

**实验报告**

**题目： C语言编译器设计与实现**

**课程名称： 编译原理**

**专业班级： CS1607**

**学 号： U201614702**

**姓 名： 樊俊超**

**指导教师： 徐丽萍 刘海坤**

**报告日期： 2019/6/15**

**计算机科学与技术学院**

**目录**

[1选题背景 1](#_Toc11578167)

[2系统关键定义 2](#_Toc11578168)

[2.1单词文法描述 2](#_Toc11578169)

[2.2语句文法描述 3](#_Toc11578170)

[2.3 符号表结构描述 4](#_Toc11578171)

[2.4 错误类型码描述 5](#_Toc11578172)

[3系统设计与实现 6](#_Toc11578173)

[3.1编译程序符号表结构 6](#_Toc11578174)

[3.2编译程序报错功能 7](#_Toc11578175)

[3.3词法语法分析器（实验一） 7](#_Toc11578176)

[3.4语义分析（实验二） 8](#_Toc11578177)

[4系统测试与评价 10](#_Toc11578178)

[4.1 测试用例 10](#_Toc11578179)

[4.2 正确性测试 11](#_Toc11578180)

[4.3 报错功能测试 15](#_Toc11578181)

[4.4 系统的优点 20](#_Toc11578182)

[4.5 系统的缺点 20](#_Toc11578183)

[5实验小结或体会 21](#_Toc11578184)

[**参考文献** 22](#_Toc11578185)

[**附件：源代码** 23](#_Toc11578186)

[**附录一 GNU Flex 工具说明** 76](#_Toc11578187)

[**附录二 GNU Bison 工具说明** 77](#_Toc11578188)

[**附录三 抽象语法树（AST）** 80](#_Toc11578189)

[**附录四 作用域与符号表操作** 83](#_Toc11578190)

# 1选题背景

本次课程设计是构造一个高级语言的子集的编译器，目标代码可以是汇编语言也可以是其他形式的机器语言。按照任务书，实现的方案可以有很多种选择。

可以根据自己对编程语言的定义选择实现语言的特定功能。建议大家选用decaf语言。

编译器的语法和词法分析采用课程的课堂实验的结果，重点在语义分析、符号表结构设计、中间代码、目标代码存储结构设计、代码优化等阶段的实现。

课设的任务主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高自己对系统软件编写的兴趣。

实验内容：

* 源语言定义：或采用教材中Decaf语言，或采用C语言(或C++语言或C#语言

JAVA语言)部分关键语法规则。源语言要求至少包含的语言成分如下：

数据类型至少包括char类型、int类型和float类型

基本运算至少包括算术运算、比较运算、逻辑运算、自增自减运算和复合赋值运算

控制语句至少包括if语句和while语句

其它选项：数组、结构，for循环等等

* 实验内容：完整可运行的自定义语言编译器

# 2系统关键定义

## 2.1单词文法描述

词法分析程序的主要任务是将输入文件中的字符流组织成为词法单元流，在某些字符不符合程序设计语言词法规范时要有能力报告相应的错误，本次实验采用词法生成器自动化生成工具GNU Flex（见附录一），具体要求以正则表达式（正规式）的形式给出词法规则，遵循上述技术线路，Flex自动生成给定的词法规则的词法分析程序。于是，设计能准确识别各类单词的正则表达式就是关键。

高级语言的词法分析器，需要识别的单词有五类：关键字（保留字）、运算符、界符、常量和标识符。依据mini-c语言的定义，在此给出各单词的种类码和相应符号说明：

|  |  |  |
| --- | --- | --- |
| 单词 | 种类码 | 符号说明 |
| INT | 258 | 整型常量 |
| FLOAT | 263 | 浮点型常量 |
| CHAR | 264 | 字符型常量 |
| ID | 259 | 标识符 |
| STRUCT | 262 | 结构体 |
| ASSIGNOP | 277 | = |
| RELOP | 260 | > | >= | < | <= | == | != |
| PLUS | 273 | + |
| MINUS | 274 | - |
| STAR | 275 | \* |
| DIV | 276 | / |
| INCASS | 287 | += |
| DECASS | 288 | -= |
| INC | 285 | ++ |
| DEC | 286 | -- |
| AND | 278 | && |
| OR | 279 | || |
| NOT | 280 | ！ |
| TYPE | 261 | int|float |
| RETURN | 284 | return |
| IF | 281 | if |
| ELSE | 282 | else |
| WHILE | 283 | while |
| SEMI | 271 | ; |
| COMMA | 272 | ， |
| LP | 265 | ( |
| RP | 266 | ) |
| LC | 269 | { |
| RC | 270 | } |
| LB | 267 | [ |
| RB | 268 | ] |
| UMINUS | 289 | - |

(行级注释（用//进行单行注释）和块级注释（用/\*和\*/进行块级注释）用正则表达式完成，分析时忽略该部分)大部分单词描述由实验给出，根据实验要求，添加char类型和struct结构体，添加复合运算符+=和-=，自增运算符++，自减运算符--，添加while循环，if语句，添加中括号符号‘[‘和‘]’进而实现数组。

## 2.2语句文法描述

语法分析程序的主要任务就是读入词法单元流、判断输入程序是否匹配程序设计语言的语法规范，并在匹配规范的情况下构造出该结构的分析树或语法树。

本次实验生成抽象语法树，见附录二。本次实验采用LALR（1）（自左向右向前查看一个记号），但它不能处理有二义性的语法，比如相同的输入可以匹配多棵语法分析树的情况，他也不能处理，需要向前查看多个记号才能确定是否匹配规则的语法

但是，Bison对于有二义性的文法，有自己一套隐式的冲突解决方案（一旦出现归约/归约冲突，Bison总会选择靠前的产生式；一旦出现移入/归约冲突，Bison总会选择移入）从而产生相应的语法程序。由于直接转化来的生成式存在大量的移进-规约冲突或规约规约冲突，需要通过一定的方法来消除冲突。对二义性与冲突处理，则是通过显示规定优先级和结合性来解决。

经过排查移进-规约冲突和规约-规约冲突的来源，不难发现大部分的冲突来自于运算符，造成的原因便是分析器不知道运算符的优先级和结合性。例如，对于算数表达式“1+2-3”，分析器并不知道是先算“1+2”还是“2-3”。当然，如果告诉分析器“+”和“-”都是左结合，那么分析器自然知道是要先计算“1+2”而不是是“2-3”。除了算数表达式外，IF-ELSE 语句的语法也会带来移进-规约冲突。

为了解决二义性问题，需要显示的指定运算符的优先级与结合性。在Bison源代码中，需要采用“%left”，“%right”，“%nonassoc”,分别表示左结合，右结合，没有结合性。%nonassoc一般与%prec结合使用表示该操作有同样的优先级,对终结符的结合性进行规定。

## 2.3 符号表结构描述

在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括： 上述名字的种类、 具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

符号表上的操作包括填表和查表两种。当分析到程序中的说明或定义语句时, 应将说明或定义的名字, 以及与之有关的特性信息填入符号表中，这便是填表操作。查表操作被使用得更为广泛，需要使用查表操作的情况有：填表前查表，检查在程序的同一作用域内名字 是否重复定义；检查名字的种类是否与说明一致；对于那些类型要求更强的语言，要检查表达式中各变量的类型是否一致；生成目标指令时，要取得所需要的地址或 者寄存器编号，等等。符号表的组织方式也有多种多样，你可以将程序中出现的所有符号组织成一张表，也可以将不同种类的符号组织成不同的表（例如，所有变量 名组织成一张表，所有函数名组织成一张表，所有临时变量组织成一张表，所有结构体定义组织成一张表等等）；你可以针对每一个语句块、每一个结构体都新建一 张表，也可以将所有语句块中出现的符号全部插入到同一张表；你的表可以仅支持插入不支持删除（此时如果要实现作用域的话需要将符号表组织成层次结构），也 可以组织一张既可以插入又可以删除的、支持动态更新的表。

由于实验只有很少的符号数量本，所以本次实验采用线性链表，符号表里所有的符号都用一条链表串起来,插入一个新的符号只需将这个符号放在链表的表头,时间效率为O(1)；删除和查找时间效率都是O(n)。链表的最大问题就在于它的查找和删除效率太低，一旦符号表中的符号数量增大之后，查表操作将变得十分耗时。不过，使用链表的好处也显而易见：它的结构简单、编程复杂度极低，可以被快速地、无错地实现。

为了支持变量的作用域，本次实验采用栈方法，即维护一个符号表栈。假设当前函数f有一个符号表，表内有变量a的定义，当编译器发现函数中出现复合语句即由“{”和“}”包含的语句块时，它会将f的符号表压栈，新建一个符号表，这个符号表里只有变量a的定义。当语句块中出现任何表达式使用到某个变量时，编译器会优先查找当前符号表，如果找到，就是用当前找到的变量，如果找不到则顺着符号表栈向下逐个符号表查找。当编译器离开某个语句块时，会销毁当前符号表，从栈中弹出一个符号表作为当前的符号表,作用表与符号表操作见附录四。

## 2.4 错误类型码描述

语义分析这部分完成的是静态语义分析，主要包括：

（1）控制流检查。控制流语句必须使得程序跳转到合法的地方。例如一个跳转语句会使控制转移到一个由标号指明的后续语句。如果标号没有对应到语句，那么久就出现一个语义错误。再者，break、continue语句必须出现在循环语句当中。

（2）唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。

（3）名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。

（4）类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

错误类型：

1. 变量或重复定义
2. 数组变量重复定义
3. 函数未定义或重复定义
4. 函数调用参数数量不对
5. 函数类型不匹配
6. 函数参数类型不匹配
7. 赋值错误

实验中还有很多错误需要细分到上述错误中的某些类型中，对于发现的语义错误会被准确定位，并针对错误类型给出错误提示，从而有助于快速修正。

# 3系统设计与实现

## 3.1编译程序符号表结构

struct symbol

{ //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥

char name[33]; //变量或函数名

int level; //层号，外部变量名或函数名层号为0，形参名为1，每到1个复合语句层号加1，退出减1

int type; //变量类型或函数返回值类型

int paramnum; //形式参数个数

char alias[10]; //别名，为解决嵌套层次使用，使得每一个数据名称唯一

char flag; //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T'

char offset; //外部变量和局部变量在其静态数据区或活动记录中偏移量

//或函数活动记录大小，目标代码生成时使用

};

//符号表，是一个顺序栈，index初值为0

struct symboltable

{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

/\*\*

\* 当前作用域的符号在符号表的起始位置序号,这是一个栈结构

\* 每到达一个复合语句，将符号表的index值进栈，离开复合语句时，取其退栈值修改符号表的index值，完成删除该复合语句中的所有变量和临时变量

\*/

struct symbol\_scope\_begin

{

int TX[30];

int top;

} symbol\_scope\_TX;

## 3.2编译程序报错功能

实验时,主要都在semantic.c文件中，在函数Exp和函数boolExp，match\_param中均有错误检测。具体实现已在代码中给出，此处不再赘述。

## 3.3词法语法分析器（实验一）

词法分析器采用的工具是自动化生成工具GUN Flex，该工具要求词法规则以正则表达式（正规式）给出，并根据给定的此法规则生成相应的词法分析程序。Flex的原理是有穷自动机，即Flex会将用正则表达式表示的词法规则等价转化为相应的有穷自动机FA，生成对应的词法分析程序。所以，设计词法分析器的关键是设计能准确识别各类单词的正则表达式。

|  |  |
| --- | --- |
| 单词 | 正则表达式 |
| id | [A-Za-z][A-Za-z0-9]\* |
| int | [0-9]+ |
| char | ('[A-Za-z0-9]') |
| float | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| 行级注释 | (\/\/).\* |
| 块级注释 | \/\\*([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*\\*\/ |

为了能在词法分析和语法分析报时提供错误的详细位置信息，运用了Flex的部分高级特性，例如：开启yylineno选项，从而全局变量yylineno会记录当前正在分析的词法单元在源程序中的行号，并由Flex自动维护（初值设为1）。

语法分析器的实现采用的是自动化生成工具GUN Bison，Bison可以根据给定的语法规则，自动化生成对应的语法分析程序。语法分析不仅仅是判断源程序的语句是否符合语法规则，还应该构造源程序对应的语法分析树，用于编译的后续阶段。Bison和Flex可以无缝对接，即将Flex进行词法分析后得到的单词序列作为Bison的输入，从而来进行语法分析。

