编译原理课程设计

小型编译程序的构造

学 院（系）：电子信息与电气工程学部

学 生 姓 名：应振强

学　　号：201181086

班　　级：电计 1101

同 组 人 ：一人完成

大连理工大学

Dalian University of Technology

题目

样本 S 语言编译器的构造

实现的功能

严格按照样本 S 语言的定义实现了：

- 词法分析
- 语法分析
- 语义分析及中间代码生成
- 错误恢复

编译过程各阶段的关键问题及其解决方法

词法分析的关键问题

1.单词定义

如何定义单词类别及其内部表示

单词类别：保留字、标识符、分界符、运算符、常量（样本 S 语言只有整型常量）

类别编码　　单词类别　　单词值

单词结构体定义如下：

```
typedef struct {
  tk_e  type; // tk_e 为单词类别的枚举类型
  char* value;
}Token;
```

单词类别编码通过枚举实现，单词类别字符串存储在 tokenString 数组中，而没有存放在单词结构体中，避免冗余存储。而单词值的存储通过一个指向字符串的指针实现，如果单词的值没有意义，则指针为空。

2.单词识别

分析并画出状态转化图，根据状态转化图画出流程图，由流程图手工构造词法分析程序。其中保留字和标识符的识别，通过构造保留字表，每识别出一个标识符就转到查询保留字表，若匹配，则识别为保留字，否则识别为标识符。

语法分析的关键问题

语法分析方案：由于是手工编写代码，所以可采用自顶向下递归下降分析方法来分析，同时由于采用了该方法，需要对文法进行改造。

算法

（例如，程序流程图、状态转换图、数据结构设计、文法定义、文法改造、翻译方案设计）

词法分析

状态转换图和 ppt 中一致，故略去。获取下一个单词的步骤如下：

1. 读入下一个字符，
2. 判断是否为空白字符，如果是换行符则行标记加一，如匹配则返回；
3. 判断是否为文件结束符，如是则返回结束符；
4. 判断是否文件结束符，如匹配则返回；
5. 判断是否为字母，查找保留字表，判断是保留字还是标识符，如匹配则返回；
6. 判断是否为数字，解析数字，如匹配则返回；
7. 判断是否为比较符号，如匹配则返回；
8. 识别其他单符号 token，如匹配则返回；
9. 返回错误符号

语法分析

约定：进入某函数前，待匹配符已经读进 sym 中， 函数结束时， 下一个待匹配符也已经读进 sym 中。

定义函数：针对每个非终极符编写函数，根据规则候选式的结构， 按从左到右的顺序定义函数体。对于规则中的终极符，sym 匹配上了规则中的一个终极符，就把下一个待匹配符读进 sym 中；若匹配不成功， 表明有语法错误，语法检查停止。对于规则右部的非终极符，就调用该非终极符所对应的函数。

语法分析中，对于{XXX}结构（XXX 可重复多次），采用 while 结构比较简单，不要局限于 if-else 使代码复杂化了

翻译方案

把语义检查和翻译成四元式代码的语义动作插入到语法分析程序中。↑代表文法符号或语义过程具有综合属性，↓代表继承属性。 对于语义过程而言，继承属性是要传入过程的参数，综合属性是过程的返回值。 所有的终极符具有综合属性， 来自词法分析的结果。

