

**实验报告**

**题目： C语言实现的minic编译器**

**课程名称： 编译原理**

**专业班级： ACM1601**

**学 号： U201614684**

**姓 名： 余颖雪**

**指导教师： 徐丽萍**

**报告日期： 2019.6.18**

**计算机科学与技术学院**

**目录**

[1选题背景 1](#_Toc12174109)

[2系统关键定义 2](#_Toc12174110)

[2.1单词文法描述 2](#_Toc12174111)

[2.2语句文法描述 3](#_Toc12174112)

[2.3 符号表结构描述 3](#_Toc12174113)

[2.4 错误类型码描述 4](#_Toc12174114)

[2.5中间代码描述 5](#_Toc12174115)

[2.6 目标代码描述 6](#_Toc12174116)

[3系统设计与实现 7](#_Toc12174117)

[3.1编译程序符号表结构 7](#_Toc12174118)

[3.2编译程序报错功能 8](#_Toc12174119)

[3.3词法语法分析器（实验一） 8](#_Toc12174120)

[3.4语义分析（实验二） 11](#_Toc12174121)

[3.5中间代码生成功能（实验三） 12](#_Toc12174122)

[3.6汇编代码生成功能（实验四） 13](#_Toc12174123)

[4系统测试与评价 16](#_Toc12174124)

[4.1 测试用例 16](#_Toc12174125)

[4.2 正确性测试 18](#_Toc12174126)

[4.3 系统的优点 24](#_Toc12174127)

[4.4 系统的缺点 24](#_Toc12174128)

[5实验小结或体会 25](#_Toc12174129)

[**参考文献** 26](#_Toc12174130)

[**附件：源代码** 27](#_Toc12174131)

# 1选题背景

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

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

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

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

# 2系统关键定义

## 2.1单词文法描述

单词有五类：关键字（保留字）、运算符、界符、常量和标识符。依据mini-c语言的定义，在此给出各单词的种类码和相应符号说明：

INT → 整型常量

CHAR → 字符型常量

FLOAT → 浮点型常量

ID → 标识符

ASSIGNOP → =

COMPASSIGN → += | -= |\*= | /=

RELOP → > | >= | < | <= | == | !=

PLUSONE → ++

MINUSONE → --

PLUS → +

MINUS → -

STAR → \*

DIV → /

AND → &&

OR → ||

NOT → !

TYPE → int | float| char

RETURN → return

IF → if

ELSE → else

WHILE → while

SEMI → ；

COMMA → ，

SEMI → ；

LP → （

RP → ）

LC → {

RC → }

minic 中的关键字（它们都是保留字）包括：基本类型关键字int、 float、 char，分支与循环语句涉及的 if、else、while，返回语句 return。运算符包括：“+ - \* / < <= > >= = == != && || ! ++ -- += -= \*= /=”，涉及到运算和逻辑运算。界符包括： “; {} （）”。常量包括：浮点数常量、整型常量（十进制）、以及字符型常量。浮点数常量可以用一般的小数形式表示，例如“6.66666、8888.8”等，注意，在本词法设计中，小数“.0001”与“1. ” 都算合法小数。一个整型常量都是十进制整数，一个十进制整数是一个十进制数字（0-9）的序列，例如“68686”； 一个字符型常量是被一对单引号包围的单个大写或小写英语字母。标识符是以字母开头的字母、数字的序列。除此之外，在词法分析阶段，为了后续程序设计以及错误提示的需要，本词法析程序还会识别注释、空白字符、换行符以及错误的字符。注释包括单行注释和多行注释，单行注释是以//开头直到该行的结尾，多行注释是用 “/\*”和“\*/”包含的所有字符（自身除外）；空白字符包括空格、制表符、Tab；错误的字符是无法与其他项匹配的字符。

## 2.2语句文法描述

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

Specifier→int | float | char

ExtDecList→VarDec | VarDec , ExtDecList

VarDec→ID

FucDec→ID ( VarList ) | ID ( )

VarList→ParamDec , VarList | ParamDec

ParamDec→Specifier VarDec

CompSt→{ DefList StmList }

StmList→Stmt StmList | ε

Stmt→Exp ; | CompSt | return Exp ;

| 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 | ID | INT | FLOAT

| ( Exp ) | - Exp | ! Exp | ID ( Args ) | ID ( )

Args→Exp , Args | Exp

## 2.3 符号表结构描述

使用符号表来记录源程序中各种名字的特性信息。实验中顺序表结构来实现：符号表symbolTable是一个顺序栈，栈顶指针index初始值为0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加1。每一个符号对应一个symbol结构体，特性信息记录在symbol的成员中，定义如下所示:

struct symbol {

char name[33]; //变量或函数名

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

int type; //变量类型或函数返回值类型

int paramnum; //形式参数个数

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

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

char offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量,或函数活动记录大小，目标代码生成时使用

};

## 2.4 错误类型码描述

**表2-1 错误类型码描述**

|  |  |  |
| --- | --- | --- |
| 编号 | 错误类型 | 错误所在行 |
| 1 | 变量重复定义 | 6 |
| 2 | 函数调用参数太少 | 51 |
| 3 | 函数调用参数类型不匹配 | 47 |
| 4 | 函数调用参数太多 | 46 |
| 5 | 将函数名当做变量使用 | 49 |
| 6 | 变量未定义 | 23 |
| 7 | 赋值运算两边类型不匹配 | 43 |
| 8 | 不支持布尔值和其他数据类型的转换 | 31 |
| 9 | 复合赋值操作需要一个左值 | 35 |
| 10 | 运算操作左右类型不一致 | 50 |
| 11 | 自增自减操作需要一个左值 | 40 |
| 12 | 只有整型数据支持自增自减操作 | 39 |
| 13 | 单运算符减操作需要一个左值 | 45 |
| 14 | 字符型数据不能为负值 | 38 |
| 15 | 函数未定义 | 48 |
| 16 | （进行函数调用的）不是一个函数 | 33 |
| 17 | 函数重复定义 | 21 |
| 18 | （函数）参数名重复定义 | 16 |
| 19 | （函数）返回值类型错误 | 19 |

每种错误类型举例对应在4.1中testerr.c 对应行的代码。

## 2.5中间代码描述

采用三地址代码TAC作为中间语言，中间语言代码的定义如表2-2所示。

**表2-2 中间代码定义**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **语法** | **描述** | **Op** | **Opn1** | **Opn2** | **Result** |
| LABEL x | 定义标号x | LABEL |  |  | X |
| FUNCTION f: | 定义函数f | FUNCTION |  |  | F |
| x := y | 赋值操作 | ASSIGN | X |  | 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 |
| GOTO x | 无条件转移 | GOTO |  |  | X |
| IF x [relop] y GOTO z | 条件转移 | [relop] | X | Y | Z |
| RETURN x | 返回语句 | RETURN |  |  | X |
| ARG x | 传实参x | ARG |  |  | X |
| x:=CALL f | 调用函数 | CALL | F |  | X |
| PARAM x | 函数形参 | PARAM |  |  | X |
| READ x | 读入 | READ |  |  | X |
| WRITE x | 打印 | WRITE |  |  | X |

三地址中间代码TAC是一个4元组，逻辑上包含（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。后续还需要根据TAC序列生成目标代码，所以设计其存储结构时，每一部分要考虑目标代码生成是所需要的信息。

（1）运算符：表示这条指令需要完成的运算，可以用枚举常量表示，如PLUS表示双目加，JLE表示小于等于，PARAM表示形参，ARG表示实参等。

（2）操作数与运算结果：这些部分包含的数据类型有多种，整常量，实常量，还有使用标识符的情况，如变量的别名、变量在其数据区的偏移量和层号、转移语句中的标号等。类型不同，所以考虑使用联合。为了明确联合中的有效成员，将操作数与运算结果设计成结构类型，包含kind，联合等几个成员，kind说明联合中的有效，联合成员是整常量，实常量或标识符表示的别名或标号或函数名等。

（3）为了配合后续的TAC代码序列的生成，将TAC代码作为数据元素，用双向循环链表表示TAC代码序列。

## 2.6 目标代码描述

目标语言选定MIPS32指令序列，可以在SPIM Simulator上运行。

包括指令li sw lw add sub mul div mflo jr beq bne bge ble blt addi jal move。

# 3系统设计与实现

## 3.1编译程序符号表结构

符号表symbolTable是一个顺序栈，栈顶指针index初始值为0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加1。每一个符号对应一个symbol结构体，特性信息记录在symbol的成员中，定义如下所示:

struct symbol {

char name[33]; //变量或函数名

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

int type; //变量类型或函数返回值类型

int paramnum; //形式参数个数

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

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

char offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量,或函数活动记录大小，目标代码生成时使用

};

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

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

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

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

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

可以调用用prn\_IR函数打印符号表信息。

## 3.2编译程序报错功能

静态语义分析时，若发现错误，调用semantic\_error()函数报错。进行的检查主要包括：

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

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

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

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

## 3.3词法语法分析器（实验一）

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

根据 2.1 的分析，合法单词包括关键字、运算符、界符、常量和标识符，以及其他一些辅助“单词”。关键字的正则表达式十分简单，例如：对于关键字 int 而言，其正则表达式就是“int”（包括引号）。以此类推，不难得到所有关键字的正则表达式。

运算符与界符的正则表达式与关键字的正则表达式类似，都是用引号括起自身即可，于是不再累述。例如+的正则表达式是“+”，{的正则表达式是“{”。

常量的表达式相对复杂,对于一般小数形式的浮点数常量，因为要考虑“36.”和“.36”这样的特殊形式，所以形式比较复杂，要分情况讨论并用“|”将规则相或，最后设计出的正则表达式为“([0-9]\*\.[0-9]+)|([0-9]+\.)”（不包括引号，下同）；对于十进制的整型常量，实际上就是 0-9 的序列再加上正负号，所以正则表达式相对简单为“[0-9]+”; 根据标识符的定义，设计其正则表达式时需要对开头第一个字符作限制，即第一个字符只能是 a-z 或 A-Z，由此得到其正则表达式为“[A-Za-z][A-Za-z0-9]\*”。不过，这里需要注意的是，对标识符的识别规则应放到关键字的识别规则的后面，否则会将所有的关键字当作标识符处理。空白字符和换行符的正则表达式十分简单，同关键字方法相同。

