

**课 程 实验 报 告**

**题目： Mini-C编译器设计与实现**

**课程名称： 编译原理实验**

**专业班级： ACM1701**

**学 号： U201714780**

**姓 名： 刘晨彦**

**指导教师： 刘海坤**

**报告日期： 2020年6月19日**

**计算机科学与技术学院**

**目录**

目录

[1. 概述 1](#_Toc43709014)

[2. 系统描述 2](#_Toc43709015)

[2.1. 自定义语言概述 2](#_Toc43709016)

[2.2. 单词文法与语言文法 2](#_Toc43709017)

[2.3. 符号表结构定义 2](#_Toc43709018)

[2.4. 错误类型码定义 3](#_Toc43709019)

[2.5. 中间代码结构定义 3](#_Toc43709020)

[2.6. 目标代码指令集选择 4](#_Toc43709021)

[3. 系统设计与实现 5](#_Toc43709022)

[3.1. 词法分析器 5](#_Toc43709023)

[3.2. 语法分析器 6](#_Toc43709024)

[3.3. 符号表管理 6](#_Toc43709025)

[3.4. 语义检查 6](#_Toc43709026)

[3.5. 报错功能 6](#_Toc43709027)

[3.6. 中间代码生成 6](#_Toc43709028)

[3.7. 代码优化 6](#_Toc43709029)

[3.8. 汇编代码生成 6](#_Toc43709030)

[4. 系统测试与评价 7](#_Toc43709031)

[4.1. 测试用例 7](#_Toc43709032)

[4.2. 正确性测试 7](#_Toc43709033)

[4.3. 报错功能测试 7](#_Toc43709034)

[4.4. 系统的优点 7](#_Toc43709035)

[4.5. 系统的缺点 7](#_Toc43709036)

[5. 实验小结或体会 8](#_Toc43709037)

[6. 参考文献 9](#_Toc43709038)

**[附件：源代码](#_Toc43709039)** [10](#_Toc43709039)

# 概述

本次实验是构造一个高级语言的子集的编译器，目标代码是汇编语言。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喜好选择实现。建议大家选用decaf语言或C语言的简单集合SC语言。

实验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，提高学生系统软件研发技术。

# 系统描述

## 自定义语言概述

自定义语言是基于C语言的简化语言，具有C语言中最为基础的内容，包括基本的运算操作、if-else、for、while循环语句，自定义函数和变量等。

## 单词文法与语言文法

Mini-C的文法如下：

G[program]:

program → ExtDefList

ExtDefList → ExtDef ExtDefList | ε

ExtDef → Specifier ExtDecList ;| SpecifierFunDecCompSt

Specifier → int | float | char | StructSpecifier

ExtDecList → VarDec | VarDec , ExtDecList

VarDec → ID | VarDec ( Exp )

FucDec → ID ( VarList ) | ID ()

VarList → ParamDec ,VarList | ParamDec

ParamDec → Specifier VarDec

CompSt → { DefList StmList }

StmList → Stmt StmList | ε

Stmt → Exp ; | CompSt | returnExp ; | FOR ( Exp ; Exp ; Exp ) Stmt | if ( Exp )Stmt | if( Exp ) Stmt else Stmt| while ( Exp ) Stmt

DefList → Def DefList | ε

Def →Specifier DecList ;

DecList → Dec | Dec ,DecList

Dec → VarDec | VarDec =Exp

Exp → Exp=Exp | Exp && Exp | Exp | |Exp | Exp < Exp| Exp <= Exp | Exp == Exp | Exp != Exp | Exp > Exp | Exp >= Exp | Exp +Exp| Exp -Exp | Exp \*Exp | Exp /Exp | INT | FLOAT | CHAR | VarDec | Exp ++ | Exp -- | ++ Exp | --Exp | VarDec += Exp | VarDec -= Exp | VarDec \*= Exp | VarDec /= Exp | BREAK | CONTINUE | Exp . ID | ( Exp ) | -Exp | ! Exp | ID ( Args ) | ID ()

Args → Exp , Args | Exp

StructSpecifier → STRUCT StructName { ExtDefList } | STRUCT ID

StructName → ID | ε

## 符号表结构定义

符号表采用顺序表结构，符号表作为顺序栈记录符号。使用LEV全局变量和栈symbol\_scope\_TX来记录每个作用域在符号表的起始序号。

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。

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，是否有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。

每次遍历完一个复合语句结点COM\_STM的所有子树，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，同时LEV减一，很简单地完成了符号表的符号删除操作。

对符号表的查找，只需从当前符号表从上向下遍历，直到超出当前作用域为止。找到变量返回其序号，否则返回-1。程序可以根据序号重新在符号表中查询该变量的更多信息，包括其类型是函数或变量等信息，用于检查静态语义错误。

## 错误类型码定义

支持的静态错误检测如下所示：

1. 定义变长数组
2. 同一作用域内的变量重复定义
3. 同一作用域内的函数重复定义
4. 函数调用参数数量少于/多于函数定义
5. 函数调用参数类型不匹配函数定义
6. 函数定义内参数名重复定义
7. 函数返回值类型错误
8. 函数无返回值
9. 使用未定义的变量
10. 使用未定义的函数
11. 对函数名采用非函数调用形式
12. 对非函数名采用函数调用形式
13. 数组变量采用非整型表达式
14. 对非数组变量采取下标变量的形式访问
15. 赋值语句无左值
16. 自增自减运算语句无左值
17. 运算符两端类型不匹配
18. 定义数组变量的同时赋值
19. 在循环语句外使用break语句
20. 在循环语句外使用continue语句

## 中间代码结构定义

使用三地址码TAC作为中间语言，其中包含（op、opn1、opn2、result），其中op表示操作类型说明，opn1和opn2表示2个操作数，result表示运算结果。TAC代码序列将使用循环链表进行连接。

中间语言代码的定义如下表所示：

表 2.1：中间代码定义

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 语法 | 描述 | Op | Opn1 | Opn2 | Result |
| x := y | 赋值操作 | ASSIGNOP | y |  | x |
| x := x ++ | 递增操作 | INC | x |  | x |
| x := x -- | 递减操作 | DEC | x |  | x |
| -x | 取反操作 | UMINUS | x |  | x |
| x := x + y | 自增操作 | SELFPLUS | x | y | x |
| x := x - y | 自减操作 | SELFMINUS | x | y | x |
| x := x \* y | 自乘操作 | SELFSTAR | x | y | x |
| x := x / y | 自除操作 | SELFDIV | x | y | x |
| x := y + z | 加法操作 | PLUS | y | z | x |
| x := y - z | 减法操作 | MINUS | y | z | x |
| x := y \* z | 乘法操作 | STAR | y | z | x |
| x := y / z | 除法操作 | DIV | y | z | x |
| FUNCTION f: | 定义函数f | FUNCTION |  |  | f |
| PARAM x | 函数形参 | PARAM |  |  | x |
| LABEL x | 定义标签x | LABEL |  |  | x |
| GOTO x | 无条件转移 | GOTO |  |  | x |
| IF x [relop] y GOTO z | 条件转移 | [relop] | x | y | z |
| ARG x | 传递实参x | ARG |  |  | x |
| CALL f | 调用函数(无返回值) | CALL | f |  |  |
| x := CALL f | 调用函数(有返回值) | CALL | f |  | x |
| RETURN x | 返回语句 | REUTRN |  |  | x |

## 目标代码指令集选择

目标代码指令集选用的是MIPS32指令序列。

# 系统设计与实现

## 词法分析器

词法分析在文件lex.l中实现。在规则部分使用正则表达式进行识别，并执行对应操作。在该部分中支持识别的词法有：

表 3.1：支持的词法

|  |  |  |
| --- | --- | --- |
| 类型 | 词法 | 正则语句 |
| 常量 | {int} | [0-9]+ |
| {float} | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| {char} | '.' |
| {id} | [A-Za-z][A-Za-z0-9]\* |
| 保留字 | int | “int” |
| float | “float” |
| char | “char” |
| return | “return” |
| if | “if” |
| else | “else” |
| while | “while” |
| for | “for” |
| break | “break” |
| continue | “continue” |
| switch | “switch” |
| case | “case” |
| default | “default” |
| struct | “struct” |
| 界符 | ; | “;” |
| , | “,” |
| : | “:” |
| . | “.” |
| ( | “(” |
| ) | “)” |
| [ | “[” |
| ] | “]” |
| { | “{” |
| } | “}” |
| 运算符 | >, <, >=, <=, ==, != | ">"|"<"|">="|"<="|"=="|"!=" |
| + | “+” |
| - | “-” |
| \* | “\*” |
| / | “/” |
| ++ | “++” |
| -- | “--" |
| += | “+=” |
| -= | “-=” |
| \*= | “\*=” |
| /= | “/=” |
| = | “=” |
| && | “&&” |
| || | “||” |
| ! | “!” |
| 转义字符 | \n | [\n] |
| \r\t | [\r\t] |
| 注释 | 单行注释 | "//"[^\n]\* |
| 多行注释 | "/\*"(.\*|[\n])\*"\*/" |

## 语法分析器

语法分析在parser.y中实现。该部分主要的内容是：添加词法中对应的终结符和非终结符，给出运算符的优先级，给出规则对应的语义动作，并给出在规约完成后需要执行的语义动作，将代码文档转换为一棵抽象语法树。

在语义动作中，添加了如下内容：

VarDec → VarDec LB Exp RB：用于支持数组的定义。通过使用递归的方法，使分析器可以识别任意维数的数组名。对于该节点，有两个子节点，分别是VarDec和Exp，将其类型标记为ARRAY数组类型。当处理该类型的节点时，需要递归查找，直到找到其数组名为止。

Specifier → StructSpecifier：用于支持结构类型的变量定义。在识别变量的类型时，可以通过该语法识别出结构类型。

Stmt → FOR LP Exp SEMI Exp SEMI Exp RP Stmt：用于识别For循环语句，该节点有四个子节点，分别是循环初始化语句、循环判断语句、循环更新语句和循环体。节点类型为FOR。

Exp → CHAR：用于支持char类型的常数作为表达式。

Exp → Exp INC | Exp DEC | INC Exp | DEC Exp：用于支持递增和递减表达式的识别。该节点有唯一的子节点，用于记录递增或递减的对象。节点类型记为INC或DEC。

Exp → VarDec SELFPLUS Exp | VarDec SELFMINUS Exp | VarDec SELFSTAR Exp | VarDec SELFDIV Exp：用于识别自增、自减、自乘、自除表达式的识别。该节点有两个子节点，分别是运算的变量和自增、自减、自乘、自除的量。该节点类型为SELFPLUS或SELFMINUS或SELFSTAR或SELFDIV。

Exp → BREAK：用于识别break语句。该节点无子节点，类型为BREAK。对于BREAK语句只能在循环体中出现的语法定义，不在此处进行约束，而在语义检查中完成。

Exp → CONTINUE：用于识别continue语句。该节点无子节点，类型为CONTINUE。对于CONTINUE语句只能在循环体中出现的语法定义，不在此处进行约束，而在语义检查中完成。

Exp → Exp DOT ID：用于识别结构体变量在表达式中的使用。该节点有两个子节点，分别是DOT左侧的结构体变量名，和DOT右侧的结构内变量名，定义该节点类型为STRUCT\_VISIT，代表该节点需要访问结构体的定义。

StructSpecifier → STRUCT StructName LC ExtDefList RC | STRUCT ID：用于识别结构体变量的定义，包含两种情况：拥有结构体内变量定义的详细定义，和直接使用结构体变量的缺省定义。对于详细定义，节点拥有两个子节点，分别是结构体名和内部定义列表。对于缺省定义，则只有一个用于记录结构体名的子节点。两类定义方式类型均为STRUCT\_DEF。

StructName → ID | ε：用于识别结构体名。结构体名可以缺省。当结构体名不缺省时，记录结构体名，类型为ID。当结构体名缺省时，节点即为叶节点。

通过添加以上语义动作，能够将Mini-C代码文件转化为一棵抽象语法树。通过递归调用，可以打印整棵抽象语法树，并将每个节点内的单词以（单词种类码，自身值）的方式显示。

## 符号表管理

在该部分，通过对符号表的管理记录程序中各类变量的名字和类型信息，用于后期的使用。符号表内包含的信息应包括：变量名、类型、层号、函数的形参个数、变量别名、符号标记（函数：'F' 变量：'V' 参数：'P' 临时变量：'T'）以及变量相对于该作用域的偏移量。