实验代码如下,优先级从上到下依次增高。

%left ASSIGNOP

%left OR

%left AND

%left RELOP //运算符

%left INC DEC INCASS DECASS

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT //UMINUS

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

一旦语法分析器创建了一棵抽象语法树，就可以直接编写递归子程序来遍历整棵树，抽象语法树由多个节点组成，每个节点都有一个节点类型，不同的节点可以有不同的域，它会去掉分析树中我们不需要关注的规则节点。

实验中定义的终结符如下：

%token <type\_int> INT

//指定INT的语义值是type\_int，有词法分析得到的数值

%token <type\_id> ID RELOP TYPE STRUCT

//指定ID,RELOP 的语义值是type\_id，有词法分析得到的标识符字符串

%token <type\_float> FLOAT

//指定ID的语义值是type\_id，有词法分析得到的标识符字符串

%token <type\_char> CHAR

//指定CHAR的语义值是type\_char，有词法分析得到的数值

%token LP RP LB RB LC RC SEMI COMMA

%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT

%token IF ELSE WHILE RETURN INC DEC INCASS DECASS

实验中定义的非终结符如下：

%type <ptr>

program ExtDefList ExtDef StructSpecifier OptTag Tag Specifier ExtDecList FuncDec

CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args

## 3.4语义分析（实验二）

语义分析的第一步便是构建符号表。综合考虑下，选择了在语法分析时构建语法分析树的同时构建符号表。这要选择所考虑的原因主要有：语法分析树和符号表都是编译器在后续阶段需要多次用到的重要信息，它们记录了源程序不同方面的信息，语法分析树记录的是源程序的语法结构，而符号表记录的是源程序的符号信息，二者虽有差异但却相互弥补，同等重要，因此逻辑上应该在一个阶段生成；在语法分析时若检测到说明语句，可以获得足够的信息来完成符号表的填表操作，因此没有必要单独进行一次语法树的遍历来构造符号表。综上，符号表的建立被放在了语法分析阶段与语法分析树的构造同时进行。对于使用了 Bison 的语法分析，其过程是自底向上的规约，所以在语法分析阶段，

当规约说明语句（包括类的定义、变量定义、函数定义等）时，便为相应的符号构造一个对应作用域符号表的表项（指向该表项的指针被放到了语法分析树的对应节点中）；当说明语句作为生成式的右部作为进一步规约时，根据生成式的语义，将说明语句所带的符号表表项传递给左部语法单元或者归并左边的所有说明语句的符号表表项，将归并后的符号表传递给左部语法单元。以此类推，各符号表便可以从底层逐层传递和累积，形成最终的符号表。

语义分析的第二步便是静态语义错误检查。本编译器在语义分析阶段的静态语义检查可以分为两大类，第一类是符号作用域相关检查，第二类是类型检查，这两类检查可以在一次语法树遍历过程内完成。当完成语法分析后，语法分析树

与符号表便会均构建完毕，可以随时调用。本编译器将静态语义错误检查单独写

成了一个模块，在语法分析阶段规约到最顶层生成式时，会调用该模块进行语义

检查。语义检查整体上就是一次语法树的遍历，自底向上进行语义分析，但是由

于类型检查和作用域分析都要用到符号表，所以在遍历语法树的同时需要维护一

个作用域栈，遍历的同时进行出栈和入栈操作。

对于作用域相关检查，最重要的便是在语法树遍历时维护作用域栈。当遇到

类的定义、函数定义、对象声明、函数调用等需要开启一个新的作用域时，便将

相应的作用域入栈，当遇到类似大括号这种语法单元时，就要将关闭当前的作用

域关闭，即从作用域栈中弹出闭作用域。当遇到使用一个符号时，就从栈中从栈

顶向栈底搜索，第一个搜索到的就是当前作用域最先访问的符号，没找到表明该

符号没有声明过。

对于类型相关检查，在遇到运算符操作时，先查看类型是否相符合。本过程主要在本表达式处理函数Exp中完成，当遇到双目运算时对两操作数进行类型匹配，不成功则调用函数semantic\_error报错。对于函数检查，类型检测仍在Exp函数中进行，但参数匹配使用函数match\_param中进行。

# 4系统测试与评价

## 4.1 测试用例

测试文件test.c

int a,b,c;

float m,n;

int d[10];

struct A{

int a;

int b;

};

int fibo(int a)

{

if (a == 1 || a == 2) return (a-1);

return fibo(a-1)+fibo(a-2);

}

// int c[10]; int i;

int main(){

int m,n,i;

char ch;

while(i<=m){

n = fibo(i);

i=i+1;

}

return 1;

}

结果文件：text.txt

测试脚本：sb.bat

del parser.tab.\* lex.yy.c parser

flex lex.l

bison -d parser.y

gcc -o parser ast.c semantic.c lex.yy.c parser.tab.c

parser.exe test.c > text.txt

## 4.2 正确性测试

外部变量定义：

类型： int

变量名：

ID： a

ID： b

ID： c

外部变量定义：

类型： float

变量名：

ID： m

ID： n

外部变量定义：

类型： int

变量名：

一维数组定义：

ID： d

SIZE: 10

结构体：

结构体： struct

ID： A

LOCAL VAR\_NAME：

类型： int

VAR\_NAME：

a

LOCAL VAR\_NAME：

类型： int

VAR\_NAME：

b

函数定义：

类型： int

函数名：fibo

函数形参：

类型：int, 参数名：a

复合语句：

复合语句的变量定义：

复合语句的语句部分：

条件语句(IF\_THEN)：

条件：

OR

==

ID： a

INT：1

==

ID： a

INT：2

IF子句：

返回语句：

MINUS

ID： a

INT：1

返回语句：

PLUS

函数调用：

函数名：fibo

第1个实际参数表达式：

MINUS

ID： a

INT：1

函数调用：

函数名：fibo

第1个实际参数表达式：

MINUS

ID： a

INT：2

函数定义：

类型： int

函数名：main

无参函数

复合语句：

复合语句的变量定义：

LOCAL VAR\_NAME：

类型： int

VAR\_NAME：

m

n

i

LOCAL VAR\_NAME：

类型： char

VAR\_NAME：

ch

复合语句的语句部分：

循环语句：

循环条件：

<=

ID： i

ID： m

循环体：

复合语句：

复合语句的变量定义：

复合语句的语句部分：

表达式语句：

ASSIGNOP

ID： n

函数调用：

函数名：fibo

第1个实际参数表达式：

ID： i

表达式语句：

ASSIGNOP

ID： i

PLUS

ID： i

INT：1

返回语句：

INT：1

变量名 别 名 层 号 类 型 标记 偏移量

a v1 0 int V 0

b v2 0 int V 4

c v3 0 int V 8

m v4 0 float V 12

n v5 0 float V 20

d v6 0 int A 28

fibo v7 0 int F 0

a v8 1 int P 12

temp1 1 int T 20

temp2 1 int T 20

temp3 1 int T 24

temp4 1 int T 28

temp5 1 int T 32

temp6 1 int T 36

变量名 别 名 层 号 类 型 标记 偏移量

a v1 0 int V 0

b v2 0 int V 4

c v3 0 int V 8

m v4 0 float V 12

n v5 0 float V 20

d v6 0 int A 28

fibo v7 0 int F 40

a v8 1 int P 12

main v9 0 int F 0

m v10 1 int V 12

n v11 1 int V 20

i v12 1 int V 28

ch v13 1 char V 36

temp7 2 int T 37

temp8 2 int T 37

变量名 别 名 层 号 类 型 标记 偏移量

a v1 0 int V 0

b v2 0 int V 4

c v3 0 int V 8

m v4 0 float V 12

n v5 0 float V 20

d v6 0 int A 28

fibo v7 0 int F 40

a v8 1 int P 12

main v9 0 int F 0

m v10 1 int V 12

n v11 1 int V 20

i v12 1 int V 28

ch v13 1 char V 36

## 4.3 报错功能测试

* 测试函数参数问题，将测试用例main函数中fibo调用改为fibo(i,i)；
* 测试重复定义问题，在测试用例第一行后添加int a;
* 测试运算符类型匹配问题，在测试用例main函数中将i=i+1改为i=i+ch;

外部变量定义：

类型： int

变量名：

ID： a

ID： b

ID： c

外部变量定义：

类型： int

变量名：

ID： a

外部变量定义：

类型： float

变量名：

ID： m

ID： n

外部变量定义：

类型： int

变量名：

一维数组定义：

ID： d

SIZE: 10

结构体：

结构体： struct

ID： A

LOCAL VAR\_NAME：

类型： int

VAR\_NAME：

a

LOCAL VAR\_NAME：

类型： int

VAR\_NAME：

b

函数定义：

类型： int

函数名：fibo

函数形参：

类型：int, 参数名：a

复合语句：

复合语句的变量定义：

复合语句的语句部分：

条件语句(IF\_THEN)：

条件：

OR

==

ID： a

INT：1

==

ID： a

INT：2

IF子句：

返回语句：

MINUS

ID： a

INT：1

返回语句：

PLUS

函数调用：

函数名：fibo

第1个实际参数表达式：

MINUS

ID： a

INT：1

函数调用：

函数名：fibo

第1个实际参数表达式：

MINUS

ID： a

INT：2

函数定义：

类型： int

函数名：main

无参函数

复合语句：

复合语句的变量定义：

LOCAL VAR\_NAME：

类型： int

VAR\_NAME：

m

n

i

LOCAL VAR\_NAME：

类型： char

VAR\_NAME：

ch

复合语句的语句部分：

循环语句：

循环条件：

<=

ID： i

ID： m

循环体：

复合语句：

复合语句的变量定义：

复合语句的语句部分：

表达式语句：

ASSIGNOP

ID： n

函数调用：

函数名：fibo

第1个实际参数表达式：

ID： i

第2个实际参数表达式：

ID： i

表达式语句：

ASSIGNOP

ID： i

PLUS

ID： i

ID： ch

返回语句：

INT：1

在2行,a 变量重复定义

变量名 别 名 层 号 类 型 标记 偏移量

a v1 0 int V 0

b v2 0 int V 4

c v3 0 int V 8

m v5 0 float V 16

n v6 0 float V 24

d v7 0 int A 32

fibo v8 0 int F 0

a v9 1 int P 12

temp1 1 int T 20

temp2 1 int T 20

temp3 1 int T 24

temp4 1 int T 28

temp5 1 int T 32

temp6 1 int T 36

在21行, 函数调用参数冗余

在22行,PLUS 两加数类型不兼容

在22行, 赋值号两边表达式类型不匹配

变量名 别 名 层 号 类 型 标记 偏移量

a v1 0 int V 0

b v2 0 int V 4

c v3 0 int V 8

m v5 0 float V 16

n v6 0 float V 24

d v7 0 int A 32

fibo v8 0 int F 40

a v9 1 int P 12

main v10 0 int F 0

m v11 1 int V 12

n v12 1 int V 20

i v13 1 int V 28

ch v14 1 char V 36

temp7 2 int T 37

temp8 2 char T 37

变量名 别 名 层 号 类 型 标记 偏移量

a v1 0 int V 0

b v2 0 int V 4

c v3 0 int V 8

m v5 0 float V 16

n v6 0 float V 24

d v7 0 int A 32

fibo v8 0 int F 40

a v9 1 int P 12

main v10 0 int F 0

m v11 1 int V 12

n v12 1 int V 20

i v13 1 int V 28

ch v14 1 char V 36.