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

语法分析器的实现采用的是自动化生成工具 Bison，Bison 可以根据给定的语法规则，自动化生成对应的语法分析程序。但是，语法分析的目的不仅仅是判断源程序的语句是否符合语法规则，还应该（如果符合语法规则）构造源程序对应的语法分析树，用于编译的后续阶段。Bison 和 Flex 可以无缝对接，即将 Flex进行词法分析后得到的单词序列作为Bison的输入，从而用来进行语法分析。 为了实现这一点，需要按照实验指导书上的步骤进行修改和编译，这里不赘述。

设计语法分析器的第一步，便是设计相应的语法规则。语法规则在 2.2 中的 中已经详细给出，这里需要做的便是将语法规则按照 Bison 的标准写成相应的生成式。在具体的转化过程中，由 minic语法规范直接转化来的生成式存在移进-规约冲突或规约规约冲突，需要通过显示规定优先级和结合性来解决。经过排查移进-规约冲突和规约-规约冲突的来源，不难发现大部分的冲突来自于运算符。例如，对于算数表达式“1+2-3”，分析器并不知道是先算“1+2”还是“2-3”。当然， 如果告诉分析器“+”和“-”都是左结合，那么分析器自然知道是要先计算“1+2” 而不是是“2-3”。除了算数表达式外，IF-ELSE 语句的语法也会带来移进-规约冲突，因为对于生成式“IF OP LP EXP RP Stmt”和“IF LP Exp RP Stmt ELSE Stmt”， 在分析到“LP”时，分析器既可以选择第一条生成式进行规约，也可以选择第二条生成式进行移近，显然根据语法规则和编程常识，应该优先选择第二条生成式也就是移进，所以需要借助 Bison 的特性赋予第二条生成式更高的优先级。

设计语法分析器的第二步便是构造语法分析树。根据编译原理课程上所学的知识，为了实现在语法分析的同时构造语法树，应该为语法的每条产生式添加一定的语义动作来完成叶节点的生成和添加到已有语法树。为此，设计并定义语法分析树的节点结构体如下：

struct node

{

enum node\_kind kind; //结点类型

union

{

char type\_id[33]; //由标识符生成的叶结点

int type\_int; //由整常数生成的叶结点

float type\_float; //由浮点常数生成的叶结点

char type\_char;

};

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

struct node \*parent; //mark 增加一个父节点，用来支持各种嵌套运算

int seq\_num ; //mark 存放此节点是父亲的第几个子节点

int level; //层号

int place; //表示结点对应的变量或运算结果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 Snext[15]; //该结点对应语句执行后的下一条语句位置标号

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

char op[10];

int type; //结点对应值的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //各种数据占用的字节数

int num; //函数调用的参数个数

};

另外定义了函数mknode用来创建新的树节点。完成了语法分析树相关结构体和函数的定义和实现后，需要将其与 Bison 中的生成式代码相结合。为此，需要完成两项工作： 一是将所有终结符和非终结符的属性值类型声明为语法树节点的指针类型。二是为每条生成式添加语义动作，用来根据生成式构造语法树。需要注意的是，对于最顶层的生成式“program: ExtDefList”的语义动作，不仅要实现构造语法树的功能，还应根据语法分析的结果打印最终的语法分析树。

## 3.4语义分析（实验二）

主要工作就是通过遍历AST完成符号表的管理和静态语义分析。

一、AST遍历

AST的遍历采用的是先根遍历，在遍历过程中，访问到了说明部分的结点时，在符号表中添加新的内容；访问到执行语句部分时，根据访问的变量（或函数）名称查询符号表，并分析其静态语义的正确性。

先根遍历AST算法的框架很简单，采用递归算法实现，设T为根结点指针。

（1）如果T为空，遍历结束返回

（2）根据T->kind，即结点类型，可知道该结点有多少棵子树，依次递归访问各子树。

在语义分析阶段，通过遍历访问结点完成各种属性的计算，比如对于一个局部变量说明语句：int a，b；

当第一次访问到VAR\_DEF结点时，按遍历次序，接着访问TYPE结点，确定TYPE结点的数据类型为INT，回到VAR\_DEF后，该说明语句中的变量列表中个变量的类型确定了，可将此类型属性向下传给结点DEC\_LIST1。接着类型由DEC\_LIST1传到a这个ID结点，这时就明确了a是一个整型变量，查符号表，如果在当前作用域（根据层号）没有定义，就根据a填写一个整型的变量a到符号表中，否则报错，变量重复定义。再接着数据类型INT由DEC\_LIST1传到DEC\_LIST2结点，直到整型变量b完成查表和填写到符号表中。

上述例子的属性计算仅考虑语义分析这部分的需求，但在整个编译过程中，需要同时完成的属性计算还很多，比如访问VAR\_DEF结点时，首先到此结点，由前面的计算结果，已经得到这个说明语句的变量在活动记录中的地址偏移量（offset），这时访问过TYPE结点后，得到该类型变量的宽度值（width），这样a的地址偏移量就为offset，b的地址偏移量为offset+width；最后回到VAR\_TYPE结点时，其说明语句中变量的总宽度计算出为2\*width。所以再遇到VAR\_DEF之后的其它变量说明的结点时，地址偏移量为offset+2\*width。由此给计算出一个函数中所有变量在活动记录中的地址偏移量。

二、作用域与符号表操作

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

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

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

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

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

## 3.5中间代码生成功能（实验三）

为了完成中间代码的生成，对于AST中的结点，需要考虑设置以下属性，在遍历过程中，根据翻译模式给出的计算方法完成属性的计算。

.place 记录该结点操作数在符号表中的位置序号，这里包括变量在符号表中的位置，以及每次完成了计算后，中间结果需要用一个临时变量保存，临时变量也需要登记到符号表中。另外由于使用复合语句，可以使作用域嵌套，不同的作用域中的变量可以同名，这是在mini-c中，和C语言一样采用就近优先的原则，但在中间语言中，没有复合语句区分层次，所以每次登记一个变量到符号表中时，会多增加一个别名（alias）的表项，通过别名实现数据的唯一性。翻译时，对变量的操作替换成对别名的操作，别名命名形式为v+序号。生成临时变量时， 命名形式为temp+序号，在填符号表时，可以在符号名称这栏填写一个空串，临时变量名直接填写到别名这栏。

.type 一个结点表示数据时，记录该数据的类型，用于表达式的计算中。该属性也可用于语句，表示语句语义分析的正确性（OK或ERROR）。

.offset 记录外部变量在静态数据区中的偏移量以及局部变量和临时变量在活动记录中的偏移量。另外对函数，利用这项保存活动记录的大小。

.width 记录一个结点表示的语法单位中，定义的变量和临时单元所需要占用的字节数，方便计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小。

.code 记录中间代码序列的起始位置，如采用链表表示中间代码序列，该属性就是一个链表的头指针。

.Etrue 和.Efalse 在完成布尔表达式翻译时，表达式值为真假时要转移的程序位置（标号的字符串形式）。

.Snext 该结点的语句序列执行完后，要转移或到的的程序位置（标号的字符串形式）。

为了生成中间代码序列，定义了几个函数：

newtemp 生成一临时变量，登记到符号表中，以temp+序号的形式组成的符号串作为别名，符号名称栏用空串登记到符号表中。

newLabel 生成一个标号，标号命名形式为LABEL+序号。

genIR 生成一条TAC的中间代码语句。一般情况下，TAC中，涉及到2个运算对象和运算结果。如果是局部变量或临时变量，表示在运行时，其对应的存储单元在活动记录中，这时需要将其偏移量（offset）这个属性和数据类型同时带上，方便最后的目标代码生成。全局变量也需要带上偏移量。

genLabel 生成标号语句。

以上3个函数，在实验时，也可以合并在一起，如何处理，可自行确定。

merge 将多个语句序列顺序连接在一起。

定义完这些属性和函数后，就需要根据翻译模式表示的计算次序，计算规则右部各个符号对应结点的代码段，再按语句的语义，将这些代码段拼接在一起，组成规则左部非终结符对应结点的代码段。

## 3.6汇编代码生成功能（实验四）

采用朴素的寄存器分配算法，对应中间代码到目标代码的翻译如表2-3所示。

对于函数调用X:=CALL f，需要完成开辟活动记录的空间、参数的传递和保存返回地址等，函数调用返回后，需要恢复返回地址，读取函数返回值以及释放活动记录空间。活动记录的空间布局没有一个统一的标准，可根据自己的理解保存好数据，并能正确使用即可。

通常，使用4个寄存器完成参数的传递，多余4个的参数使用活动记录空间，这里做了简单处理，所有参数都使用活动记录空间。具体步骤：

（1）首先根据保存在函数调用指令中的offset，找到符号表中的函数定义点，获取函数的参数个数i，这样就可得到在X:=CALL f之前的i个ARG形式的中间代码，获得i个实参值所存放的单元，取出后送到形式参数的单元中。再活动记录的空间。

（2）根据符号表记录的活动记录大小，开辟活动记录空间和保存返回地址。

（3） 使用jal f 转到函数f处

（4） 释放活动记录空间和恢复返回地址。

（5） 使用sw $v0, x的偏移量($sp) 获取返回值送到X的存储单元中

**表3-1 朴素寄存器分配的翻译**

|  |  |
| --- | --- |
| **中间代码** | **MIPS32指令** |
| x :=#k | li $t3,k  sw $t3, x的偏移量($sp) |
| x := y | lw $t1, y的偏移量($sp)  move $t3,$t1  sw $t3, x的偏移量($sp) |
| x := y + z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  add $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y - z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  sub $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y \* z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  mul $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y / z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  //mul $t3,$t1,$t2  div $t1,$t2  mflo $t3  sw $t3, x的偏移量($sp) |
| RETURN x | lw $v0, x的偏移量($sp)  jr $ra |
| IF x==y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  beq $t1,$t2,z |
| IF x!=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bne $t1,$t2,z |
| IF x>y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bgt $t1,$t2,z |
| IF x>=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bge $t1,$t2,z |
| IF x<y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  ble $t1,$t2,z |
| IF x<=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  blt $t1,$t2,z |
| X:=CALL f |  |

# 4系统测试与评价

## 4.1 测试用例

用于测试词法分析和语法分析，生成AST的程序用例test.c为：

/\* I'm blahblahblahing\*/

//again

int a,b,c;

float m,n;

int fibo(int a){

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

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

}

int main(){

int m,n,i;

char d;

m=read();

i=1;

