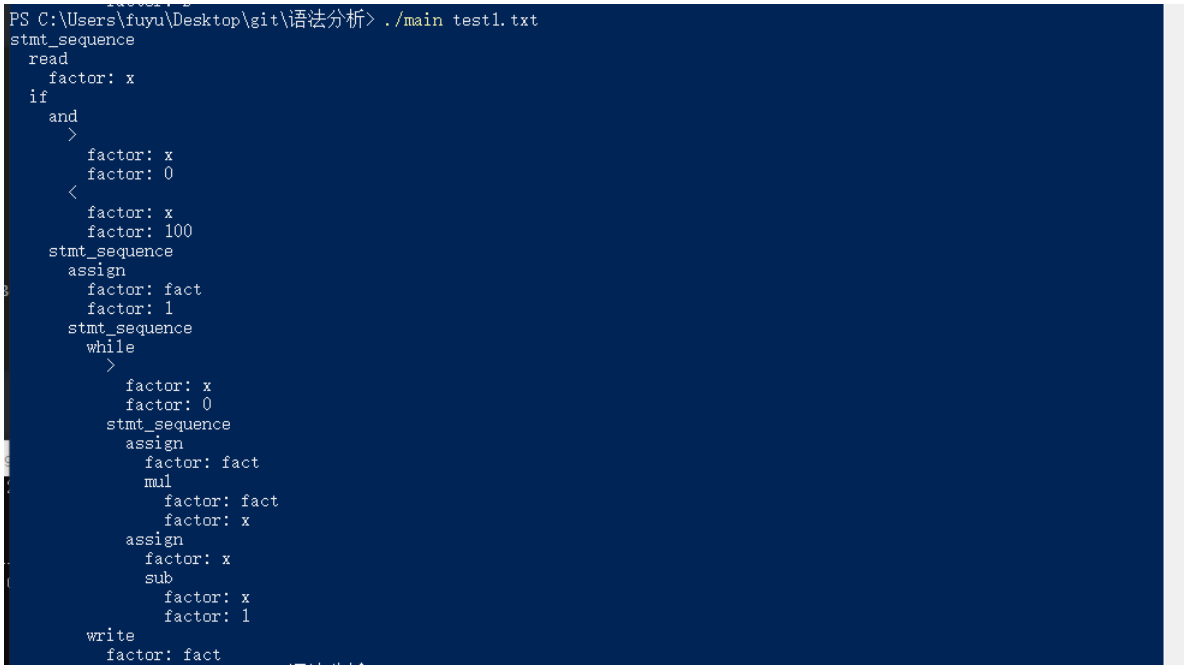
```
int A,B,C,D;
while A<C and B>D do
    if A=1  then
        A:= B*C+37
    else
        repeat
            A:=A*2
        until (A+C)<=(B+D)
    end
end
```

得到的语法树



test2

```
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
```



进行一下重复的定义

```
int A,B,C,D;
int A;
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
```

得到了错误的信息。