符号表使用顺序表实现，另有一栈symbol\_scope\_TX，用于记录每个作用域中首个变量在顺序表中的序号，用于后期检查同一作用域中“变量或函数是否已经定义”、“变量和函数的错误使用”等静态语义问题。当进入作用域时，全局变量LEV递增1，同时在symbol\_scope\_TX中压栈首个变量在顺序表中的序号。当退出当前作用域时，symbol\_scope\_TX的栈顶出栈，符号表栈顶回退至该退出作用域在符号表中的位置，后续作用域会对之前的作用域内符号进行覆盖。全局变量LEV递减1。

符号表的符号添加：对于抽象语法树，从根节点到叶节点采用递归调用的方式进行遍历。对于同一级的子节点，从左至右遵循：获得该子节点的偏移量，递归调用函数对子节点内的符号进行记录，得到子节点总长度。根据该节点偏移量和总长度，得到下一个子节点的偏移量。对于同一级的首个子节点，其偏移量即为父节点的偏移量。父节点的总长度由自身临时变量的长度和子节点长度之和决定。

符号表的符号查找：对符号表从顶部向当前作用域的底部进行遍历查找。当找到同名变量时，返回其在符号表中的序号。

## 语义检查

各类静态语义检查及实现方式如下：

定义变长数组：在ARRAY类型节点的读取过程中，通过循环，在子树中向下搜索数组名的同时，检查其每一维度类型是否为INT，对于浮点类型、char类型以及变长定义，都会被检查出语义错误并进行报错。

同一作用域内的变量重复定义：在将新的变量加入符号表的同时进行检查，利用函数fillSymbolTable从符号表顶部向下遍历，直到超出当前作用域为止。若找到重名的变量，则返回-1进行报错，否则加入符号表。

同一作用域内的函数重复定义：在新的函数加入符号表的同时进行检查，利用函数fillSymbolTable从符号表顶部向下遍历，直到超出当前作用域为止。若找到重名的函数，则返回-1进行报错，否则加入符号表。

函数调用参数数量少于/多于函数定义：在调用函数时进行检查，程序从符号表中查找到函数定义的参数个数，并比较调用中实际使用的参数个数，若个数不匹配，则报错“数量不符合”。

函数调用参数类型不匹配函数定义：在调用函数时进行检查，程序从符号表中查找到函数定义的各参数类型，并逐个比较调用函数中传入实参的类型，若类型不匹配，则进行报错。

函数定义内参数名重复定义：在函数定义时进行检查，同样是利用fillSymbolTable，检查该虚拟参数是否已经被使用过。若发生重复会导致报错。

函数返回值类型错误：在节点类型为RETURN是进行检查。利用符号表向下检查，找到该函数的类型，并与返回值的类型进行比较，若不一致进行报错。

函数无返回值：在FUNC\_DEF节点就进行检查，在子树中一路向下遍历所有节点，当所有节点都遍历过且无RETURN类型节点时，表示函数没有返回值，进行报错。

使用未定义的变量：在表达式中使用变量的过程中进行检查，使用searchSymbolTable对符号表进行搜索。若未找到该变量，则说明未定义，报错。

使用未定义的函数：在表达式中使用函数的过程中进行检查，使用searchSymbolTable对符号表进行搜索。若未找到该函数，则说明未定义，报错。

对函数名采用非函数调用形式：在表达式使用函数名的过程中检查，使用searchSymbolTable对符号表进行搜索。得到其在符号表的序号后，查看其类型是否为‘F’，若其类型是函数且被当做ID被识别，说明对函数名使用了非函数形式调用，报错。

对非函数名采用函数调用形式：在FUNC\_CALL类型节点中进行检查，使用searchSymbolTable对符号表进行搜索。得到其在符号表的序号后，查看其类型是否为‘F’，若其类型不是函数且被当做FUNC\_CALL被识别，说明对非函数名使用了函数形式调用，报错。

数组变量采用非整型表达式：在ARRAY类型节点中检查。通过循环，在子树中向下搜索数组名的同时，检查其每一维度类型是否为INT，若不是则会进行报错。

对非数组变量采取下标变量的形式访问：在ARRAY类型节点中检查。在符号表中进行搜索，查找其类型是否为‘V’，若是则说明对普通变量采用了下标访问，进行报错。

赋值语句无左值：在ASSIGNOP类型节点里检查，若该节点的左子节点的类型不是变量或数组变量，则进行报错。

自增自减运算语句无左值：在SELFPLUS类节点内进行检查，方式和赋值语句无左值的方法一致。

运算符两端类型不匹配：在运算符类节点进行检查，比较两个子节点类型是否一致，若不一致进行报错。

定义数组变量的同时赋值：在局部变量定义中，赋值节点左侧是数组类型时，说明试图对数组进行赋值，进行报错。

在循环语句外使用break语句：设置全局变量isloop，在WHILE和FOR节点分析进入循环体前递增，在结束循环体后递减。在BREAK类型节点中判断当前isloop变量是否为0，若是说明当前无循环体，报错。否则正常。

在循环语句外使用continue语句：和break语句的检查类似，在CONTINUE类型节点中判断isloop是否为0，若是则说明当前continue不在循环体内，报错。

## 报错功能

报错功能通过函数实现，需要传入错误所在行数、产生错误的变量名和错误信息。该函数会把信息进行打印输出。

## 中间代码生成

在该阶段删除了对数组和结构体的实现。中间代码使用TAC代码，利用单向循环链表进行串接。添加的中间代码翻译如下表所示：

表 3.2：新增的中间代码翻译

|  |  |  |
| --- | --- | --- |
| 节点类 | 节点类型 | 翻译动作 |
| 语句类 | FOR  T0：初始化子树  T1：条件子树  T2：更新语句子树  T3：循环体子树 | 访问到T：T1.Etrue = newlabel(), T1.Efalse = T.Snext, T3.Ebreak = T.Snext, label1 = newlabel(), T2.Snext = newlabel(); T3.Econtinue = label1, T3.Snext = label1  访问T的所有子树后：  T.code = T0.code || T2.Snext: || T1.code || T1.Etrue: || T3.code || label1: || T2.code || goto T2.Snext |
| BREAK | T.code = goto T.Ebreak |
| CONTINUE | T.code = goto T.Econtinue |
| 注：需要将出现在循环语句内的所有类型节点都将Ebreak和Econtinue都向下传递给子树，使其能够获得正确的break位置和continue位置。 | |
| 布尔表达式类 | INT/FLOAT/CHAR | if(常数的值) T.code = goto T.Etrue  else T.code = goto T.Efalse |
| ID | 在符号表中查找ID  T.code = if T.alias != 0 goto T.Etrue || goto T.Efalse |
| ASSIGNOP  T0：左值表达式子树  T1：右值表达式子树 | 访问T的所有子树后：  T.code = T0.code || T1.code || T0.alias = T1.alias  T.code = T.code || if T0.alias != 0 goto T.Etrue || goto T.Efalse |
| OP(PLUS/MINUS/STAR/DIV  SELFPLUS/SELFMINUS/  SELFSTAR/SELFDIV)  T0：第一操作数  T1：第二操作数 | t = newtemp, t的入口赋值给T.place  访问T的所有子树后：  T.code = T0.code || T1.code || t = T0.alias op T1.alias  T.code = T.code || if t != 0 goto T.Etrue || goto T.Efalse |
| INC/DEC  T0：左值表达式子树 | t = newtemp, t的入口赋值给T.place  访问T的所有子树后：  T.code = T0.code || t = T0.alias +/- 1  T.code = T.code || if t != 0 goto T.Etrue || goto T.Efalse |
| UMINUS  T0：左值表达式子树 | t = newtemp, t的入口赋值给T.place  访问T的所有子树后：  T.code = T0.code || t = 0 - T0.alias  T.code = T.code || if t != 0 goto T.Etrue || goto T.Efalse |
| 基本表达式类 | OP(PLUS/MINUS/STAR/DIV  SELFPLUS/SELFMINUS/  SELFSTAR/SELFDIV)  T0：第一操作数  T1：第二操作数 | t = newtemp  访问T的所有子树后：  T.code = T0.code || T1.code || t = T0.alias OP T1.alias |
| INC/DEC  T0：左值表达式子树 | t = newtemp  访问T的所有子树后：  T.code = T0.code || T0.alias = T0.alias +/- 1 || t = T0.alias |
| UMINUS  T0：左值表达式子树 | t = newtemp  访问T的所有子树后：  T.code = T0.code || t = 0 – T0.alias |
| RELOP  T1：第一操作数子树  T2：第二操作数子树 | t = newtemp, label1 = newlabel(), label2 = newlabel()  访问T的所有子树后：  T.code = T0.code || T1.code || if T0.alias RELOP T1.alias goto label1 || t = 0 || goto label2 || label1: || t = 1 || label2: |
| AND  T1：第一操作数子树  T2：第二操作数子树 | t = newtemp, label1 = newlabel(),  访问T的所有子树后：  T.code = T0.code || T1.code || t = T0.alias \* T1.alias || if t == 0 goto label1 || t = 1 || goto label1 |
| OR  T1：第一操作数子树  T2：第二操作数子树 |  |
| NOT  T1：第一操作数子树  T2：第二操作数子树 | t = newtemp, label1 = newlabel(), T.Snext = newlabel()  访问T的所有子树后：  T.code = T0.code || if T0.alias == 0 goto label1 || t = 0 || goto T.Snext || label1: || t = 1 || T.Snext: |

