编译器课程设计

源码文档

目录

2014-7-6

应振强
电计 1101
201181086

源码说明

- token_keywords.h 样本 S 语言的关键字
- token.h 样本 S 语言的单词
- lex 词法分析器
- code 代码生成部分函数
- gen 单遍编译核心部分，调用词法分析器获取下一个单词，进行语法分析，调用代码生成部分函数生成四元式代码

token_keywords.h

```
/** ============================================================================
@author YingZhenqiang  yingzhenqiang@gmail.com
@brief 样本S语言关键字
----------------------------------------------------------------------------
采用单遍编译，为了提高效率，传递的是枚举值，而在需要调试输出时打印对应
的字符串
============================================================================ */

#ifndef TOKEN
#error "You must define TOKEN macro before include this file"
#endif

TOKEN(TK_INT,          "int")
TOKEN(TK_IF,           "if")
TOKEN(TK_THEN,         "then")
TOKEN(TK_ELSE,         "else")
TOKEN(TK_WHILE,        "while")
TOKEN(TK_DO,           "do")
TOKEN(TK_WRITE,        "write")
TOKEN(TK_READ,         "read")
```

token.h

```
/** ============================================================================
@author YingZhenqiang  yingzhenqiang@gmail.com
----------------------------------------------------------------------------
注意这不是头文件，而是数据配置文件，可以重复包含
务必在最前面包含#include "token_keywords.h"否则keywords[]不能正确匹配
关系运算符对语法分析无区别，改为一类
// TOKEN(TK_GREAT,        ">")
// TOKEN(TK_LESS,         "<")
// TOKEN(TK_GREAT_EQ,      ">=")
// TOKEN(TK_LESS_EQ,       "<=")
```

```
// TOKEN(TK_EQUAL,         "==")
// TOKEN(TK_UNEQUAL,       "!=")
================================================================================ */
//保留字
#include "token_keywords.h" // 注意务必在最前面
//标志符
TOKEN(TK_IDENT,         "IDENT")
//常数
TOKEN(TK_NUM,           "NUM")
//算术运算符
TOKEN(TK_ADD,           "+")
TOKEN(TK_SUB,           "-")
TOKEN(TK_MUL,           "*")
TOKEN(TK_DIV,           "/")
//关系运算符
TOKEN(TK_RELOP,         "RELOP")
//逻辑运算符
TOKEN(TK_LOG_OR,        "|")
TOKEN(TK_LOG_AND,       "&")
//其他运算符
TOKEN(TK_ASSIGN,        "=")
//分隔符
TOKEN(TK_LBRACE,        "{")
TOKEN(TK_RBRACE,        "}")
TOKEN(TK_SEMICOLON,     ";")
TOKEN(TK_LPAREN,        "(")
TOKEN(TK_RPAREN,        ")")
//特殊符号
TOKEN(TK_UNDEF,         "UNDEF")
TOKEN(TK_EOF,           "EOF")
```

common.h

```
#ifndef COMMON_H_INCLUDED
#define COMMON_H_INCLUDED

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h> // getch

#include "debug.h"
```

```
#define REVISE
#define MAX_SYM_NUM 80

typedef enum{
    false = 0,
    true = 1,
}bool;

typedef enum{
// 取token.h文件中左侧的一栏，即tk枚举值
#define TOKEN(k, s) k,
#include "token.h"
#undef  TOKEN
}tk_e;

char* itos(int n);

#endif // COMMON_H_INCLUDED
```

lex.h

```
/** ===============================================================================
@author YingZhenqiang  yingzhenqiang@gmail.com
-------------------------------------------------------------------------------
多遍编译
Lex_run("src.c","src.o");
单遍编译
初始化选择分析的文件Lex_selectFile("src.c");
取下一个单词tk = Lex_getNextToken(); 如果文件结束则返回TK_EOF
输出到屏幕Lex_dispToken(tk);
输出写入文件Lex_writeToken(tk,fout);
================================================================================ */

#ifndef LEX_H_INCLUDED
#define LEX_H_INCLUDED

#include "common.h"

typedef struct {
  tk_e  type;
  char* value;
```

```
}Token;

extern const char* keywords[];
extern const char* tokenName[];
extern int curLine;

void    Lex_run(const char* inFilename,const char* outFilename);

void    Lex_selectFile(const char* filename);
Token*  Lex_getNextToken();
void    Lex_dispToken(Token* tk);
void    Lex_writeToken(Token* tk,FILE* fout);

#endif // LEX_H_INCLUDED
```