++i;

d='x';

i+=2;

while(i<=m)

{

n=fibo(i);

write(n);

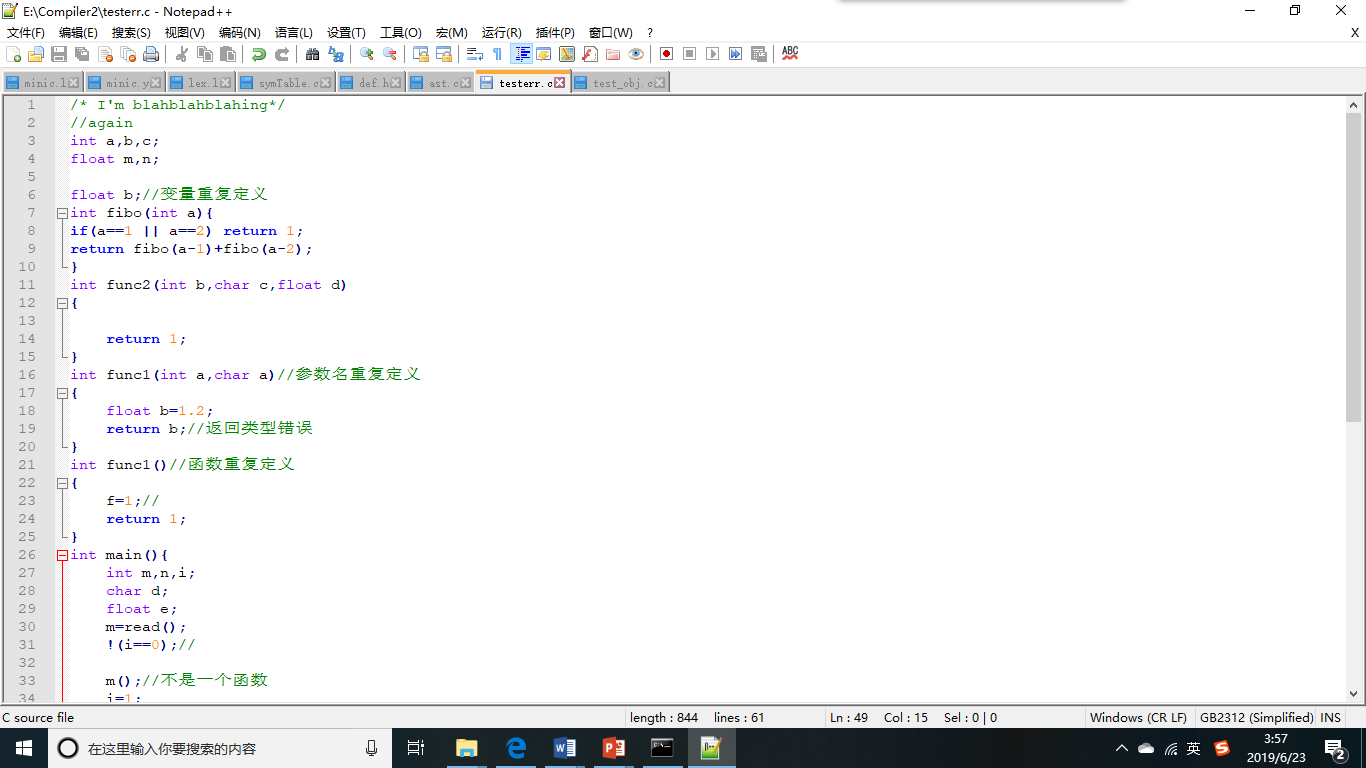
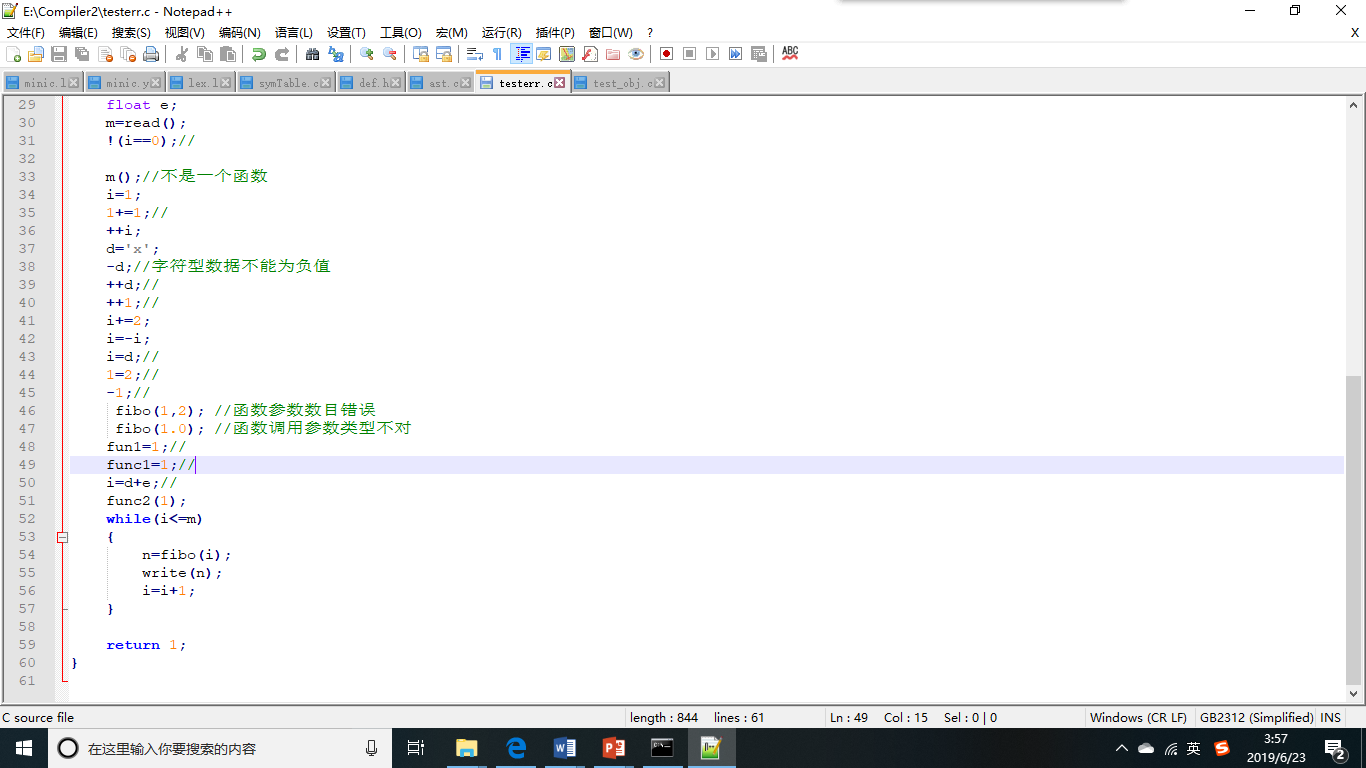
i=i+1;

}

return 1;

}

用于语义分析测试的测试程序testerr.c：

**图4-1 testerr.c源代码**

用于生成中间代码和目标代码的源程序test\_obj.c：

int a,b,c;

float m,n;

int fibo(int a)

{

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

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

}

int main()

{

int m,n,i;

m = read();

i=1;

while(i<=m)

{

n = fibo(i);

write(n);

i=i+1;

}

return 1;

