

编译原理

课程设计报告

题 目：**Simple PASCAL编译器的设计与实现**

学 院： 计算机与信息安全

专 业： 计算机科学与技术

学生姓名： 廖健生

学 号： 1400310219

指导教师： 王慧娇

题目类型：〇理论研究 √实验研究 〇工程设计 〇应用研究 〇软件开发

2017年 6 月 27 日

摘 要

编译原理课程是一门计算机专业重要的专业课，目的在于介绍编译程序构造的一般原理和方法，本门课程课程设计引导学生通过学习到的内容来实现简单的编译器系统设计。本门课程设计内容包括语言和文法、词法分析、语法分析、[语法制导翻译](http://baike.baidu.com/item/%E8%AF%AD%E6%B3%95%E5%88%B6%E5%AF%BC%E7%BF%BB%E8%AF%91)、中间代码生成、[存储管理](http://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E7%AE%A1%E7%90%86)、代码优化和目标代码生成。而其课程设计目的是让学生能够自己动手实现简单的编译器具有的词法、语法、语义分析，锻炼学生动手、将理论运用于实践的能力。

PASCAL语言是一种计算机通用的高级[程序设计语言](http://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1%E8%AF%AD%E8%A8%80)，其具有简洁的语法，结构化的程序结构，是一门结构化编程语言。由于PASCAL语言的重要性以及其语法结构特点，适合当做本门课程设计的目标编译语言。在本门课程设计中，本人将会使用C语言编写一套能够将简单PASCAL语言编译成为汇编语言并且能够简单模拟运行的课程设计。

关键词：编译原理；编译器；词法分析；语法分析；语义分析

目 录

[引言 1](#_Toc486523794)

[1 Simple-PASCAL语法 1](#_Toc486523795)

[1.1 文法 1](#_Toc486523796)

[1.2 语法说明 2](#_Toc486523797)

[2 编译器系统框架 2](#_Toc486523798)

[2.1 通用链表 3](#_Toc486523799)

[2.2 二元式结构 4](#_Toc486523800)

[2.3 语句、语句表结构 5](#_Toc486523801)

[2.4 汇编指令结构 5](#_Toc486523802)

[3 词法分析 5](#_Toc486523803)

[3.1 二元式 6](#_Toc486523804)

[3.2 词的分割 6](#_Toc486523805)

[3.3 词的类型 7](#_Toc486523806)

[3.4 二元式的生成 9](#_Toc486523807)

[3.5 词法错误提示信息 9](#_Toc486523808)

[4 语法分析 10](#_Toc486523809)

[4.1 语句的链表 10](#_Toc486523810)

[4.2 语句分割 11](#_Toc486523811)

[4.3 表达式合法判断 12](#_Toc486523812)

[4.4 逐语句判断 14](#_Toc486523813)

[4.5 语法分析提示信息 15](#_Toc486523814)

[5 语义分析 16](#_Toc486523815)

[5.1 符号表 17](#_Toc486523816)

[5.2 汇编指令表 18](#_Toc486523817)

[5.3 表达式中间代码生成 18](#_Toc486523818)

[5.4 赋值语句 18](#_Toc486523819)

[5.5 IF语句 19](#_Toc486523820)

[5.6 WHILE语句 20](#_Toc486523821)

[5.7 汇编代码输出 20](#_Toc486523822)

[5.8 代码执行 21](#_Toc486523823)

[结论 22](#_Toc486523824)

[参考文献 23](#_Toc486523825)

[附 录 24](#_Toc486523826)

[1 词法分析 24](#_Toc486523827)

[2 语法分析 27](#_Toc486523828)

[3 语义分析 32](#_Toc486523829)

[4 通用链表 36](#_Toc486523830)

引言

为了能够系统化、结构化进行PASCAL语言编译器的设计，本文将从文法分析、词法分析、语法分析、语义分析、中间代码生成、汇编代码四元式模拟执行几个方面顺序介绍PASCAL语言编译器的设计。为了能够更加深入的了解编译完成后符号表、模拟内存的实现方法以及通用链表的实现方法，本文使用C语言作为编译器的编程语言，而不使用其他简单结构化、数组化的编程语言。

# 1 Simple-PASCAL语法

为了适应本门课程设计特点，本人对PASCAL语言的生成式做了简单的修改，例如，增加了INPUT、OUTPUT关键字。

## 1.1 文法

G[<语句表>]

<语句表>→<语句> | <语句>;<语句表>

<语句>→<赋值语句>|<条件语句>|<WHILE语句>|<复合语句>|<输入输出语句>

<赋值语句>→<变量>:=<算术表达式>

<条件语句>→IF<关系表达式>THEN<语句>(ELSE<语句>)

<WHILE语句>→WHILE<关系表达式>DO<语句>

<复合语句>→BEGIN<语句表>END

<算术表达式>→<项>|<算术表达式>+<项>|<算术表达式>-<项>

<项>→<因式>|<项>\*<因式>|<项>/<因式>

<因式>→<变量>|<常数>|(<算术表达式>)

<关系表达式>→<算术表达式><关系符><算术表达式>

<变量>→<标识符>

<标识符>→<标识符><字母>|<标识符><数字>|<字母>

<常数>→<整数>

<整数>→0|<非零数字><泛整数>

<泛整数>→<数字>|<数字><泛整数>|ε

<关系符>→<|<=|==|>|>=|<>

<字母>

→A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<非零数字>→1|2|3|4|5|6|7|8|9

<数字>→<非零数字>|0

## 1.2 语法说明

Simple-PASCAL语言由赋值语句、判断语句、循环语句构成，BEGIN以及END包裹的语句合为语句表。

### 1.2.1 BEGIN END

BEGIN ... END 包裹的是一个语句表，其用法如下：

BEGIN （语句|语句表） END

可嵌套，在其中声明的变量作用于仅在其变量的最内层嵌套之中，在此之外变量无效。

### 1.2.2 赋值语句

在赋值语句中，使用形式为：

x:=(表达式);

其中，x必须为标识符，赋值语句本身也是表达式的一种，其值为x在赋值之后的值。

表达式的值应该与X的类型相同或表达式的值可由X的类型充分表示。

### 1.2.3 IF语句

在判断语句中，使用形式为：

IF (表达式) THEN (语句)

其中，表达式的值如果为真（值不为0），则将执行随后的第一个语句（或语句表），否则表达式的值如果为假（值等于0），则跳过随后的第一个语句（或语句表）。

### 1.2.4 WHILE语句

在循环语句中，使用形式为：

WHILE (表达式) DO (语句)

其中，表达式的值如果为真（值不为0），则将执行随后的第一个语句（或语句表），否则表达式的值如果为假（值等于0），则跳过随后的第一个语句（或语句表）。如果表达式为真时，执行完接下来的语句体后，程序跳转至WHILE语句的表达式判断处，重新判断。

直到WHILE语句中的表达式的值为假，循环才会退出。

# 2 编译器系统框架

编译器的主要流程包括：读入文件、词法分析、语法分析、语义分析、汇编指令生成。其流程图如图2.1所示。

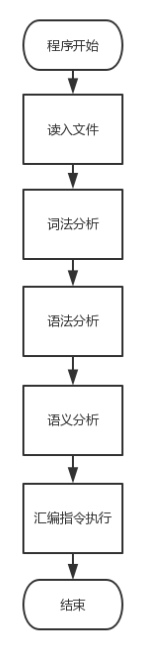


图2.1

## 2.1 通用链表

利用通用链表索引我们的数据结构对象。由于我们在本门课程设计中需要用到各种各样的链表（简单包括了词法二元式链表、语句语句表链表、四元式链表、汇编程序链表、符号表链表等等），因此我设计了一套能够存储各种数据类型的指针的通用链表。与普通的通用链表的区别在于，通用链表存储的不是数据本身，而是数据对象的指针，因此我们可以通过一套通用的链表来实现各种各样的链表的操作，节省重复的代码。

通用链表包含头结点，头结点以及链表节点结构定义如下：

typedef struct LinkNode{ //定义一个用于存储目标数据指针的链表节点类型

void \*data; //用于存储目标数据对象的指针

LinkNode \*next; //下一个节点指针

LinkNode \*pro; //上一个节点指针

}LinkNode;

typedef struct{ //定义一个链表结构

int len; //链表长度

LinkNode \*head;//链表头结点指针

LinkNode \*tail; //链表尾节点指针

}Link;

本处使用的通用链表是一个双向链表类型，为通用链表提供了各种各样操作链表的方法，无论在何时何地，只需要提供对应的数据对象的地址即可进行存储，提供的方法有：

//创建链表

Link \*link\_create();

//插入数据，后插

void link\_add(Link \*link,void \*p);

//获取第i个节点

void \*link\_get\_i(Link \*link,int i);

//删除第i个节点

void link\_del\_i(Link \*link,int i);

//销毁整个链表

void link\_destroy(Link \*\*link);

//在toLink后面添加sorLink

void link\_add\_link(Link \*toLink,Link \*sorLink);

//为link创建备份的新的link

Link \*link\_back(Link \*link);

