词法分析和语法分析实验报告

王若琪　18340166

1. 实验目的

扩充已有的样例语言TINY，为扩展TINY语言TINY + 构造词法分析和语法分析程序，从而掌握词法分析和语法分析程序的构造方法。

2. 实验内容

了解样例语言TINY及TINY编译器的实现，了解扩展TINY语言TINY + ，用EBNF描述TINY + 的语法，用C语言扩展TINY的词法分析和语法分析程序，构造TINY + 的语法分析器。

3. 实验要求

- 将TINY + 源程序翻译成对应的TOKEN序列，并能检查一定的词法错误。
- 将TOKEN序列转换成语法分析树，并能检查一定的语法错误。

4. TINY+ 扩充内容

- 增加运算符：% (求余数)
- 增加比较运算符：<, >
- 增加关键词：WHILE, DO, FOR, UPTO, DOWNTO
- 用 EBNF 描述 **TINY +** 的语法如下：

High-level program structures

```
Program -> MethodDecl MethodDecl*
Type -> INT | REAL |STRING
MethodDecl -> Type [MAIN] Id '(' FormalParams ')' Block
FormalParams -> [FormalParam ( ',' FormalParam )* ]
FormalParam -> Type Id
```

Statements

```
Block -> BEGIN Statement+ END

Statement -> Block
           | LocalVarDecl
           | AssignStmt
           | ReturnStmt
           | IfStmt
           | WriteStmt
           | ReadStmt
           | WhileStmt
           | DoWhileStmt
           | ForStmt

LocalVarDecl -> Type Id ';' | Type AssignStmt
```

```
AssignStmt  -> Id := Expression ';'
           |  Id := QString ';'
ReturnStmt -> RETURN Expression ';'
IfStmt    -> IF '(' BoolExpression ')' Statement
            | IF '(' BoolExpression ')' Statement ELSE Statement
WriteStmt -> WRITE '(' Expression ',' QString ')' ';'
ReadStmt  -> READ '(' Id ',' QString ')' ';'
WhileStmt -> WHILE '(' BoolExpression ')' Statement
  DoWhileStmt -> DO Statement WHILE '(' BoolExpression ')'
  ForStmt -> For AssignStmt UPTO Expression DO Statement
           | For AssignStmt DOWNTO Expression DO Statement

  QString is any sequence of characters except double quote itself,
enclosed in double quotes.
```

Expressions

```
Expression -> MultiplicativeExpr  (( '+' | '-' ) MultiplicativeExpr)*
  MultiplicativeExpr -> PrimaryExpr (( '*' | '/'| '%' ) PrimaryExpr)*
  PrimaryExpr -> Num  // Integer or Real numbers
               | Id
               | '(' Expression ')'
               | Id '(' ActualParams ')'
  BoolExpression -> Expression '==' Expression
                   |Expression '!=' Expression
                   |Expression '<' Expression
                   |Expression '>' Expression
  ActualParams -> [Expression ( ',' Expression)*]
```

5. 算法描述

5.1 词法分析

构造词法分析器，是从文件中依次读取字符，并按照 Tiny+ 设定的规则来将它们识别为 TOKEN。

先定义 TOKEN 类型：

```
//token type
typedef enum {
    IF, ELSE, WRITE, READ, RETURN, BEGIN, END, MAIN, INT, REAL, WHILE, DO,
FOR, UPTO, DOWNTO,
    SEMI, COMMA, LEFTPAREN, RIGHTPAREN,
    ADD, SUB, MUL, DIV, MOD, ASSIGN, EQUAL, UNEQUAL, GT, LT,
    ID, NUM, QSTR,
    ERROR, ENDFILE
}TokenType;
```

接下来定义 DFA 的状态：

```
// DFA status
typedef enum {
    ST_START,
    ST_ASSIGN,
    ST_EQUAL,
    ST_UNEQUAL,
    ST_COMMENT,
    ST_NUMBER,
    ST_REAL,
    ST_STRING,
    ST_ID,
    ST_FINISH
}dfaStatus;
```

设置 Key words：

```
// keywords
struct {
    char* str;
    TokenType token;
} keyWords[NUM_KEY] = { {"WHILE", WHILE}, {"DO",DO}, {"FOR", FOR}, {"UPTO",
UPTO}, {"DOWNTO", DOWNTO}, {"IF", IF}, {"ELSE", ELSE}, {"READ", READ},
{"WRITE", WRITE}, {"BEGIN", BEGIN}, {"END", END}, {"MAIN", MAIN}, {"RETURN",
RETURN}, {"INT", INT}, {"REAL", REAL} };
```

