

**编译技术课程设计**

院 系 软 件 学 院

专业班级 软件工程1603班

姓 名 刘俊傲

学 号 U201617047

指导教师 祝 建 华

2018年 11月 18日

**目录**

1. **选题背景**
   1. 任务.........................................................
   2. 目标.........................................................
   3. 源语言定义...................................................
2. **实验一 词法分析和语法分析**

2.1 单词文法描述.................................................

2.2 语言文法描述.................................................

2.3 词法分析器的设计.............................................

2.4 语法分析器的设计.............................................

2.5 语法分析器实现结果展示.......................................

**3. 实验二 符号表管和语义检查**

3.1 语义表示方法描述.............................................

3.2 符号表结构定义...............................................

3.3 错误分析实现技术.............................................

3.4 语义分析实现技术.............................................

3.5 语义分析结果展示.............................................

**4. 实验三 中间代码生成**

4.1 中间代码格式定义.............................................

4.2 翻译模式.....................................................

4.3 中间代码生成过程.............................................

4.4 中间代码生成结果展示.........................................

**5. 实验四 目标代码生成**

5.1 目标语言的指令定义......................................................................................

5.2 寄存器的分配.................................................

5.3 目标代码的生成代码及结果展示.................................

**6. 参考文献...............................................**

**1 选题背景**

**1.1 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，提高学生对系统软件编写的能力。

**1.2 目标**

本次课程实践目标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。

**1.3 源语言定义**

在本次编译原理实践中，选mini-c作为源语言。

**2 实验一 词法分析和语法分析**

**2.1 单词文法描述**

单词是mini-c中具有独立意义的最小单位，可分为 5 大类：关键字（保留字）、运算符、界符、常量和标识符。

mini-c中的关键字（它们都是保留字）包括：基本类型关键字int、 float、 bool、struct 和 char的，分支与循环语句涉及的 if、else、while，返回语句 return。

mini-c中的运算符包括：“+ - ++ -- += -= \* / && || ! < > >= <= == !=”，涉及到了算术运算和逻辑运算，值得注意的是，在具体的设计时，小括号（）和中括号[]被当作了运算符而非界符，因为它们在函数调用、数组访问时都有特殊意义，不能当作单纯的分隔符看待。

mini-c 中界符包括：“; , {}”， 小括号（）和中括号[]并不算界符（原因已经解释）。mini-c中的常量包括：整型常量、浮点数常量、字符常量。其中，浮点数常量可以用一般的小数形式表示，例如“6.66666、8888.8”等； 一个整型常量可以是十进制整数，一个十进制整数是一个十进制数字（0-9）的序列，例如“68686”；一个字符常量是被一对单引号包围的可打印 ASCII 字符序列，字符常量中不可以包含换行符，也不可以分成若干行。 mini-c是大小写敏感的，例如if是一个关键字，但是IF却是一个标识符，yibai和 YIBAI 是两个不同的标识符。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，本词法分析程序还会识别注释、空白字符、换行符以及错误的标识符、字符串。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）。

**2.2 语言文法描述**

文法描述以 EBNF 的扩展形式给出，用到的元符号如表 1 所示

表 1 元符号说明表

|  |  |
| --- | --- |
| x（粗体） | 表示x 是一个终结符（即“单词”）。除个别关键字以外，本节中的终结符名字均为小写字母 |
| y（常规） | 表示y 是一个非终结符。非终结符的名字均为首字母大写 |
| <x> | 表示0 或1 个x 的出现，也就是说，x 是可选的 |
| x\* | 表示0、1 或多个x 的出现 |
| x+， | 表示一个或多个以逗号分隔的x |
| | | 表示并列关系 |
| ε | 表示没有，即不存在任何符号 |

**代码规范如下所示：**

//初始语法单元，表示整个程序

program: ExtDefList;

//表示零个或多个 ExtDef

ExtDefList: {$$=NULL;}

|ExtDef ExtDefList

;

//一个全局变量、结构体或函数的定义

ExtDef: Specifier ExtDecList SEMI

|Specifier SEMI

|Specifier FuncDec CompSt

| error SEMI

;

//定义结构体

StructSpecifier: STRUCT OptTag LC DefList RC

| STRUCT Tag

;

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

Specifier: TYPE

| StructSpecifier

;

//结构体名：struct OptTag{...}

OptTag: ID

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

;

//已定义结构体名：OptTag Tag；像 int a一样

Tag: ID

;

ExtDecList: VarDec

| VarDec COMMA ExtDecList

;

//表示对一个变量的定义

VarDec: ID

| VarDec LB INT RB

;

//表示对一个函数头的定义

FuncDec:ID LP VarList RP

|ID LP RP

;

//包括一个或多个ParamDec

VarList: ParamDec

| ParamDec COMMA VarList

;