## 汇编代码生成

根据生成的TAC四元组指令，转换为对应的MIPS32指令。使用朴素寄存器分配方式进行翻译，主要使用$t1，$t2和$t3三个寄存器。

首先在首行加入“.text”和“jal main”两条指令，使程序能够转跳至主程序。然后向目标文件写入read和wirte函数以供调用。

表 3.3：汇编代码翻译表

|  |  |  |  |
| --- | --- | --- | --- |
| 操作符类型 | 中间代码 | MIPS指令 | 意义 |
| ASSIGNOP | x := #k | addi $t3,$0,%d  sw $t3, %d($fp) | 常数传递给寄存器t3  传回存储器内x对应位置 |
| x := y | lw $t1, %d($fp)  move $t3, $t1  sw $t3, %d($fp) | 从存储器调出y的值  将值从t1传给寄存器t3  传回存储器内x对应位置 |
| INC | x := x + #1 | lw $t1, %d($fp)  addi $t2, $0,1  add $t3, $t1,$t2  sw $t3, %d($fp) | 从存储器调出x的值  在t2寄存器内保存常数1  计算递增后的结果  传回存储器内x对应位置 |
| DEC | x := x + #-1 | lw $t1, %d($fp)  addi $t2, $0,-1  add $t3, $t1,$t2  sw $t3, %d($fp) | 从存储器调出x的值  在t2寄存器内保存常数-1  计算递减后的结果  传回存储器内x对应位置 |
| UMINUS | x := #0 - y | lw $t3, %d($fp)  sub $t1, $0,$t3  sw $t1, %d($fp) | 从存储器调出y的值  对值取负数  传回存储器内x对应位置 |
| SELFPLUS | x := x + y | lw $t1, %d($fp)  lw $t2, %d($fp)  add $t3, $t2,$t1  sw $t3, %d($fp)  sw $t3, %d($fp) | 从存储器调出x的值  从存储器调出y的值  加法运算  传回存储器内x对应位置  传回临时变量对应位置 |
| SELFMINUS | x := x - y | lw $t1, %d($fp)  lw $t2, %d($fp)  sub $t3, $t2,$t1  sw $t3, %d($fp)  sw $t3, %d($fp) | 从存储器调出x的值  从存储器调出y的值  减法运算  传回存储器内x对应位置  传回临时变量对应位置 |
| SELFSTAR | x := x \* y | lw $t1, %d($fp)  lw $t2, %d($fp)  mul $t3, $t2,$t1  sw $t3, %d($fp)  sw $t3, %d($fp) | 从存储器调出x的值  从存储器调出y的值  乘法运算  传回存储器内x对应位置  传回临时变量对应位置 |
| SELFDIV | x := x / y | lw $t1, %d($fp)  lw $t2, %d($fp)  div $t1,$t2  mflo $t3  sw $t3, %d($fp)  sw $t3, %d($fp) | 从存储器调出x的值  从存储器调出y的值  除法运算  获得商送入t3寄存器  传回存储器内x对应位置  传回临时变量对应位置 |
| PLUS | x := y + z | lw $t1, %d($fp)  lw $t2, %d($fp)  add $t3, $t2,$t1  sw $t3, %d($fp) | 从存储器调出y的值  从存储器调出z的值  加法运算  传回存储器内x对应位置 |
| MINUS | x := y - z | lw $t1, %d($fp)  lw $t2, %d($fp)  sub $t3, $t1,$t2  sw $t3, %d($fp) | 从存储器调出y的值  从存储器调出z的值  减法运算  传回存储器内x对应位置 |
| STAR | x := y \* z | lw $t1, %d($fp)  lw $t2, %d($fp)  mul $t3, $t1,$t2  sw $t3, %d($fp) | 从存储器调出y的值  从存储器调出z的值  乘法运算  传回存储器内x对应位置 |
| DIV | x := y / z | lw $t1, %d($fp)  lw $t2, %d($fp)  div $t1,$t2  mflo $t3  sw $t3, %d($fp) | 从存储器调出y的值  从存储器调出z的值  除法运算  获得商送入t3寄存器  传回存储器内x对应位置 |
| FUNCTION  (main函数) | main: | main:  addi $t2, $0, 4  addi $t1, $0, %d  sw $sp, ($fp)  add $fp, $fp, $t2  add $sp, $fp,$t1 | 转跳目标标签  向寄存器t2保存升栈高度  向t1保存函数长度  在栈底保存该栈顶位置  栈底升高  栈顶指向正确位置 |
| FUNCTION  (非main函数) | func: | %s:  addi $t1, $0, 4  add $sp, $sp,$t1  sw $ra,($sp)  add $sp, $sp,$t1  addi $t2, $0, %d  sw $t2,($sp)  add $sp, $sp,$t1  sw $fp, ($sp)  move $fp, $sp  add $sp, $sp,$t2  根据形参个数循环：  lw $t1, %d($fp)  move $t3, $t1  sw $t3, %d($fp) | 转跳目标标签  向寄存器t2保存升栈高度  升栈  返回地址入栈  升栈  t2中保存函数总长度  函数长度入栈  升栈  调用函数的基址(栈底)入栈  栈底移动，形成新的栈帧  升栈  取实参的值  寄存器间转移  保存至该函数对应位置 |
| LABEL | label: | %s: | 转跳目标标签 |
| GOTO | j label | j %s | 转跳至对应位置 |
| JLE | IF x <= y  GOTO z | lw $t1, %d($fp)  lw $t2, %d($fp)  ble $t1,$t2, %s | 从存储器调出x的值  从存储器调出y的值  判断转跳 |
| JLT | IF x < y  GOTO z | lw $t1, %d($fp)  lw $t2, %d($fp)  blt $t1,$t2, %s | 从存储器调出x的值  从存储器调出y的值  判断转跳 |
| JGE | IF x >= y  GOTO z | lw $t1, %d($fp)  lw $t2, %d($fp)  bge $t1,$t2, %s | 从存储器调出x的值  从存储器调出y的值  判断转跳 |
| JGT | IF x > y  GOTO z | lw $t1, %d($fp)  lw $t2, %d($fp)  bgt $t1,$t2, %s | 从存储器调出x的值  从存储器调出y的值  判断转跳 |
| EQ | IF x == y  GOTO z | lw $t1, %d($fp)  lw $t2, %d($fp)  beq $t1,$t2, %s | 从存储器调出x的值  从存储器调出y的值  判断转跳 |
| IF x== #0  GOTO z | lw $t1, %d($fp)  add $t2, $0,$0  beq $t1,$t2, %s | 从存储器调出x的值  将0存入寄存器t2  判断转跳 |
| NEQ | IF x != y  GOTO z | lw $t1, %d($fp)  lw $t2, %d($fp)  bne $t1,$t2, %s | 从存储器调出x的值  从存储器调出y的值  判断转跳 |
| ARG | ARG x | addi $t1, $0, 4  add $sp, $sp,$t1  lw $t1, %d($fp)  sw $t1, ($sp) | 向寄存器t1保存升栈高度  升栈顶  取参数  压栈 |
| CALL | X:=CALL f | jal %s  sw $v0, %d($fp) | 转跳到被调用函数  保存返回值至存储器 |
| RETURN | return x | lw $v0, %d($fp)  move $sp, $fp  lw $t2, -8($fp)  move $ra, $t2  lw $t1, ($fp)  move $fp, $t1  lw $t1, -4($fp)  add $t1, $fp,$t1  move $sp, $t1  jr $ra | 保存返回值  栈顶回退到栈底  取返回地址  返回地址送入$ra  取调用函数的栈底  回到调用函数的栈底  取调用函数的长度  计算调用函数栈顶位置  栈顶回到对应位置  返回 |

设计的调用函数、传参、创建栈帧、返回等操作，可以用下图表示：

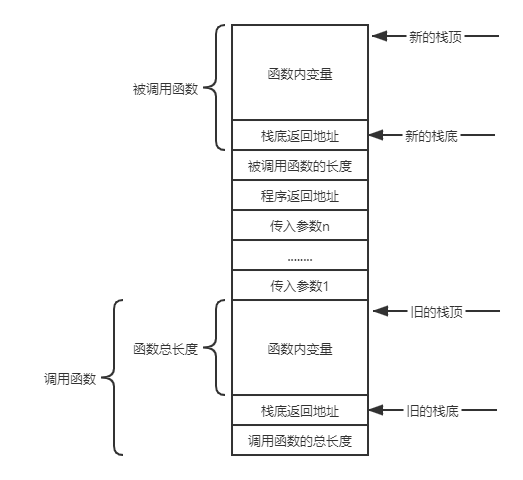


图 3.1：栈的设计

# 系统测试与评价

## 测试用例

测试用例分为正确性测试用例和报错功能测试用例。

正确性测试用例如下：

|  |
| --- |
| int fibo(int a) {      int i, j;      if(a == 1 || a == 2){          return 1;      }else{          i = fibo(a --) + fibo(a - 2);          return i;      }      return a;  }  int main(){      int m, n, i;      for (i = 1; i < 8; i ++){          m = fibo(i);          write(m);      }      i = 0;      while(1){          i ++;          if(i == 5){              continue;          }else if(i == 10){              write(i);              break;          }          write(i);      }      return 1;  } |

报错功能测试用例如下：

|  |
| --- |
| //以下用于静态语义检查  int a, b, c;  float m, n;  int X[5][10];//array  int print(int a){      int j;      char j;//变量重复定义      i ++; //使用未定义的变量      getchar();//调用未声明的函数      //函数没有返回值  }  int print(int a){//函数重复定义      int array[2][5][a];//变长数组      float j;      write ++;//对函数名采用非函数调用      a();//对非函数名采用函数调用      write(a,b,c);//实参和形参数目不匹配      write(j);//实参和形参类型不匹配不匹配      j[2] = 3; //对非数组变量采用下标访问      array[2.5][2][9];//采用非整型变量做下标，不能识别数组越界或维数缺失      2 = 3;//赋值号左边不是赋值表达式      2 += 3;//自增自减运算仅限变量，否则无法识别      return 1.5;//返回值类型不匹配函数定义  }  int fibo(int a,int a) {//参数名重复定义      int i;      return 1;//if else  }  int main(){      int m, n, i;      char y;      int array[5][3] = 2;//定义数组的同时赋值      m = read();      i = 1;      n = ++ X[0][2];      y = 'T';      break;      continue;//break和continue不再循环语句中      i += 1.5;//运算符两端类型冲突      for(i = 1; i <= m; i ++){//for          //Test this comment          n = fibo(i);          write(n);          if (i = 4){              break;//break          }else{              continue;//continue          }      }      return 1;  } |