## 4.4 系统的优点

可以实现基本的复合运算与错误检测

## 4.5 系统的缺点

功能少，类型少

# 5实验小结或体会

本次实验，大部分工作都已经由老师给出的文档给出，我们需要做的主要是读懂实验材料，其次才是在上面实现一些基本的赋值运算和条件语句。这次实验我只完成了前两个，就是词法语法分析和语义分析，后面的没有完成，希望能在暑假有空自己重新写一遍，才能真正体会到编译的困难和精彩之处。

写实验一还好，基本上就是按照老师给的参考文档和文件，添加char正则表达式和一些单词，主要工作是在parser.y中实现，需要我们自己写一些规则，这里比较困难，要先读懂ast.c文件和def.h文件中的定义。实验二主要是在semantic.c文件中实现，首先知道整个符号表是顺序表，作用域操作需要用到栈，然后就是了解表结构，作用域压栈出栈的时间，在这个过程中同时实现静态语义检查，对可能发生的错误给出错误提示。

**参考文献**

[1] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008

**附件：源代码**

**def.h**

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include "stdarg.h"

#include "parser.tab.h"

enum node\_kind

{

EXT\_DEF\_LIST, //外部定义列表

EXT\_VAR\_DEF, //外部变量定义

FUNC\_DEF, //函数定义

FUNC\_DEC, //函数声明？

EXT\_DEC\_LIST, //外部声明列表？

STRUCT\_DEF, //结构体定义

ARRAY\_DEF, //数组定义

TWO\_ARRAY\_DEF, //二维数组定义

STRUCT\_SPECIFIER, //结构体的描述符

PARAM\_LIST, //形参列表

PARAM\_DEC, //形参声明

VAR\_DEF, //变量定义

DEC\_LIST, //声明列表

DEF\_LIST, //定义列表

COMP\_STM,

STM\_LIST,

EXP\_STMT,

IF\_THEN,

IF\_THEN\_ELSE,

FUNC\_CALL,

ARGS,

FUNCTION,

PARAM,

ARG,

CALL,

LABEL,

GOTO,

JLT,

JLE,

JGT,

JGE,

EQ,

NEQ,

SCHAR,

SFLOAT,

SINT

};

#define MAXLENGTH 1000 //定义符号表的大小

#define DX 3 \* sizeof(int) //活动记录控制信息需要的单元数

#define MAXDIM 10 //定义数组最大维数

struct opn

{

int kind; //标识操作的类型

int type; //标识操作数的类型

union {

int const\_int; //整常数值，立即数

float const\_float; //浮点常数值，立即数

char const\_char; //字符常数值，立即数

char id[33]; //变量或临时变量的别名或标号字符串

};

int level; //变量的层号，0表示是全局变量，数据保存在静态数据区

int offset; //变量单元偏移量，或函数在符号表的定义位置序号，目标代码生成时用

};

struct codenode

{ //三地址TAC代码结点,采用双向循环链表存放中间语言代码

int op; //TAC代码的运算符种类

struct opn opn1, opn2, result; //2个操作数和运算结果

struct codenode \*next, \*prior;

};

struct node

{ //以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性

enum node\_kind kind; //结点类型

union {

char type\_id[33]; //由标识符生成的叶结点

int type\_int; //由整常数生成的叶结点

float type\_float; //由浮点常数生成的叶结点

char type\_char; //由字符生成的叶节点

};

struct node \*ptr[3]; //子树指针，由kind确定有多少棵子树

int level; //层号

int place; //表示结点对应的变量或运算结果符号表的位置序号

char Etrue[15], Efalse[15]; //对布尔表达式的翻译时，真假转移目标的标号

char Snext[15]; //该结点对饮语句执行后的下一条语句位置标号

struct codenode \*code; //该结点中间代码链表头指针

char op[10];

int type; //结点对应值的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //占数据字节数

int num; //若为变量列表，记录变量个数

};

struct symbol

{ //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥

char name[33]; //变量或函数名

int level; //层号，外部变量名或函数名层号为0，形参名为1，每到1个复合语句层号加1，退出减1

int type; //变量类型或函数返回值类型

int paramnum; //形式参数个数

char alias[10]; //别名，为解决嵌套层次使用，使得每一个数据名称唯一

char flag; //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T'

char offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量

//或函数活动记录大小，目标代码生成时使用

};

//符号表，是一个顺序栈，index初值为0

struct symboltable

{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

/\*\*

\* 当前作用域的符号在符号表的起始位置序号,这是一个栈结构

\* 每到达一个复合语句，将符号表的index值进栈，离开复合语句时，取其退栈值修改符号表的index值，完成删除该复合语句中的所有变量和临时变量

\*/

struct symbol\_scope\_begin

{

int TX[30];

int top;

} symbol\_scope\_TX;

struct node \*mknode(int kind, struct node \*first, struct node \*second, struct node \*third, int pos);

void semantic\_Analysis0(struct node \*T);

void semantic\_Analysis(struct node \*T);

void boolExp(struct node \*T);

void Exp(struct node \*T);

void objectCode(struct codenode \*head);

**lex.l**

%{

#include "parser.tab.h"

#include "string.h"

#include "def.h"

int yycolumn=1;

#define YY\_USER\_ACTION yylloc.first\_line=yylloc.last\_line=yylineno; yylloc.first\_column=yycolumn; yylloc.last\_column=yycolumn+yyleng-1; yycolumn+=yyleng;

typedef union {

int type\_int;

int type\_float;

char type\_char;

char type\_id[32];

struct node \*ptr;

} YYLVAL;

#define YYSTYPE YYLVAL

//%option yylineno 记录符号所在行号

%}

%option yylineno

id [A-Za-z][A-Za-z0-9]\*

int [0-9]+

float ([0-9]\*\.[0-9]+)|([0-9]+\.)

char ('[A-Za-z0-9]')

%%

{int} {yylval.type\_int=atoi(yytext); return INT;}

{float} {yylval.type\_float=atof(yytext); return FLOAT;}

{char} {yylval.type\_char=yytext[1];return CHAR;}

"int" {strcpy(yylval.type\_id, yytext);return TYPE;}

"float" {strcpy(yylval.type\_id, yytext);return TYPE;}

"char" {strcpy(yylval.type\_id, yytext);return TYPE;}

"struct" {strcpy(yylval.type\_id, yytext);return STRUCT;}

"return" {return RETURN;}

"if" {return IF;}

"else" {return ELSE;}

"while" {return WHILE;}

{id} {strcpy(yylval.type\_id, yytext); return ID;}/\*由于关键字的形式也符合表示符的规则，所以把关键字的处理全部放在标识符的前面，优先识别\*/

";" {return SEMI;}

"," {return COMMA;}

">"|"<"|">="|"<="|"=="|"!=" {strcpy(yylval.type\_id, yytext);;return RELOP;}

"=" {return ASSIGNOP;}

"+" {return PLUS;}

"-" {return MINUS;}

"++" {return INC;}

"--" {return DEC;}

"+=" {return INCASS;}

"-=" {return DECASS;}

"\*" {return STAR;}

"/" {return DIV;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"[" {return LB;}

"]" {return RB;}

"{" {return LC;}

"}" {return RC;}

[\n] {yycolumn=1;}

[ \r\t] {}

. {printf("Error type A :Mysterious character \"%s\"\n\t at Line %d\n",yytext,yylineno);}

(\/\/).\* {}

\/\\*([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*\\*\/ {}

%%

/\* 和bison联用时，不需要这部分\*/

//当Lex处理到输入串的文件尾时，自动地调用yywrap（），如果 yywrap（）返回值是 1，那么Lex就认为对输入的处理完全结束

int yywrap()

{

return 1;

}

**parser.y**

%error-verbose

%locations

%{

#include "stdio.h"

#include "math.h"

#include "string.h"

#include "def.h"

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

void yyerror(const char\* fmt, ...);

void display(struct node \*,int);

%}

//改变YYSTYPE的类型

%union {

int type\_int;

float type\_float;

char type\_char;

char type\_id[32];

struct node \*ptr;

};

/\*\*

可以把标志（token）绑定到YYSTYPE的某个域

如：%token <iValue> INTEGER

%type <nPtr> expr

把expr绑定到nPtr，把INTEGER绑定到iValue

yacc处理时会做转换

如：expr: INTEGER { $$ = con($1); }

转换结果为：yylval.nPtr = con(yyvsp[0].iValue);

其中yyvsp[0]是值栈（value stack）当前的头部

\*/

// %type 定义非终结符的语义值类型，如把program绑定到ptr

%type <ptr>

program ExtDefList ExtDef StructSpecifier OptTag Tag Specifier ExtDecList FuncDec

CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args

//% token 定义终结符的语义值类型

%token <type\_int> INT //指定INT的语义值是type\_int，有词法分析得到的数值

%token <type\_id> ID RELOP TYPE STRUCT //指定ID,RELOP 的语义值是type\_id，有词法分析得到的标识符字符串

%token <type\_float> FLOAT //指定ID的语义值是type\_id，有词法分析得到的标识符字符串

%token <type\_char> CHAR //指定CHAR的语义值是type\_char，有词法分析得到的数值

//用bison对该文件编译时，带参数-d，生成的exp.tab.h中给这些单词进行编码，可在lex.l中包含parser.tab.h使用这些单词种类码

%token LP RP LB RB LC RC SEMI COMMA

%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT

%token IF ELSE WHILE RETURN INC DEC INCASS DECASS

%left ASSIGNOP

%left OR

%left AND

%left RELOP //运算符

%left INC DEC INCASS DECASS

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT //UMINUS

//%nonassoc的含义是没有结合性。它一般与%prec结合使用表示该操作有同样的优先级

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

//display在ast.c中定义，semantic\_Analysis0在def.h中定义

//$$表示规约后的值;$1表示右边的第一个标记的值，$2表示右边的第二个标记的值，依次类推

/\*

如：expr: INTEGER { $$ = con($1); } 转换结果为：yylval.nPtr = con(yyvsp[0].iValue);

其中yyvsp[0]是值栈（value stack）当前的头部

\*/

//初始语法单元，表示整个程序

program: ExtDefList { display($1,0);semantic\_Analysis0($1);} /\*显示语法树,语义分析\*///

;

//表示零个或多个 ExtDef

ExtDefList: {$$=NULL;}

| ExtDef ExtDefList {$$=mknode(EXT\_DEF\_LIST,$1,$2,NULL,yylineno);} //每一个EXTDEFLIST的结点，其第1棵子树对应一个外部变量声明或函数

;

//一个全局变量、结构体或函数的定义

ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT\_VAR\_DEF,$1,$2,NULL,yylineno);} //该结点对应一个外部变量声明（全局变量）

|Specifier SEMI {$$=mknode(STRUCT\_DEF,$1,NULL,NULL,yylineno);} //为定义结构体

|Specifier FuncDec CompSt {$$=mknode(FUNC\_DEF,$1,$2,$3,yylineno);} //该结点对应一个函数定义

| error SEMI {$$=NULL; }

;

//定义结构体

StructSpecifier: STRUCT OptTag LC DefList RC {$$=mknode(STRUCT\_DEF,$2,$4,NULL,yylineno);strcpy($$->type\_id,$1);}//定义结构体的基本格式

| STRUCT Tag {$$=mknode(STRUCT\_DEF,$2,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}//定义结构体变量

;

//类型描述符：TYPE-->int/float等；StructSpecifier-->结构体

Specifier: TYPE {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);if(!strcmp($1, "int"))$$->type=INT;if(!strcmp($1, "float"))$$->type=FLOAT;if(!strcmp($1, "char"))$$->type=CHAR;}

| StructSpecifier {$$=$1;}

;

//结构体名：struct OptTag{...}

OptTag: ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}

| {printf("OptTag");$$=NULL;}

;

//已定义结构体名：OptTag Tag；像 int a一样

Tag: ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}

;

ExtDecList: VarDec {$$=$1;} /\*每一个EXT\_DECLIST的结点，其第一棵子树对应一个变量名(ID类型的结点),第二棵子树对应剩下的外部变量名\*/

| VarDec COMMA ExtDecList {$$=mknode(EXT\_DEC\_LIST,$1,$3,NULL,yylineno);}

;

//表示对一个变量的定义

VarDec: ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);} //ID结点，标识符符号串存放结点的type\_id

| VarDec LB INT RB {$$=mknode(ARRAY\_DEF,$1,NULL,NULL,yylineno);$$->type\_id[0]=$3;}//数组

;

//表示对一个函数头的定义

FuncDec: ID LP VarList RP {$$=mknode(FUNC\_DEC,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}//函数名存放在$$->type\_id

|ID LP RP {$$=mknode(FUNC\_DEC,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}//函数名存放在$$->type\_id

;

//包括一个或多个ParamDec

VarList: ParamDec {$$=mknode(PARAM\_LIST,$1,NULL,NULL,yylineno);}

| ParamDec COMMA VarList {$$=mknode(PARAM\_LIST,$1,$3,NULL,yylineno);}

;

//对一个形参的定义

ParamDec: Specifier VarDec {$$=mknode(PARAM\_DEC,$1,$2,NULL,yylineno);}

;

//表示由一对花括号括起来的语句块；因此必须在每个语句块的开头才可以定义变量

CompSt: LC DefList StmList RC {$$=mknode(COMP\_STM,$2,$3,NULL,yylineno);}

;

//零个或多个Stmt的组合，即语句定义集合

StmList: {$$=NULL; }

| Stmt StmList {$$=mknode(STM\_LIST,$1,$2,NULL,yylineno);}

;