//删除链表中的节点，但不删除链表本身

void link\_clere(Link \*link);

各个函数的实现请参见附录。

## 2.2 二元式结构

二元式是用来存储词法分析生成结果的数据结构，每一个二元式具有类型、内容基本属性，例如：BEGIN是一个关键字，IF是另一个关键字，为了区分其不同，需要为BEGIN和IF分配不同的类型标志，并且将类型以及其字符串存进一个有序的二元式链表中，以便语法和语义分析的按源代码顺序读入处理。

由此可知，对于不同的源代码，其二元式对象个数是不定的，因此我们可以利用2.1设计的通用链表来存储每一个二元式对象的指针。

## 2.3 语句、语句表结构

语句是源程序的抽象构成，源程序是由语句以及语句表直接构成的。其中，语句表是语句的集合。而语句本身是由一个一个词构成的，在词法分析之后，每一个词都变成了一个二元式对象，因此我们的语句可以说是由二元式构成的。由此可知，我们可以用一条有序的二元式链表来表示语句或语句表。用以给语法分析以及语义分析作处理源。

根据语句的需求，我们同样可以利用一条通用链表对象来表示语句。

## 2.4 汇编指令结构

对于语义分析生成的中间代码，利用汇编来表示是很恰当的做法。汇编指令中，包含了操作码、源操作数、被操作数、立即数几个属性，每个汇编指令都可以用这个结构表示。如同二元式，对每个源程序，其生成的汇编指令条数也是不定的，因此应该使用链表来存储每一条汇编指令对象。

同样，我们可以像二元式一样，利用通用链表来存储每一个汇编指令对象的指针，使链表的代码复用率增加，使我们的程序设计更加严谨规范。

# 3 词法分析

对于词法分析，其主要的流程包括了对输入字符串流进行分割，将输入文件分割成为正确的单词序列（注意，类似“:=”、“<=”等有效的运算符以及分号等分隔符也是单词的一种，下同），随后判断单词的类型，生成二元式链表，最后对词法分析中发生的错误进行提示，其流程图如图3.1所示。

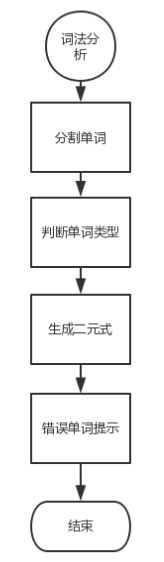


图3.1 词法分析流程图

## 3.1 二元式

二元式是用来存储每一个词类型以及参数等的数据结构。其定义如下：

typedef struct {

int dual\_type; //类型

union { //内容 字符串|整数

char lexme\_text[100]; //字符串

int lexme\_num; //整数

}lexme; //内容

int line; //所在行号

}Dual;

其中，联合（Union）使用的目的是使一个词的内容更加清晰，联合中的（lexme\_text和lexme\_num）共同使用一段内存，联合lexme的大小由最大的元素决定，在此处其大小为100个char型大小。

二元式是贯穿整个编译器设计的数据对象，它由词法分析生成，语法分析以以及语义分析均使用其中的内容作参考。由于整个目标PASCAL程序生成的二元式数量不应预测，所以其应该用链表方式链接索引。在本文中其每个二元式的指针索引由通用链表维护。

## 3.2 词的分割

由语法生成式，我们可以知道各种词法的结束符：

常数的结束符为非数字；

英文单词结束符为非英文且非数字；

其余各种运算符/分割符的结束符为其后不可能的组合，其结束符可以是数字/字母/符号。

其主要算下如下（全部算法请见附录：词法分析）：

int get\_word(char \*word,char \*s)

{

int len=0,i=0;

//跳过可忽略字符

while(s[i]==' '||s[i]=='\t'||s[i]=='\r'||s[i]=='\n'){

i++;

}

if ((s[i]>='0'&&s[i]<='9')||(s[i]>='a'&&s[i]<='z')//数字/标识符

||(s[i]>='A'&&s[i]<='Z')){//num/w

while ((s[i]>='0'&&s[i]<='9')||(s[i]>='a'&&s[i]<='z')

||(s[i]>='A'&&s[i]<='Z')){

word[len++]=s[i++];

}

}else if (s[i]==':'){ //:

word[len++]=s[i++];

if (s[i]=='='){ //:=

word[len++]=s[i++];

}

}

}

else if (s[i]=='/'){ ///

word[len++]=s[i++];

if (s[i]=='/'||s[i]=='\*'){ //// /\*

word[len++]=s[i++];

}

}else{//其他

while (s[i]&&s[i]!=' '&&s[i]!=';'&&s[i]!='\t'&&s[i]!='\r'&&s[i]!='\n'&&s[i]!=':'&&s[i]!='='

&&s[i]!='<'&&s[i]!='>'&&s[i]!='/'&&s[i]!=','){

word[len++]=s[i++];

}

}

word[len]=0;

return i;

}

## 3.3 词的类型

为了区分不同的词的类型，我将词分为：标识符/常数INT/IF/THEN/WHILE/DO/BEGIN/END/赋值/等于/大于/大于等于/小于/小于等于/不等于/加/减/乘/除。

在程序中，词的类型如下定义：

#define T\_ERR 0 //错误

#define T\_ID 1 //标识符

#define T\_EQ 2 //相等

#define T\_IF 3 //IF

#define T\_THEN 4 //THEN

#define T\_ELSE 5 //ELSE

#define T\_BEGIN 6 //BEGIN

#define T\_END 7 //END

#define T\_WHILE 8 //WHILE

#define T\_DO 9 //DO

#define T\_INT 10 //常数

#define T\_LT 11 //小于

#define T\_LE 12 //小于等于

#define T\_NE 13 //不等于

#define T\_GT 14 //大于

#define T\_GE 15 //大于等于

#define T\_N\_LINE 16 //行注释符

#define T\_N\_BEGIN 17//段注释符开始

#define T\_N\_END 18 //段注释符结束

#define T\_CMP0 19 //相等

#define T\_ADD 20 //加

#define T\_SUB 21 //减

#define T\_MUL 22 //乘

#define T\_DIV 23 //除

#define T\_CUT 24 //分隔符 ;

#define T\_POINT 25 //逗号

#define T\_CUT\_LEFT 27//左括号

#define T\_CUT\_RIGHT 28//右括号

#define T\_INPUT 100 //INPUT

#define T\_OUTPUT 101//OUTPUT

详细地区分词的类型，使得每个词的类型能够代表其独有的含义。

在词法分析的判断词的类型的过程中，如果碰到了不能够被识别的单词（例如123abc），其类型将会被置为ERR错误类型，用于给后续流程做参考。

简单识别词的类型算法如下（全部算法请见附录：词法分析）：

int get\_type(char \*word)

{

int i=0;

//关键字

if (strcmp(word,"BEGIN")==0){

return T\_BEGIN;

}else if (strcmp(word,"END")==0){

return T\_END;

}else if (strcmp(word,"IF")==0){

return T\_IF;

}else if (strcmp(word,"THEN")==0){

return T\_THEN;

}else if (strcmp(word,"ELSE")==0){

return T\_ELSE;

}else if (strcmp(word,"WHILE")==0){

return T\_WHILE;

}else if (strcmp(word,"DO")==0){

return T\_DO;

}

//常数

else if (word[i]>='0'&&word[i]<='9'){//int

while (word[i]>='0'&&word[i]<='9'){

i++;

}

if (word[i]==0){

return T\_INT;

}

}

//标识符

else if ((word[i]>='a'&&word[i]<='z')||(word[i]>='A'&&word[i]<='Z')){//w

while ((word[i]>='a'&&word[i]<='z')||(word[i]>='A'&&word[i]<='Z') ||(word[i]>='0'&&word[i]<='9')){

i++;

}

if (word[i]==0){

return T\_ID;

}

}

return T\_ERR;

}

## 3.4 二元式的生成

由于需要记录词所在的行，必须对在每次取出词的时候判断我们跳过了几行或者是使用一种方法记录下我们访问的行的数量。

## 3.5 词法错误提示信息

将所有的词区分出来之后，我们需要对词列表中的错误的单词做出相应的信息的提示，例如我们需要提示类似@、123abc等不能被识别的词。词法错误提示如图3.2。

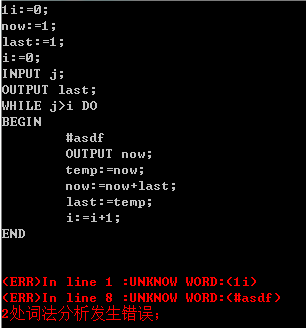


图3.2 词法错误样例

# 4 语法分析

为了对语法进行模块化的分析，方便将来代码的修改，语法分析需要模块化地按代码块进行分析。

语句的分割作为语法分析的第一步，其目的与词法分析的第一步（分割词）类似，我们需要根据语法生成式来判断一个语句将由哪几个二元式构成，将这几个二元式当做一条语句（或语句表）进行处理。