## 正确性测试

运行代码，得到MIPS指令目标代码，代码的部分截图如下：

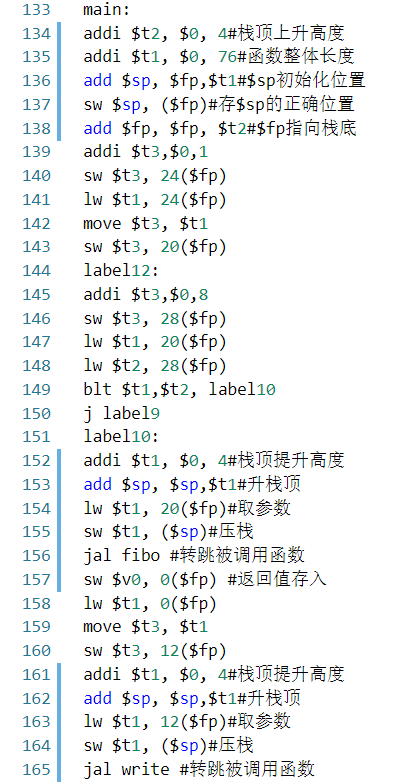


图 4.1：目标代码部分截图

将目标代码在Mars中打开，运行，得到输出结果如下：

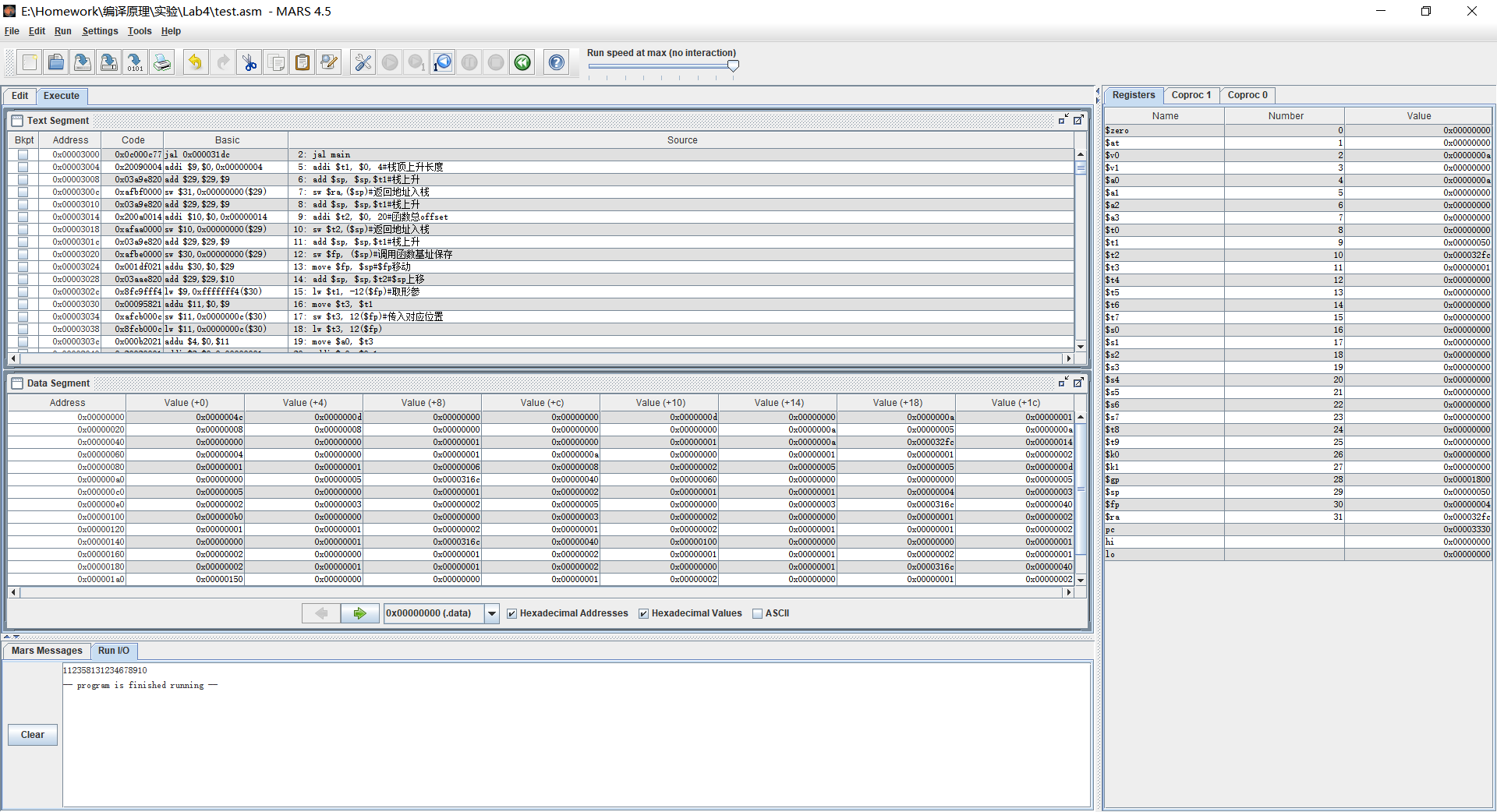


图 4.2：目标代码测试结果

可知正确打印出了斐波那契数列的前7个数，同时能利用while循环和break、continue语句，从1~10，跳过5，逐个打印。可知正确性颜色通过

## 报错功能测试

报错功能测试结果如下图所示：

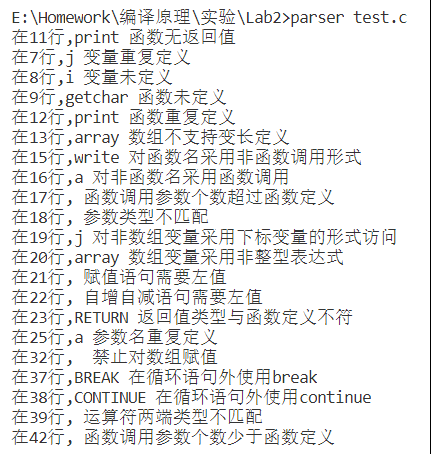


图 4.3：报错功能测试截图

根据上图可知，能够对静态语义错误进行识别和报错，报错功能正确性验证通过。

## 系统的优点

该系统的优点是：编译器轻量，所占空间小。支持对简单代码的不考虑外部库的情况下的快速编译运行，能够高效检查代码的质量。

## 系统的缺点

该系统的缺点是：不支持数组、结构体等高级数据结构，以及后期对float和char类型的忽视。故只能运行最为基本的数据结构和数据内容。

# 实验小结或体会

实验一共分为四个部分：文法和语义识别、符号表的建立、中间代码的生成以及目标代码的生成。

在第一个实验内，通过新增文法，向Mini-C中增加了一部分规则，使其更好用。在语义识别中，通过规约和移进，生成一棵抽象语法树。

在实验二中，通过对语法树的递归遍历，向符号表中添加变量和函数，并通过栈的方式进行作用域控制，已检查重名和未定义的情况。同时在该处了解到了如何根据变量类型进行变量偏移量的计算。

在实验三中，利用实验二生成的抽象语法树，同样进行遍历，将语法树转化为TAC四元组，并通过链表首尾相连，将语法树转换为TAC链表。针对TAC链表，可以根据其类型，逐个转化为中间代码，并为之后转化为目标代码做准备。

在最后一个实验中，把之前的TAC链表最终转化为目标代码，根据每个操作符的功能将其转换为对应汇编代码，同时针对函数调用部分，设计栈、参数压栈等功能，也重温了计算机系统基础的知识。

通过上述四个实验，能够对复杂的编译器设计工作管中窥豹，了解到整体的设计和编写流程，并从中了解到设计的复杂和困难。通过本次实验，加深了在理论课堂中学习到的关于编译的知识，并能将其灵活应用在实验中，收获匪浅。

# 参考文献

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016

[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**

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; //存储变量具体的值

    int type\_float;

    int type\_char;

    char type\_id[32];//变量的名字

    struct node \*ptr;

} YYLVAL;

#define YYSTYPE YYLVAL

%}

%option yylineno

id   [A-Za-z][A-Za-z0-9]\*

int    [0-9]+

float  ([0-9]\*\.[0-9]+)|([0-9]+\.)

char  '.'

%%

{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;}

"return"     {return RETURN;}

"if"         {return IF;}

"else"       {return ELSE;}

"while"      {return WHILE;}

"for"        {return FOR;}

"break"      {return BREAK;}

"continue"   {return CONTINUE;}

"switch"     {return SWITCH;}

"case"       {return CASE;}

"default"    {return DEFAULT;}

"struct"    {return STRUCT;}

{id}       {strcpy(yylval.type\_id,  yytext); return ID;/\*由于关键字的形式也符合标识符的规则，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 STAR;}

"/"         {return DIV;}

"++"        {return INC;}

"--"        {return DEC;}

"+="        {return SELFPLUS;}

"-="        {return SELFMINUS;}

"\*="        {return SELFSTAR;}

"/="        {return SELFDIV;}

"&&"        {return AND;}

"||"        {return OR;}

"!"         {return NOT;}

"("         {return LP;/\*parenthesis\*/}

")"         {return RP;}

"["         {return LB;/\*bracket\*/}

"]"         {return RB;}

"{"         {return LC;}

"}"         {return RC;}

":"         {return COLON;}

"."         {return DOT;}

[\n]        {yycolumn=1;}

[ \r\t]     {}

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

"//"[^\n]\*  {}

"/\*"(.\*|[\n])\*"\*/" {}

%%

/\* 和bison联用时，不需要这部分

void main()

{

yylex();

return 0;

}

\*/

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, ...);

extern void display(struct ASTNode \*,int);

extern struct ASTNode \* mknode(int num,int kind,int pos,...);

%}

%union {

    int    type\_int;

    float  type\_float;

    char   type\_char;

    char   type\_id[32];

    struct ASTNode \*ptr;

};

//  %type 定义非终结符的语义值类型

%type  <ptr> program ExtDefList ExtDef Specifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args

// %type  <ptr> CaseStmtList0 CaseStmtList

%type  <ptr> StructSpecifier StructName

//% 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    /\*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

%token LP RP LB RB LC RC SEMI COMMA      /\*用bison对该文件编译时，带参数-d，生成的.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 PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR BREAK CONTINUE DOT

%token SWITCH CASE COLON DEFAULT /\*未使用到的类型\*/

// %token CaseStmtList0 CaseStmtList

/\*以下为接在上述token后依次编码的枚举常量，作为AST结点类型标记\*/

%token EXT\_DEF\_LIST EXT\_VAR\_DEF FUNC\_DEF FUNC\_DEC EXT\_DEC\_LIST PARAM\_LIST PARAM\_DEC VAR\_DEF DEC\_LIST DEF\_LIST COMP\_STM STM\_LIST EXP\_STMT IF\_THEN IF\_THEN\_ELSE

%token FUNC\_CALL ARGS FUNCTION PARAM ARG CALL LABEL GOTO JLT JLE JGT JGE EQ NEQ

%token ARRAY STRUCT INC DEC SELFPLUS SELFMINUS SELFSTAR SELFDIV STRUCT\_DEF STRUCT\_TYPE STRUCT\_VISIT

// %token CASE\_STMT\_LIST CASE\_STMT\_LIST0 /\*未使用到的节点类型\*/

%left INT FLOAT ID

%left STRUCT

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS SELFPLUS SELFMINUS

%left STAR DIV SELFSTAR SELFDIV

%left LB RB

%right UMINUS NOT INC DEC

%left DOT

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

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

            ;

ExtDefList: {$$=NULL;}

            | ExtDef ExtDefList   {$$=mknode(2,EXT\_DEF\_LIST,yylineno,$1,$2);}   //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

            ;

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

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

            | error SEMI   {$$=NULL;}

            ;

Specifier:  TYPE   {$$=mknode(0,TYPE,yylineno);

                    strcpy($$->type\_id,$1);

                    if(!strcmp($1,"int")) $$->type = INT;

                    if(!strcmp($1,"float")) $$->type = FLOAT;

                    if(!strcmp($1,"char")) $$->type = CHAR;}

            | StructSpecifier {$$=mknode(1,STRUCT\_TYPE,yylineno,$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(2,EXT\_DEC\_LIST,yylineno,$1,$3);}

            ;

VarDec:     ID   {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);}   //ID结点，标识符符号串存放结点的type\_id

            | VarDec LB Exp RB {$$=mknode(2,ARRAY,yylineno,$1,$3);strcpy($$->type\_id,"ARRAY");} //数组类型的定义

            ;

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

            | ID LP RP   {$$=mknode(0,FUNC\_DEC,yylineno);strcpy($$->type\_id,$1);$$->ptr[0]=NULL;}//函数名存放在$$->type\_id

            ;

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

            | ParamDec COMMA  VarList  {$$=mknode(2,PARAM\_LIST,yylineno,$1,$3);}

            ;

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

            ;

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

            ;

StmList:    {$$=NULL;}

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

            ;

Stmt:       Exp SEMI   {$$=mknode(1,EXP\_STMT,yylineno,$1);}

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

            | RETURN Exp SEMI   {$$=mknode(1,RETURN,yylineno,$2);}

            | IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE   {$$=mknode(2,IF\_THEN,yylineno,$3,$5);}

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

            | WHILE LP Exp RP Stmt   {$$=mknode(2,WHILE,yylineno,$3,$5);}

            | FOR LP Exp SEMI Exp SEMI Exp RP Stmt {$$=mknode(4,FOR,yylineno,$3,$5,$7,$9);}

            ;

DefList:    {$$=NULL; }

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

            | error SEMI   {$$=NULL;}

            ;

Def:        Specifier DecList SEMI   {$$=mknode(2,VAR\_DEF,yylineno,$1,$2);}

            ;

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

            | Dec COMMA DecList   {$$=mknode(2,DEC\_LIST,yylineno,$1,$3);}

            ;

Dec:        VarDec   {$$=$1;}

            | VarDec ASSIGNOP Exp   {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}

            ;

Exp:        Exp ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}//$$结点type\_id空置未用，正好存放运算符

            | Exp AND Exp   {$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type\_id,"AND");}

            | Exp OR Exp    {$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type\_id,"OR");}

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

            | Exp PLUS Exp  {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type\_id,"PLUS");}

            | Exp MINUS Exp {$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type\_id,"MINUS");}

            | Exp STAR Exp  {$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type\_id,"STAR");}

            | Exp DIV Exp   {$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type\_id,"DIV");}

            | LP Exp RP     {$$=$2;}

            | MINUS Exp %prec UMINUS   {$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type\_id,"UMINUS");}

            | NOT Exp       {$$=mknode(1,NOT,yylineno,$2);strcpy($$->type\_id,"NOT");}

            | ID LP Args RP {$$=mknode(1,FUNC\_CALL,yylineno,$3);strcpy($$->type\_id,$1);}

            | ID LP RP      {$$=mknode(0,FUNC\_CALL,yylineno);strcpy($$->type\_id,$1);}

            // | Exp LB Exp RB {$$=mknode(2, ARRAY, yylineno, $1, $3); strcpy($$->type\_id,"ARRAY");} //数组

            // | ID            {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);}

            | VarDec        {$$=$1;}//数组或变量

            | INT           {$$=mknode(0,INT,yylineno);$$->type\_int=$1;$$->type=INT;}

            | FLOAT         {$$=mknode(0,FLOAT,yylineno);$$->type\_float=$1;$$->type=FLOAT;}

            | CHAR          {$$=mknode(0,CHAR,yylineno);$$->type\_char=$1;$$->type=CHAR;}

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

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

            | INC Exp       {$$=mknode(1,INC,yylineno,$2);strcpy($$->type\_id,"INC");}

            | DEC Exp       {$$=mknode(1,DEC,yylineno,$2);strcpy($$->type\_id,"DEC");}

            | Exp SELFPLUS Exp  {$$=mknode(2,SELFPLUS,yylineno,$1,$3);strcpy($$->type\_id,"SELFPLUS");}

            | Exp SELFMINUS Exp {$$=mknode(2,SELFMINUS,yylineno,$1,$3);strcpy($$->type\_id,"SELFMINUS");}

            | Exp SELFSTAR Exp {$$=mknode(2,SELFSTAR,yylineno,$1,$3);strcpy($$->type\_id,"SELFSTAR");}

            | Exp SELFDIV Exp {$$=mknode(2,SELFDIV,yylineno,$1,$3);strcpy($$->type\_id,"SELFDIV");}

            | BREAK         {$$=mknode(0,BREAK,yylineno);strcpy($$->type\_id,"BREAK");}

            | CONTINUE      {$$=mknode(0,CONTINUE,yylineno);strcpy($$->type\_id,"CONTINUE");}

            | Exp DOT ID    {$$=mknode(1,STRUCT\_VISIT,yylineno,$1);strcpy($$->type\_id,$3);}

            // | SWITCH LP Exp RP LC CaseStmtList RC {$$=mknode(2,SWITCH,yylineno,$3,$6);strcpy($$->type\_id,"SWITCH");}

            ;

// CaseStmtList0: LC CaseStmtList RC {$$=mknode(1,CASE\_STMT\_LIST0,yylineno,$2);}

//             ;

// CaseStmtList: {$$=NULL;}

//             | CASE INT COLON StmList CaseStmtList {$$=mknode(3,CASE\_STMT\_LIST,yylineno,$2,$4,$5);}

//             | DEFAULT COLON StmList               {$$=mknode(1,DEFAULT,yylineno,$3);strcpy($$->type\_id,"DEFAULT");}