//表示一条语句

Stmt: Exp SEMI {$$=mknode(EXP\_STMT,$1,NULL,NULL,yylineno);}//在末尾添加了分号的表达式

| CompSt {$$=$1;} //复合语句结点直接最为语句结点，不再生成新的结点；另一个语句块

| RETURN Exp SEMI {$$=mknode(RETURN,$2,NULL,NULL,yylineno);}//一条返回语句

| IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE {$$=mknode(IF\_THEN,$3,$5,NULL,yylineno);}//一条IF语句

| IF LP Exp RP Stmt ELSE Stmt {$$=mknode(IF\_THEN\_ELSE,$3,$5,$7,yylineno);}//一条IF-ELSE语句

| WHILE LP Exp RP Stmt {$$=mknode(WHILE,$3,$5,NULL,yylineno);}//一条WHILE语句

;

//由多个Def组成

DefList: {$$=NULL; }

| Def DefList {$$=mknode(DEF\_LIST,$1,$2,NULL,yylineno);}

;

//变量定义

Def: Specifier DecList SEMI {$$=mknode(VAR\_DEF,$1,$2,NULL,yylineno);}//如：int a,b,c;

;

DecList: Dec {$$=mknode(DEC\_LIST,$1,NULL,NULL,yylineno);}

| Dec COMMA DecList {$$=mknode(DEC\_LIST,$1,$3,NULL,yylineno);}

;

Dec: VarDec {$$=$1;}

| VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}//如：int a = 5;

;

Exp: Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}//赋值表达式；$$结点type\_id空置未用，正好存放运算符

| Exp AND Exp {$$=mknode(AND,$1,$3,NULL,yylineno);strcpy($$->type\_id,"AND");}//逻辑与

| Exp OR Exp {$$=mknode(OR,$1,$3,NULL,yylineno);strcpy($$->type\_id,"OR");}//逻辑或

| Exp RELOP Exp {$$=mknode(RELOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,$2);} //关系表达式；词法分析关系运算符号自身值保存在$2中

| Exp PLUS Exp {$$=mknode(PLUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"PLUS");}//四则运算：加

| Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"MINUS");}//四则运算：减

| Exp STAR Exp {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type\_id,"STAR");}////四则运算：乘

| Exp DIV Exp {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DIV");}//四则运算：除

| Exp INC {$$=mknode(INC,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"INC");}

| Exp DEC {$$=mknode(DEC,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"DEC");}

| Exp INCASS Exp {$$=mknode(INCASS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"INCASS");}

| Exp DECASS Exp {$$=mknode(INCASS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DECASS");}

| LP Exp RP {$$=$2;} //括号表达式

| MINUS Exp %prec UMINUS {$$=mknode(UMINUS,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"UMINUS");}//取负

| NOT Exp {$$=mknode(NOT,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"NOT");}//逻辑非

| ID LP Args RP {$$=mknode(FUNC\_CALL,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}//函数调用表达式：带参数

| ID LP RP {$$=mknode(FUNC\_CALL,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}//函数调用表达式：不带参数

| Exp LB Exp RB {$$=mknode(FUNC\_CALL,$1,NULL,$3,yylineno);} //数组访问表达式

| Exp STRUCT ID {$$=mknode(ID,$1,NULL,NULL,yylineno);strcpy($$->type\_id,$2);}//结构体访问表达式

| ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}

| INT {$$=mknode(SINT,NULL,NULL,NULL,yylineno);$$->type\_int=$1;$$->type=INT;}

| FLOAT {$$=mknode(SFLOAT,NULL,NULL,NULL,yylineno);$$->type\_float=$1;$$->type=FLOAT;}

| CHAR {$$=mknode(SCHAR,NULL,NULL,NULL,yylineno);$$->type\_char=$1;$$->type=CHAR;}

;

//实参列表，每个实参都可以变为一个表达式Exp

Args: Exp COMMA Args {$$=mknode(ARGS,$1,$3,NULL,yylineno);}

| Exp {$$=mknode(ARGS,$1,NULL,NULL,yylineno);}

;

%%

//当yacc解析出错时，会调用函数yyerror()

//调用yacc解析入口函数yyparse()

int main(int argc, char \*argv[]){

yyin=fopen(argv[1],"r");

if (!yyin) return 1;

yylineno=1;

// yydebug=1;

yyparse();

return 0;

}

#include<stdarg.h>

void yyerror(const char\* fmt, ...)

{

va\_list ap;

va\_start(ap, fmt);

fprintf(stderr, "Grammar Error at Line %d Column %d: ", yylloc.first\_line,yylloc.first\_column);

vfprintf(stderr, fmt, ap);

fprintf(stderr, ".\n");

}

**ast.c**

#include "def.h"

struct node \*mknode(int kind, struct node \*first, struct node \*second, struct node \*third, int pos)

{

struct node \*T = (struct node \*)malloc(sizeof(struct node));

T->kind = kind;

T->ptr[0] = first;

T->ptr[1] = second;

T->ptr[2] = third;

T->pos = pos;

return T;

}

void display(struct node \*T, int indent)

{ //对抽象语法树的先根遍历

int i = 1;

struct node \*T0;

if (T)

{

switch (T->kind)

{

case EXT\_DEF\_LIST:

display(T->ptr[0], indent); //显示该外部定义列表中的第一个

display(T->ptr[1], indent); //显示该外部定义列表中的其它外部定义

break;

case EXT\_VAR\_DEF:

printf("%\*c外部变量定义：\n", indent, ' ');

display(T->ptr[0], indent + 3); //显示外部变量类型

printf("%\*c变量名：\n", indent + 3, ' ');

display(T->ptr[1], indent + 6); //显示变量列表

break;

case STRUCT\_SPECIFIER:

printf("find STRUCT\_SPECIFIER");

break;

case STRUCT\_DEF:

printf("%\*c结构体： %s\n", indent, ' ', T->type\_id);

display(T->ptr[0], indent + 3);

display(T->ptr[1], indent + 3);

break;

case TYPE:

printf("%\*c类型： %s\n", indent, ' ', T->type\_id);

break;

case EXT\_DEC\_LIST:

display(T->ptr[0], indent); //依次显示外部变量名，

display(T->ptr[1], indent); //后续还有相同的，仅显示语法树此处理代码可以和类似代码合并

break;

case FUNC\_DEF:

printf("%\*c函数定义：\n", indent, ' ');

display(T->ptr[0], indent + 3); //显示函数返回类型

display(T->ptr[1], indent + 3); //显示函数名和参数

display(T->ptr[2], indent + 3); //显示函数体

break;

case FUNC\_DEC:

printf("%\*c函数名：%s\n", indent, ' ', T->type\_id);

if (T->ptr[0])

{

printf("%\*c函数形参：\n", indent, ' ');

display(T->ptr[0], indent + 3); //显示函数参数列表

}

else

printf("%\*c无参函数\n", indent + 3, ' ');

break;

case PARAM\_LIST:

display(T->ptr[0], indent); //依次显示全部参数类型和名称，

display(T->ptr[1], indent);

break;

case PARAM\_DEC:

if (T->ptr[0]->type == INT)

printf("%\*c类型：%s, 参数名：%s\n", indent, ' ', "int", T->ptr[1]->type\_id);

if (T->ptr[0]->type == FLOAT)

printf("%\*c类型：%s, 参数名：%s\n", indent, ' ', "float", T->ptr[1]->type\_id);

if (T->ptr[0]->type == CHAR)

printf("%\*c类型：%s, 参数名：%s\n", indent, ' ', "char", T->ptr[1]->type\_id);

break;

case EXP\_STMT:

printf("%\*c表达式语句：\n", indent, ' ');

display(T->ptr[0], indent + 3);

break;

case RETURN:

printf("%\*c返回语句：\n", indent, ' ');

display(T->ptr[0], indent + 3);

break;

case COMP\_STM:

printf("%\*c复合语句：\n", indent, ' ');

printf("%\*c复合语句的变量定义：\n", indent + 3, ' ');

display(T->ptr[0], indent + 6); //显示定义部分

printf("%\*c复合语句的语句部分：\n", indent + 3, ' ');

display(T->ptr[1], indent + 6); //显示语句部分

break;

case STM\_LIST:

display(T->ptr[0], indent); //显示第一条语句

display(T->ptr[1], indent); //显示剩下语句

break;

case WHILE:

printf("%\*c循环语句：\n", indent, ' ');

printf("%\*c循环条件：\n", indent + 3, ' ');

display(T->ptr[0], indent + 6); //显示循环条件

printf("%\*c循环体：\n", indent + 3, ' ');

display(T->ptr[1], indent + 6); //显示循环体

break;

case IF\_THEN:

printf("%\*c条件语句(IF\_THEN)：\n", indent, ' ');

printf("%\*c条件：\n", indent + 3, ' ');

display(T->ptr[0], indent + 6); //显示条件

printf("%\*cIF子句：\n", indent + 3, ' ');

display(T->ptr[1], indent + 6); //显示if子句

break;

case IF\_THEN\_ELSE:

printf("%\*c条件语句(IF\_THEN\_ELSE)：\n", indent, ' ');

printf("%\*c条件：\n", indent + 3, ' ');

display(T->ptr[0], indent + 6); //显示条件

printf("%\*cIF子句：\n", indent + 3, ' ');

display(T->ptr[1], indent + 6); //显示if子句

printf("%\*cELSE子句：\n", indent + 3, ' ');

display(T->ptr[2], indent + 6); //显示else子句

break;

case DEF\_LIST:

display(T->ptr[0], indent); //显示该局部变量定义列表中的第一个

display(T->ptr[1], indent); //显示其它局部变量定义

break;

case VAR\_DEF:

printf("%\*cLOCAL VAR\_NAME：\n", indent, ' ');

display(T->ptr[0], indent + 3); //显示变量类型

display(T->ptr[1], indent + 3); //显示该定义的全部变量名

break;

case DEC\_LIST:

printf("%\*cVAR\_NAME：\n", indent, ' ');

T0 = T;

while (T0)

{

if (T0->ptr[0]->kind == ID)

printf("%\*c %s\n", indent + 3, ' ', T0->ptr[0]->type\_id);

else if (T0->ptr[0]->kind == ASSIGNOP)

{

printf("%\*c %s ASSIGNOP\n ", indent + 3, ' ', T0->ptr[0]->ptr[0]->type\_id);

//显示初始化表达式

display(T0->ptr[0]->ptr[1], indent + strlen(T0->ptr[0]->ptr[0]->type\_id) + 4);

}

T0 = T0->ptr[1];

}

break;

case ID:

printf("%\*cID： %s\n", indent, ' ', T->type\_id);

break;

case SINT:

printf("%\*cINT：%d\n", indent, ' ', T->type\_int);

break;

case SFLOAT:

printf("%\*cFLAOT：%f\n", indent, ' ', T->type\_float);

break;

case SCHAR:

printf("%\*cCHAR：%c\n", indent, ' ', T->type\_char);

case ARRAY\_DEF:

printf("%\*c一维数组定义： \n", indent, ' ');

display(T->ptr[0], indent + 3);

printf("%\*cSIZE: %d\n", indent, ' ', T->type\_id[0]);

break;

case TWO\_ARRAY\_DEF:

printf("%\*c二维数组定义： \n", indent, ' ');

display(T->ptr[0], indent + 3);

printf("%\*cSIZE: %d %d\n", indent, ' ', T->type\_id[0], T->type\_id[1]);

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case INC:

case DEC:

case INCASS:

case DECASS:

case STAR:

case DIV:

printf("%\*c%s\n", indent, ' ', T->type\_id);

display(T->ptr[0], indent + 3);

display(T->ptr[1], indent + 3);

break;

case NOT:

case UMINUS:

printf("%\*c%s\n", indent, ' ', T->type\_id);

display(T->ptr[0], indent + 3);

break;

case FUNC\_CALL:

printf("%\*c函数调用：\n", indent, ' ');

printf("%\*c函数名：%s\n", indent + 3, ' ', T->type\_id);

display(T->ptr[0], indent + 3);

break;

case ARGS:

i = 1;

while (T)

{ //ARGS表示实际参数表达式序列结点，其第一棵子树为其一个实际参数表达式，第二棵子树为剩下的。

struct node \*T0 = T->ptr[0];

printf("%\*c第%d个实际参数表达式：\n", indent, ' ', i++);

display(T0, indent + 3);

T = T->ptr[1];

}

// printf("%\*c第%d个实际参数表达式：\n",indent,' ',i);

// display(T,indent+3);

printf("\n");

break;

}

}

return;

}

**semantic.c**

#include "def.h"

char \*strcat0(char \*s1, char \*s2)

{

static char result[10];

strcpy(result, s1);

strcat(result, s2);

return result;

}

char \*newAlias()

{

static int no = 1;

char s[10];

itoa(no++, s, 10);

return strcat0("v", s);

}

char \*newLabel()

{

static int no = 1;

char s[10];

itoa(no++, s, 10);

return strcat0("label", s);

}

char \*newTemp()

{

static int no = 1;

char s[10];

itoa(no++, s, 10);

return strcat0("temp", s);

}

//生成一条TAC代码的结点组成的双向循环链表，返回头指针

struct codenode \*genIR(int op, struct opn opn1, struct opn opn2, struct opn result)

{

struct codenode \*h = (struct codenode \*)malloc(sizeof(struct codenode));

h->op = op;

h->opn1 = opn1;

h->opn2 = opn2;

h->result = result;

h->next = h->prior = h;

return h;

}

//生成一条标号语句，返回头指针

struct codenode \*genLabel(char \*label)

{

struct codenode \*h = (struct codenode \*)malloc(sizeof(struct codenode));

h->op = LABEL;

strcpy(h->result.id, label);

h->next = h->prior = h;

return h;

}

//生成GOTO语句，返回头指针

struct codenode \*genGoto(char \*label)

{

struct codenode \*h = (struct codenode \*)malloc(sizeof(struct codenode));

h->op = GOTO;

strcpy(h->result.id, label);

h->next = h->prior = h;

return h;

}

//合并多个中间代码的双向循环链表，首尾相连

struct codenode \*merge(int num, ...)