总结

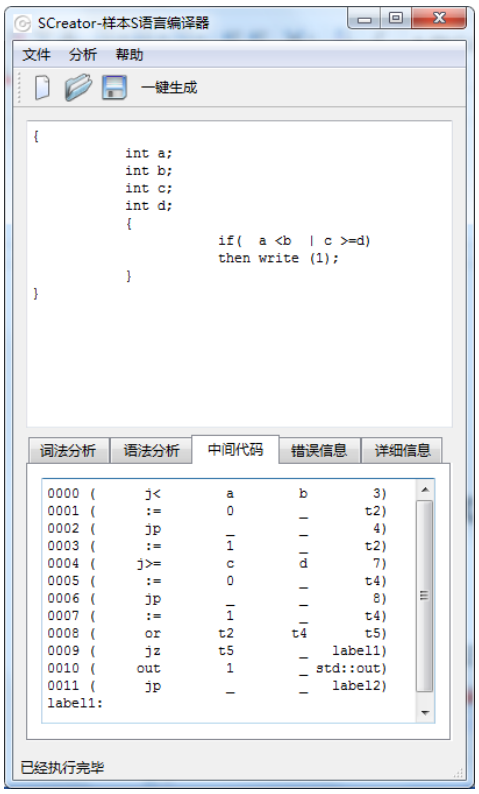
（谈实践的体会等）

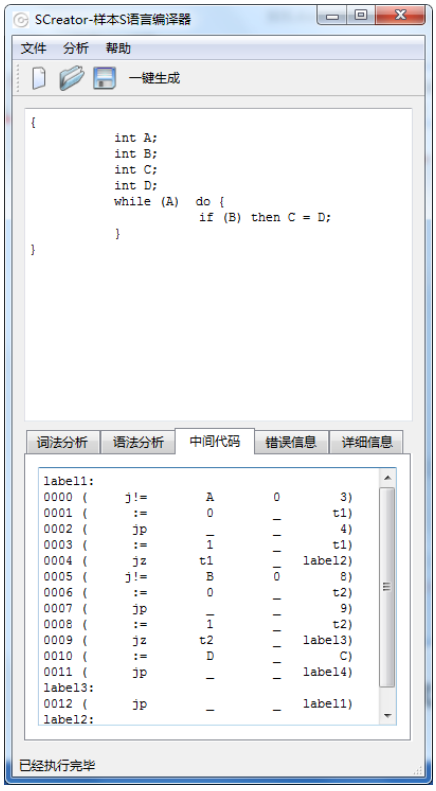
1. 不应当纠结于效率，而要优先考虑程序清晰度。采用多遍分析虽然效率低，但模块耦合低，易于维护。
2. 增量式开发，保持代码可发布。
3. 千万不要一直埋头写程序，而不编译运行，多编写一些测试程序，边编写边测试，可以促进效率。过程中的错误是很多的，推迟测试可能造成程序 bug 难以处理或处理时间大幅增加。
4. 抓住核心功能。不要做没必要的事情，前期不值得花费精力的如界面美化，应当着眼于简洁清晰朴实，不需要事先用户自定义配置，应当着眼于完善功能，并提供默认良好的配置。
5. 数据结构设计的重要性。在开源 C 编译器程序 ucc 中，语法分析调用词法分析时只返回索引，这样显然是高效的，由于语法分析不需要单词的值，本程序采用结构体而非简单的索引，虽然有损效率，但更加直观，易于维护。设计数据结构对组织易于扩展和维护的程序是非常重要的。
6. 记录的重要性。在编写程序的过程中同时撰写文档，并及时进行程序备份，模块测试程序的保留，测试截图等等都是对开发有益的。
7. 模块式开发要点：非工程接口头文件（库文件）建议放在 C 文件中，否则出现 warning，