lex.c

```
#include "lex.h"

FILE* fin = NULL;
FILE* fout = NULL;
char* filename = NULL;

void Lex_selectFile(const char* f){
  if(fin!=NULL)fclose(fin);

  fin=fopen(f,"rt");

  if(fin==NULL)printf("error in fopen %s\n",f);
  ASSERT(fin!=NULL);

  filename = (char*)f;
  curLine = 1;
}

char currentChar = ' ';// 空格会继续读入 由于存储当前指向的位置，所以必须为全局量
char* pString = NULL;// 采用fscanf可以自动过滤空格，但是不能计量行号，故不采用

#define CURRENT_CHAR     (currentChar)
#define MOVE_NEXT_CHAR   if( EOF == (currentChar=fgetc(fin)) ){currentChar =
EOF;}
```

```
#define isSpace(c)        (c == '\t' || c == ' ' || c == '\n')
#define isDigit(c)        (c >= '0' && c <= '9')
#define isAlpha(c)        ((c >= 'a' && c <= 'z') || (c == '_') || (c >= 'A' && c
<= 'Z'))
#define isRelop(c)        (c == '!' || c == '<' || c == '>')

#define error()          return tk/*exit(1)*/

const char* keywords[]={
#define TOKEN(k, s) s,
#include "token_keywords.h"
#undef  TOKEN
};
#define KEYWORDS_NUM sizeof(keywords)/sizeof(keywords[0])

const char* tokenName[]={
#define TOKEN(k, s) s,
#include "token.h"
#undef  TOKEN
};

int curLine = 0;

#define MAX_STR_LEN      40 // 支持的最长的存储值的字符串
#define NEW_STRING       pString = (char*) malloc ( MAX_STR_LEN * sizeof(char))
#define APPEND_CHAR(c)    *pString = c; pString++;

// 设已经弹入一个字符，CURRENT_CHAR指向当前待分析的字符（由于可能需要向后看一个字符）
// 试探下一个字符，若是自己的一部分，则读入下一个字符再返回，否则直接返回
Token* Lex_getNextToken(){
  Token* tk = (Token*) malloc (sizeof(Token));
  tk -> type = TK_UNDEF;
  tk ->value = NULL;

  while(isSpace(CURRENT_CHAR)){
    if(CURRENT_CHAR == '\n')curLine++;
    MOVE_NEXT_CHAR;
  }
  if(CURRENT_CHAR == EOF ){ // 注意先跳过格式字符再看是否EOF，否则文件最后的空格会识别错误
    tk -> type = TK_EOF;
    // 删除：curLine=0; 原因：造成最后一个错误报告行为0
    return tk;
  }
  if(isAlpha(CURRENT_CHAR)){
```

```
int i = 0;

  tk -> value = NEW_STRING;
  while(isAlpha(CURRENT_CHAR)||isDigit(CURRENT_CHAR)){
    APPEND_CHAR(CURRENT_CHAR);
    MOVE_NEXT_CHAR;
  }
  APPEND_CHAR('\0');
  // 保留字和标志符的识别
  for(i=0;i< KEYWORDS_NUM ;i++){
    if(0==strcmp(tk -> value ,keywords[i])){
      tk -> type =  i; //(tk_e)
      free(tk -> value);// 如果是关键字，就释放存储值的空间
      return tk;
    }
  }
  tk -> type = TK_IDENT;
  return tk;
}
else if(isDigit(CURRENT_CHAR)){ // 无符号整数识别
  tk -> value = NEW_STRING;
  while(isDigit(CURRENT_CHAR)){
    APPEND_CHAR(CURRENT_CHAR);
    MOVE_NEXT_CHAR;
  }
  APPEND_CHAR('\0');
  tk -> type = TK_NUM;
  return tk;
}
else if(isRelop(CURRENT_CHAR)){
  tk -> value = NEW_STRING;
  APPEND_CHAR(CURRENT_CHAR);
  MOVE_NEXT_CHAR;
  if('=' == CURRENT_CHAR){
    if(CURRENT_CHAR == '!'){
      tk -> type = TK_UNDEF;
    }
    else {
      APPEND_CHAR(CURRENT_CHAR);
      MOVE_NEXT_CHAR;
      tk -> type = TK_RELOP;
    }
  }
  APPEND_CHAR('\0');
```

```
    return tk;
  }
  else if( '=' == CURRENT_CHAR ){
    tk -> value = NEW_STRING;
    APPEND_CHAR(CURRENT_CHAR);
    MOVE_NEXT_CHAR;
    if('=' == CURRENT_CHAR){
      APPEND_CHAR(CURRENT_CHAR);
      MOVE_NEXT_CHAR;
      tk -> type = TK_RELOP;
    }
    else{
      tk -> type = TK_ASSIGN;
    }
    APPEND_CHAR('\0');
    return tk;
  }
  else {
    switch(CURRENT_CHAR){
      case '+':tk -> type = TK_ADD;break;
      case '-':tk -> type = TK_SUB;break;
      case '*':tk -> type = TK_MUL;break;
      case '/':tk -> type = TK_DIV;break;
      case '{':tk -> type = TK_LBRACE;break;
      case '}':tk -> type = TK_RBRACE;break;
      case '(':tk -> type = TK_LPAREN;break;
      case ')':tk -> type = TK_RPAREN;break;
      case ';':tk -> type = TK_SEMICOLON;break;
      case '|':tk -> type = TK_LOG_OR;break;
      case '&':tk -> type = TK_LOG_AND;break;
      default:
        tk -> type = TK_UNDEF;
        tk -> value = NEW_STRING;
        APPEND_CHAR(CURRENT_CHAR);
        APPEND_CHAR('\0');
        break;
    }
    MOVE_NEXT_CHAR;
    return tk;
  }
}
#define OUT_TOKEN(tk)                                    \
do{                                                      \
    _OUT("%10s",tokenName[(unsigned int)(tk->type)]);       \
```

```
    _OUT(" ");                                          \
    switch(tk->type){                                    \
        case TK_IDENT:                                   \
        case TK_RELOP:                                   \
        case TK_NUM:                                     \
        case TK_UNDEF:                                   \
        _OUT("\"%s\"",tk->value);                          \
        break;                                          \
        default:                                         \
        _OUT("_");break;                                  \
    }                                                   \
    _OUT("\n");                                          \
}while(0)
// 用tab不整齐，改为空格间隔
void Lex_dispToken(Token* tk){
#define _OUT(FORMAT,...) printf(FORMAT,##__VA_ARGS__)
  OUT_TOKEN(tk);
#undef _OUT
}
void Lex_writeToken(Token* tk,FILE* fout){
#define _OUT(FORMAT,...) fprintf(fout,FORMAT,##__VA_ARGS__)
  OUT_TOKEN(tk);
#undef _OUT
}
void Lex_run(const char* inFilename,const char* outFilename){
Token* tk;

    Lex_selectFile(inFilename);
    fout = fopen( outFilename ,"w");
    ASSERT(fout!=NULL);
    do{
        tk = Lex_getNextToken();
        Lex_dispToken(tk);
        Lex_writeToken(tk,fout);
    }while(tk->type!=TK_EOF);
    fclose(fout);
}
```