{

struct codenode \*h1, \*h2, \*p, \*t1, \*t2;

va\_list ap;

va\_start(ap, num);

h1 = va\_arg(ap, struct codenode \*);

while (--num > 0)

{

h2 = va\_arg(ap, struct codenode \*);

if (h1 == NULL)

h1 = h2;

else if (h2)

{

t1 = h1->prior;

t2 = h2->prior;

t1->next = h2;

t2->next = h1;

h1->prior = t2;

h2->prior = t1;

}

}

va\_end(ap);

return h1;

}

//输出中间代码

void prnIR(struct codenode \*head)

{

char opnstr1[32], opnstr2[32], resultstr[32];

struct codenode \*h = head;

do

{

if (h->opn1.kind == INT)

sprintf(opnstr1, "#%d", h->opn1.const\_int);

if (h->opn1.kind == FLOAT)

sprintf(opnstr1, "#%f", h->opn1.const\_float);

if (h->opn1.kind == CHAR)

sprintf(opnstr1, "#%s", h->opn1.const\_char);

if (h->opn1.kind == ID)

sprintf(opnstr1, "%s", h->opn1.id);

if (h->opn2.kind == INT)

sprintf(opnstr2, "#%d", h->opn2.const\_int);

if (h->opn2.kind == FLOAT)

sprintf(opnstr2, "#%f", h->opn2.const\_float);

if (h->opn2.kind == CHAR)

sprintf(opnstr1, "#%s", h->opn2.const\_char);

if (h->opn2.kind == ID)

sprintf(opnstr2, "%s", h->opn2.id);

sprintf(resultstr, "%s", h->result.id);

switch (h->op)

{

case ASSIGNOP:

printf(" %s := %s\n", resultstr, opnstr1);

break;

case PLUS:

case MINUS:

case STAR:

case DIV:

printf(" %s := %s %c %s\n", resultstr, opnstr1,

h->op == PLUS ? '+' : h->op == MINUS ? '-' : h->op == STAR ? '\*' : '\\', opnstr2);

break;

case FUNCTION:

printf("\nFUNCTION %s :\n", h->result.id);

break;

case PARAM:

printf(" PARAM %s\n", h->result.id);

break;

case LABEL:

printf("LABEL %s :\n", h->result.id);

break;

case GOTO:

printf(" GOTO %s\n", h->result.id);

break;

case JLE:

printf(" IF %s <= %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case JLT:

printf(" IF %s < %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case JGE:

printf(" IF %s >= %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case JGT:

printf(" IF %s > %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case EQ:

printf(" IF %s == %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case NEQ:

printf(" IF %s != %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case ARG:

printf(" ARG %s\n", h->result.id);

break;

case CALL:

printf(" %s := CALL %s\n", resultstr, opnstr1);

break;

case RETURN:

if (h->result.kind)

printf(" RETURN %s\n", resultstr);

else

printf(" RETURN\n");

break;

}

h = h->next;

} while (h != head);

}

void semantic\_error(int line, char \*msg1, char \*msg2)

{

//这里可以只收集错误信息，最后在一次显示

printf("在%d行,%s %s\n", line, msg1, msg2);

}

void prn\_symbol()

{ //显示符号表

int i = 0;

printf("%6s %6s %6s %6s %4s %6s\n", "变量名", "别 名", "层 号", "类 型", "标记", "偏移量");

for (i = 0; i < symbolTable.index; i++)

printf("%6s %6s %6d %6s %4c %6d\n", symbolTable.symbols[i].name,

symbolTable.symbols[i].alias, symbolTable.symbols[i].level,

symbolTable.symbols[i].type == INT ? "int" : (symbolTable.symbols[i].type == FLOAT ? "float" : "char"),

symbolTable.symbols[i].flag, symbolTable.symbols[i].offset);

}

int searchSymbolTable(char \*name)

{

int i;

for (i = symbolTable.index - 1; i >= 0; i--)

if (!strcmp(symbolTable.symbols[i].name, name))

return i;

return -1;

}

/\*\*

\* 向符号表添加内容

\*

\* name: 变量或函数名

\* alias: 变量或函数别名

\* level：层号，外部变量名或函数名层号为0，形参名为1，每到1个复合语句层号加1，退出减1

\* type：变量类型或函数返回值类型

\* flag：符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T' 数组变量：A

\* offset：外部变量和局部变量在其静态数据区或活动记录中的偏移量

\*/

int fillSymbolTable(char \*name, char \*alias, int level, int type, char flag, int offset)

{

//首先根据name查符号表，不能重复定义 重复定义返回-1

int i;

/\*符号查重，考虑外部变量声明前有函数定义，

其形参名还在符号表中，这时的外部变量与前函数的形参重名是允许的\*/

for (i = symbolTable.index - 1; symbolTable.symbols[i].level == level || (level == 0 && i >= 0); i--)

{

if (level == 0 && symbolTable.symbols[i].level == 1)

continue; //外部变量和形参不必比较重名

if (!strcmp(symbolTable.symbols[i].name, name))

return -1;

}

//填写符号表内容

strcpy(symbolTable.symbols[symbolTable.index].name, name);

strcpy(symbolTable.symbols[symbolTable.index].alias, alias);

symbolTable.symbols[symbolTable.index].level = level;

symbolTable.symbols[symbolTable.index].type = type;

symbolTable.symbols[symbolTable.index].flag = flag;

symbolTable.symbols[symbolTable.index].offset = offset;

return symbolTable.index++; //返回的是符号在符号表中的位置序号，中间代码生成时可用序号取到符号别名

}

//填写临时变量到符号表，返回临时变量在符号表中的位置

int fill\_Temp(char \*name, int level, int type, char flag, int offset)

{

strcpy(symbolTable.symbols[symbolTable.index].name, "");

strcpy(symbolTable.symbols[symbolTable.index].alias, name);

symbolTable.symbols[symbolTable.index].level = level;

symbolTable.symbols[symbolTable.index].type = type;

symbolTable.symbols[symbolTable.index].flag = flag;

symbolTable.symbols[symbolTable.index].offset = offset;

return symbolTable.index++; //返回的是临时变量在符号表中的位置序号

}

int LEV = 0; //层号

int func\_size; //1个函数的活动记录大小

void ext\_var\_list(struct node \*T)

{ //处理变量列表

int rtn, num = 1;

switch (T->kind)

{

case EXT\_DEC\_LIST:

T->ptr[0]->type = T->type; //将类型属性向下传递变量结点

T->ptr[0]->offset = T->offset; //外部变量的偏移量向下传递

T->ptr[1]->type = T->type; //将类型属性向下传递变量结点

T->ptr[1]->offset = T->offset + T->width; //外部变量的偏移量向下传递

T->ptr[1]->width = T->width;

ext\_var\_list(T->ptr[0]);

ext\_var\_list(T->ptr[1]);

T->num = T->ptr[1]->num + 1;

break;

case ID:

rtn = fillSymbolTable(T->type\_id, newAlias(), LEV, T->type, 'V', T->offset); //最后一个变量名

if (rtn == -1)

semantic\_error(T->pos, T->type\_id, "变量重复定义");

else

T->place = rtn;

T->num = 1;

T->width = T->type == CHAR ? 1 : (T->type == INT ? 4 : 8);

break;

case ARRAY\_DEF:

rtn = fillSymbolTable(T->ptr[0]->type\_id, newAlias(), LEV, T->type, 'A', T->offset); //最后一个变量名

if (rtn == -1)

semantic\_error(T->pos, T->type\_id, "数组变量重复定义");

else

T->place = rtn;

T->num = 1;

T->width = T->type == CHAR ? 1 : (T->type == INT ? 4 : 8);

break;

}

}

int match\_param(int i, struct node \*T)

{

int j, num = symbolTable.symbols[i].paramnum;

int type1, type2;

if (num == 0 && T == NULL)

return 1;

for (j = 1; j <= num; j++)

{

if (!T)

{

semantic\_error(T->pos, "", "函数调用参数不足");

return 0;

}

type1 = symbolTable.symbols[i + j].type; //形参类型

type2 = T->ptr[0]->type;

if (type1 != type2)

{

semantic\_error(T->pos, "", "参数类型不匹配");

return 0;

}

T = T->ptr[1];

}

if (T)

{ //num个参数已经匹配完，还有实参表达式

semantic\_error(T->pos, "", "函数调用参数冗余");

return 0;

}

return 1;

}

void boolExp(struct node \*T)

{ //布尔表达式，参考文献[2]p84的思想

struct opn opn1, opn2, result;

int op;

int rtn;

if (T)

{

switch (T->kind)

{

case INT:

if (T->type\_int != 0)

T->code = genGoto(T->Etrue);

else

T->code = genGoto(T->Efalse);

T->width = 0;

break;

case FLOAT:

if (T->type\_float != 0.0)

T->code = genGoto(T->Etrue);

else

T->code = genGoto(T->Efalse);

T->width = 0;

break;

case CHAR:

if (T->type\_char != 0)

T->code = genGoto(T->Etrue);

else

T->code = genGoto(T->Efalse);

T->width = 0;

break;

case ID: //查符号表，获得符号表中的位置，类型送type

rtn = searchSymbolTable(T->type\_id);

if (rtn == -1)

semantic\_error(T->pos, T->type\_id, "变量未定义");

if (symbolTable.symbols[rtn].flag == 'F')

semantic\_error(T->pos, T->type\_id, "是函数名，类型不匹配");

else

{

opn1.kind = ID;

strcpy(opn1.id, symbolTable.symbols[rtn].alias);

opn1.offset = symbolTable.symbols[rtn].offset;

opn2.kind = INT;

opn2.const\_int = 0;

result.kind = ID;

strcpy(result.id, T->Etrue);

T->code = genIR(NEQ, opn1, opn2, result);

T->code = merge(2, T->code, genGoto(T->Efalse));

}

T->width = 0;

break;

case RELOP: //处理关系运算表达式,2个操作数都按基本表达式处理

T->ptr[0]->offset = T->ptr[1]->offset = T->offset;

Exp(T->ptr[0]);

T->width = T->ptr[0]->width;

Exp(T->ptr[1]);

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

opn1.kind = ID;

strcpy(opn1.id, symbolTable.symbols[T->ptr[0]->place].alias);

opn1.offset = symbolTable.symbols[T->ptr[0]->place].offset;

opn2.kind = ID;

strcpy(opn2.id, symbolTable.symbols[T->ptr[1]->place].alias);

opn2.offset = symbolTable.symbols[T->ptr[1]->place].offset;

result.kind = ID;

strcpy(result.id, T->Etrue);

if (strcmp(T->type\_id, "<") == 0)

op = JLT;

else if (strcmp(T->type\_id, "<=") == 0)

op = JLE;

else if (strcmp(T->type\_id, ">") == 0)

op = JGT;

else if (strcmp(T->type\_id, ">=") == 0)

op = JGE;

else if (strcmp(T->type\_id, "==") == 0)

op = EQ;

else if (strcmp(T->type\_id, "!=") == 0)

op = NEQ;

T->code = genIR(op, opn1, opn2, result);

T->code = merge(4, T->ptr[0]->code, T->ptr[1]->code, T->code, genGoto(T->Efalse));

break;

case AND:

case OR:

if (T->kind == AND)

{

strcpy(T->ptr[0]->Etrue, newLabel());

strcpy(T->ptr[0]->Efalse, T->Efalse);