//对一个形参的定义

ParamDec: Specifier VarDec

;

//表示由一对花括号括起来的语句块；必须在每个语句块的开头才可以定义变量

CompSt: LC DefList StmList RC

;

//零个或多个Stmt的组合，即语句定义集合

StmList: {$$=NULL; }

| Stmt StmList

;

//表示一条语句

Stmt: Exp SEMI

| CompSt

| RETURN Exp SEMI

| IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE

| IF LP Exp RP Stmt ELSE Stmt

| WHILE LP Exp RP Stmt

;

//由多个Def组成

DefList: {$$=NULL; }

| Def DefList

;

//变量定义

Def: Specifier DecList SEMI

;

DecList: Dec

| Dec COMMA DecList

;

Dec: VarDec

| VarDec ASSIGNOP Exp

;

Exp: Exp ASSIGNOP Exp | Exp AND Exp | Exp OR Exp

| Exp RELOP Exp

| Exp PLUS Exp

| Exp MINUS Exp

| Exp STAR Exp

| Exp DIV Exp

| Exp INC

| Exp DEC

| Exp INCASS Exp

| Exp DECASS Exp

| LP Exp RP

| MINUS Exp %prec UMINUS

| NOT Exp

| ID LP Args RP

| ID LP RP

| Exp LB Exp RB

| Exp STRUCT ID

| ID

| INT

| FLOAT

| DOUBLE

| CHAR

;

//实参列表，每个实参都可以变为一个表达式Exp

Args: Exp COMMA Args

| Exp

;

**作用域：**

mini-c支持多种层次的作用域。最高层是全局作用域，其中只包含变量和函数定义。每个函数有一个用于声明参数表的参数作用域和存放函数体的局部作用域。局部作用域中一对大括号建立了一个嵌套的局部作用域。内层作用域屏蔽外层作用域。需要注意以下4 点：

1. 局部作用域中的变量必须先声明后使用
2. 同一个作用域中的标识符是唯一的（mini-c不支持函数重载）
3. 在嵌套的作用域中重新声明的标识符屏蔽外层的同名标识符，但不允许在局部作用域中声明与外层的局部作用域或参数作用域中的变量同名的变量
4. 不可访问在一个已经关闭的作用域中声明的标识符

**类型：**

预定义好的基本类型有 int，float，double, char

**变量：**

变量的类型可以是已定义的基本类型、数组类型或者结构体类型之一。在函数参数表中声明的变量具有参数作用域，而在函数体中声明的变量具有局部作用域。一旦被声明，则该变量保持可见直到该作用域关闭。需要注意的是：局部变量可以在语句序列的任意地方声明，而且在声明点到该声明所在的作用域末之间的区域可访问。

**函数：**

函数必须定义在一个作用域中，函数之间不允许嵌套。函数可以有零或者多个形参。形参的类型可以是已定义的基本类型、数组类型或者结构体类型。用在形参表中的标识符必须唯一（即形参不能重名）。函数的形参是声明在函数体关联的局部作用域之外的另一个作用域中。函数的返回类型可以是任何的基本类型、数组类型或者结构体类型。一个函数只能被定义一次。不支持函数的重载（overload），函数重载是指使用名字相同但类 型签名不同的函数。一个函数的任何 return 语句必须返回一个与该返回类型兼容的值。函数调用中实参的个数必须与函数所需形参的个数相匹配。函数调用中每个实参的类型必须与对应形参的类型相匹配。函数调用时实参的求值顺序是从左至右。函数调用结果的类型是函数声明时候的返回值类型。

**2.3 词法分析器的设计**

词法分析器采用的工具是自动化生成工具 GNUFlex，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NFA，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。

常量的表达式相对复杂。对于空指针和布尔表达式，由于本质上它们属于关 键字，所以正则表达式是引号加自身；对于一般小数形式的浮点数常量，不用考虑特殊形式，则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，所以正则表达式相对简单为“[0-9]+”；对于字符常量，其为由两个单引号扩起来的不包含换行符的字符序列，由此设计其正则表达式为“'.'|[0-9]+”。

根据关键字的定义，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]\*”。

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）。

**2.4 语法分析器设计**

语法分析器的实现采用的是自动化生成工具GNUBison，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。

**设计语法分析器的第一步：**是设计相应的语法规则。

语法规则在 2.2 中的 mini-c 语法规范中已经详细给出，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，我遇到了一个比较大的问题：由mini-c语法规范直接转化来的生成式存在大量的移进-规约冲突或规约规约冲突，需要通过一定的方法来消除冲突。