code.h

```
#ifndef CODE_H_INCLUDED
```

```h
#define CODE_H_INCLUDED

#include "common.h"

typedef tk_e sym_e;
typedef struct{
    sym_e type;
    char* value;
}IdentSym;

typedef struct{
    char* value;
}ConstSym;

char*       newTemp();
char*       getLabel();
void        symTab_insert(sym_e type, char* value);
IdentSym*   symTab_lookup(char* value);
ConstSym*   constTab_lookInt(char* value);

#endif // CODE_H_INCLUDED
```

code.c

```c
#include "code.h"

static IdentSym symTab[MAX_SYM_NUM];
static ConstSym constTab[MAX_SYM_NUM];
static int pos_id = 0;
static int pos_const = 0;

void symTab_insert(sym_e type, char* value){
    symTab[pos_id].type = type;
    ASSERT(value!=0);
    symTab[pos_id].value = value;
    ASSERT(pos_id<MAX_SYM_NUM);
    pos_id ++;
}
IdentSym* symTab_lookup(char* value){
int i;
    for(i=0;i<pos_id;i++){
        if(0==strcmp(symTab[i].value,value)){
```

```
            return symTab+i;
        }
    }
    return NULL;
}
ConstSym* constTab_lookInt(char* value){
int i;
    for(i=0;i<pos_const;i++){
        if(0==strcmp(constTab[i].value,value)){
            return constTab+i;
        }
    }
    if(i==pos_const){
        constTab[pos_const].value = value;
        ASSERT(pos_const<MAX_SYM_NUM);
        pos_const ++;
    }
    return constTab+pos_const;
}
//函数用来产生临时变量, 如产生临时变量t1 :
static int tempNum=0;
char* newTemp(){
  //N=N+1;
  //"t"|| ITOS(N) ;
  tempNum ++ ;
  char* tempName = (char*) malloc(8*sizeof(char));
  *tempName='t';
  strcpy(tempName+1,itos(tempNum) );
  return tempName;
}

static int labelNum=0;
char* getLabel(){
  //N=N+1;
  //"t"|| ITOS(N) ;
  labelNum ++ ;
  char* labelName = (char*) malloc(8*sizeof(char));
  *labelName='t';
  strcpy(labelName+1,itos(labelNum) );
  return labelName;
}
char* itos(int n) {
    char* str= (char*)malloc(10*sizeof(char));
    int radix=10;
```

```
    int i = 0;
    int m = n;
    int f = 0;
    if (n == 0)  //如果是0，直接赋值{
        str[0] = '0';
        str[1] = '\0';
        return str;
    }
    else if (n < 0){
        str[0] = '-';
        n = -n;
        f = 1;
    }
    while (m){
        m /= radix;
        i++;
    }
    str[i + f] = '\0';
    i--;
    while (n){
        str[i + f] = n % radix;
        if (str[i + f] < 10){
            str[i + f] += '0';
        }
        else{
            str[i + f] += ('a' - 10);
        }
        n /= radix;
        i--;
    }
    return str;
}
```

gen.h

```
#ifndef GEN_H_INCLUDED
#define GEN_H_INCLUDED

#include "common.h"

void Gen_run(char* filename);
```

```
#endif // GEN_H_INCLUDED
```