}

## 4.2 正确性测试

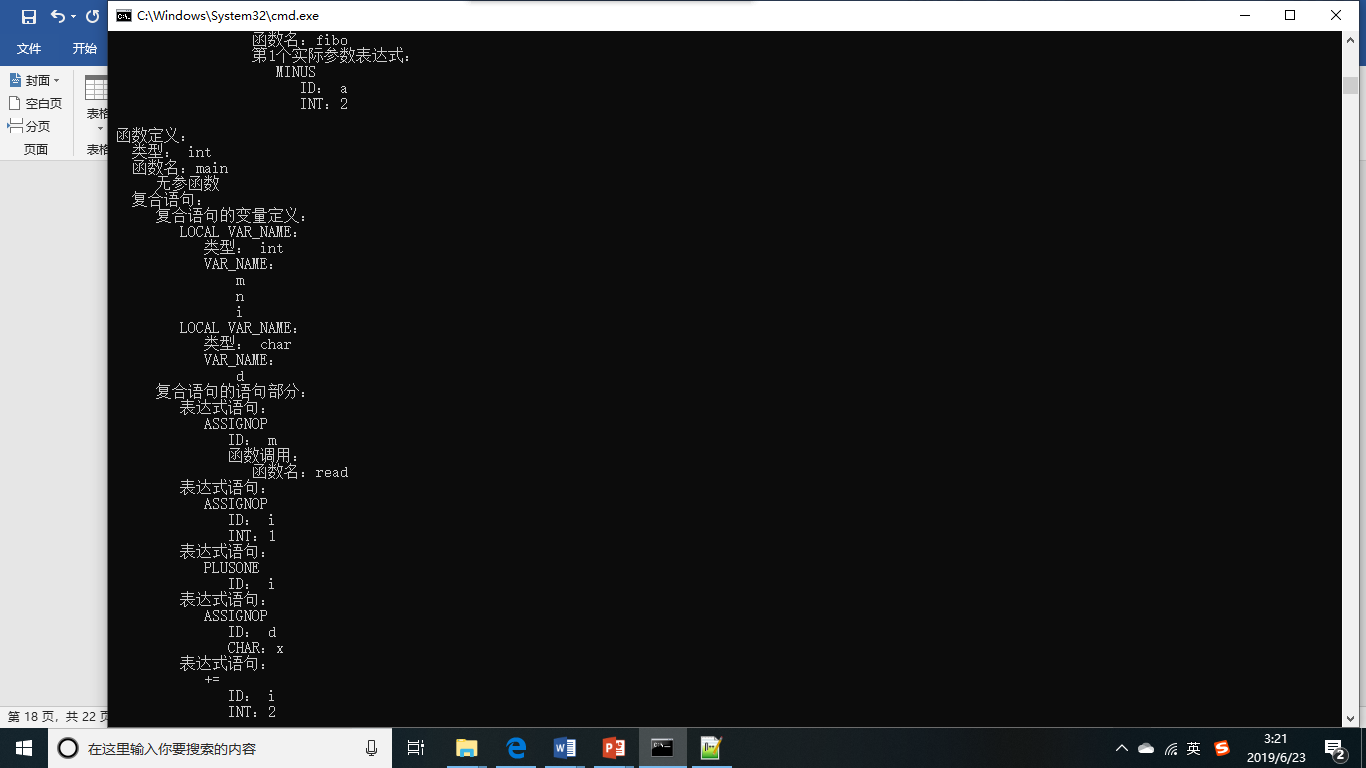
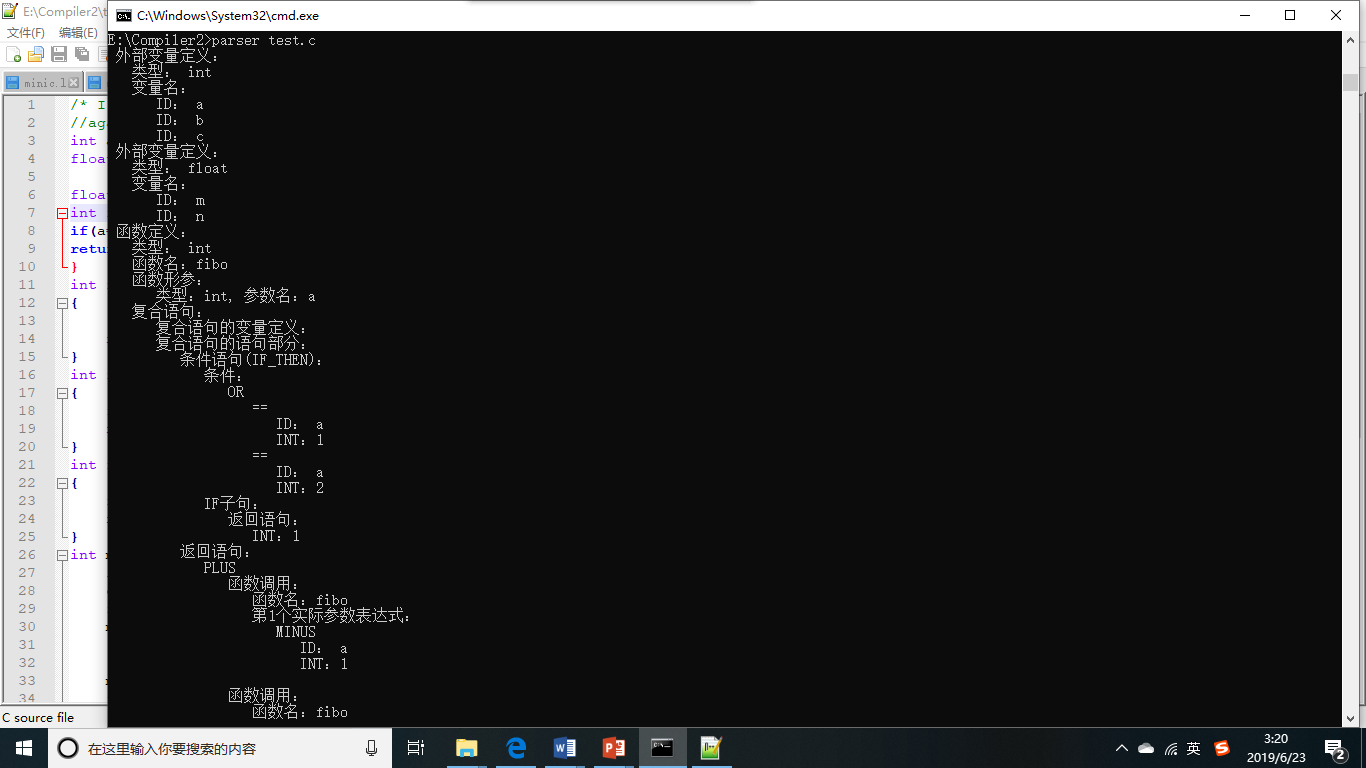
parser.exe生成AST