调试实例

正常翻译情况

左图为赋值语句的翻译，右图为控制语句的翻译

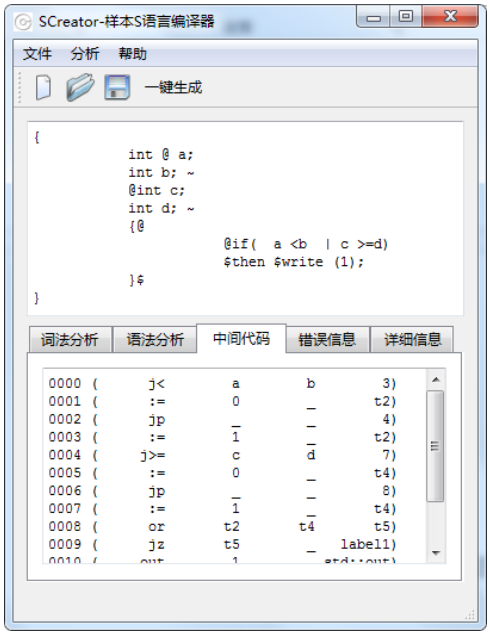


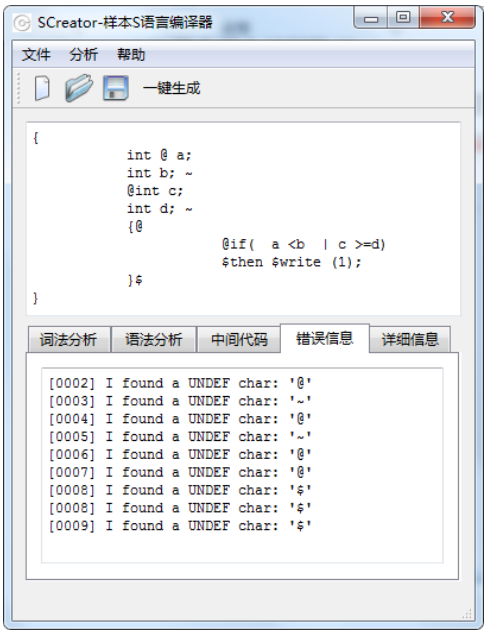


错误检查及恢复特性测试

屏蔽非法字符的影响

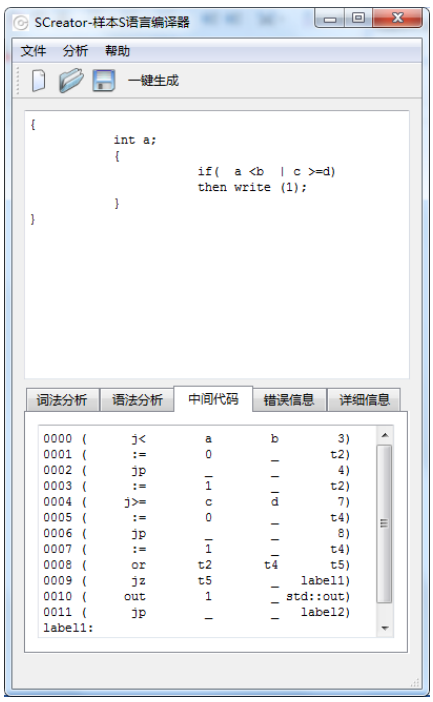
可以任意插入非法字符，不影响正确代码的生成

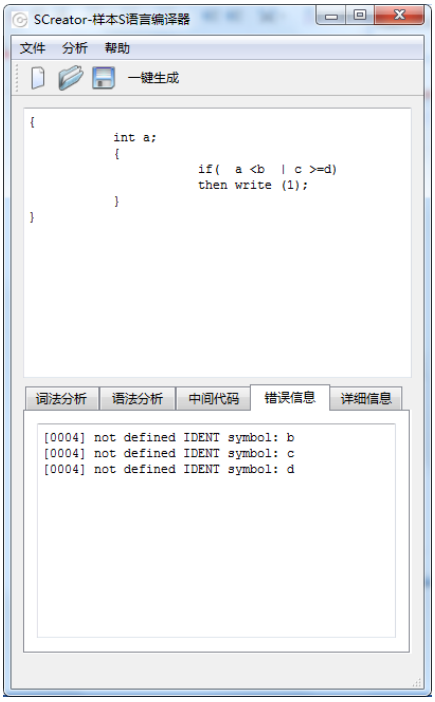




未定义的标识符补充定义

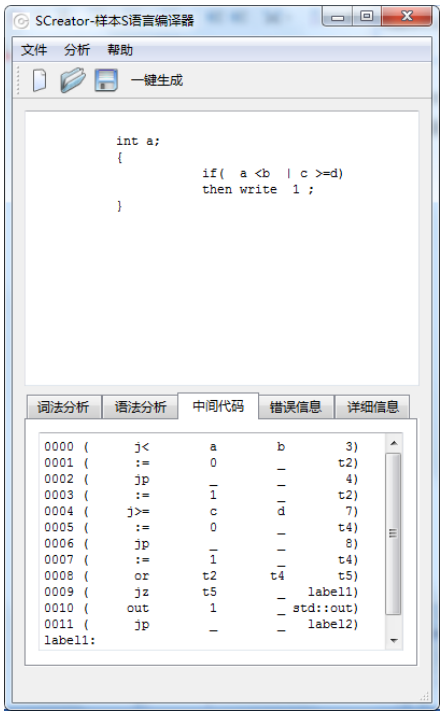
提示错误并补充定义继续执行，但是生成的四元式仍然是正确的 。

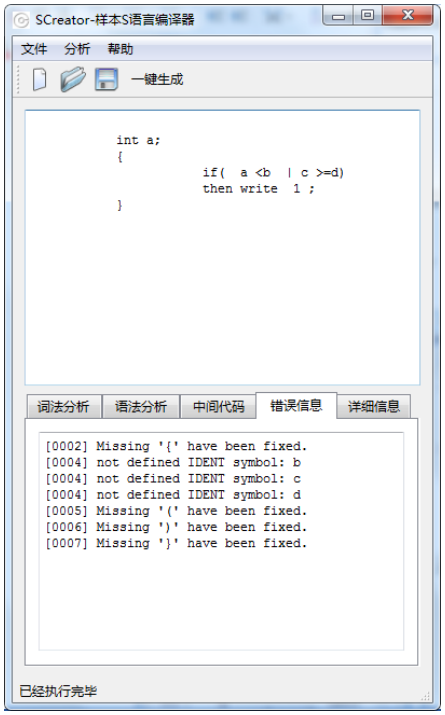




多一个或少一个字符的情况处理

在前面的基础上，再加上错误，程序忘了要用花括号包裹，write 语句也忘了圆括号，可以看出程序可以进行正确的错误恢复

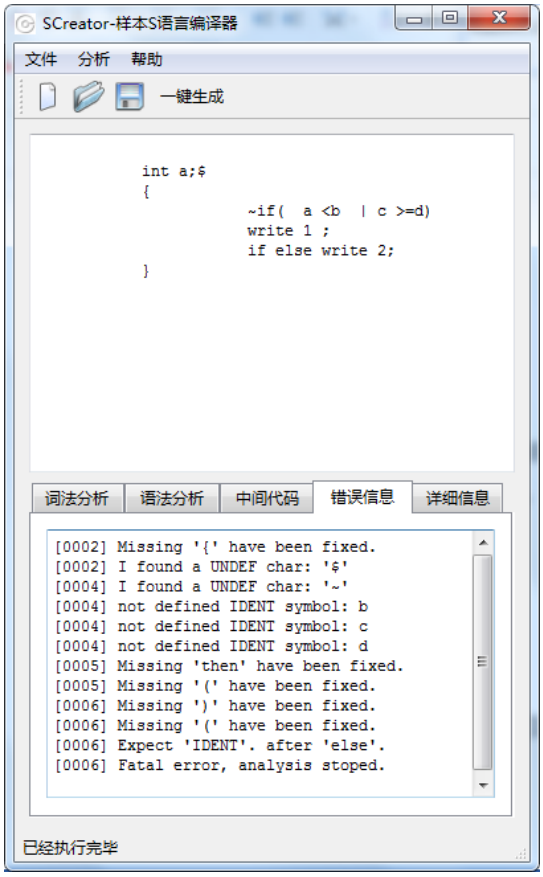




严重错误终止

如果有难以处理的错误，程序会终止分析，防止分析发生混乱。

可以看出程序修复了大多数错误，但对于 if else 这句严重错误，程序终止了。



程序代码

详细代码已提交到网上学习中心，这里贴出关键部分

错误恢复

```
static FILE* errFile = NULL; // 错误写入文件中
// 非法字符直接在 getSym 时跳过，不再在这里处理
static int tokenBackNum = 0; // 尝试恢复时向前看几个单词，恢复完成回退单词
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(-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]);
            }
        }
        // else 正常情况
}
static void Error_ifIdentNotDefined(char* id){
        // 对可以补充定义的情况进行补充定义工作
        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);
            }
        }
}
```

词法分析核心

```
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;
  }
}
```

语法语义分析及代码生成部分

```
// 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;
}
```