gen.c

```
/** ================================================================================
@author YingZhenqiang  yingzhenqiang@gmail.com
--------------------------------------------------------------------------------
错误恢复的范围是向前向后看一个字符，所以能力有限。除非遇到可能发生混乱的
情况，程序持续分析直到文件结束
（需要标识符时找不到，如果在向后看一个字符找到了，就不会出错；
如果没有定义，可以恢复定义）
================================================================================
*/

#include "lex.h"
#include "code.h"

/** 调试信息输出和写入文件 ------------------------------------------------------
-------- */

#include <windows.h>
#define COLOR_LEX   FOREGROUND_INTENSITY|FOREGROUND_GREEN
#define COLOR_PARSE FOREGROUND_INTENSITY|FOREGROUND_RED|FOREGROUND_BLUE
#define COLOR_CODE  FOREGROUND_INTENSITY|FOREGROUND_BLUE

static int pos=0;
#define _MOVE_POS           do{int p=pos;while(p){printf(" ");p--;}}while(0)
#define _MOVE_NEXT_POS      do{_MOVE_POS;pos++;}while(0)
#define _MOVE_BACK_POS      do{pos--;_MOVE_POS;}while(0)

static FILE* lexFile = NULL;
static FILE* parseFile = NULL;
static FILE* codeFile = NULL;
#define Out_LexToken(tk)           \
do{Format_setColor(COLOR_LEX);Lex_dispToken(tk);\
Lex_writeToken(tk,lexFile);}while(0)
// _MOVE_POS;
#define Out_parseBeginMark()      \
do{Format_setColor(COLOR_PARSE);\
_MOVE_NEXT_POS;printf("<%s>{\n",__FUNCTION__);\
fprintf(parseFile,"<%s>{\n",__FUNCTION__);}while(0)
```

```
#define Out_parseEndMark()        \
do{Format_setColor(COLOR_PARSE);\
_MOVE_BACK_POS;printf("<%s>}\n",__FUNCTION__);\
fprintf(parseFile,"<%s>}\n",__FUNCTION__);}while(0)

static int nextStat = 0;
#define Out_CodeEmit(x1,x2,x3,x4)    \
do{Format_setColor(COLOR_CODE);printf("%04d
(%8s %8s %8s %8s)\n",nextStat,x1,x2,x3,x4);\
fprintf(codeFile,"%04d
(%8s %8s %8s %8s)\n",nextStat,x1,x2,x3,x4);nextStat++;}while(0)
#define Out_SetLabel(label)         \
do{Format_setColor(COLOR_CODE);printf("%-4s:\n",label);\
fprintf(codeFile,"%-4s:\n",label);}while(0)


/** 内部变量声明 ------------------------------------------------------------ */
static Token* g_lastToken=NULL;// 需要保存上一个token
static Token* g_thisToken=NULL;
static Token* g_nextToken=NULL;
#define g_sym   (g_thisToken->type)
#define g_value (g_thisToken->value)

/** 内部函数声明 ------------------------------------------------------------ */
static void getNextSym();
static void P();
static void D();
static void S();
static void L();
#ifndef REVISE
static void Lprime();
#endif

static char* B();
static char* Tprime();
static char* Fprime();

static char* E();
static char* T();
static char* F();
/** 错误处理 ---------------------------------------------------------------- */
static FILE* errFile = NULL;
// 非法字符直接在getSym时跳过，不再在这里处理
#define _Error_print(FORMAT,...)       \
```

```
do{Format_setColor(COLOR_ERROR);printf("[%04d] " FORMAT
"\n",curLine,##__VA_ARGS__);\
fprintf(errFile,"[%04d] " FORMAT "\n",curLine,##__VA_ARGS__);}while(0)
#define Error_parseFail()              \
do{_Error_print("Fail in parsing '%s'.Fatal error, analysis stoped."\
,__FUNCTION__);exit(-1);}while(0)
#define Error_notEof()                 _Error_print("Expect EOF.")

static int tokenBackNum = 0;
static void Error_expect(sym_e sym){
Token* lastToken = NULL;
// 保存现场环境
    lastToken  = g_lastToken;

    if(g_sym!=sym){
        getNextSym();
        if(g_sym == sym){// 多了一个字符