针对这个问题，即二义性与冲突处理，则是通过显示规定优先级和结合性来解决。经过排查移进-规约冲突和规约-规约冲突的来源，不难发现大部分的冲突来自于运算符，造成的原因便是分析器不知道运算符的优先级和结合性。例如，对于算数表达式“1+2-3”，分析器并不知道是先算“1+2”还是“2-3”。当然，如果告诉分析器“+”和“-”都是左结合，那么分析器自然知道是要先计算“1+2” 而不是是“2-3”。为了解决二义性与冲突，在Bison 中定义的结合性和优先级如下所示：

%left ASSIGNOP

%left OR

%left AND

%left RELOP //运算符

%left INC DEC INCASS DECASS

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT

**设计语法分析器的第二步：**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。这就要求，终结符和非终结符应该都有相关的“属性”作为语义动作的对象。再结合最终目的是构造语法分析树，不难想到这里的“属性”应该选择语法分析树的节点指针。考虑到语法分析树是一棵各节点度都不相等的树，所以决定采用“孩子兄弟法”表示语法分析树，这样可以通过二叉树的相关操作来实现语法分析树的相关操作。

为此，设计并定义语法分析树的节点结构体（C 语言），如下所示：

struct node

{ //以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性

enum node\_kind kind; //结点类型

union {

char type\_id[33]; //由标识符生成的叶结点

int type\_int; //由整常数生成的叶结点

float type\_float; //由浮点常数生成的叶结点

double type\_double;//由双精度浮点数生成的叶节点

char type\_char;//由字符生成的叶节点

};

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

int level; //层号

int place; //表示结点对应的变量或运算结果符号表的位置序号

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

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

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

char op[10];

int type; //结点对应值的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //占数据字节数

};

定义了语法分析树节点结合体，还需设计并实现语法分析树的生成函数和遍历函数。其中，语法分析树生成函数是生成一颗以生成式左部语法单元（非终结符）为根节点、生成式右部所有语法单元为子节点的语法树并返回根节点指针。 两个函数的函数原型如下所示。生成函数的设计思想是借助 C 语言的可变参数（因为不确定产生式的右部到底有多少个语法单元），将生成式右部所有语法单元对应的语法树指针作为可变参数传入。

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

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

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，具体的做法如图 2-3 所示（只给出了部分声明）；

%union {

        int type\_int;

        float type\_float;

double type\_double;

char type\_char;

        char type\_id[32];

        struct node \*ptr;

};

// %type 定义非终结符的语义值类型，如把program绑定到ptr

%type <ptr>

program //初始语法单元

ExtDefList //零个或多个ExtDef

ExtDef //一个全局变量、结构体或函数的定义

StructSpecifier //结构体描述符

OptTag

Tag

Specifier //类型描述符

ExtDecList //零个或多个VarDec

FuncDec //函数头

CompSt //函数体、由花括号括起来的语句块

VarList //形参列表

VarDec //一个变量的定义

ParamDec //一个形参的定义

Stmt //一条语句

StmList //语句列表

DefList //变量定义列表

Def //一条变量定义

DecList

Dec

Exp //一个表达式

Args //实参列表

//% token 定义终结符的语义值类型

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

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

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

%token <type\_double> DOUBLE

%token <type\_char> CHAR

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

%token LP //(

RP //)

LB //[

RB //]

LC //{

RC //}

SEMI //;

COMMA //,

%token PLUS //+

MINUS //-

STAR //\*

DIV // /

ASSIGNOP //=

AND //&&

OR // ||

NOT //!

IF

ELSE

WHILE

RETURN

INC

DEC

INCASS

DECASS

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。对于一般生成式 的语义动作如图 2-4 所示，对于顶层生成式“program: ExtDefList”的语义动作如下所示：

program: ExtDefList { display($1,0);}

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

{ //对抽象语法树的先根遍历

int i = 1;

struct node \*T0;

if (T)

{

switch (T->kind)

{

case EXT\_DEF\_LIST:

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

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

break;

case EXT\_VAR\_DEF:

printf("%\*c外部变量定义：\n", indent, ' ');

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

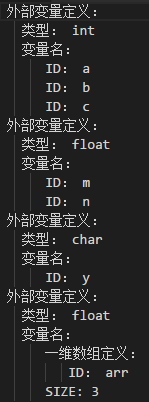
printf("%\*c变量名：\n", indent + 3, ' ');

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

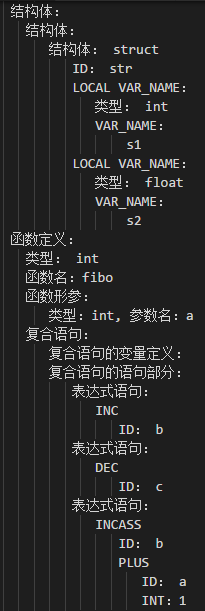
break;