语法分析的主要流程如图4.1所示。

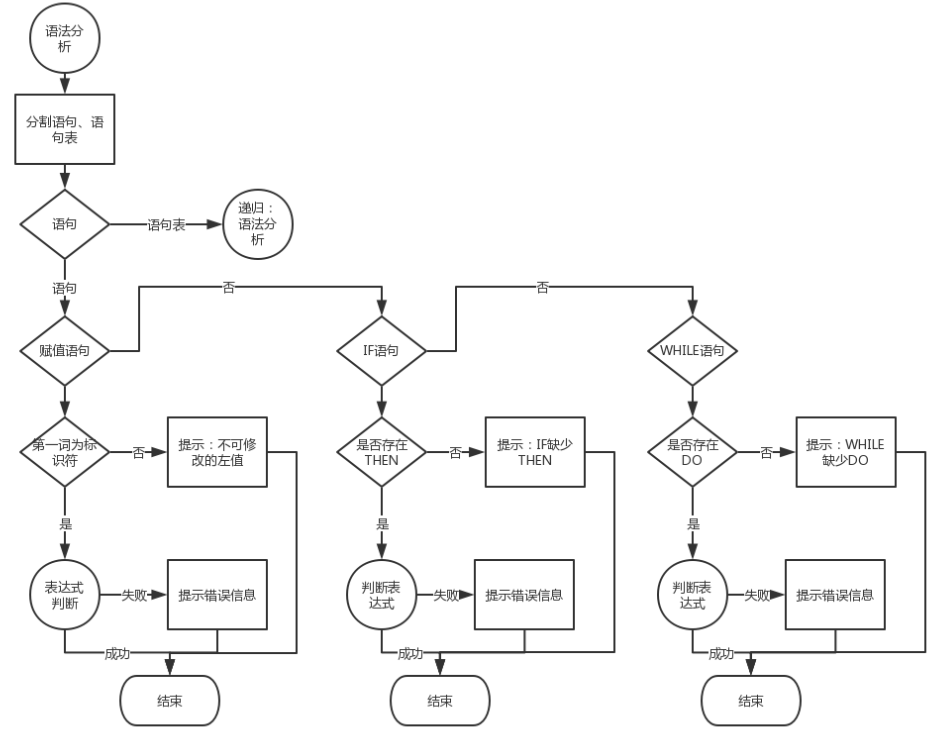


图4.1 语法分析流程图

## 4.1 语句的链表

由于一个句子的二元式数量不定，需要利用链表保存一个句子中的所有二元式，为了保存所有的句子。由于目标程序的语句数量也是不定的，因此也需要利用一个链表来储存所有的句子（二元式链表），即链表的链表。此时，通用链表的优势得以体现。

## 4.2 语句分割

分割程序将从第一个二元式开始往下扫描，当遇到以下操作时，将语句二元式链表添加到所有语句链表当做一个句子并且生成一个新的语句二元式链表：

分割附“ ; ”：正常句子的结尾；

THEN：IF语句的结尾；

DO：WHILE语句的结尾；

END：最外层BEGIN与之配套的END，为了匹配最外层的BEGIN...END，需要利用一个变量记录匹配数量，当遇到BEGIN时+1，当遇到END时-1，当该变量在遍历完二元式仍未回到0时说明缺少END，应提示错误信息。

将所有的词法二元式分割成为语句或语句表，其中，BEGIN以及NED匹配包含的内容为语句表，其余（如赋值语句、IF语句、WHILE语句）单语句结构将作为一条语句，语句使用语句链表存储，链表中仅仅包含本语句、语句表的二元式，用来给后面逐语句判断。

语句分割主要算法如下：

//判断语句表开头,跳过二元式最外层的begin 和 end

if (get1->dual\_type==T\_BEGIN&&get2->dual\_type==T\_END){

iner=1;

i=1;

}else

i=0;

have\_begin=0;

for (;i<link->len;){

ju=link\_create();

have\_begin=0;

do{

get=(Dual \*)link\_get\_i(link,i);

if (get==NULL){

printf("缺少关键元素！\n");

return 1;

}

if(get->dual\_type==T\_BEGIN){

have\_begin++;

}else if(get->dual\_type==T\_END){

if(have\_begin==0){

printf("缺少开始符！（在第%d行）\n",get->line);

return -1;

}else {

have\_begin--;

}

}

link\_add(ju,get);

if(iner==1&&i==link->len-2){

i+=2;

break;

}

i++;

}while (((get->dual\_type!=T\_ELSE&&get->dual\_type!=T\_DO&&get->dual\_type!=T\_THEN&&get->dual\_type!=T\_CUT&&get->dual\_type!=T\_END)||(have\_begin))&&i<link->len);

link\_add(all\_ju,ju);

}

## 4.3 表达式合法判断

首先，表达式是由二元式构成的，一条表达式可由一条二元式链表表示。

对于表达式语法是否合法的判断，主要从以下几点判断：

括号是否匹配，是否存在不匹配的左括号或右括号；

运算符后面只能跟着表达式，运算符后面只能跟着标识符、常数、左括号；

标识符、常数后面只能跟着运算符、右括号、空；

左括号后面只能跟着标识符、常数；

右括号后面只能跟着运算符、空。

由上述条件可知，其判断算法可直接利用判断的方法，也可以使用可达矩阵表示，这里我们进行直接判断，其主要算法如下：

//判断表达式是否合法

int biaodashi\_hefa(Link \*link)

{

int i=0,t,last\_t,kuo\_sum=0;

Dual \*p;

t=last\_t=get\_type\_in\_link(link,0);

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_LEFT){

printf("错误的表达式开始！在第%d词！\n",i);

return -1;

}

for (i=1;i<link->len;i++){

p=(Dual \*)link\_get\_i(link,i);

t=p->dual\_type;

switch(last\_t){

case T\_ID: case T\_INT: //操作数

if (t!=T\_ADD&&t!=T\_SUB&&t!=T\_MUL&&t!=T\_DIV&&t!=T\_CUT\_RIGHT&&

t!=T\_LT&&t!=T\_LE&&t!=T\_NE&&t!=T\_GT&&t!=T\_GE&&t!=T\_CMP0){

printf("错误的运算符！在第%d词！\n",i);

return -2;

}

break;

case T\_CUT\_LEFT: //左括号

kuo\_sum++;

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_LEFT){

printf("错误的操作数！在第%d词！\n",i);

return -3;

}

break;

case T\_CUT\_RIGHT: //右括号

if (kuo\_sum==0){

printf("错误的右括号！在第%d词！\n",i);

return -12;

}else

kuo\_sum--;

if (t!=T\_ADD&&t!=T\_SUB&&t!=T\_MUL&&t!=T\_DIV&&t!=T\_CUT\_RIGHT&&

t!=T\_LT&&t!=T\_LE&&t!=T\_NE&&t!=T\_GT&&t!=T\_GE&&t!=T\_CMP0){

printf("错误的运算符！在第%d词！\n",i);

return -4;

}

break;

case T\_ADD:case T\_SUB: case T\_MUL: case T\_DIV: case T\_LT: case T\_LE: case T\_NE:

case T\_GT: case T\_GE: case T\_CMP0: //运算符

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_LEFT){

printf("错误的操作数！在第%d词！\n",i);

return -5;

}

break;

default:

return -100;

break;

}

last\_t=t;

}

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_RIGHT){ //错误的表达式结尾

return -6;

}

if (t==T\_CUT\_RIGHT)kuo\_sum--;//最后右括号回复

if (kuo\_sum!=0){ //判断括号是否匹配

printf("错误的括号匹配！在第%d词！\n",i);

return -7;

}

//表达式合法

return 0;

}

## 4.4 逐语句判断

轮询语句链表中的所有语句，对所有的单条语句进行语法判断处理，对所有语句表（由最外层BEGIN...END包裹的代码块）进行递归下降。

当一条语句的第一个词为BEGIN且最后一个词为END时，认为其是一个语句表，将最外层的BEGIN和END去掉之后直接递归进语法分析函数。

根据语法生成式，对不同的单语句采取不同的策略，例如：

若第二个词为赋值符，应将本条语句按赋值语句处理；

若第一个词为IF，本条语句为判断语句，语句的最后一个词必须为THEN，IF与THEN之间为表达式，应将表达式送入表达式检查函数；

若第一个词为WHILE，本条语句为循环语句，语句的最后一个词必须为DO，WHILE与DO之间为表达式，应将表达式送入表达式检查函数；

若第一个词为INPUT，本条语句为输入语句，后面应该仅跟一个标识符；

若第一个词为OUTPUT，本条语句为输出语句，后面应该仅跟一个标识符或者常数。

语句处理的错误，并且根据语句的错误进行判断。

其主要算法如下：

//遍历所有语句

for(i=0;i<all\_ju->len;i++){

ju=(Link \*)link\_get\_i(all\_ju,i);

get=(Dual \*)link\_get\_i(ju,0);

if ((get->dual\_type==T\_IF||get->dual\_type==T\_WHILE)&&i==all\_ju->len-1){

printf("\n不完整的语句体！\n");

err\_sum++;