}

else

{

strcpy(T->ptr[0]->Etrue, T->Etrue);

strcpy(T->ptr[0]->Efalse, newLabel());

}

strcpy(T->ptr[1]->Etrue, T->Etrue);

strcpy(T->ptr[1]->Efalse, T->Efalse);

T->ptr[0]->offset = T->ptr[1]->offset = T->offset;

boolExp(T->ptr[0]);

T->width = T->ptr[0]->width;

boolExp(T->ptr[1]);

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

if (T->kind == AND)

T->code = merge(3, T->ptr[0]->code, genLabel(T->ptr[0]->Etrue), T->ptr[1]->code);

else

T->code = merge(3, T->ptr[0]->code, genLabel(T->ptr[0]->Efalse), T->ptr[1]->code);

break;

case NOT:

printf("NOT %s", T->type\_id);

if (T->type\_id != "int" || T->ptr[0]->type\_id != "float")

semantic\_error(T->pos, "按位取反", "参数类型错误");

strcpy(T->ptr[0]->Etrue, T->Efalse);

strcpy(T->ptr[0]->Efalse, T->Etrue);

boolExp(T->ptr[0]);

T->code = T->ptr[0]->code;

break;

}

}

}

void Exp(struct node \*T)

{ //处理基本表达式，参考文献[2]p82的思想

int rtn, num, width;

struct node \*T0;

struct opn opn1, opn2, result;

if (T)

{

switch (T->kind)

{

case ID: //查符号表，获得符号表中的位置，类型送type

rtn = searchSymbolTable(T->type\_id);

if (rtn == -1)

semantic\_error(T->pos, T->type\_id, "变量未定义");

if (symbolTable.symbols[rtn].flag == 'F')

semantic\_error(T->pos, T->type\_id, "是函数名，类型不匹配");

if (symbolTable.symbols[rtn].flag == 'A')

semantic\_error(T->pos, T->type\_id, "是数组名，类型不匹配");

else

{

T->place = rtn; //结点保存变量在符号表中的位置

// T->code = NULL; //标识符不需要生成TAC

T->type = symbolTable.symbols[rtn].type;

T->offset = symbolTable.symbols[rtn].offset;

T->width = 0; //未再使用新单元

}

break;

case ARRAY\_DEF: //查符号表，获得符号表中的位置，类型送type

rtn = searchSymbolTable(T->type\_id);

if (rtn == -1)

semantic\_error(T->pos, T->type\_id, "变量未定义");

if (symbolTable.symbols[rtn].flag == 'F')

semantic\_error(T->pos, T->type\_id, "是函数名，类型不匹配");

if (symbolTable.symbols[rtn].flag == 'V')

semantic\_error(T->pos, T->type\_id, "是变量名，类型不匹配");

else

{

T->place = rtn; //结点保存变量在符号表中的位置

// T->code = NULL; //标识符不需要生成TAC

T->type = symbolTable.symbols[rtn].type;

T->offset = symbolTable.symbols[rtn].offset;

T->width = 0; //未再使用新单元

}

break;

case INT:

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset); //为整常量生成一个临时变量

T->type = INT;

T->width = 4;

break;

case FLOAT:

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset); //为浮点常量生成一个临时变量

T->type = FLOAT;

T->width = 8;

break;

case CHAR:

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset); //为浮点常量生成一个临时变量

T->type = CHAR;

T->width = 1;

break;

case ASSIGNOP:

if (T->ptr[0]->kind != ID)

{

semantic\_error(T->pos, "", "赋值语句需要左值");

}

else

{

if (T->ptr[0]->type == ID)

{

rtn = searchSymbolTable(T->ptr[0]->type\_id);

if (rtn == -1)

{

semantic\_error(T->pos, T->type\_id, "变量未定义");

break;

}

}

Exp(T->ptr[0]); //处理左值，例中仅为变量

T->ptr[1]->offset = T->offset;

Exp(T->ptr[1]);

if (T->ptr[0]->type != T->ptr[1]->type)

{

semantic\_error(T->pos, "", "赋值号两边表达式类型不匹配");

}

T->type = T->ptr[0]->type;

T->width = T->ptr[1]->width;

}

break;

case AND:

case OR:

case RELOP:

T->type = INT;

T->ptr[0]->offset = T->ptr[1]->offset = T->offset;

Exp(T->ptr[0]);

Exp(T->ptr[1]);

break;

case PLUS:

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

Exp(T->ptr[1]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if ((T->ptr[0]->type == FLOAT && T->ptr[1]->type == INT) || (T->ptr[0]->type == INT && T->ptr[1]->type == FLOAT))

T->type = FLOAT, T->width = T->ptr[0]->width + T->ptr[1]->width + 4;

else if (T->ptr[0]->type == INT && T->ptr[1]->type == INT)

T->type = INT, T->width = T->ptr[0]->width + T->ptr[1]->width + 2;

else

semantic\_error(T->pos, "PLUS ", "两加数类型不兼容");

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset + T->ptr[0]->width + T->ptr[1]->width);

T->width = T->ptr[0]->width + T->ptr[1]->width + (T->type == CHAR ? 1 : (T->type == INT ? 4 : 8));

break;

case MINUS:

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

Exp(T->ptr[1]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if ((T->ptr[0]->type == FLOAT && T->ptr[1]->type == INT) || (T->ptr[0]->type == INT && T->ptr[1]->type == FLOAT))

T->type = FLOAT, T->width = T->ptr[0]->width + T->ptr[1]->width + 4;

else if (T->ptr[0]->type == INT && T->ptr[1]->type == INT)

T->type = INT, T->width = T->ptr[0]->width + T->ptr[1]->width + 2;

else

semantic\_error(T->pos, "MINUS ", "两减数类型不兼容");

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset + T->ptr[0]->width + T->ptr[1]->width);

T->width = T->ptr[0]->width + T->ptr[1]->width + (T->type == CHAR ? 1 : (T->type == INT ? 4 : 8));

break;

case STAR:

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

Exp(T->ptr[1]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if ((T->ptr[0]->type == FLOAT && T->ptr[1]->type == INT) || (T->ptr[0]->type == INT && T->ptr[1]->type == FLOAT))

T->type = FLOAT, T->width = T->ptr[0]->width + T->ptr[1]->width + 4;

else if (T->ptr[0]->type == INT && T->ptr[1]->type == INT)

T->type = INT, T->width = T->ptr[0]->width + T->ptr[1]->width + 2;

else

semantic\_error(T->pos, "STAR ", "两乘数类型不兼容");

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset + T->ptr[0]->width + T->ptr[1]->width);

T->width = T->ptr[0]->width + T->ptr[1]->width + (T->type == CHAR ? 1 : (T->type == INT ? 4 : 8));

break;

case DIV:

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

Exp(T->ptr[1]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if ((T->ptr[0]->type == FLOAT && T->ptr[1]->type == INT) || (T->ptr[0]->type == INT && T->ptr[1]->type == FLOAT))

T->type = FLOAT, T->width = T->ptr[0]->width + T->ptr[1]->width + 4;

else if (T->ptr[0]->type == INT && T->ptr[1]->type == INT)

T->type = INT, T->width = T->ptr[0]->width + T->ptr[1]->width + 2;

else

semantic\_error(T->pos, "DIV ", "两除数类型不兼容");

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset + T->ptr[0]->width + T->ptr[1]->width);

T->width = T->ptr[0]->width + T->ptr[1]->width + (T->type == CHAR ? 1 : (T->type == INT ? 4 : 8));

break;

case NOT:

case UMINUS:

case FUNC\_CALL: //根据T->type\_id查出函数的定义，如果语言中增加了实验教材的read，write需要单独处理一下

rtn = searchSymbolTable(T->type\_id);

if (rtn == -1)

{

semantic\_error(T->pos, T->type\_id, "函数未定义");

break;

}

if (symbolTable.symbols[rtn].flag != 'F')

{

semantic\_error(T->pos, T->type\_id, "不是一个函数");

break;

}

T->type = symbolTable.symbols[rtn].type;

width = T->type == CHAR ? 1 : (T->type == INT ? 4 : 8); //存放函数返回值的单数字节数

if (T->ptr[0])

{

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]); //处理所有实参表达式求值，及类型

T->width = T->ptr[0]->width + width; //累加上计算实参使用临时变量的单元数

// T->code = T->ptr[0]->code;

}

else

{

T->width = width;

// T->code = NULL;

}

match\_param(rtn, T->ptr[0]); //处理所以参数的匹配

//处理参数列表的中间代码

T0 = T->ptr[0];

while (T0)

{

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T0->ptr[0]->place].alias);

result.offset = symbolTable.symbols[T0->ptr[0]->place].offset;

// T->code = merge(2, T->code, genIR(ARG, opn1, opn2, result));

T0 = T0->ptr[1];

}

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset + T->width - width);

// opn1.kind = ID;

// strcpy(opn1.id, T->type\_id); //保存函数名

// opn1.offset = rtn; //这里offset用以保存函数定义入口,在目标代码生成时，能获取相应信息

// result.kind = ID;

// strcpy(result.id, symbolTable.symbols[T->place].alias);

// result.offset = symbolTable.symbols[T->place].offset;

// T->code = merge(2, T->code, genIR(CALL, opn1, opn2, result)); //生成函数调用中间代码

break;

case ARGS: //此处仅处理各实参表达式的求值的代码序列，不生成ARG的实参系列

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

T->width = T->ptr[0]->width;

// T->code = T->ptr[0]->code;

if (T->ptr[1])

{

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

Exp(T->ptr[1]);

T->width += T->ptr[1]->width;

// T->code = merge(2, T->code, T->ptr[1]->code);

}

break;

}

}

}

//对抽象语法树的先根遍历,按display的控制结构修改完成符号表管理和语义检查和TAC生成（语句部分）

void semantic\_Analysis(struct node \*T)

{

int rtn, num, width;

struct node \*T0;

struct opn opn1, opn2, result;

if (T)

{

switch (T->kind)

{

case EXT\_DEF\_LIST:

if (!T->ptr[0])

break;

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //访问外部定义列表中的第一个

// T->code = T->ptr[0]->code;

if (T->ptr[1])

{

T->ptr[1]->offset = T->ptr[0]->offset + T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]); //访问该外部定义列表中的其它外部定义

// T->code = merge(2, T->code, T->ptr[1]->code);

}

break;

case EXT\_VAR\_DEF: //处理外部说明,将第一个孩子(TYPE结点)中的类型送到第二个孩子的类型域

if (!strcmp(T->ptr[0]->type\_id, "int"))

T->type = T->ptr[1]->type = INT;

else if (!strcmp(T->ptr[0]->type\_id, "float"))

T->type = T->ptr[1]->type = FLOAT;

else

T->type = T->ptr[1]->type = CHAR;

T->ptr[1]->offset = T->offset; //这个外部变量的偏移量向下传递

T->ptr[1]->width = T->type == CHAR ? 1 : (T->type == INT ? 4 : 8); //将一个变量的宽度向下传递

ext\_var\_list(T->ptr[1]); //处理外部变量说明中的标识符序列

T->width = (T->type == CHAR ? 1 : (T->type == INT ? 4 : 8)) \* T->ptr[1]->num; //计算这个外部变量说明的宽度

// T->code = NULL; //这里假定外部变量不支持初始化

break;

case FUNC\_DEF:

if (!strcmp(T->ptr[0]->type\_id, "int")) //获取函数返回类型送到含函数名、参数的结点

T->ptr[1]->type = INT;

else if (!strcmp(T->ptr[0]->type\_id, "float"))

T->ptr[1]->type = FLOAT;

else

T->ptr[1]->type = CHAR;

T->width = 0; //函数的宽度设置为0，不会对外部变量的地址分配产生影响

T->offset = DX; //设置局部变量在活动记录中的偏移量初值

semantic\_Analysis(T->ptr[1]); //处理函数名和参数结点部分，这里不考虑用寄存器传递参数

T->offset += T->ptr[1]->width; //用形参单元宽度修改函数局部变量的起始偏移量

T->ptr[2]->offset = T->offset;

strcpy(T->ptr[2]->Snext, newLabel()); //函数体语句执行结束后的位置属性

semantic\_Analysis(T->ptr[2]); //处理函数体结点

//计算活动记录大小,这里offset属性存放的是活动记录大小，不是偏移

symbolTable.symbols[T->ptr[1]->place].offset = T->offset + T->ptr[2]->width;