//             ;

Args:       Exp COMMA Args    {$$=mknode(2,ARGS,yylineno,$1,$3);}

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

            ;

StructSpecifier:

            STRUCT StructName LC ExtDefList RC {$$=mknode(2,STRUCT\_DEF,yylineno,$2,$4);}

            | STRUCT ID {$$=mknode(1,STRUCT\_DEF,yylineno,$2);strcpy($$->type\_id,$2);$$->ptr[0]=NULL;}

            ;

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

            | {$$ = NULL;}

            ;

%%

int main(int argc, char \*argv[]){

    yyin=fopen(argv[1],"r");

    if (!yyin) return;

    yylineno=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");

}

def.h：

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include "stdarg.h"

#include "parser.tab.h"

#define MAXLENGTH   200

#define DX 3\*sizeof(int)          /\*活动记录控制信息需要的单元数，这个根据实际系统调整\*/

//以下语法树结点类型、三地址结点类型等定义仅供参考，实验时一定要根据自己的理解来定义

int LEV;      //层号

struct opn{

    int kind;    //标识联合成员的属性

    int type;    //标识操作数的数据类型

    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;                 //偏移量，目标代码生成时用

    };

struct codenode {   //三地址TAC代码结点,采用单链表存放中间语言代码

        int  op;

        struct opn opn1,opn2,result;

        struct codenode  \*next,\*prior;

    };

struct ASTNode {

        //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

    int kind;

    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;

          };

    struct ASTNode \*ptr[4];         //由kind确定有多少棵子树

    int place;                      //存放（临时）变量在符号表的位置序号

    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 Ebreak[15],Econtinue[15];

    char Snext[15];                 //结点对应语句S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

    int type;                       //用以标识表达式结点的类型

    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;                        //计数器，可以用来统计形参个数

    };

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

    int  type;       //变量类型或函数返回值类型

    int  paramnum;   //对函数适用，记录形式参数个数

    char alias[10];  //别名，为解决嵌套层次使用

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

    int  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  weight;     //或记录函数活动记录大小，目标代码生成时使用

                     //函数入口等实验可能会用到的属性...

};

//符号表

struct symboltable{

    struct symbol symbols[MAXLENGTH];

    int index;

} symbolTable;

struct symbol\_scope\_begin {

    //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,

    // 当使用顺序表作为符号表时，进入、退出一个作用域时需要对其操作，

    // 以完成符号表的管理。对其它形式的符号表，不一定需要此数据结构

    int TX[30];

    int top;

} symbol\_scope\_TX;

extern struct ASTNode \* mknode(int num,int kind,int pos,...);

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

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

void boolExp(struct ASTNode \*T);

void Exp(struct ASTNode \*T);

void objectCode(struct codenode \*head);

ast.c：

#include "def.h"

#include "parser.tab.h"

struct ASTNode \* mknode(int num,int kind,int pos,...);

void display(struct ASTNode \*T,int indent);

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

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

struct ASTNode \* mknode(int num,int kind,int pos,...){

    struct ASTNode \*T=(struct ASTNode \*)calloc(sizeof(struct ASTNode),1);

    int i=0;

    T->kind=kind;

    T->pos=pos;

    va\_list pArgs = NULL;

    va\_start(pArgs, pos);

    for(i=0;i<num;i++)

        T->ptr[i]= va\_arg(pArgs, struct ASTNode \*);

    while (i<4) T->ptr[i++]=NULL;

    va\_end(pArgs);

    return T;

}

void semantic\_Analysis0(struct ASTNode \*T){

    printf("\n\n");

    printf("单词序列展示：\n");

    semantic\_Analysis(T);

}

void semantic\_Analysis(struct ASTNode \*T){

    int i;

    struct ASTNode \*T0;

    switch(T->kind){

        case ID:        printf("(%s, ID)\n", T->type\_id);

                        return;

        case ARRAY:     T0 = T->ptr[0];

                        while(T0->kind == ARRAY){

                            T0 = T0->ptr[0];

                        }

                        printf("(%s, ARRAY)\n", T0->type\_id);

                        return;

        case STRUCT\_DEF:if(!T->ptr[1]){

                            printf("(%s, ID)\n", T->type\_id);

                        }

                        break;

        case STRUCT\_VISIT:

                        printf("(%s, ID)\n", T->type\_id);

                        break;

        case FUNC\_CALL:

        case FUNC\_DEC:  printf("(%s, FUNC)\n", T->type\_id);

                        break;

        case ASSIGNOP:  printf("(=, ASSIGNOP)\n");

                        break;

        case AND:       printf("($$, AND)\n");

                        break;

        case OR:        printf("(||, OR)\n");

                        break;

        case RELOP:     printf("(%s, RELOP)\n", T->type\_id);

                        break;

        case PLUS:      printf("(+, PLUS)\n");

                        break;

        case MINUS:     printf("(-, MINUS)\n");

                        break;

        case STAR:      printf("(\*, STAR)\n");

                        break;

        case DIV:       printf("(/, DIV)\n");

                        break;

        case UMINUS:    printf("(-, UMINUS)\n");

                        break;

        case NOT:       printf("(!, NOT)\n");

                        break;

        case INT:       printf("(%d, CONSTANT INT)\n", T->type\_int);

                        break;

        case FLOAT:     printf("(%f, CONSTANT FLOAT)\n", T->type\_float);

                        break;

        case CHAR:      printf("(%c, CONSTANT CHAR)\n", T->type\_char);

                        break;

        case INC:       printf("(++, INC)\n");

                        break;

        case DEC:       printf("(--, DEC)\n");

                        break;

        case SELFPLUS:  printf("(+=, SELFPLUS)\n");

                        break;

        case SELFMINUS: printf("(-=, SELFMINUS)\n");

                        break;

        case BREAK:     printf("(break, BREAK)\n");

                        break;

        case CONTINUE:  printf("(continue, CONTINUE)\n");

                        break;

    }

    for(i = 0; i < 3; i ++){

        if(T->ptr[i]){

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

        }

    }

    return;

}

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

{//对抽象语法树的先根遍历

  int i=1;

  struct ASTNode \*T0;

  if (T)

    {

    switch (T->kind) {

    case EXT\_DEF\_LIST:  display(T->ptr[0],indent);    //显示该外部定义（外部变量和函数）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：(%d)\n",indent,' ',T->pos);

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

                        break;

    case STRUCT\_TYPE:   printf("%\*c类型： STURCT\n",indent,' ');

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

                        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：(%d)\n",indent,' ',T->pos);

                        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:     printf("%\*c类型：%s, 参数名：%s\n",indent,' ',T->ptr[0]->type==INT?"int":"float",T->ptr[1]->type\_id);

                        break;

    case EXP\_STMT:      printf("%\*c表达式语句：(%d)\n",indent,' ',T->pos);

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

                        break;

    case RETURN:        printf("%\*c返回语句：(%d)\n",indent,' ',T->pos);

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

                        break;

    case COMP\_STM:      printf("%\*c复合语句：(%d)\n",indent,' ',T->pos);

                        printf("%\*c复合语句的变量定义部分：\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("%\*cwhile循环语句：(%d)\n",indent,' ',T->pos);

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

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

                        printf("%\*cwhile循环体：(%d)\n",indent+3,' ',T->pos);

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

                        break;

    case FOR:           printf("%\*cfor循环语句：(%d)\n",indent,' ',T->pos);

                        printf("%\*cfor循环起始状态:\n",indent+3, ' ');

                        display(T->ptr[0],indent+6);      //显示循环起始状态

                        printf("%\*cfor循环判断:\n",indent+3, ' ');

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

                        printf("%\*cfor循环更新:\n",indent+3, ' ');

                        display(T->ptr[2],indent+6);      //显示循环判断

                        printf("%\*cfor循环体：(%d)\n",indent+3,' ',T->pos);

                        display(T->ptr[3],indent+6);      //显示循环体

                        break;

    case IF\_THEN:       printf("%\*c条件语句(IF\_THEN)：(%d)\n",indent,' ',T->pos);

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

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

                        printf("%\*cIF子句：(%d)\n",indent+3,' ',T->pos);

                        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)：(%d)\n",indent,' ',T->pos);

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

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

                        printf("%\*cIF子句：(%d)\n",indent+3,' ',T->pos);

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

                        printf("%\*cELSE子句：(%d)\n",indent+3,' ',T->pos);

                        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("%\*c局部变量定义：(%d)\n",indent,' ',T->pos);

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

                        T0=T;

                        while (T0) {

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

                                //printf("%\*c %s\n",indent+6,' ',T0->ptr[0]->type\_id);

                                display(T0->ptr[0], indent+6);

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

                                {

                                printf("%\*c %s ASSIGNOP\n ",indent+6,' ',T0->ptr[0]->ptr[0]->type\_id);

                                display(T0->ptr[0]->ptr[1],indent+strlen(T0->ptr[0]->ptr[0]->type\_id)+7);        //显示初始化表达式

                                }

                            T0=T0->ptr[1];

                            }

                        break;

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

                        break;

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

                        break;

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

                        break;

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

                        break;

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

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

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

                        break;

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

                        break;

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

                        break;

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

    //                     printf("%\*c控制语句：\n",indent,' ');

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

    //                     printf("%\*cCase语句：\n",indent,' ');

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

    //                     break;

    // case CASE\_STMT\_LIST:printf("Case条件：\n");

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

    //                     printf("对应操作：\n");

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

    //                     display(T->ptr[2],indent);

    //                     break;

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

    //                     printf("对应操作：\n");

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

    //                     break;

    case STRUCT\_DEF:    if (T->ptr[1]) {

                            printf("%\*c结构名：%s\n",indent,' ',T->ptr[0]->type\_id);

                            printf("%\*c结构定义：\n", indent, ' ');

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

                        } else{

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

                            printf("%\*c结构定义：无\n", indent, ' ');

                        }

                        break;

    case STRUCT\_VISIT:  T0 = T->ptr[0];

                        printf("%\*c结构名：%s\n",indent,' ',T0->type\_id);

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

                        break;

    case ASSIGNOP:

    case AND:

    case OR:

    case RELOP:

    case PLUS:

    case MINUS:

    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 INC:

    case DEC:       display(T->ptr[0],indent);

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

                    return;

    case SELFSTAR:

    case SELFDIV:

    case SELFPLUS:

    case SELFMINUS: display(T->ptr[0],indent);

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

                    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：(%d)\n",indent,' ',T->pos);

                    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 ASTNode \*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;

         }

      }

}

SymbolTable.c：（包含实验3、4）

#include "def.h"

#include "parser.tab.h"

#define DEBUG 1

int INLOOP=0;

int blockid = 0;

struct ASTNode \* mknode(int num,int kind,int pos,...){

struct ASTNode \*T=(struct ASTNode \*)calloc(sizeof(struct ASTNode),1);

int i=0;

T->kind=kind;

T->pos=pos;

va\_list pArgs = NULL;

va\_start(pArgs, pos);

for(i=0;i<num;i++)

T->ptr[i]= va\_arg(pArgs, struct ASTNode \*);

while (i<4) T->ptr[i++]=NULL;

va\_end(pArgs);

return T;

}

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==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==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 INC:

case DEC: printf(" %s := %s %c #1\n",resultstr,opnstr1,\

h->op==INC? '+':'-');

break;

case UMINUS: printf(" %s := 0 - %s\n",resultstr,opnstr1);

break;

case SELFPLUS:

case SELFMINUS:

case SELFSTAR:

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

h->op==SELFPLUS?'+':h->op==SELFMINUS?'-':h->op==SELFSTAR?'\*':'/',opnstr2);

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: if (!strcmp(opnstr1,"write"))

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

else

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 objectCode(struct codenode \*head){

FILE \*fp;

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

struct codenode \*h=head;

int rtn, i;

int ismain = 0;

if((fp = fopen("test.asm", "w")) == NULL){

printf("File open failure\n");

return;