}

if (get->dual\_type==T\_BEGIN){

if ((temp=gram\_control(ju,level+1))!=0){//递归遍历语句集

get=(Dual \*)link\_get\_i(ju,0);

err\_sum+=temp;

}

}else {

for (j=0;j<ju->len;j++){

get=(Dual \*)link\_get\_i(ju,j);

}

if (test\_ju(ju)!=0){

printf("\n语法错误！\n");

err\_sum++;

}

}

}

## 4.5 语法分析提示信息

当发生语法错误时，提示当前语法错误的位置，报告其错误的原因，对简单的语法错误提示修改方法。部分语法错误如图4.2所示。

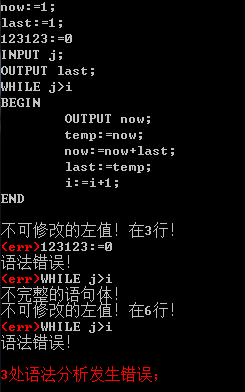


图4.2 语法错误提示

为了能够直观看到语法分析的过程以及结果，对不同的二元式、递归层次，可以在分析过程中将语法树显示出来，例如，对于源程序：

now:=1;

last:=1;

i:=0;

INPUT j;

OUTPUT last;

WHILE j>i DO

BEGIN

OUTPUT now;

temp:=now;

now:=now+last;

last:=temp;

i:=i+1;

END

其生成的语法树应该如图4.3所示。

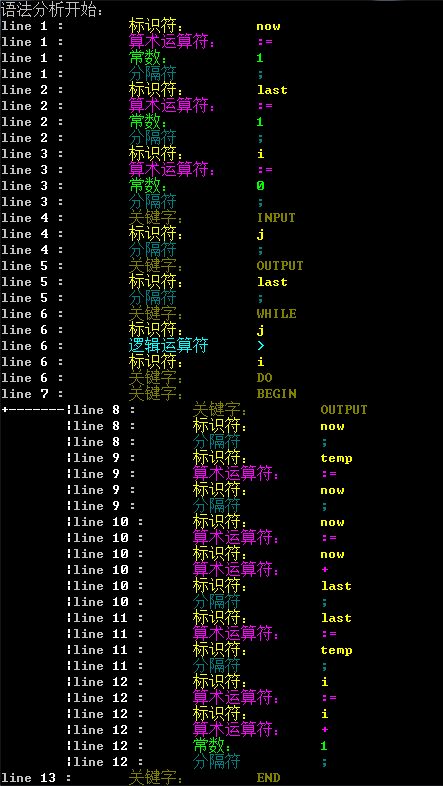


图4.3 显示语法树

# 5 语义分析

语义分析的流程与语法分析相似，同样需要对句子进行操作，对不同的句子采取不同的判断策略，其主要流程如图5.1所示。

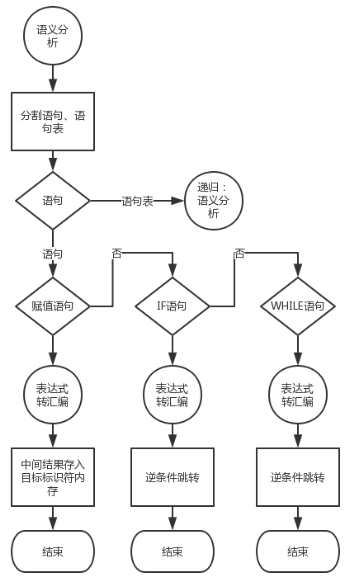


图5.1 语义分析流程图

## 5.1 符号表

在PASCAL源程序中，每一个标识符都是一个符号，其表示一个地址，我们实际是需要将数据存于这个符号对应的内存地址中，因此需要设计符号表，应该具有添加、删除、获取一个符号对应地址等功能，可以用链表存取，因此同样使用通用链表存放每个符号对象指针。其中符号结构设计为：

typedef struct {

int addr;

char name[100];

}Id;

其中，addr表示该符号表示的地址，name表示该符号的名字。

提供的符号表操作的方法有：

//获取一个符号名对应的内存地址，不存在该符号时返回-1

int get\_id\_addr(Link \*id\_link,char \*id\_name);

//将一个符号添加到符号表中，并返回该符号的地址，如果符号以存在则返回-1

int add\_id\_addr(Link \*id\_link,char \*id\_name);

## 5.2 汇编指令表

汇编指令的结构可以总结为一个包含操作码、源寄存器、目标寄存器、立即数的结构体，其结构体设计如下：

typedef struct {

int type;

int op;

int op\_ed;

int num;

}Asm;

由于生成的汇编指令的数量不定，因此我们可以利用链表来存放每一个汇编指令，可以使用通用链表来存放每一个汇编结构体的指针。

在语法分析的基础上，利用语法分析的方法将二元式分割成为语句或者语句表，语句将会在本层递归处理，语句表则递归下降处理。

## 5.3 表达式中间代码生成

在赋值语句、条件语句、循环语句中都有表达式的存在，表达式的值是真正用于赋值、判断的值，由于表达式是可以由表达式复合的，并且有优先级的区分，因此需要对其进行中间值、递归处理。

表达式中，所有的标识符必须在之前声明赋值，否则不能用于取值，当表达式中存在未声明的变量时应该报错。

由于这部分代码偏长，请参加附录：语义分析。

## 5.4 赋值语句

赋值语句的特点是语句的第二个单词为":="赋值符，否则不可认为其是赋值语句。赋值语句的第3个至结束符之间的词为表达式，应该将其取出，交由表达式处理函数进行处理，约定表达式的结果存放于某个寄存器中，在表达式分析结束后，将寄存器中的值存放于表达式目标标识符的内存单元中，即可完成赋值任务。

关键代码如下：

if(have\_type(ju,T\_EQ)){//:= 赋值语句

//获取表达式

link=link\_create();

i=2;

while (get\_type\_in\_link(ju,i)!=T\_CUT&&i<ju->len){

get=(Dual \*)link\_get\_i(ju,i);

link\_add(link,get);

i++;

}

//产生四元式

get=(Dual \*)link\_get\_i(ju,0);

biaodashi\_to\_asm(link,ju\_asm,id\_table);

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R2,0,add\_id\_addr(id\_table,get->lexme.lexme\_text)));//MOV R2,addr

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_MEM\_REG,REG\_R2,REG\_R1,0));//mov (R2),R1

link\_destroy(&link);

}

## 5.5 IF语句

判断语句的特点是第一个词为IF类型，否则不可认为其是判断语句，IF至THEN之间的内容为表达式，应该将其取出，交由表达式处理函数处理，处理完即可进行跳转的处理。

在IF语句的条件表达式运算完成后，应该在条件不满足时跳转至IF语句体之后，这就需要知道IF语句的下一个语句（语句表）的汇编指令长度，因此跳转指令的目的地待定，将IF的下一条语句取出，交由语义分析递归处理完成后，计算该条语句生成的汇编代码的长度，然后再回头设置跳转出口。

为了能够知道跳转的长度（PC寄存器的值）为多少，需要将IF的下一个语句（或语句表）语义分析之后，计算其中汇编代码长度，再进行跳转长度的设置，具体代码如下（部分函数定义未列出，请参见附录：语义分析）：

else if (have\_type(ju,T\_WHILE)||have\_type(ju,T\_IF)){//IF/WHILE语句

//获取表达式

link=link\_create();

i=1;

while (get\_type\_in\_link(ju,i)!=T\_DO&&i<ju->len){

get=(Dual \*)link\_get\_i(ju,i);

link\_add(link,get);

i++;

}

//产生四元式

get=(Dual \*)link\_get\_i(ju,0);

biaodashi\_to\_asm(link,ju\_asm,id\_table);

if (have\_type(ju,T\_LT)){

link\_add(ju\_asm,asm\_word(T\_ASM\_NB,0,0,0));//JB \* //小于->不小于

}else if (have\_type(ju,T\_GT)){

link\_add(ju\_asm,asm\_word(T\_ASM\_BE,0,0,0));//NB \* //大于->不大于(小于等于)

}else if (have\_type(ju,T\_NE)){

link\_add(ju\_asm,asm\_word(T\_ASM\_E,0,0,0));//NB \* //不等于->等于

}else if (have\_type(ju,T\_LE)){

link\_add(ju\_asm,asm\_word(T\_ASM\_JG,0,0,0));//NB \* //小于等于->大于

}else if (have\_type(ju,T\_GE)){

link\_add(ju\_asm,asm\_word(T\_ASM\_JB,0,0,0));//NB \* //大于等于->小于

}else if (have\_type(ju,T\_CMP0)){

link\_add(ju\_asm,asm\_word(T\_ASM\_NE,0,0,0));//NB \* //等于->不等于

}

link\_destroy(&link);

}

## 5.6 WHILE语句

对于循环语句来说，与IF语句相似。区别仅仅在于，在循环体结束之后，需要加一条JMP语句，使之跳转至WHILE条件判断之前，其他部分与IF语句的处理一致。

## 5.7 汇编代码输出

为了了解生成的汇编指令是否正确，将语义分析中生成的汇编指令表以汇编代码形式输出显示出来是很必要的，对于源程序：

now:=1;

last:=1;

i:=0;