error.exe产生报错信息并在部分位置打印符号表

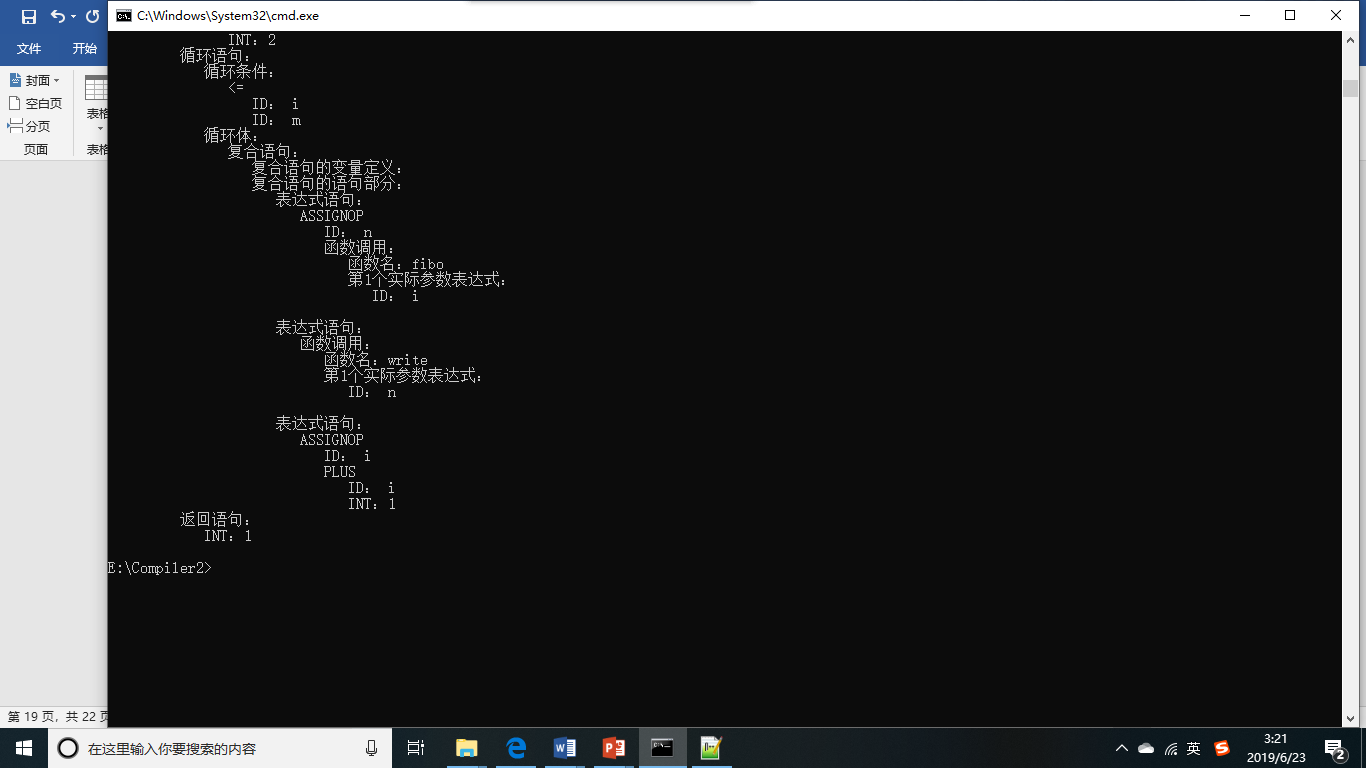
midcode.exe打印中间代码

objcode.exe产生目标代码到文件objectcode.S

对于源代码test.c,parser.exe生成的AST显示如下：

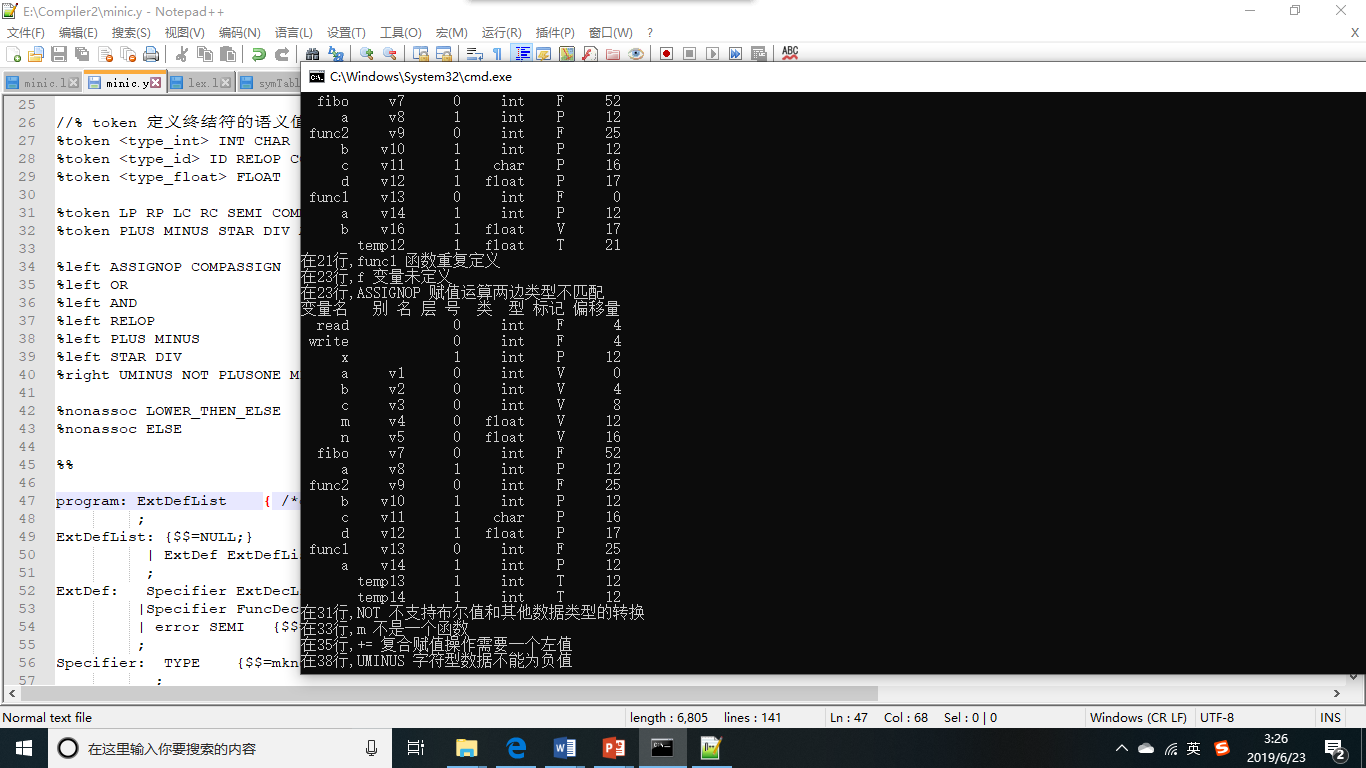


**图4-2 test.c输出AST1**



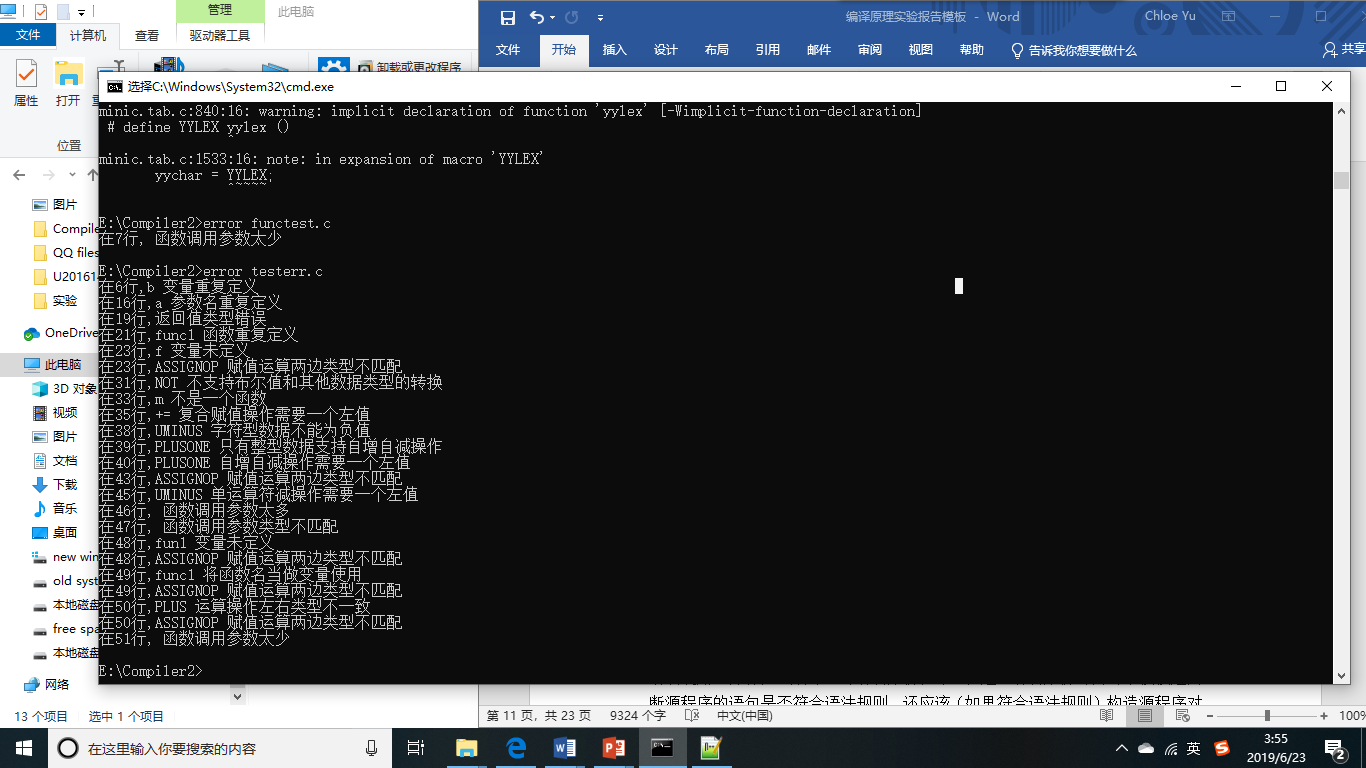
**图4-3 test.c 输出AST2**

对于源代码testerr.c ,error.exe打印出的其中一个符号表如图4-4所示



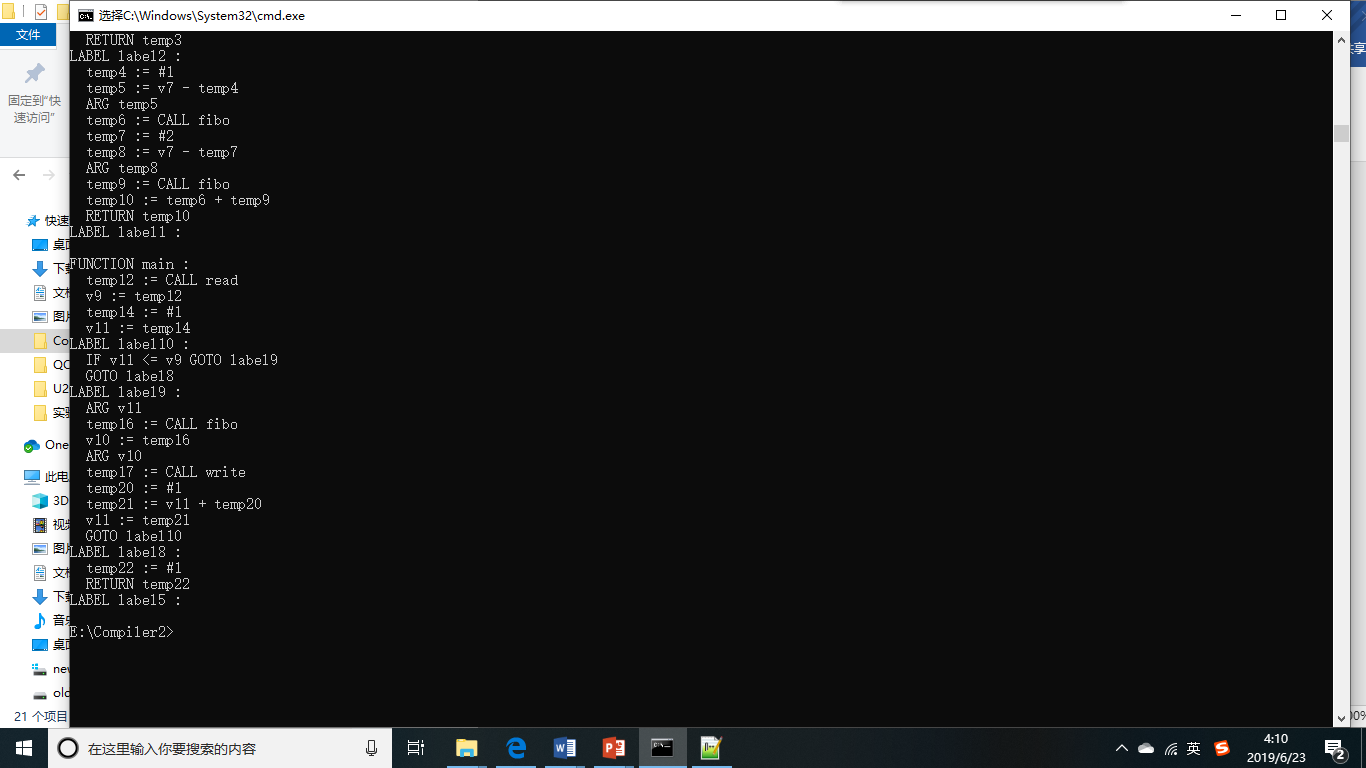
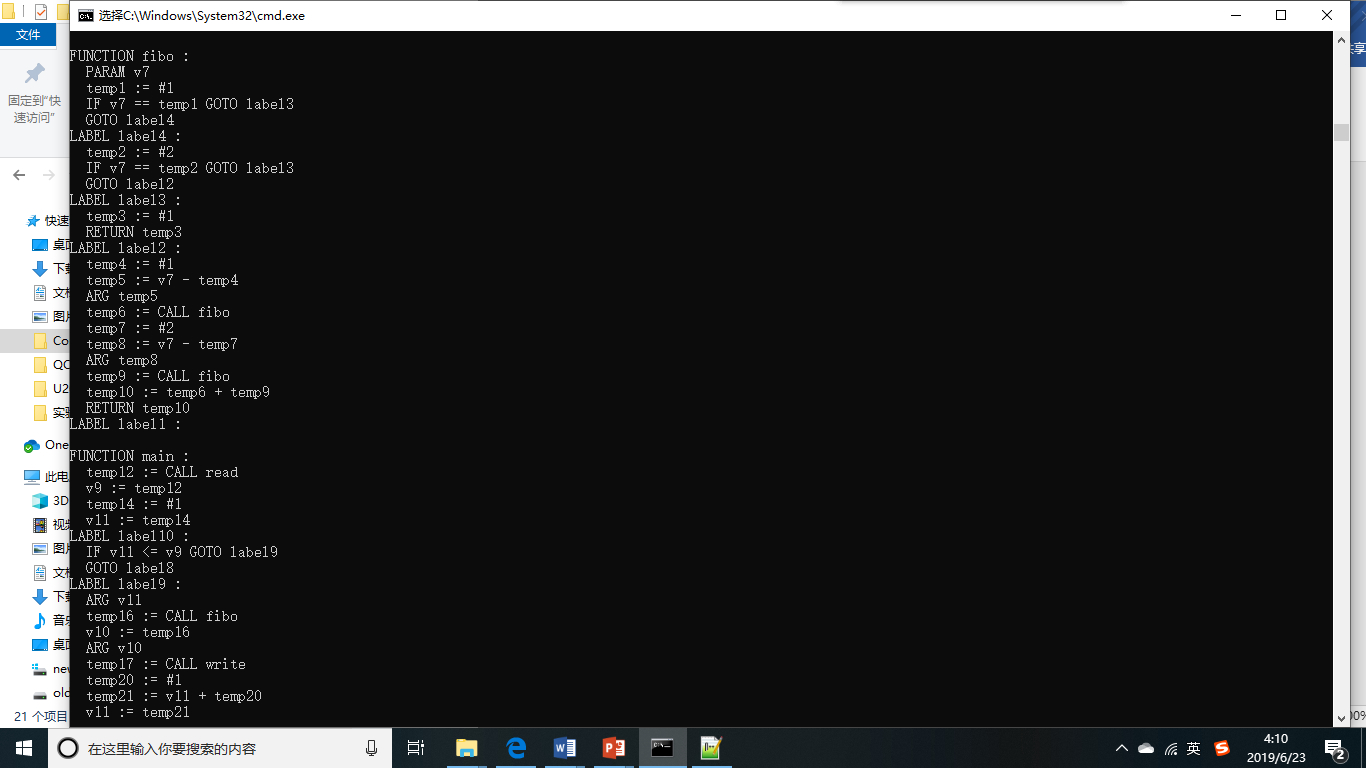
**图4-4符号表打印**