}

do{

fprintf(fp,".text\n");

fprintf(fp,"jal main\n\n");

fprintf(fp,"write:\naddi $t1, $0, 4#栈顶上升长度\n");

fprintf(fp,"add $sp, $sp,$t1#栈上升\nsw $ra,($sp)#返回地址入栈\n");

fprintf(fp,"add $sp, $sp,$t1#栈上升\naddi $t2, $0, 20#函数总offset\n");

fprintf(fp,"sw $t2,($sp)#返回地址入栈\nadd $sp, $sp,$t1#栈上升\n");

fprintf(fp,"sw $fp, ($sp)#调用函数基址保存\nmove $fp, $sp#$fp移动\n");

fprintf(fp,"add $sp, $sp,$t2#$sp上移\nlw $t1, -12($fp)#取形参\n");

fprintf(fp,"move $t3, $t1\nsw $t3, 12($fp)#传入对应位置\n");

fprintf(fp,"lw $t3, 12($fp)\nmove $a0, $t3\n");

fprintf(fp,"addi $v0, $0,1\nsyscall\n");

fprintf(fp,"addi $t3,$0,0\nsw $t3, 16($fp)\n");

fprintf(fp,"lw $v0, 16($fp)#存返回值\nmove $sp, $fp #$sp回退到$fp\n");

fprintf(fp,"lw $t2, -8($fp)#取返回地址\nmove $ra, $t2#送入$ra\n");

fprintf(fp,"lw $t1, ($fp)#读取调用函数的栈底\nmove $fp, $t1#回到栈底\n");

fprintf(fp,"lw $t1, -4($fp)\nadd $t1, $fp,$t1\n");

fprintf(fp,"move $sp, $t1\njr $ra\n\n");

}while(0);

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,"#%c",h->opn1.const\_float);

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

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

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

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

// fprintf(fp,"\n");

i = 0;

switch (h->op) {

case ASSIGNOP: if(h->opn1.kind == INT){//x := #k

fprintf(fp,"addi $t3,$0,%d\n",h->opn1.const\_int);

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

}else if(h->opn1.kind == FLOAT){

fprintf(fp,"addi $t3,$0,%d\n",h->opn1.const\_float);

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

}else if(h->opn1.kind == CHAR){

fprintf(fp,"addi $t3,$0,%d\n",h->opn1.const\_char);

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

}else if(h->opn1.kind == ID){//x := y

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"move $t3, $t1\n");

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

}

break;

case INC: // v := v + #1

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);//load v

fprintf(fp,"addi $t2, $0,1\n");//load 1

fprintf(fp,"add $t3, $t1,$t2\n");//add

fprintf(fp,"sw $t3, %d($fp)\n",h->opn1.offset);//store into v

break;

case DEC: // temp := v - #1

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);//load v

fprintf(fp,"addi $t2, $0,-1\n");//load -1

fprintf(fp,"add $t3, $t1,$t2\n");//add

fprintf(fp,"sw $t3, %d($fp)\n",h->opn1.offset);//store into v

break;

case UMINUS: // temp := #0 - v

fprintf(fp,"lw $t3, %d($fp)\n",h->opn1.offset);

fprintf(fp,"sub $t1, $0,$t3\n");

fprintf(fp,"sw $t1, %d($fp)\n",h->result.offset);

break;

case SELFPLUS: // v := v + #x

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);//load v

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);//load constant

fprintf(fp,"add $t3, $t2,$t1\n");//add

fprintf(fp,"sw $t3, %d($fp)\n",h->opn1.offset);//store into v

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);//store into temp

break;

case SELFMINUS: // v := v - #x

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);//load v

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);//load constant

fprintf(fp,"sub $t3, $t2,$t1\n");//sub

fprintf(fp,"sw $t3, %d($fp)\n",h->opn1.offset);//store into v

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);//store into temp

break;

case SELFSTAR: // v := v \* #x

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);//load v

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);//load constant

fprintf(fp,"mul $t3, $t2,$t1\n");//multiple

fprintf(fp,"sw $t3, %d($fp)\n",h->opn1.offset);//store into v

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);//store into temp

break;

case SELFDIV: // v := v / #x

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);//load v

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);//load constant

fprintf(fp,"div $t1,$t2\n");//div

fprintf(fp,"mflo $t3\n");//div

fprintf(fp,"sw $t3, %d($fp)\n",h->opn1.offset);//store into v

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);//store into temp

break;

case PLUS: // t := vt + vt

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"add $t3, $t2,$t1\n");

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

break;

case MINUS: // t := vt - vt

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"sub $t3, $t1,$t2\n");

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

break;

case STAR: // t := vt \* vt

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"mul $t3, $t1,$t2\n");

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

break;

case DIV: // t := vt / vt

fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"div $t1,$t2\n");

fprintf(fp,"mflo $t3\n");//div

fprintf(fp,"sw $t3, %d($fp)\n",h->result.offset);

break;

case FUNCTION: //打标签

rtn = searchSymbolTable(h->result.id);

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

if(!strcmp(h->result.id,"main")){

fprintf(fp,"addi $t2, $0, 4#栈顶上升高度\n");

fprintf(fp,"addi $t1, $0, %d#函数整体长度\n",symbolTable.symbols[rtn].offset + 4);

fprintf(fp,"add $sp, $fp,$t1#$sp初始化位置\n");//$sp += 该函数内长度

//把$sp在函数中的位置放到$fp里

fprintf(fp,"sw $sp, ($fp)#存$sp的正确位置\n");

fprintf(fp,"add $fp, $fp, $t2#$fp指向栈底\n");

ismain = 1;

break;

}

//把返回地址存入栈

fprintf(fp,"addi $t1, $0, 4#栈顶上升长度\n");

fprintf(fp,"add $sp, $sp,$t1#栈上升\n");//$sp += 4

fprintf(fp,"sw $ra,($sp)#返回地址入栈\n");

//该函数的$sp位置入栈

fprintf(fp,"add $sp, $sp,$t1#栈上升\n");//$sp += 4

fprintf(fp,"addi $t2, $0, %d#函数总offset\n",symbolTable.symbols[rtn].offset);

fprintf(fp,"sw $t2,($sp)#返回地址入栈\n");

//保存调用函数的$fp

fprintf(fp,"add $sp, $sp,$t1#栈上升\n");//$sp += 4

fprintf(fp,"sw $fp, ($sp)#调用函数基址保存\n");//保存基址位置

//$fp更新到$sp

fprintf(fp,"move $fp, $sp#$fp移动\n");

//更新$sp

fprintf(fp,"add $sp, $sp,$t2#$sp上移\n");//$sp += 该函数内长度

//读取形参

int stored\_param\_offset = -4 \* (symbolTable.symbols[rtn].paramnum + 2);

int new\_param\_offset = 12;

for (i = 0; i < symbolTable.symbols[rtn].paramnum; i++){

fprintf(fp,"lw $t1, %d($fp)#取形参\n",stored\_param\_offset);

fprintf(fp,"move $t3, $t1\n");

fprintf(fp,"sw $t3, %d($fp)#传入对应位置\n",new\_param\_offset);

stored\_param\_offset += 4;

new\_param\_offset += 4;

}

// if(!strcmp(h->result.id,"print")){

// fprintf(fp,"lw $t3, 12($fp)\n");

// fprintf(fp,"move $a0, $t3\n");

// fprintf(fp,"addi $v0, $0,1\n");

// fprintf(fp,"syscall\n");

// }

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

break;

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

// break;

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

break;

case GOTO: fprintf(fp,"j %s\n",h->result.id);

break;

case JLE: fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"ble $t1,$t2, %s\n",resultstr);

break;

case JLT: fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"blt $t1,$t2, %s\n",resultstr);

break;

case JGE: fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"bge $t1,$t2, %s\n",resultstr);

break;

case JGT: fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"bgt $t1,$t2, %s\n",resultstr);

break;

case EQ: fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

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

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

}else{

fprintf(fp,"add $t2, $0,$0\n");

}

fprintf(fp,"beq $t1,$t2, %s\n",resultstr);

break;

case NEQ: fprintf(fp,"lw $t1, %d($fp)\n",h->opn1.offset);

fprintf(fp,"lw $t2, %d($fp)\n",h->opn2.offset);

fprintf(fp,"bne $t1,$t2, %s\n",resultstr);

break;

case ARG: fprintf(fp,"addi $t1, $0, 4#栈顶提升高度\n");

fprintf(fp,"add $sp, $sp,$t1#升栈顶\n");

fprintf(fp,"lw $t1, %d($fp)#取参数\n",h->result.offset);

fprintf(fp,"sw $t1, ($sp)#压栈\n");

break;

case CALL: fprintf(fp,"jal %s #转跳被调用函数\n",opnstr1);//保存地址后跳转

//调用结束后

fprintf(fp,"sw $v0, %d($fp) #返回值存入\n",h->result.offset);

break;

case RETURN: if(ismain == 1){

// fprintf(fp,"+++++++++++\n");

ismain = 0;

break;

}

// fprintf(fp,"-------\n");

if (h->result.kind)

fprintf(fp,"lw $v0, %d($fp)#存返回值\n",h->result.offset);

//$sp 回退到 $fp

fprintf(fp,"move $sp, $fp #$sp回退到$fp\n");//$fp -> $sp

//取返回地址

fprintf(fp,"lw $t2, -8($fp)#取返回地址\n");

fprintf(fp,"move $ra, $t2#送入$ra\n");

//$fp 回退到 调用函数的 $fp

fprintf(fp,"lw $t1, ($fp)#读取调用函数的栈底\n");

fprintf(fp,"move $fp, $t1#回到栈底\n");

//$sp取($sp) 回退

fprintf(fp,"lw $t1, -4($fp)\n");

fprintf(fp,"add $t1, $fp,$t1\n");//new added

fprintf(fp,"move $sp, $t1\n");

//返回

fprintf(fp,"jr $ra\n");

break;

}

h=h->next;

} while (h!=head);

fprintf(fp,"addi $v0,$0,10\n");

fprintf(fp,"syscall\n\n");

}

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

//这里可以只收集错误信息，最后一次显示

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

}

void prn\_symbol(){ //显示符号表

int i=0;

char type[10];

printf("%6s %8s %7s %7s %4s %6s\n","变量名","别 名","层 号","类 型","标记","偏移量");

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

if(symbolTable.symbols[i].type==INT){

strcpy(type, "int");

}else if (symbolTable.symbols[i].type==FLOAT) {

strcpy(type, "float");

}else {

strcpy(type, "char");

}

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

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

type,symbolTable.symbols[i].flag,\

symbolTable.symbols[i].offset);

}

}

int searchSymbolTable(char \*name) {

int i,flag=0;

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

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

flag=1;

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

continue; //跳过前面函数的形式参数表项

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

}

return -1;

}

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; i>=0 && (symbolTable.symbols[i].level==level||level==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

}

void ext\_var\_list(struct ASTNode \*T){ //处理变量列表

int rtn,num=1;

struct ASTNode \*T0;

int totallength = 1;

int flag = 0;

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

totallength = 1;

if (T->ptr[0]->kind == ARRAY){//解决 int a, b[10], c; 的情况

//当类型为ARRAY时，记录ARRAY总width，当出现变长数组，产生报错

T0 = T->ptr[0];

flag = 0;

while (T0->kind == ARRAY){

if(T0->ptr[1]->kind == INT){

totallength \*= T0->ptr[1]->type\_int; //仅当INT类型定义数组时，记录数组长度

} else{

flag = 1;//否则报错

}

T0 = T0->ptr[0];

}

if (flag){

semantic\_error(T0->pos,T0->type\_id, "数组不支持变长定义");

}

}

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

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

if (flag != 1){//当已经对变长数组报错后，就不需要进入函数报第二次错

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

}

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

T->num= T->ptr[1]->num + totallength;//加上 totallength 才能确定下一行外部定义变量的正确偏移量

break;

case ARRAY:

// 解决 int y[10]; 的情况

T0 = T;

totallength = 1;

while (T0->kind == ARRAY){

T0->ptr[1]->offset = T0->offset;//偏移量传递给数组的一个维度

if(T0->ptr[1]->kind == INT){

totallength \*= T0->ptr[1]->type\_int; //仅当INT类型定义数组时，记录数组长度

} else{

flag = 1;//否则报错

}

semantic\_Analysis(T0->ptr[1]);//将数组的长度作为temp存入表中

T0->ptr[0]->offset = T0->ptr[1]->width + T0->offset;//把下一层ARRAY的offset计算出来

T0 = T0->ptr[0];

}

if (flag){

semantic\_error(T0->pos,T0->type\_id, "数组不支持变长定义");

return;

}

rtn = fillSymbolTable(T0->type\_id,newAlias(),LEV,T->type,'A',T0->offset); //offset使用最内侧ARRAY的偏移量

if (rtn==-1)

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

else T->place = rtn;

T->num=totallength;//考虑数组长度的情况下，下一行外部定义变量的正确偏移量

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;

break;

}

}

int match\_param(int i,struct ASTNode \*T){

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

int type1,type2,pos=T->pos;

T=T->ptr[0];

if (num==0 && T==NULL) return 1;