            _Error_print("I am quite sure you make a mistake, so I repair it.");
        }
        else { // 丢了 mising
            if( sym == TK_IDENT ){
                _Error_print("Expect '%s'. after '%s'.",
                             tokenName[(int)sym],tokenName[g_lastToken->type]);
                _Error_print("Fatal error, analysis stoped.");
              // 要求是标识符时，找不到标识符为致命错误
                exit(0); //exit(-1);
            }
            // 还要确保getNextToken弹回。backNextToken
            tokenBackNum = 2;
            // 恢复现场环境
            g_nextToken = g_thisToken;
            g_thisToken = g_lastToken;
            g_lastToken = lastToken;
            _Error_print("Missing '%s' have been fixed.",tokenName[(int)sym]);
        }
    }
    // 并不终止程序，继续执行
}
static void Error_ifIdentNotDefined(char* id){
    // 进行定义工作
    if(symTab_lookup(id) == NULL){
        _Error_print("not defined IDENT symbol: %s",id);
        if(id!=NULL){
            symTab_insert(TK_INT,id);
```

```
        }
        else {
            _Error_print("Fatal error, analysis stoped."); // 需要是标识符才行
            exit(-1);
        }
    }
}
/** 内部函数定义 ----------------------------------------------------------------
*/
static void getNextSym(){
// 未定义的token不会影响程序
// 进入下一个token 打印上一个token
    ASSERT(g_lastToken!=NULL);   // 初始化中读入了一个token

    if(tokenBackNum == 2 ){
        tokenBackNum --;
    }
    else if(tokenBackNum == 1){
        tokenBackNum --;
        g_lastToken = g_thisToken;
        g_thisToken = g_nextToken;
    }
    else{
        g_lastToken = g_thisToken;
        g_thisToken = Lex_getNextToken();
        /** 非法字符处理 ---------------------------------- */
        while(g_thisToken->type == TK_UNDEF){
            _Error_print("I found a UNDEF char: '%s'",g_thisToken->value);
            Out_LexToken(g_thisToken);
            g_thisToken = Lex_getNextToken();
        }
        /** ------------------------------------------------ */
        Out_LexToken(g_lastToken);
    }
}

void Gen_run(char* filename){
    Lex_selectFile(filename);

    // 先读入一个字符,给g_sym, g_value,tk赋予初值
    g_thisToken = g_lastToken = Lex_getNextToken();

    lexFile = fopen("lex.txt","w");
    parseFile = fopen("parse.txt","w");
```

```
    codeFile = fopen("code.txt","w");
    errFile = fopen("err.txt","w");

    P();

    while(g_sym!=TK_EOF){
        Error_notEof();
        P();
    }
}

// P →{DS}
void P(){
    Out_parseBeginMark();
    Error_expect(TK_LBRACE);
    getNextSym();
    D();
    S();
    Error_expect(TK_RBRACE);
    getNextSym();
    Out_parseEndMark();
}

//D →int ID ;{int ID;}  D->int ID ; D' D'->D|null
// 声明语句的翻译
void D(){
    Out_parseBeginMark();
    do{
        Error_expect( TK_INT );
        getNextSym();
            Error_expect( TK_IDENT );
            symTab_insert(TK_INT,g_value);
            getNextSym();
                Error_expect( TK_SEMICOLON );
                getNextSym();
    }while(  g_sym == TK_INT/*in First(D)*/ );
    Out_parseEndMark();
}

/**
S→if (B) then S [else S ] | while (B) do S | { L } | ID=E
      | write (E); | read ID;
*/
void S(){
```

```
char* label1 =NULL;
char* label2 =NULL;
char* place =NULL;
    Out_parseBeginMark();
    if( g_sym == TK_IF ){
        getNextSym();
        Error_expect( TK_LPAREN );
            getNextSym();