// T->code = merge(3, T->ptr[1]->code, T->ptr[2]->code, genLabel(T->ptr[2]->Snext)); //函数体的代码作为函数的代码

break;

case FUNC\_DEC: //根据返回类型，函数名填写符号表

rtn = fillSymbolTable(T->type\_id, newAlias(), LEV, T->type, 'F', 0); //函数不在数据区中分配单元，偏移量为0

if (rtn == -1)

{

semantic\_error(T->pos, T->type\_id, "函数重复定义");

break;

}

else

T->place = rtn;

result.kind = ID;

strcpy(result.id, T->type\_id);

result.offset = rtn;

// T->code = genIR(FUNCTION, opn1, opn2, result); //生成中间代码：FUNCTION 函数名

T->offset = DX; //设置形式参数在活动记录中的偏移量初值

if (T->ptr[0])

{ //判断是否有参数

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //处理函数参数列表

T->width = T->ptr[0]->width;

symbolTable.symbols[rtn].paramnum = T->ptr[0]->num;

// T->code = merge(2, T->code, T->ptr[0]->code); //连接函数名和参数代码序列

}

else

symbolTable.symbols[rtn].paramnum = 0, T->width = 0;

break;

case PARAM\_LIST: //处理函数形式参数列表

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]);

if (T->ptr[1])

{

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]);

T->num = T->ptr[0]->num + T->ptr[1]->num; //统计参数个数

T->width = T->ptr[0]->width + T->ptr[1]->width; //累加参数单元宽度

// T->code = merge(2, T->ptr[0]->code, T->ptr[1]->code); //连接参数代码

}

else

{

T->num = T->ptr[0]->num;

T->width = T->ptr[0]->width;

// T->code = T->ptr[0]->code;

}

break;

case PARAM\_DEC:

rtn = fillSymbolTable(T->ptr[1]->type\_id, newAlias(), 1, T->ptr[0]->type, 'P', T->offset);

if (rtn == -1)

semantic\_error(T->ptr[1]->pos, T->ptr[1]->type\_id, "参数名重复定义");

else

T->ptr[1]->place = rtn;

T->num = 1; //参数个数计算的初始值

T->width = T->ptr[0]->type == CHAR ? 1 : (T->type == INT ? 4 : 8); //参数宽度

result.kind = ID;

strcpy(result.id, symbolTable.symbols[rtn].alias);

result.offset = T->offset;

// T->code = genIR(PARAM, opn1, opn2, result); //生成：FUNCTION 函数名

break;

case COMP\_STM:

LEV++;

//设置层号加1，并且保存该层局部变量在符号表中的起始位置在symbol\_scope\_TX

symbol\_scope\_TX.TX[symbol\_scope\_TX.top++] = symbolTable.index;

T->width = 0;

// T->code = NULL;

if (T->ptr[0])

{

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //处理该层的局部变量DEF\_LIST

T->width += T->ptr[0]->width;

// T->code = T->ptr[0]->code;

}

if (T->ptr[1])

{

T->ptr[1]->offset = T->offset + T->width;

strcpy(T->ptr[1]->Snext, T->Snext); //S.next属性向下传递

semantic\_Analysis(T->ptr[1]); //处理复合语句的语句序列

T->width += T->ptr[1]->width;

// T->code = merge(2, T->code, T->ptr[1]->code);

}

prn\_symbol(); //c在退出一个符合语句前显示的符号表

LEV--; //出复合语句，层号减1

symbolTable.index = symbol\_scope\_TX.TX[--symbol\_scope\_TX.top]; //删除该作用域中的符号

break;

case DEF\_LIST:

// T->code = NULL;

if (T->ptr[0])

{

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //处理一个局部变量定义

// T->code = T->ptr[0]->code;

T->width = T->ptr[0]->width;

}

if (T->ptr[1])

{

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]); //处理剩下的局部变量定义

// T->code = merge(2, T->code, T->ptr[1]->code);

T->width += T->ptr[1]->width;

}

break;

case VAR\_DEF: //处理一个局部变量定义,将第一个孩子(TYPE结点)中的类型送到第二个孩子的类型域

//类似于上面的外部变量EXT\_VAR\_DEF，换了一种处理方法

// T->code = NULL;

if (!strcmp(T->ptr[0]->type\_id, "int")) //确定变量序列各变量类型

T->ptr[1]->type = INT;

else if (!strcmp(T->ptr[0]->type\_id, "float"))

T->ptr[1]->type = FLOAT;

else

T->ptr[1]->type = CHAR;

T0 = T->ptr[1]; //T0为变量名列表子树根指针，对ID、ASSIGNOP类结点在登记到符号表，作为局部变量

num = 0;

T0->offset = T->offset;

T->width = 0;

width = T->ptr[1]->type == CHAR ? 1 : (T->type == INT ? 4 : 8); //一个变量宽度

while (T0)

{ //处理所以DEC\_LIST结点

num++;

T0->ptr[0]->type = T0->type; //类型属性向下传递

if (T0->ptr[1])

T0->ptr[1]->type = T0->type;

T0->ptr[0]->offset = T0->offset; //类型属性向下传递

if (T0->ptr[1])

T0->ptr[1]->offset = T0->offset + width;

if (T0->ptr[0]->kind == ID)

{

rtn = fillSymbolTable(T0->ptr[0]->type\_id, newAlias(), LEV, T0->ptr[0]->type, 'V', T->offset + T->width); //此处偏移量未计算，暂时为0

if (rtn == -1)

semantic\_error(T0->ptr[0]->pos, T0->ptr[0]->type\_id, "变量重复定义");

else

T0->ptr[0]->place = rtn;

T->width += width;

}

if (T0->ptr[0]->kind == ARRAY\_DEF)

{

rtn = fillSymbolTable(T0->ptr[0]->type\_id, newAlias(), LEV, T0->ptr[0]->type, 'A', T->offset + T->width); //此处偏移量未计算，暂时为0

if (rtn == -1)

semantic\_error(T0->ptr[0]->pos, T0->ptr[0]->type\_id, "数组变量变量重复定义");

else

T0->ptr[0]->place = rtn;

T->width += width;

}

else if (T0->ptr[0]->kind == ASSIGNOP)

{

rtn = fillSymbolTable(T0->ptr[0]->ptr[0]->type\_id, newAlias(), LEV, T0->ptr[0]->type, 'V', T->offset + T->width); //此处偏移量未计算，暂时为0

if (rtn == -1)

semantic\_error(T0->ptr[0]->ptr[0]->pos, T0->ptr[0]->ptr[0]->type\_id, "变量重复定义");

else

{

T0->ptr[0]->place = rtn;

T0->ptr[0]->ptr[1]->offset = T->offset + T->width + width;

Exp(T0->ptr[0]->ptr[1]);

opn1.kind = ID;

strcpy(opn1.id, symbolTable.symbols[T0->ptr[0]->ptr[1]->place].alias);

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T0->ptr[0]->place].alias);

// T->code = merge(3, T->code, T0->ptr[0]->ptr[1]->code, genIR(ASSIGNOP, opn1, opn2, result));

}

T->width += width + T0->ptr[0]->ptr[1]->width;

}

T0 = T0->ptr[1];

}

break;

case STM\_LIST:

if (!T->ptr[0])

{

// T->code = NULL;

T->width = 0;

break;

} //空语句序列

if (T->ptr[1]) //2条以上语句连接，生成新标号作为第一条语句结束后到达的位置

strcpy(T->ptr[0]->Snext, newLabel());

else //语句序列仅有一条语句，S.next属性向下传递

strcpy(T->ptr[0]->Snext, T->Snext);

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]);

// T->code = T->ptr[0]->code;

T->width = T->ptr[0]->width;

if (T->ptr[1])

{ //2条以上语句连接,S.next属性向下传递

strcpy(T->ptr[1]->Snext, T->Snext);

T->ptr[1]->offset = T->offset; //顺序结构共享单元方式

semantic\_Analysis(T->ptr[1]);

//序列中第1条为表达式语句，返回语句，复合语句时，第2条前不需要标号

// if (T->ptr[0]->kind == RETURN || T->ptr[0]->kind == EXP\_STMT || T->ptr[0]->kind == COMP\_STM)

// T->code = merge(2, T->code, T->ptr[1]->code);

// else

// T->code = merge(3, T->code, genLabel(T->ptr[0]->Snext), T->ptr[1]->code);

if (T->ptr[1]->width > T->width)

T->width = T->ptr[1]->width; //顺序结构共享单元方式

}

break;

case IF\_THEN:

strcpy(T->ptr[0]->Etrue, newLabel()); //设置条件语句真假转移位置

strcpy(T->ptr[0]->Efalse, T->Snext);

T->ptr[0]->offset = T->ptr[1]->offset = T->offset;

boolExp(T->ptr[0]);

T->width = T->ptr[0]->width;

strcpy(T->ptr[1]->Snext, T->Snext);

semantic\_Analysis(T->ptr[1]); //if子句

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

// T->code = merge(3, T->ptr[0]->code, genLabel(T->ptr[0]->Etrue), T->ptr[1]->code);

break; //控制语句都还没有处理offset和width属性

case IF\_THEN\_ELSE:

strcpy(T->ptr[0]->Etrue, newLabel()); //设置条件语句真假转移位置

strcpy(T->ptr[0]->Efalse, newLabel());

T->ptr[0]->offset = T->ptr[1]->offset = T->ptr[2]->offset = T->offset;

boolExp(T->ptr[0]); //条件，要单独按短路代码处理

T->width = T->ptr[0]->width;

strcpy(T->ptr[1]->Snext, T->Snext);

semantic\_Analysis(T->ptr[1]); //if子句

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

strcpy(T->ptr[2]->Snext, T->Snext);

semantic\_Analysis(T->ptr[2]); //else子句

if (T->width < T->ptr[2]->width)

T->width = T->ptr[2]->width;

// T->code = merge(6, T->ptr[0]->code, genLabel(T->ptr[0]->Etrue), T->ptr[1]->code,

// genGoto(T->Snext), genLabel(T->ptr[0]->Efalse), T->ptr[2]->code);

break;

case WHILE:

strcpy(T->ptr[0]->Etrue, newLabel()); //子结点继承属性的计算

strcpy(T->ptr[0]->Efalse, T->Snext);

T->ptr[0]->offset = T->ptr[1]->offset = T->offset;

boolExp(T->ptr[0]); //循环条件，要单独按短路代码处理

T->width = T->ptr[0]->width;

strcpy(T->ptr[1]->Snext, newLabel());

semantic\_Analysis(T->ptr[1]); //循环体

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

// T->code = merge(5, genLabel(T->ptr[1]->Snext), T->ptr[0]->code,

// genLabel(T->ptr[0]->Etrue), T->ptr[1]->code, genGoto(T->ptr[1]->Snext));

break;

case EXP\_STMT:

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]);

// T->code = T->ptr[0]->code;

T->width = T->ptr[0]->width;

break;

case RETURN:

if (T->ptr[0])

{

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

num = symbolTable.index;

do

num--;

while (symbolTable.symbols[num].flag != 'F');

if (T->ptr[0]->type != symbolTable.symbols[num].type)

{

semantic\_error(T->pos, "返回值类型错误", "");

T->width = 0;

// T->code = NULL;

break;

}

T->width = T->ptr[0]->width;

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T->ptr[0]->place].alias);

result.offset = symbolTable.symbols[T->ptr[0]->place].offset;

// T->code = merge(2, T->ptr[0]->code, genIR(RETURN, opn1, opn2, result));

}

else

{

T->width = 0;

result.kind = 0;

// T->code = genIR(RETURN, opn1, opn2, result);

}

break;

case ID:

case ARRAY\_DEF:

case INT:

case FLOAT:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

case NOT:

case UMINUS:

case FUNC\_CALL:

Exp(T); //处理基本表达式

break;

}

}

}

void semantic\_Analysis0(struct node \*T)

{

symbolTable.index = 0;

/\* fillSymbolTable("read", "", 0, INT, 'F', 4);

symbolTable.symbols[0].paramnum = 0; //read的形参个数

fillSymbolTable("x", "", 1, INT, 'P', 12);

fillSymbolTable("write", "", 0, INT, 'F', 4);

symbolTable.symbols[2].paramnum = 1;\*/

symbol\_scope\_TX.TX[0] = 0; //外部变量在符号表中的起始序号为0

symbol\_scope\_TX.top = 1;

T->offset = 0; //外部变量在数据区的偏移量

semantic\_Analysis(T);