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

if (!T){

semantic\_error(pos,"", "函数调用参数个数少于函数定义");

return 0;

}

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

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

if (type1!=type2){

semantic\_error(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(pos,"", "函数调用参数个数超过函数定义");

return 0;

}

return 1;

}

void boolExp(struct ASTNode \*T){ //布尔表达式，参考文献[2]p84的思想

struct opn opn1,opn2,result;

struct opn opn3,opn4,result1;

int op;

int rtn;

if (T)

{

switch (T->kind) {

case INT: if(T->type\_int){

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

}else{

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

}

break;

case FLOAT: if(T->type\_float){

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

}else{

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

}

break;

case CHAR: if(T->type\_char){

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

}else{

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

}

break;

case ID: 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, "对函数名采用非函数调用形式");

}else {

T->place=rtn; //lab3,结点保存变量在符号表中的位置

T->code=NULL; //lab3,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

opn1.kind=ID;

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

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

opn2.const\_int = 0;

opn2.kind = INT;

result.kind=ID;

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

op = NEQ;

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

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

}

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

// printf("%d\n",T->offset);

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

//opn1

opn1.kind=ID;

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

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

//opn2

opn2.kind=ID;

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

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

//result

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->offset;

boolExp(T->ptr[0]);

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

boolExp(T->ptr[1]);

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

case ASSIGNOP:

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

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

boolExp(T->ptr[0]);

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

boolExp(T->ptr[1]);

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

//opn1

opn1.kind=ID;

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

opn1.offset=symbolTable.symbols[T->ptr[1]->place].offset;

//result

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

//opn1

opn3.kind=ID;

strcpy(opn3.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn3.offset=symbolTable.symbols[T->ptr[0]->place].offset;

//opn2

opn4.kind = INT;

opn4.const\_int = 0;

//result

result1.kind = ID;

strcpy(result1.id, T->Etrue);

T->code = merge(3,T->code,genIR(NEQ,opn3,opn4,result1),genGoto(T->Efalse));

break;

case INC:

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

if(T->type == INT){

T->width = 4;

}else if(T->type == FLOAT){

T->width = 8;

}else if(T->type == CHAR){

T->width = 1;

}

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

Exp(T->ptr[0]);

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

opn1.kind=ID;

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

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

//result

result.kind=ID;

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

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

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

//opn1

opn1.kind=ID;

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

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

//opn2

opn2.kind = INT;

opn2.const\_int = 0;

//result

result.kind = ID;

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

T->code = merge(3,T->code,genIR(NEQ,opn1,opn2,result),genGoto(T->Efalse));

break;

case SELFPLUS:

case SELFMINUS:

case SELFSTAR:

case SELFDIV:

case PLUS:

case MINUS:

case STAR:

case DIV:

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

if(T->type == INT){

T->width = 4;

}else if(T->type == FLOAT){

T->width = 8;

}else if(T->type == CHAR){

T->width = 1;

}

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

//opn1

opn1.kind=ID;

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

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

//opn2

opn2.kind=ID;

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

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

//result

result.kind=ID;

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

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

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

//opn1

opn1.kind=ID;

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

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

//opn2

opn2.kind = INT;

opn2.const\_int = 0;

//result

result.kind = ID;

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

T->code = merge(3,T->code,genIR(NEQ,opn1,opn2,result),genGoto(T->Efalse));

break;

case UMINUS:

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

if(T->type == INT){

T->width = 4;

}else if(T->type == FLOAT){

T->width = 8;

}else if(T->type == CHAR){

T->width = 1;

}

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

Exp(T->ptr[0]);

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

//opn1

opn1.kind=ID;

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

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

//result

result.kind=ID;

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

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

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

//opn1

opn1.kind=ID;

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

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

//opn2

opn2.kind = INT;

opn2.const\_int = 0;

//result

result.kind = ID;

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

T->code = merge(3,T->code, genIR(NEQ,opn1,opn2,result),genGoto(T->Efalse));

}

}

}

void Exp(struct ASTNode \*T){

//处理基本表达式，参考文献[2]p82的思想

int rtn,num,width;

struct ASTNode \*T0;

struct opn opn1,opn2,result;

struct opn opn3, opn4, result1, opn5, opn6, result2;

struct opn opn7, opn8, result3;

char label1[15], label2[15];

int flag;

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, "对函数名采用非函数调用形式");

else {

T->place=rtn; //lab3,结点保存变量在符号表中的位置

T->code=NULL; //lab3,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: T0 = T;

flag = 0;

while(T0->kind == ARRAY){

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

Exp(T0->ptr[1]);

if(T0->ptr[1]->type != INT){

flag = 1;

}

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

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

T0 = T0->ptr[0];

}

if (flag){

semantic\_error(T->pos,T0->type\_id, "数组变量采用非整型表达式");

}

// printf("%s\n",T0->type\_id);

rtn = searchSymbolTable(T0->type\_id);

if(rtn == -1){

semantic\_error(T->pos,T0->type\_id, "变量未定义");

}

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

semantic\_error(T->pos,T0->type\_id, "对函数名采用非函数调用形式");

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

semantic\_error(T->pos,T0->type\_id, "对非数组变量采用下标变量的形式访问");

}else {

T->place = rtn;

T->code=NULL;

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); //lab3,为整常量生成一个临时变量

// printf("line %d, int %d, offset %d\n",T->pos, T->type\_int, T->offset);

T->type = INT;

//lab3

opn1.kind = INT;

opn1.const\_int=T->type\_int;

result.kind = ID;

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

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

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

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

// printf("line %d float %f, offset %d\n",T->pos, T->type\_float, T->offset);

T->type = FLOAT;

//lab3

opn1.kind = FLOAT;

opn1.const\_float = T->type\_float;

result.kind = ID;

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

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

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

T->width=8;

break;

case CHAR: T->place = fill\_Temp(newTemp(),LEV,T->type,'T',T->offset); //为字符常量生成一个临时变量

// printf("line %d, char %c, offset %d\n",T->pos, T->type\_char, T->offset);

T->type = CHAR;

//lab3

opn1.kind = CHAR;

opn1.const\_char = T->type\_char;

result.kind = ID;

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

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

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

T->width = 1;

break;

case ASSIGNOP:

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

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

}else {

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

Exp(T->ptr[0]); //处理左值，例中仅为变量

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

Exp(T->ptr[1]);

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

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

T->code = merge(2,T->ptr[0]->code,T->ptr[1]->code);

opn1.kind = ID;

strcpy(opn1.id,symbolTable.symbols[T->ptr[1]->place].alias);//右值一定是个变量或临时变量

opn1.type = T->ptr[1]->type;

opn1.offset = symbolTable.symbols[T->ptr[1]->place].offset;

result.kind = ID;

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

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

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

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

}

break;

case AND:

T->type = INT;

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

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

strcpy(label1,newLabel());

//opn1

opn1.kind = ID;

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

opn1.type = T->ptr[0]->type;

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

//opn2

opn2.kind = ID;

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

opn2.type = T->ptr[1]->type;

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

//result

result.kind = ID;

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

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

//opn3

opn3.kind = ID;

strcpy(opn3.id,symbolTable.symbols[T->place].alias);

opn3.offset = symbolTable.symbols[T->place].offset;

//opn4

opn4.kind = INT;

opn4.const\_int = 0;

//result1

result1.kind = ID;

strcpy(result1.id,label1);

//opn5

opn5.kind = INT;

opn5.const\_int = 1;

//result2

result2.kind = ID;

strcpy(result2.id,symbolTable.symbols[T->place].alias);

result2.offset = symbolTable.symbols[T->place].offset;

T->code = merge(6,T->ptr[0]->code, T->ptr[1]->code, genIR(STAR,opn1,opn2,result),\

genIR(EQ,opn3,opn4,result1),genIR(ASSIGNOP,opn5,opn6,result2),\

genLabel(label1));

break;

case OR: T->type = INT;

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

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

strcpy(label1,newLabel());

strcpy(label2,newLabel());

//opn1

opn1.kind = INT;

opn1.const\_int = 0;

//result

result.kind = ID;

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

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

//opn3

opn3.kind = ID;

strcpy(opn3.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn3.type = T->ptr[0]->type;

opn3.offset = symbolTable.symbols[T->ptr[0]->place].offset;

//opn4

opn4.kind = INT;

opn4.const\_int = 0;

//result1

result1.kind = ID;

strcpy(result1.id, label1);

//opn5

opn5.kind = INT;

opn5.const\_int = 1;

//result2

result2.kind = ID;

strcpy(result2.id,symbolTable.symbols[T->place].alias);

result2.offset = symbolTable.symbols[T->place].offset;

//opn7

opn7.kind = ID;

strcpy(opn7.id,symbolTable.symbols[T->ptr[1]->place].alias);

opn7.type = T->ptr[0]->type;

opn7.offset = symbolTable.symbols[T->ptr[1]->place].offset;

//opn8

opn8.kind = INT;

opn8.const\_int = 0;

//result3

result3.kind = ID;

strcpy(result3.id, label2);

T->code = merge(10,T->ptr[0]->code,T->ptr[1]->code,genIR(ASSIGNOP,opn1,opn2,result),\

genIR(EQ,opn3,opn4,result1),genIR(ASSIGNOP,opn5,opn6,result2),genGoto(label2),\

genLabel(label1),genIR(EQ,opn7,opn8,result3),genIR(ASSIGNOP,opn5,opn6,result2),\

genLabel(label2));

break;

case RELOP:

T->type = INT;

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

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

strcpy(label1,newLabel());

strcpy(label2,newLabel());

//opn1

opn1.kind = ID;

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

opn1.type = T->ptr[0]->type;

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

//opn2

opn2.kind = ID;

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

opn2.type = T->ptr[1]->type;

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

//result

result.kind = ID;

strcpy(result.id,label1);

//opn3

opn3.kind = INT;

opn3.const\_int = 0;

//result1

result1.kind = ID;

strcpy(result1.id,symbolTable.symbols[T->place].alias);

result1.type = T->type;

result1.offset = symbolTable.symbols[T->place].offset;

//opn5

opn5.kind = INT;

opn5.const\_int = 1;

//result2

result2.kind = ID;

strcpy(result2.id,symbolTable.symbols[T->place].alias);

result2.type = T->type;

result2.offset = symbolTable.symbols[T->place].offset;

int op;

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 = merge(8,T->ptr[0]->code,T->ptr[1]->code,genIR(op,opn1,opn2,result),\

genIR(ASSIGNOP,opn3,opn4,result1),genGoto(label2),genLabel(label1),genIR(ASSIGNOP,opn5,opn6,result2)\

, genLabel(label2));

break;

case SELFPLUS:

case SELFMINUS:

case SELFSTAR:

case SELFDIV:

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

semantic\_error(T->pos,"", "自增自减语句需要左值");

}

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

//ptr[1]的width是常数的width

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

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

T->type = FLOAT;

T->width += 8;//此处的增量是变量自增后暂存所需要的的width

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

T->type = INT;

T->width += 4;

}else if (T->ptr[0]->type == CHAR && T->ptr[1]->type == CHAR){

T->type = CHAR;

T->width += 1;

}else {

semantic\_error(T->pos,"", "运算符两端类型不匹配");

return;

}

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

//opn1

opn1.kind = ID;

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

opn1.type = T->ptr[0]->type;

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

//opn2

opn2.kind = ID;

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

opn2.type = T->ptr[1]->type;

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

//result

result.kind = ID;

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

result.type = T->type;

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

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

break;

case PLUS:

case MINUS:

case STAR:

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

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

T->type = FLOAT;

T->width = T->ptr[0]->width + T->ptr[1]->width + 8;

}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 + 4;

}else if (T->ptr[0]->type == CHAR && T->ptr[1]->type == CHAR){

T->type = CHAR;

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

}else {

semantic\_error(T->pos,"", "运算符两端类型不匹配");

return;

}

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

//opn1

opn1.kind = ID;

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

opn1.type = T->ptr[0]->type;

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

//opn2

opn2.kind = ID;

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

opn2.type = T->ptr[1]->type;

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

//result

result.kind = ID;

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

result.type = T->type;

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

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

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

break;

case INC:

case DEC: T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

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

T->type = FLOAT;

T->width = T->ptr[0]->width + 8;

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

T->type = INT;

T->width = T->ptr[0]->width + 4;

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

T->type = CHAR;

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

}

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

opn1.kind = ID;

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

opn1.type = T->ptr[0]->type;

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

result.kind = ID;

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

result.type = T->type;