            place = B();
            Error_expect( TK_RPAREN );
                getNextSym();

                label1=getLabel();
                Out_CodeEmit("jz",place,"_",label1);

                Error_expect( TK_THEN );
                    getNextSym();
                    S();

                    label2=getLabel();
                    Out_CodeEmit("jp","_","_",label2);
                    Out_SetLabel(label1);

                    if( g_sym == TK_ELSE ){
                        getNextSym();
                        S();
                        Out_SetLabel(label2);
                    }
                    else { /*null*/ }
    }
    else if( g_sym == TK_WHILE ){
        label1 = getLabel();
        Out_SetLabel(label1);

        getNextSym();
        Error_expect( TK_LPAREN );
            getNextSym();
            place = B();
            Error_expect( TK_RPAREN );

                label2 = getLabel();
                Out_CodeEmit("jz",place,"_",label2);

                getNextSym();
```

```
            Error_expect( TK_DO );
                getNextSym();
                S();

                Out_CodeEmit("jp","_","_",label1);
                Out_SetLabel(label2);
    }
    else if( g_sym == TK_LBRACE ){
        getNextSym();
        L();
        Error_expect( TK_RBRACE );
            getNextSym();
    }
    /* ID=E ------------------------------- */
    else if( g_sym == TK_IDENT ){
    char* n;
        n = g_value;
        getNextSym();
        Error_expect( TK_ASSIGN );
        char* x = NULL;
            getNextSym();
            x = E();
#ifdef REVISE
            Error_expect(TK_SEMICOLON );
                getNextSym();
#endif // REVISE
            Error_ifIdentNotDefined(n);
            Out_CodeEmit(":=",x,"_",n);
    }
    /* write <算术表达式>; ----------------- */
    else if( g_sym == TK_WRITE ){
    char* place=NULL;
        getNextSym();
        Error_expect( TK_LPAREN );
            getNextSym();
            place = E();
            Out_CodeEmit("out",place,"_","std::out");
            Error_expect( TK_RPAREN );
                getNextSym();
                Error_expect( TK_SEMICOLON );
                    getNextSym();

    }
    /* read ID; --------------------------- */
```

```
    else if( g_sym == TK_READ ){
        getNextSym();
        Error_expect(TK_IDENT);
            Error_ifIdentNotDefined(g_value);
            Out_CodeEmit("in","std::in","_",g_value);
            getNextSym();
            Error_expect( TK_SEMICOLON );
                getNextSym();
    }
    else { Error_parseFail(); }
    Out_parseEndMark();
}
// L→SL'

void L(){
    Out_parseBeginMark();

#ifdef REVISE
    while( g_sym == TK_IF || g_sym == TK_WHILE ||
    g_sym == TK_IDENT || g_sym == TK_WRITE || g_sym == TK_READ ){
        S();
    }
#else
    S();
    Lprime();
#endif // REVISE
    Out_parseEndMark();
}

#ifndef REVISE
// L' →; L | null
void Lprime(){
    Out_parseBeginMark();
    if( g_sym == TK_SEMICOLON ){
        getNextSym();
        L();
    }
    else { /*null*/ }
    Out_parseEndMark();
}
#endif
// B→T' { |T'} 可出现多次的话可采用while
// 这种方式比较麻烦 B→ T' || B | T'
```

```
// 布尔表达式的翻译

char* B(){
char* x =NULL;
char* q =NULL;
char* r =NULL;
    Out_parseBeginMark();
    x = q = Tprime();
    while( g_sym == TK_LOG_OR ){
        getNextSym();