}

**附录一 GNU Flex 工具说明**

使用工具Flex生成词法分析程序时，按照其规定的格式，生成一个Flex文件，Flex的文件扩展名为.l的文本文件，假定为lex.l，其格式为：

定义部分

%%

规则部分

%%

用户子程序部分

这里被%%分隔开的三个部分都是可选的，没有辅助过程时，第2个%%可以省略。

第一个部分为定义部分，其中可以有一个%{ 到%}的区间部分，主要包含c语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明。会直接被复制到词法分析器源程序lex.yy.c中。%{ 到%}之外的部分是一些正规式宏名的定义，这些宏名在后面的规则部分会用到。

第二个部分为规则部分，一条规则的组成为：

正规表达式 动作

表示词法分析器一旦识别出正规表达式所对应的单词，就执行动作所对应的操作，返回单词的种类码。在这里可写代码显示（种类编码，单词的自身值），观察词法分析每次识别出来的单词，作为实验检查的依据。

词法分析器识别出一个单词后，将该单词对应的字符串保存在yytext中，其长度为yyleng，

第三个部分为用户子程序部分，这部分代码会原封不动的被复制到词法分析器源程序lex.yy.c中。

**附录二 GNU Bison 工具说明**

使用工具Bison生成语法分析程序时，其格式为：

%{

声明部分

%}

辅助定义部分

%%

规则部分

%%

用户函数部分

声明部分：

%{到%}间的声明部分内容包含语法分析中需要的头文件包含，宏定义和全局变量的定义等，这部分会直接被复制到语法分析的C语言源程序中。

辅助定义部分：

在实验中要用到的几个主要内容有：

（1）语义值的类型定义，mini-c的文法中，有终结符，如ID表示的标识符，INT表示的整常数，IF表示关键字if，WHILE表示关键字while等；同时也有非终结符，如ExtDefList表示外部定义列表，CompSt表示复合语句等。每个符号（终结符和非终结符）都会有一个属性值，这个值的类型默认为整型。实际运用中，值得类型会有些差异，如ID的属性值类型是一个字符串，INT的属性值类型是整型。在下一节会介绍，语法分析时，需要建立抽象语法树，这时ExtDefList的属性值类型会是树结点（结构类型）的指针。这样各种符号就会对应不同类型，这时可以用联合将这多种类型统一起来：

%union {

. . . . . .

}

将所有符号的属性值类型用联合的方式统一起来后，某个符号的属性值就是联合中的一个成员的值。

（2）终结符定义，在Flex和Bison联合使用时，parser.y如何使用lex.l中识别出的单词的种类码？这时需要做的是在parser.y中的%token后面罗列出所有终结符(单词)的种类码标识符，如：

%token ID，INT，IF，ELSE，WHILE

这样就完成了定义终结符ID、INT、IF、ELSE、WHILE。接着可使用命令：bison -d parser.y 对语法分析的Bison文件parser.y进行翻译，当使用参数-d时，除了会生成语法分析器的c语言源程序文件parser.tab.c外，还会生成一个头文件parser.tab.h，在该头文件中，将所有的这些终结符作为枚举常量，从258开始，顺序编号。这样在lex.l中，使用宏命令 #include “parser.tab.h”，就可以使用这些枚举常量作为终结符（单词）的种类码返回给语法分析程序，语法分析程序接收到这个种类码后，就完成了读取一个单词。

（3）非终结符的属性值类型说明，对于非终结符，如果需要完成语义计算时，会涉及到非终结符的属性值类型，这个类型来源于（1）中联合的某个成员，可使用格式：%type <union的成员名> 非终结符。例如parser.y中的：

%type <ptr> program ExtDefList

这表示非终结符ExtDefList属性值的类型对应联合中成员ptr的类型，在本实验中对应一个树结点的指针。

（4）优先级与结合性定义。对Bison文件进行翻译，得到语法分析分析程序的源程序时，通常会出现报错，大部分是移进和归约(shift/reduce)，归约和归约(reduce /reduce)的冲突类的错误。为了改正这些错误，需要了解到底什么地方发生错误，这是，需要在翻译命令中，加上一个参数-v，即命令为： ：bison -d -v parser.y 这时，会生成一个文件parser.output。打开该文件，开始几行说明（LALR（1）分析法）哪几个状态有多少个冲突项，再根据这个说明中的状态序号去查看对应的状态进行分析、解决错误，常见的错误一般都能通过单词优先级的设定解决，例如对表达式Exp，其部分文法规则有：

Exp →Exp + Exp | Exp - Exp | Exp \* Exp | Exp \* Exp

在文法介绍时，明确过该文法是二义性的，这样对于句子a+b\*c，到了符号\*时，可能的操作一个是移进\*，一个是对前面的a+b所对应的Exp+Exp进行归约。同样，对于句子a+b+c，读到第二个+号时，是移进，还是把前面的归约？

这样对文件parser.y进行翻译时，会出现移进和归约的冲突，在parser.output文件中，其对应的某个状态会出现说明：

'+' shift, and go to state 16

'-' shift, and go to state 17

'\*' shift, and go to state 18

'/' shift, and go to state 18

'+' [reduce using rule 12 (exp)]

'-' [reduce using rule 12 (exp)]

'\*' [reduce using rule 12 (exp)]

'/' [reduce using rule 12 (exp)]

前面4条表示遇到这些符号要做的操作是移进，后面4条表示遇到这些符号要做的操作是归约，所以产生冲突。这时的解决方法就是通过设定优先级和结合性来实现：

%left '+' '-'

%left '\*' '/'

left表示左结合，前面符号的优先级低。

另外就是对： Exp →-Exp 单目-的运算优先级高于\*与/，而词法分析时，无论是单目-还是双目-，种类码都是MINUS，为此，需要在定义一个优先级搞得单目-符号UMINUS：

%left '+' '-'

%left '\*' '/'

%right UMINUS

相应对其规则处理为：

Exp →-Exp %prec UMINUS

表示这条规则的优先级等同于UMINUS，高于乘除，这样对于句子-a\*b就会先完成-a归约成Exp，即先处理单目-，后处理\*。

**附录三 抽象语法树（AST）**

在语法分析阶段，一个很重要任务就是生成待编译程序的抽象语法树AST，AST不同于推导树，去掉了一些修饰性的单词部分，简明地把程序的语法结构表示出来，后续的语义分析、中间代码生成都可以通过遍历抽象语法树来完成。

例如对语句： while (a>=1) a=a-1;

推导树和抽象语法树分别如图的左右2棵树所示。

 

推导树和抽象语法树的形式

其中，根据处理方式不同，定义的AST形式可能会存在一些差异。但从形式上看，AST简洁很多，这样后续处理相应就方便得多。所以语法分析过程中，尽量不要考虑生成推导树的形式。

为了创建AST，需要对文法的各个符号规定一些属性值，如表所示列出了终结符绑定词法分析得到的值，非终结符绑定AST中对应的树结点指针。

文法符号对应属性

|  |  |  |
| --- | --- | --- |
| **符 号** | **属 性** |  |
| ID | 标识符的字符串 |  |
| INT | 整常数数字 |  |
| FLOAT | 浮点常数数字 |  |
| 所有非终结符 | 抽象语法树的结点指针 |  |
| 其它终结符 | 可忽略 |  |

由上表可见，不同的符号绑定的属性值的类型不一定相同。例如，词法分析器识别出一个正常数123，返回的单词种类码INT，同时INT对应的终结符要对应一个单词自身值(整数123)。

为了将要用到的各种非终结符和终结符的类型统一在一个类型下，如2.2.2中已叙述可采用联合（共用体）这个类型。例如在lex.l和parser.y中，同时定义：

typedef union {

int type\_int;

int type\_float;

char type\_id[32];

struct node \*ptr;

......

} YYLVAL;

这样所有符号的属性值的类型就是联合类型YYLVAL了。如何实现不同符号绑定不同的成员哪？ 对终结符，可采用：%token <type\_int> INT，表示INT的属性值对应联合的成员type\_int。例如在lex.l的词法分析中，识别到整常数后，在返回给语法分析器一个整常数的种类码INT的同时，通过yylval.type\_int=atoi(yytext);将整常数的值保存在yylval的成员type\_in中，这里yylval是一个Flex和Bison共用的内部变量，类型为YYLVAL，按这样的方式，在Flex中通过yylval的成员保存单词属性值，在Bison中就可以通过yylval的成员取出属性值，实现了数据的传递。由于已经建立了绑定关系，语法分析的规则部分的语义处理时，通过终结符INT绑定的属性值可直接取到这个常数值。比如对规则：Exp →INT， 由于终结符INT是规则右部的第一个符号，就可通过$1简单方便地取到识别出的整常数值，不必采用yylval.type\_int的形式提取这个整常数值。

同样可采用：%token <type\_id> ID，表示识别出来一个标识符后，标识符的字符串串值保存在成员type\_id中。对非终结符，如果采用%type <ptr> Exp，表示Exp对应绑定成员ptr，即Exp的属性值是一个结点的指针。

在parser.y中处理AST时，所有结点的类型也是统一的，所以为区分结点的属性，在定义结点时，要有一个属性kind，用以标识结点类型，明确结点中存放了哪些有意义的信息。结点定义参考附录3中node的定义。

AST的逻辑结构就是一棵多叉树，下面就需要考虑其物理结构，这个就需要灵活地运用数据结构课程的知识，可采用：（1）孩子表示法，本指导书中，基于简明以及让读者理解方便的原则，采用的就是结点大小固定的孩子表示法，每个结点有3个指针域，可指向3棵子树。由结点类型kind确定有多少棵子树，显然这会有很多空指针域。如果已经掌握了C++，利用类的封装，继承与多态来定义结点会更好。（2）孩子兄弟表示法（二叉链表），这种方法存储效率要高一些，实现时要清楚结点之间的关系的转换。

在parser.y中，在完成归约的过程中，完成抽象语法树的构造。例如处理下面规则，即完成将INT归约成Exp时。

Exp ： INT

需要调用函数mknode生成一个类型为INT的叶结点，指针赋值给Exp的属性$$（$$=mknode(INT,NULL,NULL,NULL,yylineno)）;，同时将INT的属性值$1（一个整常数）写到结点中type\_int成员域保存（$$->type\_int=$1;）。

当处理下面规则，即完成将Exp1+Exp2归约成Exp时。

Exp ： Exp1 PLUS Exp2

需要调用函数mknode生成一个类型为PLUS的非叶子结点，结点指针赋值给Exp的属性$$（$$=mknode(PLUS,$1,$3,NULL,yylineno);）。将Exp1表示的树$1作为Exp的第一棵子树，将Exp2表示的树$3作为Exp的第二棵子树。

如果没有语法错误，最后归约到了文法开始符号，这样就可以获得抽象语法树的根结点指针。再调用display以缩进编排的格式进行显示AST。

**附录四 作用域与符号表操作**

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量LEV来管理，LEV的初始值为0。这样在处理外部变量名，以及函数名时，对应符号的层号值都是1；处理函数形式参数时，固定形参名在填写符号表时，层号为1。由于mini\_C中允许有复合语句，复合语句中可定义局部变量，函数体本身也是一个复合语句，这样在AST的遍历中，通过LEV的修改来管理不同的作用域。

（1）每次遇到一个复合语句的结点COM\_STM，首先对LEV加1，表示准备进入一个新的作用域，为了管理这个作用域中的变量，使用栈symbol\_scope\_TX，记录该作用域变量在符号表中的起点位置，即将符号表symbolTable的栈顶位置symbolTable.index保存在栈symbol\_scope\_TX中。

（2）每次要登记一个新的符号到符号表中时，首先在symbolTable中，从栈顶向栈底方向查层号为LEV的符号是否和当前待登记的符号重名，是则报重复定义错误，否则使用LEV作为层号将新的符号登记到符号表中。

（3） 每次遍历完一个复合语句的结点COM\_STM的子树，准备回到其父结点时，这时该复合语句语义分析完成，需要从符号表中删除该复合语句的变量，方法是首先symbol\_scope\_TX退栈，取出该复合语句作用域的起点，再根据这个值修改symbolTable.index，很简单地完成了符号表的符号删除操作。

（4）符号表的查找操作，在AST的遍历过程中，当分析各种表达式，遇到变量的访问时，在symbolTable中，从栈顶向栈底方向查询是否有相同的符号定义，如果全部查询完后没有找到，就是该符号没有定义；如果相同符号在符号表中有多处定义，按查找的方向可知，符合就近优先的原则。如果查找到符号后，就进一步进行语义分析，如：（1）函数调用时，根据函数名在符号表找到的是一个变量，不是函数，需要报错；（2）函数调用时，根据函数名找到这个函数，需要判断参数个数、类型是否匹配；（3）根据变量名找的的是一个函数。等等，需要做出各种检查。