并报错如下图4-5所示，可看出表2-1中设计的所有类型的错误均可进行报错。



**图4-5报错信息**

对于源代码test\_obj.c , midcode.exe打印出的中间代码如下所示：



**图4-6中间代码**

对于源代码test\_obj.c , objcode.exe生成的目标代码如下：

.data

\_Prompt: .asciiz "Enter an integer: "

\_ret: .asciiz "\n"

.globl main

.text

read:

li $v0,4

la $a0,\_Prompt

syscall

li $v0,5

syscall

jr $ra

write:

li $v0,1

syscall

li $v0,4

la $a0,\_ret

syscall

move $v0,$0

jr $ra

fibo:

li $t3, 1

sw $t3, 16($sp)

lw $t1, 12($sp)

lw $t2, 16($sp)

beq $t1,$t2,label3

j label4

label4:

li $t3, 2

sw $t3, 16($sp)

lw $t1, 12($sp)

lw $t2, 16($sp)

beq $t1,$t2,label3

j label2

label3:

li $t3, 1

sw $t3, 20($sp)

lw $v0,20($sp)

jr $ra

label2:

li $t3, 1

sw $t3, 24($sp)

lw $t1, 12($sp)

lw $t2, 24($sp)

sub $t3,$t1,$t2

sw $t3, 28($sp)

move $t0,$sp

addi $sp, $sp, -52

sw $ra,0($sp)

lw $t1, 28($t0)

move $t3,$t1

sw $t3,12($sp)

jal fibo

lw $ra,0($sp)

addi $sp,$sp,52

sw $v0,32($sp)

li $t3, 2

sw $t3, 36($sp)

lw $t1, 12($sp)

lw $t2, 36($sp)

sub $t3,$t1,$t2

sw $t3, 40($sp)

move $t0,$sp

addi $sp, $sp, -52

sw $ra,0($sp)

lw $t1, 40($t0)

move $t3,$t1

sw $t3,12($sp)

jal fibo

lw $ra,0($sp)

addi $sp,$sp,52

sw $v0,44($sp)

lw $t1, 32($sp)

lw $t2, 44($sp)

add $t3,$t1,$t2

sw $t3, 48($sp)

lw $v0,48($sp)

jr $ra

label1:

main:

addi $sp, $sp, -52

addi $sp, $sp, -4

sw $ra,0($sp)

jal read

lw $ra,0($sp)

addi $sp, $sp, 4

sw $v0, 24($sp)

lw $t1, 24($sp)

move $t3, $t1

sw $t3, 12($sp)

li $t3, 1

sw $t3, 28($sp)

lw $t1, 28($sp)

move $t3, $t1

sw $t3, 20($sp)

label10:

lw $t1, 20($sp)

lw $t2, 12($sp)

ble $t1,$t2,label9

j label8

label9:

move $t0,$sp

addi $sp, $sp, -52

sw $ra,0($sp)

lw $t1, 20($t0)

move $t3,$t1

sw $t3,12($sp)

jal fibo

lw $ra,0($sp)

addi $sp,$sp,52

sw $v0,32($sp)

lw $t1, 32($sp)

move $t3, $t1

sw $t3, 16($sp)

lw $a0, 16($sp)

addi $sp, $sp, -4

sw $ra,0($sp)

jal write

lw $ra,0($sp)

addi $sp, $sp, 4

li $t3, 1

sw $t3, 40($sp)

lw $t1, 20($sp)

lw $t2, 40($sp)

add $t3,$t1,$t2

sw $t3, 44($sp)

lw $t1, 44($sp)

move $t3, $t1

sw $t3, 20($sp)

j label10

label8:

li $t3, 1

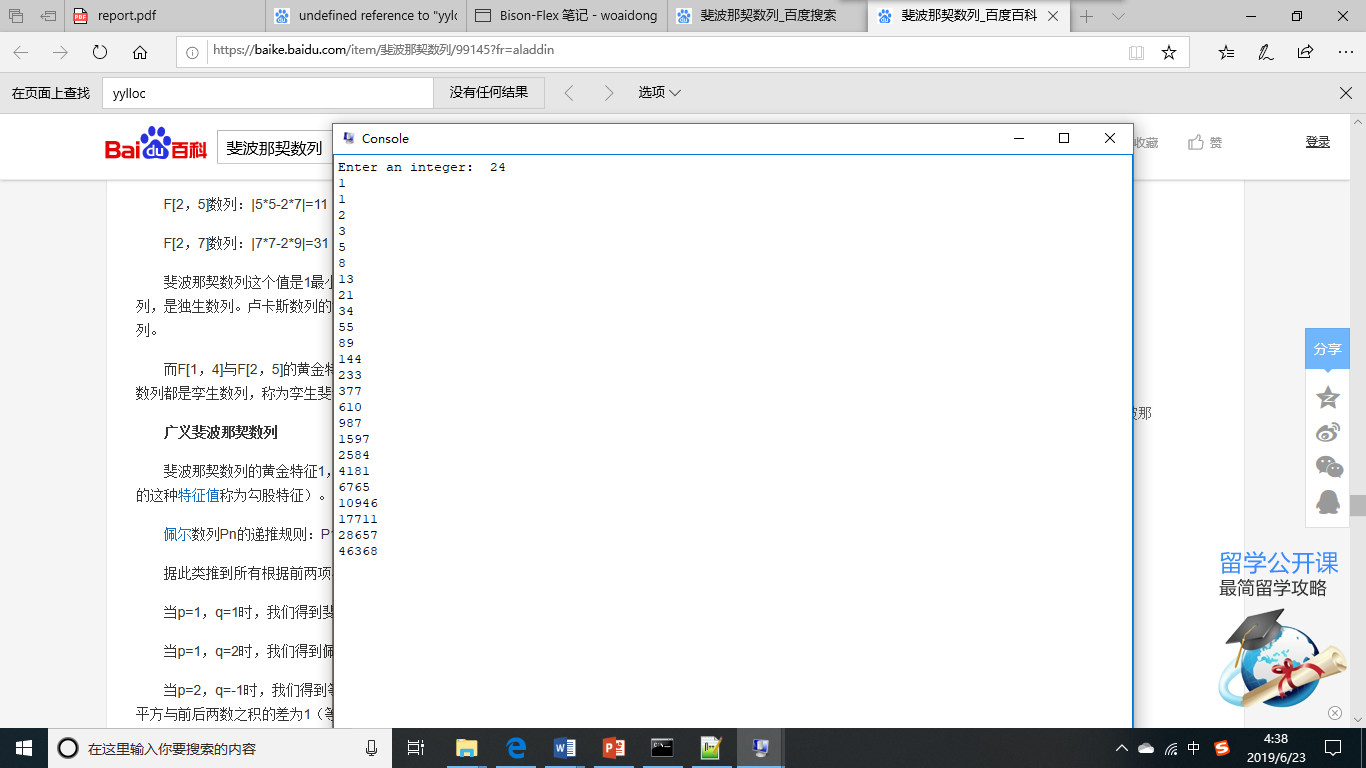
sw $t3, 48($sp)

lw $v0,48($sp)

jr $ra

label5:

将以上代码加载到QtSpim中，并运行，结果如下图所示：



**图4-7目标代码运行**

## 4.3 系统的优点

1、数据类型包括char类型、int类型和float类型；基本运算包括算术运算、比较运算、逻辑运算、自增自减运算和复合赋值运算；控制语句包括if语句和while语句。

2、能够以形式化、结构化、表格化的形式直观美观地打印出编译阶段所用 到的符号表。

## 4.4 系统的缺点

不支持比较复杂的数据类型，以及比较复杂的控制语句。

# 5实验小结或体会

由于指导书附录中提供了参考代码，编译原理实验虽然实际代码量不算太大，但是要真正弄懂编译每一步的实现耗时还是很长的。在实验中，我学会了如何编写词法分析器的正规式（正规表达式）和使用自动化生成工具 Flex 得到词法分析程序；学会了如何使用自动化生成工具 Bison 得到语法分析程序，以及如何在 Bison 环境中使用属性文法、语义动作增强文法的表达能力；学会了如何利用源程序的语法分析树，对其进行语义分析；学会了如何对源程序进行分析而能得到其符号表；学会了中间代码的常用形式三地址码，以及如何将源代码转化为中间代码；学会了如何将中间代码翻译成目标代码，以及如何理解函数调用过程包括栈帧维护、现场保存等。

自己从头到尾实现一个简易的编译器，真的十分有助于自己理解整个编译过 程。在课堂上，感觉上课的重心是词法和语法的理论知识，但是实践起来才能体会到编译的整个工程的复杂性。要确定遍历次数，每一遍的工作划分。要思考语义分析和生成中间代码需要哪些属性，并将其加入树的节点结构。只有当自己亲身去实践去体会时，才能将课本上所讲述的编译器的每一阶段的方法真正实现出来，才能真正理解这些方法的有效性。

本次实验只是实现了一个非常简单的编译器，如果要实现完全支持某面向对象语言，并且还能支持不同级别优化的编译器可想是非常复杂的。而一门编程语言，如果没有好的编译器支持，是没有生命的。虽然目前有一些国产的编译器，包括华为刚刚推出的据称大大提高了安卓使用流畅度的方舟编译器，但是整体还是依赖于国外开发出的编译器。希望我们国家能够渐渐地，把过去落后于别人的基础技术补回来，至少能有自己的一系列硬件生产技术与软件的开发环境，能够不靠人不求人。

**参考文献**

[1] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

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

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008

**附件：源代码**

**minc.l**

%{

#include "minic.tab.h"

#include "string.h"

#include "def.h"

int yycolumn=1;

#define YY\_USER\_ACTION yylloc.first\_line=yylloc.last\_line=yylineno; \

yylloc.first\_column=yycolumn; yylloc.last\_column=yycolumn+yyleng-1; yycolumn+=yyleng;

typedef union {

int type\_int;

int type\_float;

char type\_id[32];

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 ['][A-Za-z][']