**2.5 语法分析器实现结果展示**

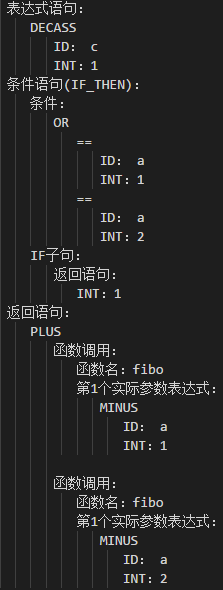
在展示词法分析和语法分析的测试结果前，首先给出所用的测试用例，是一 份合法的 mini-c代码，如图所示：



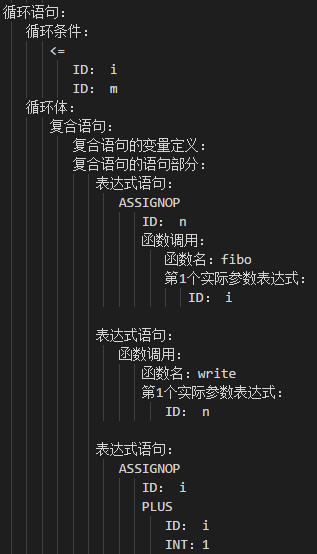
从上图中可知，词法分析器成功地识别并记录了常量int,float,char以及数组等。



从上图中可知，词法分析器成功地识别并记录了常量结构体，函数，局部变量等。



从上图中可知，词法分析器成功地识别并记录了自增，IF 语句，函数调用等。



从上图中可知，词法分析器成功地识别并记录了WHILE 语句，函数调用等。

IMG_256

从上图中可知，词法分析器成功地识别并给出错误提示

通过上图不难看出，词法分析器不仅支持错误提示，还能准确定位错误的词法单元的位置信息，给出其在源程序中的行数和列数，从而方便程序员的修改。

**词法分析功能测试：**

语法分析的输出为语法树。语法树是一颗多叉树，但在设计时采用了孩子兄弟法，因此其在程序中的存储形式为二叉树。打印语法树时，需要对其进行深度优先遍历，显示所有的节点（语法单元：非终结符或者终结符）。

语法分析阶段还可以对源程序的语法正确性检测，并根据检测结果给出相应的报错提示。

**3符号表管和语义检查**

**3.1 语义表示方法描述**

进行语义分析的理论工具是属性文法，属性值可以分成不相交的两类：综合 属性和继承属性。在语法树中，一个节点的综合属性值是从其子节点的属性值计 算而来的，而一个结点的继承属性则是由该节点的父节点和兄弟结点的属性值计 算而来的。为了能够自底向上地进行语义分析，所以在节点的属性方面，主要选择的是综合属性。相对于继承属性，由于一个节点的综合属性的值只依赖于其子节点的属性值，与自底向上的分析方向完全相同，因此更适合自底向上的语义分析。

从编程实现的角度来看，语义分析可以作为编译器里单独的一个模块，也可以并入前面的词法分析模块或者后面的中间代码生成模块。不过，由于这样做其牵扯到的内容较多而且较为复杂，所以还是决定将语义分析单独作为一个模块。在完成了实验一的词法分析和语法分析后，将会得到源程序的语法分析树，自底向上的语义分析完全可以通过对语法树进行深度优先遍历来实现。这样，语义分析便可以作为独立模块而单独进行，而不用在语法分析时进行。实际上，遍历语法树进行语义分析和在语法分析时进行语义分析完全是等价的。

为了进行语义分析，除了需要语法分析阶段得到的语法分析树，还需要借助 符号表进行类型检查等分析。

符号表的构建时机也是一个需要考虑的问题。符号表可以在语法分析时与语法分析树一同构建，也可以在语法分析树得到后对语法树进行一次或多次遍历来构建。综合考虑下，选择了在语法分析时构建语法分析树的同时构建符号表。这要选择所考虑的原因主要有：语法分析树和符号表都是编译器在后续阶段需要多次用到的重要信息，它们记录了源程序不同方面的信息，语法分析树记录的是源程序的语法结构，而符号表记录的是源程序的符号信息，二者虽有差异但却相互弥补，同等重要，因此逻辑上应该在一个阶段生成；在语法分析时若检测到说明语句，可以获得足够的信息来完成符号表的填表。

除了借助符号表进行类型检查外，语义分析阶段还有一个重要职责是符号作用域合法性的检查。对符号作用域合法性的检查，仅仅只靠符号表是不够的，需要通过一个作用域栈来维持当前对程序可见的作用域。因此，符号表也需要按照作用域来进行分类，具体的介绍见3.2节。

**3.2 符号表结构定义**

3.1 节中已经对符号表的分类动机进行了简略介绍，即为了实现符号作用域合法性的检查，需要将符号表按作用域进行分类。

struct symbol

{ //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥

char name[33]; //变量或函数名

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

int type; //变量类型或函数返回值类型

int paramnum; //形式参数个数

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

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

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

//或函数活动记录大小，目标代码生成时使用

//其它...