设定初始状态为 ST_START，表示开始状态，接下来依次读入字符，根据读入字符进行状态转移的判断。

例如，在此状态时读取下一个字符，如果判断该字符是字母，那么状态可转变为ST_ID 状态，如果下一个字母判断为数字或字母，那么状态不需要改变，如果判断为既不是字母也不是数字，那么说明这个 TOKEN 结束了，则状态转换为 ST_FINISH，TOKEN 判断为 ID。

再例如，如果开始时读到的第一个字符是数字，那么久跳转到 ST_NUMBER 状态，如果读入下一个字符仍然是数字，那么状态不需要改变，如果读入下一个字符是小数点，说明这可能是一个小数，所以跳转到状态 ST_REAL，如果下一个数字是字母，就不符合语法规定，于是就进入报错流程，对错误进行处理。

另外，对于 "!=" 和 "==" 此类的运算符，可以设置状态 UNEQUAL 和 EQUAL 等，进行判断操作。

此外，单个字符的运算符比如 "+"、"-"、"*"、"/"、"%" 等，就可以直接判断 TOKEN，不需要进行状态转换。

以判断整数、小数或者错误为例，状态转换方式实现如下：

```
        case ST_NUMBER:
            if (!isdigit(c)) {
                if (c == '.') {
                    curStatus = ST_REAL;
                }
                else if (isalpha(c)) {
                    if (!IF_EOF) Col--;
                    save = 0;
                    curStatus = ST_FINISH;
                    curToken = ERROR;
                }
                else {
                    if (!IF_EOF) Col--;
```

```
                    curStatus = ST_FINISH;
                    curToken = NUM;
                }
            }
            break;
```

报错编写：

```
void lexError()
{
    printf("\n[ERROR] Error in line %d, column %d!\n", Row, Col);
    getchar();
    exit(1);
}
```

5.2 语法分析

采用前看一个 token 的预测分析法进行自顶向下的语法分析。

设计语法树的结构：

- 语句之间通过链表相连，第一条之后的语句可以通过前一个语句的 sibling 连接到。
- Assign 语句有一个孩子，位于 child[0]。
- If 语句分为三个孩子，test 位于 child[0]，then 位于 child[1]，else 位于 child[2]。
- write、read 语句有一个孩子，位于 child[0]。
- while 有两个孩子，test 和 do 分别位于 child[0] 和 child[1]。
- write 和 read 有一个孩子，位于 child[0]。
- 运算符号有两个孩子，分别位于 child[0] 和 child[1]。
- Define 分别位于某节点和他的 sibling 中。

节点结构设计：

定义树节点 Node 结构体如下：

```
// parse tree
typedef struct
{
    struct Node* child[MAXCHILDREN];
    struct Node* sibling;
    NodeKind nodekind;
    union { MethodKind method; TypeKind type; StmtKind stmt; ExpKind exp; }
kind;
    union {
        TokenType token;
        float val;
        char* name;
    } attr;
}Node;
```

语法树的树节点有不同的类型，对树节点的类型进行定义：

```
typedef enum { MethodK, TypeK, ParamK, StmtK, ExpK } NodeKind;
typedef enum { MainK, NormalK } MethodKind;
typedef enum { FormalK, ActualK, NoneK } ParamKind;
typedef enum { ReturnTypeK, IntTypeK, RealTypeK } TypeKind;
typedef enum { WhileK, DoWhileK, ForK, IfK, ReturnK, AssignK, ReadK, WriteK,
IntDeclareK, RealDeclareK } StmtKind;
typedef enum { OpK, ConstK, IdK, MethodCallK } ExpKind;
```

先以整个函数为单位，函数之间进行串行分析：

```
Node* MethodDecl_Sequence(void) {
    Node* t = MethodDecl();
    Node* p = t;
    while (token != ENDFILE) {
        Node* q;
        q = MethodDecl();
        if (q != NULL) {
            if (t == NULL) t = p = q;
            else {
                p->sibling = q;
                p = q;
            }
        }
    }
    return t;
}
```