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

result1.kind = ID;

strcpy(result1.id,symbolTable.symbols[T->place].alias);

result1.type = T->type;

result1.offset = symbolTable.symbols[T->place].offset;

T->code = merge(3,T->ptr[0]->code,genIR(T->kind,opn1,opn2,result),genIR(ASSIGNOP,opn1,opn2,result1));

break;

case NOT: T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

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

T->type = FLOAT;

T->width = T->ptr[0]->width + 8;

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

T->type = INT;

T->width = T->ptr[0]->width + 4;

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

T->type = CHAR;

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

}

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

strcpy(label1,newLabel());

strcpy(T->Snext,newLabel());

//opn1

opn1.kind = ID;

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

opn1.type = T->ptr[0]->type;

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

//opn2

opn2.kind = INT;

opn2.const\_int = 0;

//result

result1.kind = ID;

strcpy(result.id, label1);

//opn3

opn3.kind = INT;

opn3.const\_int = 0;

//result1

result1.kind = ID;

strcpy(result1.id,symbolTable.symbols[T->place].alias);

result1.type = T->type;

result1.offset = symbolTable.symbols[T->place].offset;

//opn5

opn5.kind = INT;

opn5.const\_int = 1;

//opn4 is empty

T->code = merge(7,T->ptr[0]->code,genIR(EQ,opn1,opn2,result),genIR(ASSIGNOP,opn3,opn4,result1),\

genGoto(T->Snext),genLabel(label1),genIR(ASSIGNOP,opn5,opn6,result1),genLabel(T->Snext));

break;

case UMINUS:T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

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

T->type = FLOAT;

T->width = T->ptr[0]->width + 8;

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

T->type = INT;

T->width = T->ptr[0]->width + 4;

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

T->type = CHAR;

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

}

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

opn1.kind = ID;

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

opn1.type = T->ptr[0]->type;

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

result.kind = ID;

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

result.type = T->type;

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

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

break;

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, "对非函数名采用函数调用");

break;

}

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

if (T->type == INT) {

width = 4;

}else if (T->type == FLOAT) {

width = 8;

}else if (T->type == CHAR) {

width = 1;

}else{}

// width=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); //处理所有参数的匹配

//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;

}

}

}

int isreturnexist = 0;

int findreturn(struct ASTNode \*T){

int i;

for(i = 0; i < 3; i ++){

if(T->ptr[i]){

if(T->ptr[i]->kind == RETURN){

isreturnexist = 1;

}else{

findreturn(T->ptr[i]);

}

}

}

}

void semantic\_Analysis(struct ASTNode \*T){

//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）

int rtn,num,width;

struct ASTNode \*T0, \*T1;

struct opn opn1,opn2,result;

char label1[15];

if (T){

switch (T->kind) {

case EXT\_DEF\_LIST://外部定义列表

// [0]: ExtDef, [1]: ExtDefList or [0] == NULL

if (!T->ptr[0]){

break;

}

T->ptr[0]->offset = T->offset; //继承偏移量

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]){//外部定义列表还没结束

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

// [0]: Specifier, [1]: ExtDecList, [2]: SEMI

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

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

}else {

}//以上将Specifier的类型传给之后定义的变量

// 不考虑CHAR的情况:

// T->type=T->ptr[1]->type=!strcmp(T->ptr[0]->type\_id,"int")?INT:FLOAT;

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

if(T->type == INT){

T->ptr[1]->width = 4;

}else if(T->type == FLOAT){

T->ptr[1]->width = 8;

}else if(T->type == CHAR){

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

}else{

// T->ptr[1]->width = struct\_type\_process(T->ptr[0]->ptr[0]);

}//将一个变量的宽度向后传递

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

T->width = T->ptr[1]->width \* T->ptr[1]->num;

//计算这个外部变量说明的宽度

// T->width=(T->type==INT?4:8)\* T->ptr[1]->num;

T->code=NULL; //这里假定外部变量不支持初始化

break;

case FUNC\_DEF: //填写函数定义信息到符号表,

// [0]:Specifier, [1]:FuncDec, [2]:CompSt

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

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

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

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

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

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

} else{

semantic\_error(T->pos,T->ptr[1]->type\_id, "函数返回类型错误");

}

//获取函数返回类型送到含函数名、参数的结点

// T->ptr[1]->type=!strcmp(T->ptr[0]->type\_id,"int")?INT:FLOAT;

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

isreturnexist = 0;

findreturn(T->ptr[2]);

if (isreturnexist == 0){

semantic\_error(T->pos,T->ptr[1]->type\_id, "函数无返回值");

}

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

// [0]: VarList or NULL

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;//rtn 是符号表的index

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

// [0]: ParamDec, [1]: VarList or NULL

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:

// [0]: Specifier, [1]: VarDec

//PARAM\_DEC是函数形式参数，不用考虑ARRAY的情况。

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

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

T->width = 4;

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

T->width = 8;

}else {

T->width = 1;

}

// T->width = T->ptr[0]->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:

// [0]: DefList, [1]: StmList

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]){

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

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

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);

}

#if (DEBUG)

prn\_symbol(); //c在退出一个符合语句前显示的符号表

// system("pause");

#endif

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:

// NULL or [0]:Def, [1]:DefList

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

// [0]:Specifier, [1]:DecList

T->code = NULL;

//根据类型确定变量列表内变量类型，用于传递

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

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;

}

//确定变量序列各变量类型

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

//T0's kind is DecList

num = 0;

T0->offset = T->offset;

T->width=0;

if (T->ptr[1]->type==INT) {

width = 4;

}else if (T->ptr[1]->type==FLOAT){

width = 8;

}else{

width = 1;

}//一个变量宽度

while (T0) { //处理所有DEC\_LIST结点

// [0]:Dec, [1]:DecList or NULL

num++;

T0->offset = T->offset + T->width;

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

if (T0->ptr[1]) {

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

}

//首先判断T0->ptr[0]作为VarDec的kind是ID或ARRAY，不考虑变长数组

T1 = T0->ptr[0];

T1->offset = T0->offset;//偏移量传递

int totallength = 1;//记录多维数组总的长度

int flag = 0;

while (T1->kind == ARRAY){

T1->ptr[1]->offset = T1->offset;

if(T1->ptr[1]->kind == INT){

totallength \*= T1->ptr[1]->type\_int; //仅当INT类型定义数组时，记录数组长度

} else{

flag = 1;//否则报错

}

semantic\_Analysis(T1->ptr[1]);//将数组的长度作为temp存入表中

T->width += T1->ptr[1]->width;//数组各维度长度作为temp也算作VAR\_DEF节点长度

T1->ptr[0]->offset = T1->ptr[1]->width + T1->offset;

T1 = T1->ptr[0];

}

//若Dec是ARRAY，则T1指向最内侧ID，同时offset中包含了各维度常量的width

//若Dec是ID，则也指向ID

if (flag){

semantic\_error(T1->pos,T1->type\_id, "数组不支持变长定义");

}

if (T0->ptr[0]->kind == ID){//不能用T1因为T1可能经过while == ARRAY 而都为ID

//加入符号表

rtn = fillSymbolTable(T1->type\_id,newAlias(),LEV,T1->type,'V',T1->offset);

//此处偏移量未计算，暂时为0

if (rtn==-1){

semantic\_error(T1->pos,T1->type\_id, "变量重复定义");

}else {

T1->place=rtn;

}

T->width += width;

}else if(T0->ptr[0]->kind == ARRAY){

//考虑ARRAY, 禁止定义变长数组

rtn = fillSymbolTable(T1->type\_id,newAlias(),LEV,T1->type,'A',T1->offset);

//此处偏移量未计算，暂时为0

if (rtn==-1){

semantic\_error(T1->pos,T1->type\_id, "变量重复定义");

}else {

T1->place=rtn;

}

T->width += width \* totallength;

}else if (T0->ptr[0]->kind == ASSIGNOP){//变量声明的同时赋值

if(T1->ptr[0]->kind == ARRAY){//here

semantic\_error(T1->ptr[0]->pos," ", "禁止对数组赋值");

}

rtn = fillSymbolTable(T1->ptr[0]->type\_id,newAlias(),LEV,T1->type,'V',T->offset+T->width);

//此处偏移量未计算，暂时为0

// semantic\_Analysis

if (rtn==-1)

semantic\_error(T1->ptr[0]->pos,T1->ptr[0]->type\_id, "变量重复定义");

else {

T1->place = rtn;

T1->ptr[1]->offset = T->offset+T->width+width;

Exp(T1->ptr[1]);

opn1.kind = ID;

strcpy(opn1.id,symbolTable.symbols[T1->ptr[1]->place].alias);

opn1.offset = symbolTable.symbols[T1->ptr[1]->place].offset;

result.kind = ID;

strcpy(result.id,symbolTable.symbols[T1->place].alias);

result.offset = symbolTable.symbols[T1->place].offset;

T->code = merge(3,T->code,T1->ptr[1]->code,genIR(ASSIGNOP,opn1,opn2,result));

}

//width是被的变量的长度，T1->ptr[1]->width是所赋值的长度

T->width += width + T1->ptr[1]->width;

}

T0=T0->ptr[1];//回到While句首，处理剩下Dec

}

break;

case STM\_LIST:

// [0]: Stmt, [1]: StmList

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);

}

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

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

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

T->ptr[1]->offset=T->offset+T->ptr[0]->width; //顺序结构顺序分配单元方式

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

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

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

// }

T->width+=T->ptr[1]->width;//顺序结构顺序分配单元方式

}

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->offset;

boolExp(T->ptr[0]);

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

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

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

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

semantic\_Analysis(T->ptr[1]); //if子句

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

strcpy(T->Snext,newLabel());

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

boolExp(T->ptr[0]); //条件，要单独按短路代码处理

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

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

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

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

semantic\_Analysis(T->ptr[1]); //if子句

T->ptr[2]->offset = T->ptr[1]->offset + T->ptr[1]->width;

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

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

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

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

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

boolExp(T->ptr[0]); //循环条件，要单独按短路代码处理

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

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

INLOOP ++;

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

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

semantic\_Analysis(T->ptr[1]); //循环体

INLOOP --;

T->width = T->ptr[0]->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 FOR: strcpy(T->ptr[1]->Etrue,newLabel());

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

strcpy(T->ptr[3]->Ebreak,T->Snext);

strcpy(label1,newLabel());

// strcpy(T->ptr[3]->Snext,newLabel());

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

strcpy(T->ptr[3]->Econtinue,label1);

strcpy(T->ptr[3]->Snext,label1);

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

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

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

boolExp(T->ptr[1]);

T->ptr[2]->offset = T->ptr[1]->offset + T->ptr[1]->width;

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

T->ptr[3]->offset = T->ptr[3]->offset + T->ptr[3]->width;

INLOOP ++;

semantic\_Analysis(T->ptr[3]);//循环体

INLOOP --;

T->width = T->ptr[0]->width + T->ptr[1]->width + T->ptr[2]->width + T->ptr[3]->width;

T->code = merge(8,T->ptr[0]->code,genLabel(T->ptr[2]->Snext),T->ptr[1]->code,\

genLabel(T->ptr[1]->Etrue),T->ptr[3]->code,genLabel(label1) ,T->ptr[2]->code,\

genGoto(T->ptr[2]->Snext));

break;

case EXP\_STMT:

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

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

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

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

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

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

break;

case RETURN:if (T->ptr[0]){

// printf("%d\n",T->offset);

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

Exp(T->ptr[0]);

/\*需要判断返回值类型是否匹配\*/

int idx = symbolTable.index;

while(symbolTable.symbols[idx].flag != 'F'){

idx --;

}

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

semantic\_error(T->pos,"RETURN", "返回值类型与函数定义不符");

}

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 BREAK:

if(INLOOP == 0){

semantic\_error(T->pos, "BREAK", "在循环语句外使用break");

}

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

break;

case CONTINUE:

if(INLOOP == 0){

semantic\_error(T->pos, "CONTINUE", "在循环语句外使用continue");

}

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

break;

case ID:

case ARRAY:

case INT:

case FLOAT:

case CHAR:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

case INC:

case DEC:

case SELFPLUS:

case SELFMINUS:

case SELFSTAR:

case SELFDIV:

case NOT:

case UMINUS:

case FUNC\_CALL:

Exp(T); //处理基本表达式

break;

}

}

}

void semantic\_Analysis0(struct ASTNode \*T) {

symbolTable.index=0;

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

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

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

symbolTable.symbols[1].paramnum=1;

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

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);

prnIR(T->code);

objectCode(T->code);

}