        r = Tprime();
        x = newTemp();
        Out_CodeEmit("or",q,r,x);
    }
    Out_parseEndMark();
    return x;
}
// T′ →F′ { & F′ }
char* Tprime(){
char* x =NULL;
char* q =NULL;
char* r =NULL;
    Out_parseBeginMark();
    x = q = Fprime();
     while( g_sym == TK_LOG_AND ){
        getNextSym();
        r = Fprime();
        x = newTemp();
        Out_CodeEmit("and",q,r,x);
    }
    Out_parseEndMark();
    return x;
}
// F′ →ID relop ID | ID
char* Fprime(){
char* x =NULL;
char* n =NULL;
char* q =NULL;
    Out_parseBeginMark();
    Error_expect( TK_IDENT );
        n = g_value;
        Error_ifIdentNotDefined(g_value);
        x = newTemp();
        Out_CodeEmit("j!=",n,"0",itos(nextStat+3));
```

```
        Out_CodeEmit(":=","0","_",x);
        Out_CodeEmit("jp","_","_",itos(nextStat+2));
        Out_CodeEmit(":=","1","_",x);

        getNextSym();
        if ( g_sym == TK_RELOP ){
        char* m = g_value;
            getNextSym();
            Error_expect( TK_IDENT );
                q = g_value;
                Error_ifIdentNotDefined(g_value);
                x = newTemp();
                char s[4] = "";
                s[0]='j';
                strcpy(s+1,m);
                Out_CodeEmit(s,n,q,itos(nextStat+3));
                Out_CodeEmit(":=","0","_",x);
                Out_CodeEmit("jp","_","_",itos(nextStat+2));
                Out_CodeEmit(":=","1","_",x);

                getNextSym();
        }
        else { /*null*/ }
    Out_parseEndMark();
    return x;
}
// E→T{+T| -T}
char* E(){
char* q =NULL;
char* r =NULL;
char* x =NULL;
    Out_parseBeginMark();
     x = q = T(); // 这里x =也是处理单个项的情况
     while( 1 ){
        if( g_sym == TK_ADD ){
            getNextSym();
            r = T();
            x = newTemp();
            Out_CodeEmit("+",q,r,x);
        }
        else if( g_sym == TK_SUB ){
            getNextSym();
            r = T();
            x = newTemp();
```

```
            Out_CodeEmit("-",q,r,x);
        }
        else {/*null*/ break; }
    }
    Out_parseEndMark();
    return x;
}
//T→F{ * F | /F }
char* T(){
char* q =NULL;
char* r =NULL;
char* x =NULL;
    Out_parseBeginMark();
     x = q = F(); // 注意这里 x =  需要给x赋值，防止单独因子的情况 T->F
     while( 1 ){
        if( g_sym == TK_MUL ){
            getNextSym();
            r = F();
            x = newTemp();
            Out_CodeEmit("*",q,r,x);
        }
        else if( g_sym == TK_DIV ){
            getNextSym();
            r = F();
            x = newTemp();
            Out_CodeEmit("/",q,r,x);
        }
        else {/*null*/break;}
    }
    Out_parseEndMark();
    return x;
}
// F→ (E) | NUM | ID
// 在语法分析的基础上，每个识别过程添加一个返回值，用来传递属性
char* F(){
char* x =NULL;
    Out_parseBeginMark();
    if( g_sym == TK_LPAREN ){
        getNextSym();
        char* q = E();
        Error_expect( TK_RPAREN );
            getNextSym();
            x = q;
    }
```

```
    else if( g_sym == TK_NUM ){
    char* lexval = g_value;
    //ConstSym* p =
        constTab_lookInt(lexval);
        x = lexval;
        getNextSym();
    }
    else if( g_sym == TK_IDENT ){
    char* n = g_value;
        Error_ifIdentNotDefined(g_value);
        x = n ;
        getNextSym();
    }
    else { Error_parseFail(); }
    Out_parseEndMark();
    return x;
}
```