在函数中，自顶向下分析：

```
static Node* MethodDecl(void) {
    Node* t = NULL;
    Node* p = ReturnType();

    if (token == MAIN) {
        t = NewMethodNode(MainK);
        match(MAIN);
    }
    else {
        t = NewMethodNode(NormalK);
    }
    t->child[0] = p;
    //func name
    if (t != NULL && token == ID) {
        t->attr.name = my_strcpy(stringSave);
    }
    match(ID);
    match(LEFTPAREN);
    if (token == RIGHTPAREN) {
        t->child[1] = NULL;
        match(RIGHTPAREN);
        t->child[2] = Block();
    }
    else {
        t->child[1] = FormalParams();
        match(RIGHTPAREN);
        t->child[2] = Block();
```

```
    }
    return t;
}
```

在语法分析时，采用自上而下的方式，可以前看一位，来决定哪一个产生式是合理的，从而继续分析。此部分用代码实现如下：

```
static Node* Statement(void) {
    Node* t = NULL;
    switch (token) {
    case WHILE: t = WhileStmt(); break;
    case DO: t = DoWhileStmt(); break;
    case FOR: t = ForStmt(); break;
    case BEGIN: t = Block(); break;
    case INT: t = IntLocalVarDeclStmt(); break;
    case REAL: t = RealLocalVarDeclStmt(); break;
    case ID: t = AssignStmt(); break;
    case RETURN: t = ReturnStmt(); break;
    case IF: t = IfStmt(); break;
    case WRITE: t = WriteStmt(); break;
    case READ: t = ReadStmt(); break;
    default:
        parError("Unexpected token");
        break;
    }
    return t;
}
```

报错设计：

在语法分析中，设计了报错的功能，可以显示出错的具体行列位置以及错误种类。

```
void parError(char* str)
{
    printf("\n[ERROR] Parse error in line %d, column %d: %s!\n", Row, Col, str);
    getchar();
    exit(1);
}
```

6. 测试结果

6.1 对参考文件中的代码进行测试，得到输出结果。

词法分析结果：

```
[KEYWORD, INT]
[ID, f2]
[SEP, (]
[KEYWORD, INT]
[ID, x]
[SEP, ,]
[KEYWORD, INT]
[ID, y]
```

```
[SEP, )]
[KEYWORD, BEGIN]
[KEYWORD, INT]
[ID, z]
[SEP, ;]
[ID, z]
[OP, :=]
[ID, x]
[OP, *]
[ID, x]
[OP, -]
[ID, y]
[OP, *]
[ID, y]
[SEP, ;]
[KEYWORD, RETURN]
[ID, z]
[SEP, ;]
[KEYWORD, END]
[KEYWORD, INT]
[KEYWORD, MAIN]
[ID, f1]
[SEP, (]
[SEP, )]
[KEYWORD, BEGIN]
[KEYWORD, INT]
[ID, x]
[SEP, ;]
[KEYWORD, READ]
[SEP, (]
[ID, x]
[SEP, ,]
[QString, A41.input]
[SEP, )]
[SEP, ;]
[KEYWORD, INT]
[ID, y]
[SEP, ;]
[KEYWORD, READ]
[SEP, (]
[ID, y]
[SEP, ,]
[QString, A42.input]
[SEP, )]
[SEP, ;]
[KEYWORD, INT]
[ID, z]
[SEP, ;]
[ID, z]
[OP, :=]
[ID, f2]
[SEP, (]
[ID, x]
[SEP, ,]
[ID, y]
[SEP, )]
[OP, +]
[ID, f2]
```

```
[SEP, (]
[ID, y]
[SEP, ,]
[ID, x]
[SEP, )]
[SEP, ;]
[KEYWORD, WRITE]
[SEP, (]
[ID, z]
[SEP, ,]
[QString, A4.output]
[SEP, )]
[SEP, ;]
[KEYWORD, END]
```

语法分析树：

```
Parse Tree:
    [Method: f2]
        [ReturnType: INT]
        [FormalParam:]
            [INT: x]
            [INT: y]
        [Decl: INT z]
        [Assign: z]
            [OP, -]
                [OP, *]
                    [Id: x]
                    [Id: x]
                [OP, *]
                    [Id: y]
                    [Id: y]
        [Return]
            [Id: z]
    [Main Method: f1]
        [ReturnType: INT]
        [Decl: INT x]
        [Read "A41.input"]
            [Id: x]
        [Decl: INT y]
        [Read "A42.input"]
            [Id: y]
        [Decl: INT z]
        [Assign: z]
            [OP, +]
                [function Call: f2]
                    [ActualParam:]
                        [Id: x]
                        [Id: y]
                [function Call: f2]
                    [ActualParam:]
                        [Id: y]
                        [Id: x]
        [Write "A4.output"]
            [Id: z]
```