};

//符号表，是一个顺序栈，index初值为0

struct symboltable

{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

符号表的每条表项应该包括符号的名字（变量名或函数名）、符号类别（变量或函数）、符号类型、描述（对于变量而言该项为空，对于函数该项指向函数的描述）。除了为各个作用域的符号表定义结构外，在3.1节中也提到了，需要通过一个作用域栈来维持当前对程序可见的作用域，进而实现符号作用域合法性的检查。当处理到程序的某一位置时，可以访问的作用域称为开作用域，否则为闭作用域。

需要建立一个栈来管理整个程序的作用域：每打开一个作用域，就把该作用域压入栈中；每关闭一个作用域，就从栈顶弹出该作用域。这样，这个作用域栈中就记录着当前所有打开的作用域的信息，栈顶元素就是当前最内层的作用域。查找一个变量时，按照自栈顶向下的顺序查找栈中各作用域的符号表，最先找到的就是最靠近内层的变量。

//符号表，是一个顺序栈，index初值为0

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

int TX[30];

int top;

} symbol\_scope\_TX;

**3.3 错误类型码定义**

本编译器的语义分析阶段可识别的错误类型如下：

(1) 错误类型 1：变量未定义。

(2) 错误类型 2：变量重复定义。

(3) 错误类型 3：结构体未定义。

(4) 错误类型 4：结构体重复定义。

(5) 错误类型 5：对非结构体变量使用了结构体运算。

(6) 错误类型 6：等式左值为常量。

(7) 错误类型 7：赋值操作等号两边类型不匹配。

(8) 错误类型 8：使用了结构体中没有定义的变量。

(9) 错误类型 10：函数未定义便使用。

(10) 错误类型 11：将函数名作为形参使用（作用域问题）。

(11) 错误类型 12：将函数名作为变量名使用（作用域问题）。

(12) 错误类型 13：数组下标使用了非整型变量或常量。

(13) 错误类型 14：对非数组变量使用了“[]”

(14) 错误类型 15：对非数组变量使用了“[]”

需要注意的是，虽然上述仅列出了14个错误类型，但实际上错误类型数远大于14。因为对于错误类型7而言，可进一步划分，例如函数返回值与变量类型不匹配、父类对象赋给子类等。除此之外，本编译器的语义分析阶段所作的定义和声明都是全局范围的，也就是说，变量和函数可以在未声明或定义前便可使用，但是必须要有声明或定义。同样的道理，子类所继承的父类可以在子类前定义，也可以在子类后定义，但一定要定义父类，否则，语义分析会出错。

语义分析阶段发现的语义错误也会被准确定位，编译器会给出错误的具体位置，包括行号和列号。编译器还可以针对具体的错误类型，给出相关提示。例如 如果赋值运算等号两边的表达式类型不匹配，编译器会给出提示，告之等号两边 的表达式的具体类型是什么。

总之，错误类型和错误提示会帮助程序员准确地找到错误位置和发现错误原因，从而快速修正。

**3.4 语义分析实现技术**

虽然在 3.1 节中对语义分析的过程作了简要的概述，但不够详细，接下来将 详细阐述语义分析阶段的具体实现和各个过程。

**语义分析的第一步：构建符号表。**

构建符号表的重要意义在 3.1 与 3.2 节中已经说明，这里不再累述。综合考虑下，选择了在语法分析时构建语法分析树的同时构建符号表。这要选择所考虑的原因主要有：语法分析树和符号表都是编译器在后续阶段需要多次用到的重要信息，它们记录了源程序不同方面的信息，语法分析树记录的是源程序的语法结构，而符号表记录的是源程序的符号信息，二者虽有差异但却相互弥补，同等重要，因此逻辑上应该在一个阶段生成；在语法分析时若检测到说明语句，可以获得足够的信息来完成符号表的填表操作，因此没有必要单独进行一次语法树的遍历来构造符号表。

综上，符号表的建立被放在了语法分析阶段与语法分析树的构造同时进行。对于使用了 Bison 的语法分析，其过程是自底向上的规约，所以在语法分析阶段，当规约说明语句（包括结构体的定义、变量定义、函数定义等）时，便为相应的符号构造一个对应作用域符号表的表项（指向该表项的指针被放到了语法分析树的对应节点中）；当说明语句作为生成式的右部作为进一步规约时，根据生成式的语义，将说明语句所带的符号表表项传递给左部语法单元或者归并左边的所有说明语句的符号表表项，将归并后的符号表传递给左部语法单元。