INPUT j;

OUTPUT last;

WHILE j>i DO

BEGIN

OUTPUT now;

temp:=now;

now:=now+last;

last:=temp;

i:=i+1;

END

其生成的汇编指令代码如图5.1所示。

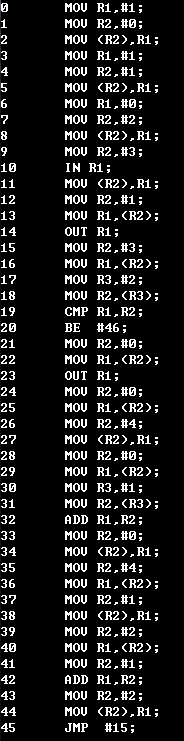


图5.1 汇编代码输出

根据源程序，初步可判断，其生成的汇编代码是正确的。

## 5.8 代码执行

为了验证Simple-PASCAL语言编译的汇编指令是否正确，需要设计一个简单的汇编语言模拟器。将汇编指令链表传入，设置PC、CF、ZF、R[0-5]、MEM[100]等变量，模拟汇编指令的执行即可。

为了验证实验结果，利用输入语句输入整数x，程序应输出前x项菲波那切数列，如输入10，则应输出：1 1 2 3 5 8 13 21 34 55 89。上述汇编代码执行结果如图5.2所示。

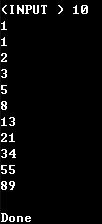


图5.2 编译目标程序执行结果

由图5.2可以看到，PASCAL源程序的目的已经正确达到。

结论

经过本次课程设计，本人成功设计出了一套能够编译Simple-PASCAL语言至汇编语言的编译器，实现了表达式、判断语句、赋值语句、循环语句的编译、运行。

在设计过程中，遇到了许多的对于自动机处理、文本处理、语法处理、汇编跳转原理等问题，但是在老师、网络以及同学的帮助下都得到了完美解决。在网络搜索答案的过程中，开源社区的帮助极大，使我吸收了许多标准的GCC编译实现方法、测试方法，这也告诉我们需要有开源、共享、共同进步的思想，因为我本身也是在别人的开源思想帮助下学习的。

本门课程设计使我对编译器的编译过程有了充分的了解。其中重要的意义是，使我对我编写的C代码编译的过程有系统性了解，由于我的学习方向是嵌入式，我编写的代码经由编译器编译之后，将直接在硬件层面运行，没有了Windows或Linux下的系统内存控制等等局限，我的嵌入式代码需要更加地高效、可靠，因此了解我的代码编译的过程以及汇编代码的生成实现原理是非常重要的。

参考文献

[1] 张素琴，吕映芝，蒋维杜，戴桂兰.编译原理.清华大学出版社，2005.

[2] 杜淑敏，王永宁.编译程序设计原理.北京大学出版社，1986.

[3] 金成植.编译程序构造原理和实现技术.高等教育出版社，2000.

[4] 陈意云.编译原理和技术.中国科技大学出版社，1997.

[5] http://gcc.hun.org/

附 录

## 1 词法分析

#include <string.h>

#include <stdio.h>

#include "word.h"

#include "show.h"

#include <malloc.h>

int get\_word(char \*word,char \*s);

int char\_to\_num(char \*s);

void get\_line(char \*line,char \*s);

int get\_line\_change(char \*s,int len);

int to\_n\_end(char \*s,int type);

int get\_type(char \*word);

//判断一个二元式链表中是否包含某种类型的二元式

int have\_type(Link \*ju,int type)

{

Dual \*get;

int i;

for (i=0;i<ju->len;i++){

get=(Dual \*)link\_get\_i(ju,i);

if (get->dual\_type==type){

return 1;

}

}

return 0;

}

//获取二元式链表中的第i个元素的类型

int get\_type\_in\_link(Link \*link,int i)

{

Dual \*get;

if (link==NULL||i>=link->len){

return -10;

}

get=(Dual \*)link\_get\_i(link,i);

return get->dual\_type;

}

//获取一个字符串的类型

int get\_type(char \*word)

{

int i=0;

//关键字

if (strcmp(word,"BEGIN")==0){

return T\_BEGIN;

}else if (strcmp(word,"END")==0){

return T\_END;

}else if (strcmp(word,"IF")==0){

return T\_IF;

}else if (strcmp(word,"THEN")==0){

return T\_THEN;

}else if (strcmp(word,"ELSE")==0){

return T\_ELSE;

}else if (strcmp(word,"WHILE")==0){

return T\_WHILE;

}else if (strcmp(word,"DO")==0){

return T\_DO;

}

//输入输出

else if (strcmp(word,"OUTPUT")==0){

return T\_OUTPUT;

}else if (strcmp(word,"INPUT")==0){

return T\_INPUT;

}

//逻辑运算符

else if (strcmp(word,"<")==0){

return T\_LT;

}else if (strcmp(word,"<=")==0){

return T\_LE;

}else if (strcmp(word,"<>")==0){

return T\_NE;

}else if (strcmp(word,">")==0){

return T\_GT;

}else if (strcmp(word,">=")==0){

return T\_GE;

}else if (strcmp(word,"==")==0){

return T\_CMP0;

}

//算数运算符

else if (strcmp(word,":=")==0){

return T\_EQ;

}else if (strcmp(word,"+")==0){

return T\_ADD;

}else if (strcmp(word,"-")==0){

return T\_SUB;

}else if (strcmp(word,"\*")==0){

return T\_MUL;

}else if (strcmp(word,"/")==0){

return T\_DIV;

}

//分隔符

else if (strcmp(word,"//")==0){

return T\_N\_LINE;

}else if (strcmp(word,"/\*")==0){

return T\_N\_BEGIN;

}else if (strcmp(word,"\*/")==0){

return T\_N\_END;

}else if (strcmp(word,";")==0){

return T\_CUT;

}else if (strcmp(word,",")==0){

return T\_POINT;

}else if (strcmp(word,"(")==0){

return T\_CUT\_LEFT;

}else if (strcmp(word,")")==0){

return T\_CUT\_RIGHT;

}

//常数

else if (word[i]>='0'&&word[i]<='9'){//int

while (word[i]>='0'&&word[i]<='9'){

i++;

}

if (word[i]==0){

return T\_INT;

}

}

//标识符

else if ((word[i]>='a'&&word[i]<='z')||(word[i]>='A'&&word[i]<='Z')){//w

while ((word[i]>='a'&&word[i]<='z')||(word[i]>='A'&&word[i]<='Z')

||(word[i]>='0'&&word[i]<='9')){

i++;

}

if (word[i]==0){

return T\_ID;

}

}

return T\_ERR;

}

int word\_control(Link \*link,char \*text)

{

char \*s=text,word[1000];

int i=0,len,type,jump\_len;

int line\_num=1;

int err\_sum=0;

if(link==NULL){

printf("??治??\n\n");

return -1;

}

while (\*s){

len=get\_word(word,s);

if (word[0]==0&&\*(s+len)==0){

break;

}

type=get\_type(word);

if (type==T\_ERR){

//get\_line(word,s);

setcolor(0x0c);

printf("(ERR)In line %d :UNKNOW WORD:(%s)\n",line\_num,word);

setcolor(0x07);

s+=len;

err\_sum++;

line\_num+=get\_line\_change(s,jump\_len+len);

}else {

jump\_len=0;

if (type==T\_N\_LINE||type==T\_N\_BEGIN){

jump\_len=to\_n\_end(s+len,type);

}

line\_num+=get\_line\_change(s,jump\_len+len);

s+=jump\_len+len;

if(type==T\_N\_BEGIN||type==T\_N\_END||type==T\_N\_LINE){

continue;

}

//printf("t:%d;\tli:%d;\tnum%d;\tword:%s\n",type,line\_num,i++,word);

Dual \*p=(Dual \*)malloc((int )sizeof(Dual));

if(p==NULL){

printf("??治??\n\n");

return -1;

}

p->dual\_type=type;

p->line=line\_num;

if (type==T\_INT){

p->lexme.lexme\_num=char\_to\_num(word);

//printf("num::%d\n",char\_to\_num(word));

}else {

strcpy(p->lexme.lexme\_text,word);

}

link\_add(link,p);

}

}

return err\_sum;

}

int get\_word(char \*word,char \*s)

{

int len=0,i=0;

//跳过可忽略字符

while(s[i]==' '||s[i]=='\t'||s[i]=='\r'||s[i]=='\n'){

i++;

}

if ((s[i]>='0'&&s[i]<='9')||(s[i]>='a'&&s[i]<='z')//数字/标识符

||(s[i]>='A'&&s[i]<='Z')){//num/w

while ((s[i]>='0'&&s[i]<='9')||(s[i]>='a'&&s[i]<='z')

||(s[i]>='A'&&s[i]<='Z')){

word[len++]=s[i++];

}

}else if (s[i]=='='){ //=

word[len++]=s[i++];

if (s[i]=='='){ //==

word[len++]=s[i++];