6.2 加入 Tiny+ 扩充的功能，进行测试，得到输出结果。

测试代码如下：

```
/** this is a comment line in the sample program **/
 INT f2(INT x, INT y )
 BEGIN
    INT z;
    z := 64;
    WHILE (z > 0)
        z := z % 2;
    RETURN z;
 END
 INT MAIN f1()
```

```
BEGIN
   INT x;
   READ(x, "A41.input");
   INT y;
   READ(y, "A42.input");
   INT z;
   z := f2(x,y) + f2(y,x);
   FOR x := 5; DOWNTO 0 DO y := x;
   WRITE (z, "A4.output");
END
```

词法分析结果：

```
[KEYWORD, INT]
[ID, f2]
[SEP, (]
[KEYWORD, INT]
[ID, x]
[SEP, ,]
[KEYWORD, INT]
[ID, y]
[SEP, )]
[KEYWORD, BEGIN]
[KEYWORD, INT]
[ID, z]
[SEP, ;]
[ID, z]
[OP, :=]
[NUM, 64;]
[SEP, ;]
[KEYWORD, WHILE]
[SEP, (]
[ID, z]
[OP, >]
[NUM, 0)]
[SEP, )]
[ID, z]
[OP, :=]
[ID, z]
[OP, ]
[NUM, 2;]
[SEP, ;]
[KEYWORD, RETURN]
[ID, z]
[SEP, ;]
[KEYWORD, END]
[KEYWORD, INT]
[KEYWORD, MAIN]
[ID, f1]
[SEP, (]
[SEP, )]
[KEYWORD, BEGIN]
[KEYWORD, INT]
[ID, x]
[SEP, ;]
[KEYWORD, READ]
[SEP, (]
```

```
[ID, x]
[SEP, ,]
[QString, A41.input]
[SEP, )]
[SEP, ;]
[KEYWORD, INT]
[ID, y]
[SEP, ;]
[KEYWORD, READ]
[SEP, (]
[ID, y]
[SEP, ,]
[QString, A42.input]
[SEP, )]
[SEP, ;]
[KEYWORD, INT]
[ID, z]
[SEP, ;]
[ID, z]
[OP, :=]
[ID, f2]
[SEP, (]
[ID, x]
[SEP, ,]
[ID, y]
[SEP, )]
[OP, +]
[ID, f2]
[SEP, (]
[ID, y]
[SEP, ,]
[ID, x]
[SEP, )]
[SEP, ;]
[KEYWORD, FOR]
[ID, x]
[OP, :=]
[NUM, 5;]
[SEP, ;]
[KEYWORD, DOWNTO]
[NUM, 0 ]
[KEYWORD, DO]
[ID, y]
[OP, :=]
[ID, x]
[SEP, ;]
[KEYWORD, WRITE]
[SEP, (]
[ID, z]
[SEP, ,]
[QString, A4.output]
[SEP, )]
[SEP, ;]
[KEYWORD, END]
```

语法分析结果：

```
Parse Tree:
    [Method: f2]
        [ReturnType: INT]
        [FormalParam:]
            [INT: x]
            [INT: y]
        [Decl: INT z]
        [Assign: z]
            [Const: 64.000000]
        [While]
            [OP, >]
                [Id: z]
                [Const: 0.000000]
            [Assign: z]
                [OP, ]
                    [Id: z]
                    [Const: 2.000000]
        [Return]
            [Id: z]
    [Main Method: f1]
        [ReturnType: INT]
        [Decl: INT x]
        [Read "A41.input"]
            [Id: x]
        [Decl: INT y]
        [Read "A42.input"]
            [Id: y]
        [Decl: INT z]
        [Assign: z]
            [OP, +]
                [function Call: f2]
                    [ActualParam:]
                        [Id: x]
                        [Id: y]
                [function Call: f2]
                    [ActualParam:]
                        [Id: y]
                        [Id: x]
        [For]
            [Assign: x]
                [Const: 5.000000]
            [Const: 0.000000]
            [Assign: y]
                [Id: x]
        [Write "A4.output"]
            [Id: z]
```

6.3 报错测试

将代码中的 4 行 9 列中的变量名改成数字开头的形式，在词法分析时会报错：

```
[ERROR] Error in line 4, column 9!
```

将代码的 13 行 8 列中句子结尾后再写一个不相关符号，在语法分析时会报错：

```
[ERROR] Parse error in line 13, column 8: Unexpected token!
```