以此类推，各符号表便可以从底层逐层传递和累积，形成最终的符号表。这里有个特殊情况，那就是继承关系中的父类指向（指针），由于是自底向上的分析无法在分析时便能确定父类的位置（有可能先引用，后定义），所以需要一个单独的小遍历确定。

**语义分析的第二步：静态语义错误检查。**

本编译器在语义分析阶段的静态 语义检查可以分为两大类，第一类是符号作用域相关检查，第二类是类型检查，这两类检查可以在一次语法树遍历过程内完成。当完成语法分析后，语法分析树与符号表便会均构建完毕，可以随时调用。

本编译器将静态语义错误检查单独写成了一个模块，在语法分析阶段规约到最顶层生成式时，会调用该模块进行语义检查。语义检查整体上就是一次语法树的遍历，自底向上进行语义分析，但是由于类型检查和作用域分析都要用到符号表，所以在遍历语法树的同时需要维护一个作用域栈，遍历的同时进行出栈和入栈操作。

对于作用域相关检查，最重要的便是在语法树遍历时维护作用域栈。当遇到类的定义、函数定义、对象声明、函数调用等需要开启一个新的作用域时，便将相应的作用域入栈，当遇到类似大括号这种语法单元时，就要将关闭当前的作用域关闭，即从作用域栈中弹出闭作用域。当遇到使用一个符号时，就从栈中从栈顶向栈底搜索，第一个搜索到的就是当前作用域最先访问的符号，没找到表明该符号没有声明过。

例如，再遇到右大括号时，表明函数作用域或语句块作用域已结束，需要从作用域栈中弹出相应的作用域符号表。如果当前作用域为类作用域，还需要判断其有没有父类，如果有父类，将父类作用域符号表也弹出；如果当前作用域为函数作用域，则还需要将形参作用域也弹出。

**3.5 语义分析结果展示**

结果展示分为两部分，第一部分为符号表信息展示，第二部分为静态语义错 误检查展示。

**展示编译器生成的符号表与错误检查：**

**测试用例如下：**

int gm(int n)

{

return 1;

}

int main()

{

int b;

int a[2];

float arr[2];

float c, d;

char m;

arr[0]++;

b = 1;

2 = 1;

a[0] = 1;

a[1] = 1.0;

a[3] = 1;

d = b + c;

(b + b)++;

++c;

c++;

m = 1;

m = 'l';

a = a + 1;

c = b;

m();

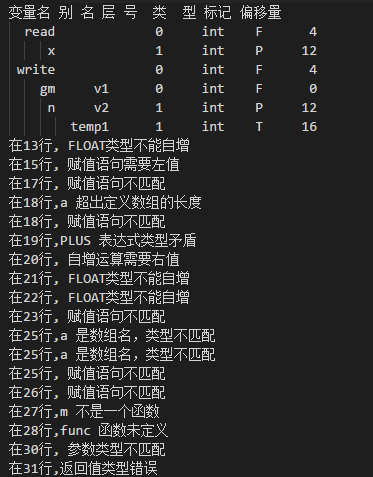
func();

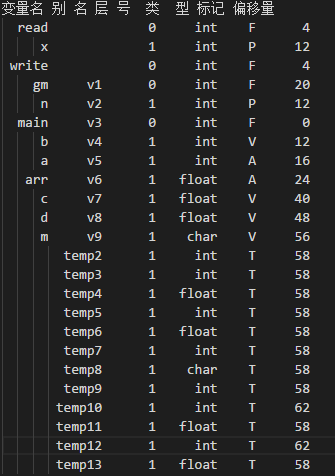
gm(1.0);

return 1.0;

}

**结果显示：**





**4 实验三 中间代码生成**

**4.1 中间代码格式定义**

中间代码是编译器从源语言到目标语言之间采用的一种过渡性质的代码形 式。一方面，中间代码将编译器自然地分为前段和后端两个部分；另一方面，在采用中间代码有利于进行机器无关的优化。

从中间代码所体现出的细节上，可以将中间代码分为高层次中间代码、中层次中间代码、低层次中间代码，而从表现形式上来看，中间代码又可以分为图形中间代码、线形中间代码和混合型中间代码。

在本编译器中，中间代码选择了线形中间代码的“四元式”，也叫三地址码。 这种结构最大的优点是表示简单、处理高效，而缺点就是代码和代码之间的先后 关系有时会模糊整段程序的逻辑，让某些优化操作变得很复杂。

在做本实验时，由于时间关系，因此一开始就没有考虑要进行太复杂的代码 优化。基于这样考虑，线性中间代码的缺点对本实验的影响并不大，再考虑到线性中间代码表现直观、实现简单的优点，因此，便选择了线形中间代码作为本编 译器中间代码的形式。