}

}else if (s[i]==':'){ //:

word[len++]=s[i++];

if (s[i]=='='){ //:=

word[len++]=s[i++];

}

}else if (s[i]=='<'){ //<

word[len++]=s[i++];

if (s[i]=='='||s[i]=='>'){ //<= <>

word[len++]=s[i++];

}

}else if (s[i]=='>'){ //>

word[len++]=s[i++];

if (s[i]=='='){ //>=

word[len++]=s[i++];

}

}else if (s[i]=='+'){ //+

word[len++]=s[i++];

}else if (s[i]==';'||s[i]==','||s[i]=='('||s[i]==')'){ //; , ( )

word[len++]=s[i++];

}else if (s[i]=='-'){ //-

word[len++]=s[i++];

}else if (s[i]=='\*'){ //\*

word[len++]=s[i++];

if (s[i]=='/'){ //\*/

word[len++]=s[i++];

}

}else if (s[i]=='/'){ ///

word[len++]=s[i++];

if (s[i]=='/'||s[i]=='\*'){ //// /\*

word[len++]=s[i++];

}

}else{//其他

while (s[i]&&s[i]!=' '&&s[i]!=';'&&s[i]!='\t'&&s[i]!='\r'&&s[i]!='\n'&&s[i]!=':'&&s[i]!='='

&&s[i]!='<'&&s[i]!='>'&&s[i]!='/'&&s[i]!=','){

word[len++]=s[i++];

}

}

word[len]=0;

return i;

}

int to\_n\_end(char \*s,int type)

{

int i=0;

if (type==T\_N\_LINE){

while (s[i]!=0&&s[i]!='\n')i++;

//printf("/////\ns:(%s)\n//////\n",s+i+1);

return i;

}else if (type==T\_N\_BEGIN){

while (s[i]!=0&&(s[i]!='\*'&&s[i+1]!='/'))i++;

if (s[i])return i;

}

return 0;

}

int get\_line\_change(char \*s,int len)

{

int i=0,j=0;

while (s[j]&&j<len){

if (s[j]=='\n')i++;

j++;

}

return i;

}

void get\_line(char \*line,char \*s)

{

while (\*s&&\*s!='\n'){

\*line=\*s;

line++;

s++;

}

\*line=0;

}

int char\_to\_num(char \*s)

{

int result=0;

while (\*s>='0'&&\*s<='9'){

result\*=10;

result+=\*s-'0';

s++;

}

return result;

}

## 2 语法分析

#include "gram.h"

#include "link.h"

#include "word.h"

#include "stdio.h"

#include "show.h"

int gram\_control(Link \*link,int level);

int test\_ju(Link \*ju);

int biaodashi\_hefa(Link \*link);

int gram\_control(Link \*link,int level)

{

Dual \*get,\*get1,\*get2;

Link \*ju;

int i,j,k;

int last\_type=T\_ERR;

int have\_begin=0;

int iner=0;

int err\_sum=0,temp,show\_line\_sum=0;

Link \*all\_ju=link\_create();

get1=(Dual \*)link\_get\_i(link,0);

get2=(Dual \*)link\_get\_i(link,link->len-1);

if (get1==NULL||get2==NULL){

printf("缺少关键元素！\n");

return 1;

}

//判断语句表开头,跳过二元式最外层的begin 和 end

if (get1->dual\_type==T\_BEGIN&&get2->dual\_type==T\_END){

iner=1;

i=1;

}else {

i=0;

}

have\_begin=0;

for (;i<link->len;){

ju=link\_create();

have\_begin=0;

do{

get=(Dual \*)link\_get\_i(link,i);

if (get==NULL){

printf("缺少关键元素！\n");

return 1;

}

if(get->dual\_type==T\_BEGIN){

have\_begin++;

}else if(get->dual\_type==T\_END){

if(have\_begin==0){

printf("缺少开始符！（在第%d行）\n",get->line);

return -1;

}else {

have\_begin--;

}

}

link\_add(ju,get);

if(iner==1&&i==link->len-2){

i+=2;

break;

}

i++;

}while (((get->dual\_type!=T\_ELSE&&get->dual\_type!=T\_DO&&get->dual\_type!=T\_THEN&&get->dual\_type!=T\_CUT&&get->dual\_type!=T\_END)||(have\_begin))&&i<link->len);

link\_add(all\_ju,ju);

}

if (have\_begin!=0||(have\_begin==0&&(get->dual\_type!=T\_CUT&&get->dual\_type!=T\_END))){

printf("缺少结束符！（在第%d行）%d\n",get->line,have\_begin);

return -1;

}

//遍历所有语句

for(i=0;i<all\_ju->len;i++){

ju=(Link \*)link\_get\_i(all\_ju,i);

get=(Dual \*)link\_get\_i(ju,0);

if ((get->dual\_type==T\_IF||get->dual\_type==T\_WHILE)&&i==all\_ju->len-1){

get=(Dual \*)link\_get\_i(ju,0);

setcolor(0x0c);

printf("(err)");

setcolor(0x07);

show\_line(get->line);

printf("\n不完整的语句体！\n");

err\_sum++;

}

if (get->dual\_type==T\_BEGIN){

for (k=0;k<level-(!!(show\_line\_sum==0));k++)printf("\t");if(level&&show\_line\_sum++==0)printf("+-------");if(level)printf("|");

show\_word(get);

if ((temp=gram\_control(ju,level+1))!=0){//递归遍历语句集

get=(Dual \*)link\_get\_i(ju,0);

err\_sum+=temp;

}

get=(Dual \*)link\_get\_i(ju,ju->len-1);

for (k=0;k<level-(!!(show\_line\_sum==0));k++)printf("\t");if(level&&show\_line\_sum++==0)printf("+-------");if(level)printf("|");

show\_word(get);

}else {

for (j=0;j<ju->len;j++){

get=(Dual \*)link\_get\_i(ju,j);

for (k=0;k<level-(!!(show\_line\_sum==0));k++)printf("\t");if(level&&show\_line\_sum++==0)printf("+-------");if(level)printf("|");

show\_word(get);

}

if (test\_ju(ju)!=0){

get=(Dual \*)link\_get\_i(ju,0);

setcolor(0x0c);

printf("(err)");

setcolor(0x07);

show\_line(get->line);

printf("\n语法错误！\n");

//return -1;

err\_sum++;

}

}

}

return err\_sum;

}

//判断表达式是否合法

int biaodashi\_hefa(Link \*link)

{

int i=0;

int t,last\_t;

int kuo\_sum=0;

Dual \*p;

last\_t=get\_type\_in\_link(link,0);

t=last\_t;

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_LEFT){

printf("错误的表达式开始！在第%d词！\n",i);

return -1;

}

for (i=1;i<link->len;i++){

p=(Dual \*)link\_get\_i(link,i);

t=p->dual\_type;

switch(last\_t){

//操作数

case T\_ID:

case T\_INT:

if (t!=T\_ADD&&t!=T\_SUB&&t!=T\_MUL&&t!=T\_DIV&&t!=T\_CUT\_RIGHT&&

t!=T\_LT&&t!=T\_LE&&t!=T\_NE&&t!=T\_GT&&t!=T\_GE&&t!=T\_CMP0){

printf("错误的运算符！在第%d词！\n",i);

return -2;

}

break;

//左括号

case T\_CUT\_LEFT:

kuo\_sum++;

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_LEFT){

printf("错误的操作数！在第%d词！\n",i);

return -3;

}

break;

//右括号

case T\_CUT\_RIGHT:

if (kuo\_sum==0){

printf("错误的右括号！在第%d词！\n",i);

return -12;

}else {

kuo\_sum--;

}

if (t!=T\_ADD&&t!=T\_SUB&&t!=T\_MUL&&t!=T\_DIV&&t!=T\_CUT\_RIGHT&&

t!=T\_LT&&t!=T\_LE&&t!=T\_NE&&t!=T\_GT&&t!=T\_GE&&t!=T\_CMP0){

printf("错误的运算符！在第%d词！\n",i);

return -4;

}

break;

//运算符

case T\_ADD:

case T\_SUB:

case T\_MUL:

case T\_DIV:

case T\_LT:

case T\_LE:

case T\_NE:

case T\_GT:

case T\_GE:

case T\_CMP0:

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_LEFT){

printf("错误的操作数！在第%d词！\n",i);

return -5;

}

break;

default:

return -100;

break;

}

last\_t=t;

}

//错误的表达式结尾

if (t!=T\_ID&&t!=T\_INT&&t!=T\_CUT\_RIGHT){

printf("错误的表达式结尾！在第%d词！\n",i);

return -6;

}

//最后右括号回复

if (t==T\_CUT\_RIGHT)kuo\_sum--;

//判断括号是否匹配

if (kuo\_sum!=0){

printf("错误的括号匹配！在第%d词！\n",i);

return -7;

}

//表达式合法

return 0;

}

//判断语句是否合法

int test\_ju(Link \*ju)

{

Dual \*get;