%%

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

{char} {yylval.type\_int=(int)yytext[1]; return CHAR;}

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

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

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

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

"return" {return RETURN;}

"if" {return IF;}

"else" {return ELSE;}

"while" {return WHILE;}

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

";" {return SEMI;}

"," {return COMMA;}

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

"=" {return ASSIGNOP;}

"+="|"-="|"\*="|"/=" {strcpy(yylval.type\_id, yytext);;return COMPASSIGN;}

"+" {return PLUS;}

"++" {return PLUSONE;}

"--" {return MINUSONE;}

"-" {return MINUS;}

"\*" {return STAR;}

"/" {return DIV;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"{" {return LC;}

"}" {return RC;}

\/\/.\* ;

\/\\*(.\*\n)\*.\*\\*\/ ;

[\n] {yycolumn=1;}

[ \r\t] {}

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

%%

/\*

void main()

{

yylex();

}

\*/

int yywrap()

{

return 1;

}

**minic.y**

%error-verbose

%locations

%{

#include "stdio.h"

#include "math.h"

#include "string.h"

#include "def.h"

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

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

void display(struct node \*,int);

%}

%union {

int type\_int;

float type\_float;

char type\_id[32];

struct node \*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

//% token 定义终结符的语义值类型

%token <type\_int> INT CHAR //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 COMPASSIGN 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 LP RP LC RC SEMI COMMA //用bison对该文件编译时，带参数-d，生成的exp.tab.h中给这些单词进行编码，可在lex.l中包含parser.tab.h使用这些单词种类码

%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN MINUSONE PLUSONE

%left ASSIGNOP COMPASSIGN

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT PLUSONE MINUSONE

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

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

;

ExtDefList: {$$=NULL;}

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

;

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

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

| error SEMI {$$=NULL; }

;

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

;

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

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

;

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

;

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

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

;

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

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

;

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

;

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

;

StmList: {$$=NULL; }

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

;

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

| 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(RETURN,$2,NULL,NULL,yylineno);}

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

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

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

;

DefList: {$$=NULL; }

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

;

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

;

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

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

;

Dec: VarDec {$$=$1;}

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

;

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

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

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

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

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

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

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

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

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

| LP Exp RP {$$=$2;}

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

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

| PLUSONE Exp %prec NOT {$$=mknode(PLUSONE,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"PLUSONE");}//前置

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

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

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

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

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

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

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

;

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

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

;

%%

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

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

if (!yyin) return -1;

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");

}

**symTable.c**

#include "def.h"

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

{

static char result[10];

strcpy(result,s1);

strcat(result,s2);

return result;

}

char \*newAlias()

{

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("v",s);

}

char \*newLabel()

{

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("label",s);

}

char \*newTemp()

{

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("temp",s);

}

//生成一条TAC代码的结点组成的双向循环链表，返回头指针

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

{

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

h->op=op;

h->opn1=opn1;

h->opn2=opn2;

h->result=result;

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

return h;

}

//生成一条标号语句，返回头指针

struct codenode \*genLabel(char \*label)

{

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

h->op=LABEL;

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

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

return h;

}

//生成GOTO语句，返回头指针

struct codenode \*genGoto(char \*label)

{

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

h->op=GOTO;

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

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

return h;

}

//合并多个中间代码的双向循环链表，首尾相连

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

{

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

va\_list ap;

va\_start(ap,num);

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

while (--num>0)

{

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

if (h1==NULL)

h1=h2;

else if (h2)

{

t1=h1->prior;

t2=h2->prior;

t1->next=h2;

t2->next=h1;

h1->prior=t2;

h2->prior=t1;

}

}

va\_end(ap);

return h1;

}

//输出中间代码

void prnIR(struct codenode \*head)

{

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

FILE \*fp;

fp = fopen("objectcode.txt","rw+");

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 PLUS:

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

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

break;

case MINUS:

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

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

break;

case STAR:

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

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

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

break;

case DIV:

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

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

break;

case FUNCTION:

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

break;

case PARAM:

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

break;

case LABEL:

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

break;

case GOTO:

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

break;

case JLE:

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

break;

case JLT:

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

break;

case JGE:

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

break;

case JGT:

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

break;

case EQ:

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

break;

case NEQ:

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

break;

case ARG:

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

break;

case CALL:

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

break;

case RETURN:

if (h->result.kind){

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

}

else

{

printf(" RETURN\n");

}

break;

}

h=h->next;

}

while (h!=head);

}

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

{

//这里可以只收集错误信息，最后一次显示

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

}

void prn\_symbol() //显示符号表

{

int i=0;

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

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

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

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

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

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

}

int searchSymbolTable(char \*name)

{

int i;

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

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

return i;

return -1;

}

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

}

int LEV=0; //层号

int func\_size; //1个函数的活动记录大小

/\*ExtDecList: VarDec {$$=$1;}

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

;

\*/

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

{

int rtn,num=1;

switch (T->kind)

{

case EXT\_DEC\_LIST:

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

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

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

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

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

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

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

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

//在ExtDef再统计width

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 node \*T)

{

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

int type1,type2;

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

return 1;

int pos=T->pos;

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

{

if (T==NULL)

{

semantic\_error(pos,"", "函数调用参数太少");

return 0;

}

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

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

if (type1!=type2)

{

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

return 0;

}

T=T->ptr[1];

}

if (T) //num个参数已经匹配完，还有实参表达式

{

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

return 0;

}

return 1;

}

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

{

struct opn opn1,opn2,result;

int op;

int rtn;

if (T)

{

switch (T->kind)

{

case INT:

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

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

else

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

T->width=0;

break;

case FLOAT:

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

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

else

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

T->width=0;

break;

case 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, "将函数名当做变量使用");

else

{

opn1.kind=ID;

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

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

opn2.kind=INT;

opn2.const\_int=0;

result.kind=ID;

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

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

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

}

T->width=0;

break;

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

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

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

opn1.kind=ID;

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

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

opn2.kind=ID;

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

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

result.kind=ID;

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

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

op=JLT;

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

op=JLE;

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

op=JGT;

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

op=JGE;

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

op=EQ;

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

op=NEQ;

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

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

break;

case AND:

case OR:

if (T->kind==AND)

{

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

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

}

else

{

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

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

}

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

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

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

boolExp(T->ptr[0]);

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

boolExp(T->ptr[1]);

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

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

if (T->kind==AND)

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

else

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

break;

case NOT:

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

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

boolExp(T->ptr[0]);

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

break;

}

}

}

void Exp(struct node \*T)

{

//处理基本表达式，参考文献[2]p82的思想

int rtn,num,width;

struct node \*T0;

struct opn opn1,opn2,result;

if (T)

{

switch (T->kind)

{

case ID: //查符号表，获得符号表中的位置，类型送type

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

if (rtn==-1)

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

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

semantic\_error(T->pos,T->type\_id, "将函数名当做变量使用");

else

{

T->place=rtn; //结点保存变量在符号表中的位置

T->code=NULL; //标识符不需要生成TAC

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

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

T->width=0; //未再使用新单元

}

break;

case INT:

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

T->type=INT;

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;

//生成 result(newAlias)=opn1;

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

T->width=4;

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

T->type=CHAR;

opn1.kind=CHAR;

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=1;

break;

case FLOAT:

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

T->type=FLOAT;

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=4;

break;

case ASSIGNOP:

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

{

semantic\_error(T->pos, T->type\_id, "赋值运算两边类型不匹配");

}

else

{

Exp(T->ptr[0]); //处理左值，例中仅为变量

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

Exp(T->ptr[1]);

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

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

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

}

break;

//不允许在非控制语句中出现&& ||等运算符,控制语句的exp都由boolexp处理了，这里直接报错

case AND:

case OR:

case RELOP:

semantic\_error(T->pos,T->type\_id, "不支持布尔值和其他数据类型的转换");

break;

case COMPASSIGN:

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

// 要求第一个子exp是ID，需要做检查

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

{

semantic\_error(T->pos,T->type\_id, "复合赋值操作需要一个左值");

}

int type = (!strcmp(T->type\_id,"+="))? PLUS:(!strcmp(T->type\_id,"-="))?MINUS:(!strcmp(T->type\_id,"\*="))?STAR:DIV;

// 将当前的节点替换成一个ASSIGNOP节点

// x=x+exp

struct node \* temp\_op = mknode(type,T->ptr[0],T->ptr[1],NULL,T->pos);

strcpy(temp\_op->type\_id,(!strcmp(T->type\_id,"+="))? "PLUS":(!strcmp(T->type\_id,"-="))?"MINUS":(!strcmp(T->type\_id,"\*="))?"STAR":"DIV");

T->parent->ptr[T->seq\_num]=mknode(ASSIGNOP,T->ptr[0],temp\_op,NULL,T->pos);

strcpy(T->type\_id,"ASSIGNOP");

T=T->parent->ptr[T->seq\_num];

T->code=NULL;

Exp(T);//调用时生成code

// prnIR(T->code);

// display(T,T->level);

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]);

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

{

semantic\_error(T->pos,T->type\_id, "运算操作左右类型不一致");

}

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

//下面的类型属性计算，没有考虑错误处理情况

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

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

else

T->type=INT,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);

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.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.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 NOT:

semantic\_error(T->pos,T->type\_id, "不支持布尔值和其他数据类型的转换");

break;

//只支持前置自增自减

case PLUSONE:

case MINUSONE:

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

// 要求子exp是ID，需要做检查

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

{

semantic\_error(T->pos,T->type\_id, "自增自减操作需要一个左值");

break;

}

Exp(T->ptr[0]); //处理ID语句，从符号表查找变量类型等

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

{

semantic\_error(T->pos,T->type\_id, "只有整型数据支持自增自减操作");

break;

}

else

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

// 将当前的节点替换成一个ASSIGNOP节点

// x=x+1

struct node \* temp1 = mknode(INT,NULL,NULL,NULL,T->pos);

temp1->type\_int=1;

T->type=INT;

struct node \* temp\_add = mknode((T->kind==PLUSONE)? PLUS:MINUS,T->ptr[0],temp1,NULL,T->pos);

strcpy(temp\_add->type\_id,(T->kind==PLUSONE)?"PLUS":"MINUS");

T->parent->ptr[T->seq\_num]=mknode(ASSIGNOP,T->ptr[0],temp\_add,NULL,T->pos);

strcpy(T->type\_id,"ASSIGNOP");