对于线形中间代码，本编译器选择了“四元式”，具体的格式为：

（OP，DST，SRC1，SRC2）

其中，OP 为操作名称，例如加 Add、减 Sub、乘 Mul 等等；DST、SRC1、SRC2 分别为目的操作数、源操作数 1、源操作数 2 的地址，因此，这种表示方 式也称为“三地址码（TAC）” 。

实现上述三地址码的方式也有很多种，可以采用不同的数据结构来实现，例如静态数组、单链表、双向链表等。本编译器采用了双向链表的形式实现三地址码，因为虽然双向链表增加了一点的实现复杂度，但却也换来了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，并且不存在代码最大行数的限制。

在具体实现过程中，并没有直接将 TAC 就定义成双向链表的形式，而是将其作为链表节点的一个属性（数据成员），之所以这样实现是为了方面后面的扩展（如果有可能）。除此之外，为了实现双向链表，每一个链表结点中还要有指向前置节点和后置节点的指针。

关于 TAC 的定义，实现的形式如前面介绍的其具体格式相同。实现三地址码（TAC）和三地址码双向链表（TAC Code）的具体代码如图 4-1和4-2 所示。

尽管中间代码在编译器的内部存储形式为三地址码（线形中间代码），但是打印出来的中间代码并不是完全按照“（OP，DST，SRC1，SRC2）”的格式，而是一种更为接近汇编语言或者是伪代码的形式，如表4-1 所示。

**表4-1 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 |

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。后续还需要根据个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。

## 4.2 翻译模式

在遍历AST的过程中，完成中间代码的生成。具体的方法是：在这些翻译模式中，每一个文法非终结符通常会对应AST中的一个结点。例如规则A → M..... X..... N的翻译模式：

A → M...... {X的继承属性计算}X..... N {A的综合属性计算}

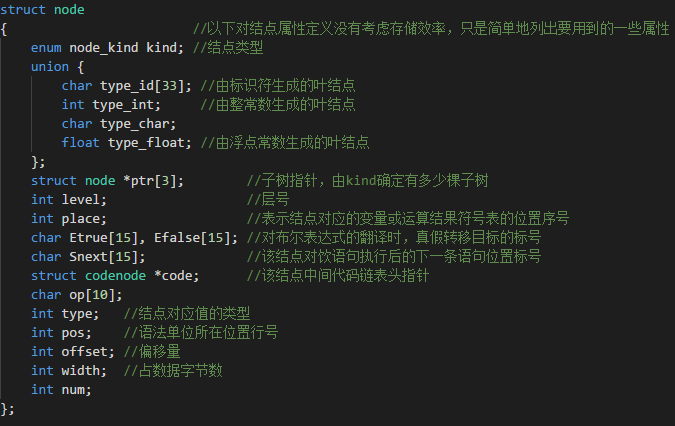
对应的AST中的形式如图4-1所示。



**图4-1 翻译模式规则部分的AST**

在非终结符X的前面有语义属性的计算，表示在遍历到结点X的父结点A，并访问完X左边的所有子树，准备访问该结点X时，可以使用结点A以及结点X之前的结点的属性，进行规则中非终结符X的语义属性计算，这里体现的是非终结符X的继承属性计算；在翻译模式中规则的最后所定义的语义属性计算，表示该规则左部的非终结符A对应的子树全部遍历完成后，从N回到父结点A时，需要完成的语义属性的计算，这里体现的是综合属性的计算。

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。

char \*newTemp()

{

static int no = 1;

char s[10];

itoa(no++, s, 10);

return strcat0("temp", s);

}

**newLabel** 生成一个标号，标号命名形式为**LABEL+序号**。

char \*newLabel()

{

static int no = 1;

char s[10];

itoa(no++, s, 10);

return strcat0("label", s);

}

**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。

//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;

}

**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。

//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;

}

**merge** 将多个语句序列顺序连接在一起。

//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;

}

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。过程可参考课件if\_then\_else语句的翻译过程。

**4.3 中间代码生成过程**

中间代码的生成过程与语义分析阶段中静态语义检查的过程相类似，都是通

过遍历语法分析树并同时维护作用域栈来实现的，区别只在于一个是通过语法单

元判断是否存在语义错误，另一个则是通过语法单元判断如何生成中间代码。所

以，生成中间代码的程序框架借用了实验2 中静态语义错误检查的程序框架，采

用自低向上的方式，递归实现语法分析树的遍历。

在遍历语法分析树的过程中，需要与实验2 一样同时维护作用域栈，以保证

当需要查找符号表时，可以准确定位符号的相关信息。具体的方法同实验2 一样，

所以这里不在累述。

在遍历语法树的过程中，有一步十分重要，那就是合并不同分支产生的中间

代码。所为合并中间代码，实际上是合并两个分支的中间代码对应的双向链表。

双向链表的合并算法在数据结构课程中已经学习过，比单项链表的合并复杂一

些，相关的代码如下所示。

//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;