main.c

```
/** ============================================================================
@author YingZhenqiang  yingzhenqiang@gmail.com
----------------------------------------------------------------------------
如果是点击exe触发的，则需要在打印完输出后等待用户，否则黑窗口一闪而过不友好，
如果出现错误，则也需要等待，断言错误不会发生，不用再断言里添加等待。
但是在上位机调用时出现问题，采用传入串的方式可行，但采用传入参数的方式就
相当于用户输入了。

TODO
完成多遍编译部分
MAYBE SOMEDAY
命令行配置功能——可以设置输出文件、输出内容、编译选项等
============================================================================ */

#include "lex.h"
#include "parse.h"
#include "gen.h"

char filename[40];
bool isSinglePass;
bool isClickExe = false ;
```

```
static void ParseCmdLine(int argc, char *argv[]){
    ASSERT(argc==1);
    strcpy(filename,argv[0]);
}
int main(int argc, char *argv[]) {
 isClickExe = false;
    if (argc <= 1){
        //ShowHelp();
        isClickExe = true;
        printf("Input filename:");
        scanf("%s",filename);
    }
    else ParseCmdLine(--argc, ++argv);

    isSinglePass = true;

    if(isSinglePass){
        Gen_run(filename);
    }
    else{
        // 多遍分析由两个相互独立的模块组成，输入文件处理后输出文件 由于没有编写配置功能，所以未编写多遍部分
        Lex_run(filename,"lex.txt");
        //Parse_run("lex.txt","code.txt");
    }
    if(isClickExe){
        printf("输出完成，按任意键退出");
        getch();
    }
    return 0;
}
```