Link \*link;

int i=0,t;

get=(Dual \*)link\_get\_i(ju,0);

if(have\_type(ju,T\_EQ)){//:= 赋值语句

if(get\_type\_in\_link(ju,0)!=T\_ID||get\_type\_in\_link(ju,1)!=T\_EQ){

printf("不可修改的左值！在%d行！\n",get->line);

return -1;

}else {

link=link\_create();

i=2;

while (get\_type\_in\_link(ju,i)!=T\_CUT&&i<ju->len){

get=(Dual \*)link\_get\_i(ju,i);

link\_add(link,get);

i++;

}

if (i==ju->len){

printf("未结束的表达式！在%d行！\n",get->line);

link\_destroy(&link);

return -2;

}else{

if ((t=biaodashi\_hefa(link))!=0){

//show\_link(ju);

printf("不合法的表达式！在%d行！错误代码：%d;\n",get->line,t);

link\_destroy(&link);

return -3;

}

}

link\_destroy(&link);

}

}else if(have\_type(ju,T\_IF)){//IF 判断语句

if(have\_type(ju,T\_THEN)==0){//如果语句中不存在THEN

printf("缺少：THEN！在%d行！\n",get->line);

return -2;

}

//获取表达式链表

link=link\_create();

i=1;

while (get\_type\_in\_link(ju,i)!=T\_THEN&&i<ju->len){

get=(Dual \*)link\_get\_i(ju,i);

link\_add(link,get);

i++;

}

//判断表达式是否合法

if (i==ju->len){//

printf("未结束的表达式！在%d行！\n",get->line);

link\_destroy(&link);

return -2;

}else{

if ((t=biaodashi\_hefa(link))!=0){

//show\_link(ju);

printf("不合法的表达式！在%d行！错误代码：%d;\n",get->line,t);

link\_destroy(&link);

return -3;

}

}

link\_destroy(&link);

}else if(have\_type(ju,T\_WHILE)){//WHILE 语句

if(have\_type(ju,T\_DO)==0){//如果语句不存在DO

printf("缺少：DO！在%d行！\n",get->line);

return -3;

}

//获取表达式链表

link=link\_create();

i=1;

while (get\_type\_in\_link(ju,i)!=T\_DO&&i<ju->len){

get=(Dual \*)link\_get\_i(ju,i);

link\_add(link,get);

i++;

}

//判断表达式是否合法

if (i==ju->len){//

printf("未结束的表达式！在%d行！\n",get->line);

link\_destroy(&link);

return -2;

}else{

if ((t=biaodashi\_hefa(link))!=0){

//show\_link(ju);

printf("不合法的表达式！在%d行！错误代码：%d;\n",get->line,t);

link\_destroy(&link);

return -3;

}

}

link\_destroy(&link);

}else if(have\_type(ju,T\_DO)){

if(have\_type(ju,T\_WHILE)==0){

printf("缺少：WHILE！在%d行！\n",get->line);

return -4;

}

}else if(have\_type(ju,T\_THEN)){

if(have\_type(ju,T\_IF)==0){

printf("缺少：IF！在%d行！\n",get->line);

return -5;

}

}else if(have\_type(ju,T\_ELSE)){

}else if(have\_type(ju,T\_INPUT)){

if(get\_type\_in\_link(ju,1)!=T\_ID){

printf("错误的输入标识符！在%d行！\n",get->line);

return -10;

}

}else if (have\_type(ju,T\_OUTPUT)){

if(get\_type\_in\_link(ju,1)!=T\_ID&&get\_type\_in\_link(ju,1)!=T\_INT){

printf("错误的输出标识符！在%d行！\n",get->line);

return -12;

}

}else {

printf("错误的语句！在%d行！\n",get->line);

return -7;

}

return 0;

}

## 3 语义分析

#include <malloc.h>

#include <string.h>

#include "run.h"

#include "link.h"

#include "word.h"

#include "stdio.h"

#include "show.h"

#include "id\_table.h"

Asm \*asm\_word(int type,int op,int op\_ed,int num);

Link \*ju\_to\_asm(Link \*ju,Link \*id\_table);

Link \*turn\_to\_asm(Link \*link,Link \*asm\_link,Link \*id\_table)

{

Dual \*get,\*get1,\*get2;

Link \*ju;

int i,j,k;

int last\_type=T\_ERR;

int have\_begin=0;

int iner=0;

int err\_sum=0,show\_line\_sum=0,i\_temp;

Link \*ju\_link,\*temp,\*temp1;

Asm \*asm\_get;

Link \*all\_ju=link\_create();

if (asm\_link==NULL)asm\_link=link\_create();

if (id\_table==NULL)id\_table=link\_create();

get1=(Dual \*)link\_get\_i(link,0);

get2=(Dual \*)link\_get\_i(link,link->len-1);

//判断语句表开头,跳过二元式最外层的begin 和 end

if (get1->dual\_type==T\_BEGIN&&get2->dual\_type==T\_END){

iner=1;

i=1;

}else {

i=0;

}

have\_begin=0;

for (;i<link->len;){

ju=link\_create();

have\_begin=0;

do{

get=(Dual \*)link\_get\_i(link,i);

if(get->dual\_type==T\_BEGIN){

have\_begin++;

}else if(get->dual\_type==T\_END){

have\_begin--;

}

link\_add(ju,get);

if(iner==1&&i==link->len-2){

i+=2;

break;

}

i++;

}while (((get->dual\_type!=T\_ELSE&&get->dual\_type!=T\_DO&&get->dual\_type!=T\_THEN&&get->dual\_type!=T\_CUT&&get->dual\_type!=T\_END)||(have\_begin))&&i<link->len);

link\_add(all\_ju,ju);

}

//遍历所有语句

for(i=0;i<all\_ju->len;i++){

ju=(Link \*)link\_get\_i(all\_ju,i);

get=(Dual \*)link\_get\_i(ju,0);

if (get->dual\_type==T\_BEGIN){

turn\_to\_asm(ju,asm\_link,id\_table);//递归遍历语句集

}else if (get->dual\_type==T\_IF){

temp=ju\_to\_asm(ju,id\_table);

link\_add\_link(asm\_link,temp);

temp1=(Link \*)link\_get\_i(all\_ju,i+1);

turn\_to\_asm(temp1,asm\_link,id\_table);

asm\_get=(Asm \*)link\_get\_i(temp,temp->len-1);

asm\_get->num=asm\_link->len;

i++;

}else if (get->dual\_type==T\_WHILE){

i\_temp=asm\_link->len;

temp=ju\_to\_asm(ju,id\_table);

link\_add\_link(asm\_link,temp);

temp1=(Link \*)link\_get\_i(all\_ju,i+1);

turn\_to\_asm(temp1,asm\_link,id\_table);

asm\_get=(Asm \*)link\_get\_i(temp,temp->len-1);

link\_add(asm\_link,asm\_word(T\_ASM\_JMP,0,0,i\_temp));//mov R1,(R2)

asm\_get->num=asm\_link->len;

link\_destroy(&temp);

i++;

}else {

ju\_link=ju\_to\_asm(ju,id\_table);

link\_add\_link(asm\_link,ju\_link);

link\_destroy(&ju\_link);

}

}

return asm\_link;

}

Asm \*asm\_word(int type,int op,int op\_ed,int num)

{

Asm \*asm\_word;

asm\_word=(Asm \*)malloc((int )sizeof(Asm));

asm\_word->type=type;

asm\_word->op=op;

asm\_word->op\_ed=op\_ed;

asm\_word->num=num;

return asm\_word;

}

void biaodashi\_to\_asm(Link \*biao,Link \*asm\_link,Link \*id\_table)

{

Link \*biao\_min;

LinkNode \*p;

int kuo\_sum,addr;

Dual \*get=NULL,\*last\_get=NULL;

p=biao->head;

get=(Dual \*)(p->data);

if (get->dual\_type==T\_ID){

addr=get\_id\_addr(id\_table,get->lexme.lexme\_text);

if(addr<0){

printf("使用了未定义的标识符！在%d行！\n",get->line);

show\_word(get);

return ;

}

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R2,0,addr));//mov R2,addr

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_MEM,REG\_R1,REG\_R2,0));//mov R1,(R2)

p=p->next;

}else if (get->dual\_type==T\_INT){

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R1,0,get->lexme.lexme\_num));//mov R1,num

p=p->next;

}

while (p){

get=(Dual \*)(p->data);

if (get->dual\_type==T\_CUT\_LEFT){//子表达式递归

kuo\_sum=1;

biao\_min=link\_create();

while (1){

p=p->next;

get=(Dual \*)(p->data);

if(get->dual\_type==T\_CUT\_LEFT){

kuo\_sum++;

}else if(get->dual\_type==T\_CUT\_RIGHT){

kuo\_sum--;

}

if(kuo\_sum==0){

break;

}

link\_add(biao\_min,get);

}

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_REG,REG\_R5,REG\_R1,addr));//mov R5,R1

biaodashi\_to\_asm(biao\_min,asm\_link,id\_table);

if(last\_get){

switch(last\_get->dual\_type){

case T\_ADD:

link\_add(asm\_link,asm\_word(T\_ASM\_ADD,REG\_R1,REG\_R5,0));//ADD R1,R5

break;

case T\_SUB:

link\_add(asm\_link,asm\_word(T\_ASM\_SUB,REG\_R5,REG\_R1,0));//SUB R1,R5

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_REG,REG\_R1,REG\_R5,addr));//MOV R1,R5

break;