}

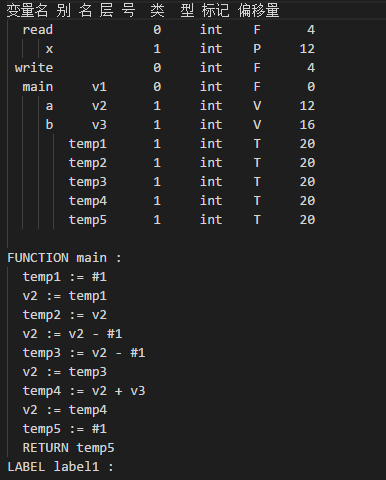
}

va\_end(ap);

return h1;

}

**4.4 中间代码生成结果展示**



**5 实验四 目标代码生成**

**5.1 目标语言的指令定义**

目标语言可选定MIPS32指令序列，可以在SPIM Simulator上运行，SPIM Simulator的安装使用参见文献[2]。TAC指令和MIPS32指令的对应关系如表5-1所示。其中reg(x)表示变量x所分配的寄存器。

**表5-1 中间代码与MIPS32指令对应关系**

|  |  |
| --- | --- |
| **中间代码** | **MIPS32指令** |
| LABEL x | x： |
| x :=#k | li reg(x),k |
| x := y | move reg(x), reg(y) |
| x := y + z | add reg(x), reg(y) , reg(z) |
| x := y - z | sub reg(x), reg(y) , reg(z) |
| x := y \* z | mul reg(x), reg(y) , reg(z) |
| x := y / z | div reg(y) , reg(z)  mflo reg(x) |
| GOTO x | j x |
| RETURN x | move $v0, reg(x)  jr $ra |
| IF x==y GOTO z | beq reg(x),reg(y),z |
| IF x!=y GOTO z | bne reg(x),reg(y),z |
| IF x>y GOTO z | bgt reg(x),reg(y),z |
| IF x>=y GOTO z | bge reg(x),reg(y),z |
| IF x<y GOTO z | ble reg(x),reg(y),z |
| IF x<=y GOTO z | blt reg(x),reg(y),z |
| X:=CALL f | jal f  move reg(x),$v0 |

## 5.2 寄存器的分配

当选择朴素的寄存器分配方案后，目标代码生成时，每当运算操作时，都需要将操作数读入到寄存器中，运算结束后将结果写到对应的单元。由于选择朴素的寄存器分配，只会用到几个寄存器，这里约定操作数使用$t1和$t2，运算结果使用$t3，翻译的方法如表5-2所示。

**表5-2 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 | move $v0, x的偏移量($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 |  |

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。思考一下main函数的活动记录如何处理？

（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。

## 5.3 目标代码的生成代码及结果展示

**输出目标代码，read和write：**

void objectCode(struct codenode \*head)

{

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

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

int i;

FILE \*fp;

fp = fopen("object.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");

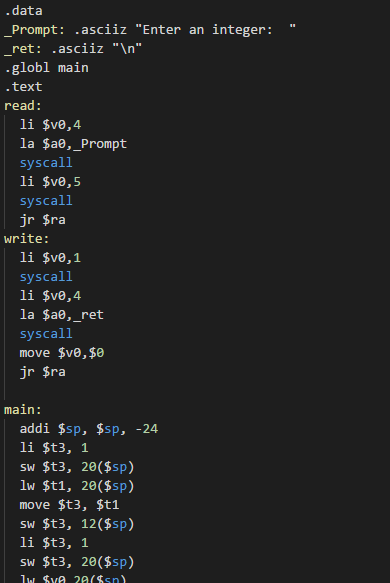
fprintf(fp, " move $v0,$0\n");

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

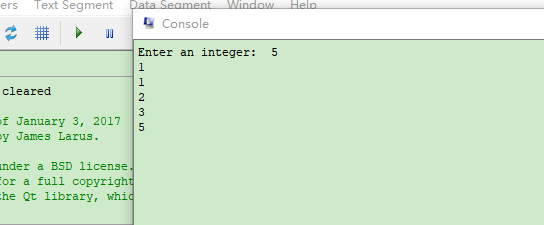
**生成程序语句的目标代码：**



**生成的汇编代码：**



**运行结果展示：**



# 6 参考文献

[1] Jobn Levine著 陆军 译. 《Flex与Bison》.东南大学出版社

[2] 许畅等编著. 《编译原理实践与指导教程》.机械工业出版社

[3] 王生原等编著. 《编译原理（第3版）》.清华大学出版社