T=T->parent->ptr[T->seq\_num];

T->code=NULL;

Exp(T);//调用时，将临时变量 INT 1 加入符号表

// prnIR(T->code);

// display(T,T->level);

break;

case UMINUS:

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

// 要求子exp是计算，需要做检查

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

{

semantic\_error(T->pos,T->type\_id, "单运算符减操作需要一个左值");

break;

}

Exp(T->ptr[0]);

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

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

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

{

semantic\_error(T->pos,T->type\_id, "字符型数据不能为负值");

break;

}

else

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

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(UMINUS,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, "不是一个函数");

break;

}

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

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

if (T->ptr[0])

{

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

Exp(T->ptr[0]); //处理所有实参表达式求值，及类型

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

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

}

else

{

T->width = width;

T->code = NULL;

}

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

//处理参数列表的中间代码

T0 = T->ptr[0];

while (T0)

{

result.kind = ID;

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

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

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

T0 = T0->ptr[1];

}

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

opn1.kind = ID;

strcpy(opn1.id, T->type\_id); //保存函数名

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

result.kind = ID;

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

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

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

break;

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

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

Exp(T->ptr[0]);

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

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

if (T->ptr[1])

{

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

Exp(T->ptr[1]);

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

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

}

break;

}

}

}

void semantic\_Analysis(struct node \*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 node \*T0;

struct opn opn1,opn2,result;

if (T)

{

switch (T->kind)

{

case EXT\_DEF\_LIST:

if (!T->ptr[0])

break;

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

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

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

if (T->ptr[1])

{

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

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

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

}

break;

//ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT\_VAR\_DEF,$1,$2,NULL,yylineno);}

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

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

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

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

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

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

T->code=NULL; //这里假定外部变量不支持初始化

break;

case FUNC\_DEF: //填写函数定义信息到符号表

T->ptr[1]->type=!strcmp(T->ptr[0]->type\_id,"int")?INT:FLOAT;//获取函数返回类型送到含函数名、参数的结点

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

T->offset=DX; //设置局部变量在活动记录中的偏移量初值

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

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

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

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

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

//计算活动记录大小,这里offset属性存放的是活动记录大小，不是偏移

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

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

break;

case FUNC\_DEC: //根据返回类型，函数名填写符号表

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

if (rtn==-1)

{

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

break;

}

else

T->place=rtn;

result.kind=ID;

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

result.offset=rtn;

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

T->offset=DX; //设置形式参数在活动记录中的偏移量初值

if (T->ptr[0]) //判断是否有参数

{

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

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

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

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

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

}

else

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

break;

case PARAM\_LIST: //处理函数形式参数列表

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

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

if (T->ptr[1])

{

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

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

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

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

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

}

else

{

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

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

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

}

break;

case PARAM\_DEC:

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

if (rtn==-1)

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

else

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

T->num=1; //参数个数计算的初始值

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

result.kind=ID;

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

result.offset=T->offset;

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

break;

case COMP\_STM:

LEV++;

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

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

T->width=0;

T->code=NULL;

if (T->ptr[0])

{

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

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

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

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

}

if (T->ptr[1])

{

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

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

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

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

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

}

//prn\_symbol(); //c在退出一个符合语句前显示的符号表

LEV--; //出复合语句，层号减1

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

break;

case DEF\_LIST:

T->code=NULL;

if (T->ptr[0])

{

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

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

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

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

}

if (T->ptr[1])

{

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

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

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

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

}

break;

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

//类似于上面的外部变量EXT\_VAR\_DEF，换了一种处理方法

T->code=NULL;

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

num=0;

T0->offset=T->offset;

T->width=0;

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

while (T0) //处理所以DEC\_LIST结点

{

num++;

T0->ptr[0]->type=T0->type; //类型属性向下传递

if (T0->ptr[1])

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

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

if (T0->ptr[1])

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

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

{

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

if (rtn==-1)

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

else

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

T->width+=width;

}

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

{

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

if (rtn==-1)

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

else

{

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

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

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

opn1.kind=ID;

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

result.kind=ID;

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

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

}

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

}

T0=T0->ptr[1];

}

break;

case STM\_LIST:

if (!T->ptr[0])

{

T->code=NULL; //空语句序列

T->width=0;

break;

}

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

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

else //语句序列仅有一条语句，S.next属性向下传递

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

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

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

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

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

if (T->ptr[1]) //2条以上语句连接,S.next属性向下传递

{

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

T->ptr[1]->offset=T->offset+T->ptr[0]->width; //顺序结构顺序分配单元方式

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

//序列中第1条为表达式语句，返回语句，复合语句时，第2条前不需要标号

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

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

else

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

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

}

break;

case IF\_THEN:

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

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

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

boolExp(T->ptr[0]);

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

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

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

semantic\_Analysis(T->ptr[1]); //if子句

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

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

break; //控制语句都还没有处理offset和width属性

case IF\_THEN\_ELSE:

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

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

//???

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

boolExp(T->ptr[0]); //条件，要单独按短路代码处理

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

//???

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

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

semantic\_Analysis(T->ptr[1]); //if子句

//???

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

T->ptr[2]->offset=T->ptr[1]->offset+T->ptr[1]->width;

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

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

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

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

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

break;

case WHILE:

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

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

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

boolExp(T->ptr[0]); //循环条件，要单独按短路代码处理

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

//???

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

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

semantic\_Analysis(T->ptr[1]); //循环体

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

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

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

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

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

break;

case EXP\_STMT:

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

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

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

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

break;

case RETURN:

if (T->ptr[0])

{

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

Exp(T->ptr[0]);

num=symbolTable.index;

do

num--;

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

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

{

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

T->width=0;

T->code=NULL;

break;

}

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

result.kind=ID;

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

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

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

}

else

{

T->width=0;

result.kind=0;

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

}

break;

case ID:

case INT:

case FLOAT:

case ASSIGNOP:

case PLUSONE:

case MINUSONE:

case COMPASSIGN:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

case NOT:

case UMINUS:

case FUNC\_CALL:

Exp(T); //处理基本表达式

break;

}

}

}

void objectCode(struct codenode \*head)

{

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

struct codenode \*h=head,\*p;

int i;

FILE \*fp;

fp=fopen("objectcode.s","w");

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

fprintf(fp,"\_Prompt: .asciiz \"Enter an integer: \"\n");

fprintf(fp,"\_ret: .asciiz \"\\n\"\n");

fprintf(fp,".globl main\n");

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

fprintf(fp,"read:\n");

fprintf(fp," li $v0,4\n");

fprintf(fp," la $a0,\_Prompt\n");

fprintf(fp," syscall\n");

fprintf(fp," li $v0,5\n");

fprintf(fp," syscall\n");

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

fprintf(fp,"write:\n");

fprintf(fp," li $v0,1\n");

fprintf(fp," syscall\n");

fprintf(fp," li $v0,4\n");

fprintf(fp," la $a0,\_ret\n");

fprintf(fp," syscall\n");

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

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

do {

switch (h->op) {

case ASSIGNOP:

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

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

else {

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

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

}

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

break;

case PLUS:

case MINUS:

case STAR:

case DIV:

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

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

if (h->op==PLUS) fprintf(fp, " add $t3,$t1,$t2\n");

else if (h->op==MINUS) fprintf(fp, " sub $t3,$t1,$t2\n");

else if (h->op==STAR) fprintf(fp, " mul $t3,$t1,$t2\n");

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

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

}

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

break;

case FUNCTION:

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

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

fprintf(fp, " addi $sp, $sp, -%d\n",symbolTable.symbols[h->result.offset].offset);

break;

case PARAM:

break;

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

break;

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

break;

case JLE:

case JLT:

case JGE:

case JGT:

case EQ:

case NEQ:

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

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

if (h->op==JLE) fprintf(fp, " ble $t1,$t2,%s\n", h->result.id);

else if (h->op==JLT) fprintf(fp, " blt $t1,$t2,%s\n", h->result.id);

else if (h->op==JGE) fprintf(fp, " bge $t1,$t2,%s\n", h->result.id);

else if (h->op==JGT) fprintf(fp, " bgt $t1,$t2,%s\n", h->result.id);

else if (h->op==EQ) fprintf(fp, " beq $t1,$t2,%s\n", h->result.id);

else fprintf(fp, " bne $t1,$t2,%s\n", h->result.id);

break;

case ARG:

break;

case CALL: if (!strcmp(h->opn1.id,"read")){

fprintf(fp, " addi $sp, $sp, -4\n");

fprintf(fp, " sw $ra,0($sp)\n");

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

fprintf(fp, " lw $ra,0($sp)\n");

fprintf(fp, " addi $sp, $sp, 4\n");

fprintf(fp, " sw $v0, %d($sp)\n",h->result.offset);

break;

}

if (!strcmp(h->opn1.id,"write")){

fprintf(fp, " lw $a0, %d($sp)\n",h->prior->result.offset);

fprintf(fp, " addi $sp, $sp, -4\n");

fprintf(fp, " sw $ra,0($sp)\n");

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

fprintf(fp, " lw $ra,0($sp)\n");

fprintf(fp, " addi $sp, $sp, 4\n");

break;

}

for(p=h,i=0;i<symbolTable.symbols[h->opn1.offset].paramnum;i++)

p=p->prior;

fprintf(fp, " move $t0,$sp\n");

fprintf(fp, " addi $sp, $sp, -%d\n", symbolTable.symbols[h->opn1.offset].offset);

fprintf(fp, " sw $ra,0($sp)\n");

i=h->opn1.offset+1;

while (symbolTable.symbols[i].flag=='P')

{

fprintf(fp, " lw $t1, %d($t0)\n", p->result.offset);

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

fprintf(fp, " sw $t3,%d($sp)\n", symbolTable.symbols[i].offset);

p=p->next; i++;

}

fprintf(fp, " jal %s\n",h->opn1.id);

fprintf(fp, " lw $ra,0($sp)\n");

fprintf(fp, " addi $sp,$sp,%d\n",symbolTable.symbols[h->opn1.offset].offset);

fprintf(fp, " sw $v0,%d($sp)\n", h->result.offset);

break;

case RETURN:fprintf(fp, " lw $v0,%d($sp)\n",h->result.offset);

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

break;

}

h=h->next;

} while (h!=head);

fclose(fp);

}

void semantic\_Analysis0(struct node \*T)

{

symbolTable.index=0;

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

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

fillSymbolTable("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);

}