}

}

link\_destroy(&biao\_min);

}else {

//取出待操作数放于R2

if (get->dual\_type==T\_ID){

addr=get\_id\_addr(id\_table,get->lexme.lexme\_text);

if(addr<0){

printf("使用了未定义的标识符！在%d行！\n",get->line);

show\_word(get);

return ;

}

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R3,0,addr));//mov R3,addr

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_MEM,REG\_R2,REG\_R3,0));//mov R2,(R3)

}else if (get->dual\_type==T\_INT){

link\_add(asm\_link,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R2,0,get->lexme.lexme\_num));//mov R2,num

}

//添加asm

if(last\_get&&last\_get->dual\_type==T\_ADD){

link\_add(asm\_link,asm\_word(T\_ASM\_ADD,REG\_R1,REG\_R2,0));//ADD R1,R2

}else if(last\_get&&last\_get->dual\_type==T\_SUB){

link\_add(asm\_link,asm\_word(T\_ASM\_SUB,REG\_R1,REG\_R2,0));//SUB R1,R2

}else if (last\_get&&(last\_get->dual\_type==T\_LT||last\_get->dual\_type==T\_GT||last\_get->dual\_type==T\_NE||last\_get->dual\_type==T\_LE||last\_get->dual\_type==T\_GE||last\_get->dual\_type==T\_CMP0)){

link\_add(asm\_link,asm\_word(T\_ASM\_CMP,REG\_R1,REG\_R2,0));//CMP R1,R5

}

}

last\_get=get;

p=p->next;

}

}

Link \*ju\_to\_asm(Link \*ju,Link \*id\_table)

{

Link \*ju\_asm;

Dual \*get;

Link \*link;//表达式

int i=0,t,addr;

ju\_asm=link\_create();

get=(Dual \*)link\_get\_i(ju,0);

if(have\_type(ju,T\_EQ)){//:= 赋值语句

//获取表达式

link=link\_create();

i=2;

while (get\_type\_in\_link(ju,i)!=T\_CUT&&i<ju->len){

get=(Dual \*)link\_get\_i(ju,i);

link\_add(link,get);

i++;

}

//产生四元式

get=(Dual \*)link\_get\_i(ju,0);

biaodashi\_to\_asm(link,ju\_asm,id\_table);

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R2,0,add\_id\_addr(id\_table,get->lexme.lexme\_text)));//MOV R2,addr

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_MEM\_REG,REG\_R2,REG\_R1,0));//mov (R2),R1

link\_destroy(&link);

}else if (have\_type(ju,T\_INPUT)){//输入语句

get=(Dual \*)link\_get\_i(ju,1);

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R2,0,add\_id\_addr(id\_table,get->lexme.lexme\_text)));//MOV R2,addr

link\_add(ju\_asm,asm\_word(T\_ASM\_IN,0,REG\_R1,0));//IN R1

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_MEM\_REG,REG\_R2,REG\_R1,0));//MOV (R2),R1

} else if (have\_type(ju,T\_OUTPUT)){//输出语句

get=(Dual \*)link\_get\_i(ju,1);

if (get->dual\_type==T\_ID){

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R2,0,add\_id\_addr(id\_table,get->lexme.lexme\_text)));//MOV R2,addr

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_REG\_MEM,REG\_R1,REG\_R2,0));//MOV R1,(R2)

}else {

link\_add(ju\_asm,asm\_word(T\_ASM\_MOV\_REG\_NUM,REG\_R1,0,get->lexme.lexme\_num));//MOV R1,(R2)

}

link\_add(ju\_asm,asm\_word(T\_ASM\_OUT,0,REG\_R1,0));//OUT R1

}else if (have\_type(ju,T\_WHILE)||have\_type(ju,T\_IF)){//IF/WHILE语句

//获取表达式

link=link\_create();

i=1;

while (get\_type\_in\_link(ju,i)!=T\_DO&&i<ju->len){

get=(Dual \*)link\_get\_i(ju,i);

link\_add(link,get);

i++;

}

//产生四元式

get=(Dual \*)link\_get\_i(ju,0);

biaodashi\_to\_asm(link,ju\_asm,id\_table);

if (have\_type(ju,T\_LT)){

link\_add(ju\_asm,asm\_word(T\_ASM\_NB,0,0,0));//JB \* //小于->不小于

}else if (have\_type(ju,T\_GT)){

link\_add(ju\_asm,asm\_word(T\_ASM\_BE,0,0,0));//NB \* //大于->不大于(小于等于)

}else if (have\_type(ju,T\_NE)){

link\_add(ju\_asm,asm\_word(T\_ASM\_E,0,0,0));//NB \* //不等于->等于

}else if (have\_type(ju,T\_LE)){

link\_add(ju\_asm,asm\_word(T\_ASM\_JG,0,0,0));//NB \* //小于等于->大于

}else if (have\_type(ju,T\_GE)){

link\_add(ju\_asm,asm\_word(T\_ASM\_JB,0,0,0));//NB \* //大于等于->小于

}else if (have\_type(ju,T\_CMP0)){

link\_add(ju\_asm,asm\_word(T\_ASM\_NE,0,0,0));//NB \* //等于->不等于

}

link\_destroy(&link);

}

//show\_asm(ju\_asm);

return ju\_asm;

}

## 4 通用链表

#include "link.h"

#include <malloc.h>

#include <stdio.h>

#define LINK\_DEBUG

Link \*link\_back(Link \*link)

{

Link \*new\_link;

new\_link=link\_create();

link\_add\_link(new\_link,link);

return new\_link;

}

void link\_add\_link(Link \*toLink,Link \*sorLink)

{

LinkNode \*p;

if (toLink==NULL||sorLink==NULL){

#ifdef LINK\_DEBUG

printf("(link debug)NULL pointer on (link\_add\_link)\n");

#endif

return ;

}

p=sorLink->head;

while (p){

link\_add(toLink,p->data);

p=p->next;

}

return ;

}

Link \*link\_create()

{

Link \*p;

p=(Link \*)malloc((int )sizeof(Link));

if (p==NULL){

#ifdef LINK\_DEBUG

printf("(link debug)NULL pointer on (link\_create)\n");

#endif

return NULL;

}

p->len=0;

p->head=NULL;

p->tail=NULL;

return p;

}

void link\_add(Link \*link,void \*p)

{

LinkNode \*np=NULL;

if(p==NULL||link==NULL){

#ifdef LINK\_DEBUG

printf("(link debug)NULL pointer on (link\_add)\n");

#endif

return ;

}

np=(LinkNode \*)malloc((int )sizeof(LinkNode));

if(np==NULL)return ;

np->data=p;

np->next=NULL;

np->pro=NULL;

if(link->len==0){

link->head=np;

link->tail=np;

}else {

link->tail->next=np;

np->pro=link->tail;

link->tail=np;

}

link->len++;

return ;

}

void \*link\_get\_i(Link \*link,int i)

{

LinkNode \*pn;

if(link==NULL||link->head==NULL||i<0||i>=link->len){

#ifdef LINK\_DEBUG

printf("(link debug)NULL pointer on (link\_get\_i) link=%x link\_len=%d i=%d\n",link,(link==NULL)?-1:link->len,i);

#endif

return NULL;

}

pn=link->head;

while (i-->0){

pn=pn->next;

}

return pn->data;

}

void link\_del\_i(Link \*link,int i)

{

LinkNode \*pn;

if(link==NULL||link->head==NULL||i<0||i>=link->len){

#ifdef LINK\_DEBUG

printf("(link debug)NULL pointer on (link\_del\_i)\n");

#endif

return ;

}

pn=link->head;

while (i-->0){

pn=pn->next;

}

if (pn==link->head){

link->head=link->head->next;

}

if(pn==link->tail){

link->tail=link->tail->pro;

}

if(pn->next!=NULL){

pn->next->pro=pn->pro;

}

if(pn->pro!=NULL){

pn->pro->next=pn->next;

}

link->len--;

free(pn);

}

void link\_clere(Link \*link)

{

int i;

LinkNode \*np;

if(link==NULL){

#ifdef LINK\_DEBUG

printf("(link debug)NULL pointer on (link\_clere)\n");

#endif

return ;

}

for (i=0;i<link->len;i++){

np=link->head;

link->head=link->head->next;

free(np);

}

link->len=0;

link->head=NULL;

link->tail=NULL;

}

void link\_destroy(Link \*\*link)

{

int i;

LinkNode \*np;

if(\*link==NULL){

#ifdef LINK\_DEBUG

printf("(link debug)NULL pointer on (link\_destroy)\n");

#endif

return ;

}

for (i=0;i<(\*link)->len;i++){

np=(\*link)->head;

(\*link)->head=(\*link)->head->next;

free(np);

}

free(\*link);

\*link=NULL;

}