華中科技大学

实验报告

题目：mini-c 语言编译器设计与实现

课程名称：编译原理

专业班级：校交 1601 班

学　　号：U201610504

姓　　名：刘逸帆

指导教师：刘铭

报告日期：2019 年 6 月 9 日

计算机科学与技术学院

目录

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

词法分析器的构造就是设计能准确表示各类单词的正则表达式。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，确定化、最小化，最后依据这个FA编写对应的词法分析程序。

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

INT → 整型常量

FLOAT → 浮点型常量

CHAR → 字符

ID → 标识符

ASSIGNOP → =

ASSIGNPLUS → +=

ASSIGNMINUS → -=

ASSIGNSTAR → *=

ASSIGNDIV → /=

RELOP → > | >= | < | <= | == | !=

PLUS → +

MINUS → -

STAR → *

DIV → /

AND → &&

OR → ||

NOT → !

TYPE → int | float | char | void

RETURN → return

IF → if

ELSE → else

WHILE → while

BREAK → break

SEMI → ；

COMMA → ，

LP → （

RP → ）

LC → {

RC → }

2.2 语句文法描述

按照mini-c语法定义列出所有的终结符以及非终结符如下。

定义以下终结符：

INT、ID、RELOP、TYPE、FLOAT、CHAR、LP、RP、LC、RC、SEMI、COMMA、PLUS、MINUS、STAR、DIV、ASSIGNOP、ASSIGNPLUS、ASSIGNMINUS、ASSIGNSTAR、ASSIGNDIV、AND、OR、NOT、IF、ELSE、WHILE、RETURN、BREAK、SELFADD、SELFDEC；

定义以下非终结符：

program、ExtDefList、ExtDef、Specifier、ExtDecList、FuncDec、CompSt、VarList、VarDec、ParamDec、Stmt、StmList、DefList、Def、DecList、Dec、Exp、Args。

参照C语言的运算符优先级表定义运算符的优先级与结合性如表2.1所示。

表2.1 mini-c运算符优先级结合性表

优先级	运算符	结合性
1	后缀++、后缀--	左结合
2	单目-、!、前缀++、前缀--	右结合
3	*、/	左结合
4	+、-	左结合
5	<、<=、>、>=	左结合
6	==、!=	左结合
7	&&	左结合
8	||	左结合
9	=、+=、-=、*=、/=	左结合

2.3 符号表结构描述

语义分析的一个非常重要的工作就是符号表的管理，在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括： 上述名字的种类、 具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

符号表可以采用多种数据结构实现，本次实验中采用顺序表这一数据结构管理符号表。此时的符号表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。符号表sysbolTable的表头信息分别是：变量名、别名、层号、类型、标记、偏移量。

2.4 错误类型码描述

静态语义分析时需要对语义错误进行检查，检查类型主要包括：

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

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

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

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

在mini-c的语义分析过程中，需要能够检测出以下15种错误类型：

1. 变量重复定义
2. 参数名重复定义
3. 参数类型不匹配

4. 函数调用参数太多
5. 函数调用参数太少
6. 变量未定义
7. 操作函数名，类型不匹配
8. 赋值语句需左值
9. 函数未定义
10. 调用符号不是函数
11. 返回值类型错误
12. 循环外使用continue
13. 循环外使用break
14. 自增语句需要对左值表达式进行操作
15. 自减语句需要对左值表达式进行操作

语义分析对于15种错误类型不设码进行标识，在检测出时直接通过输出函数输出错误类型提示信息。

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。

2.6 目标代码描述

目标语言选定 MIPS32 指令序列，可以在 MARS Simulator 上运行。TAC 指令和 MIPS32 指令的对应关系如表 2.3 所示。其中 reg(x)表示变量 x 所分配的寄存器。

表 2.3 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

在目标代码生成阶段，一个很重要的工作就是寄存器的分配，对于寄存器的分配已有不少算法可供参考，其中最为简单的就是朴素的寄存器分配算法，效率最低，也最容易实现；对于一个基本块采用的局部寄存器分配算法，实现起来相对不是太难，且效率上有一定提升；其它的算法基本上都要涉及到数据流的分析，效率会提升很多，但在实验中，由于学时的原因，对其相关的理论部分，数据流的分析介绍很少，这样实现起来会有较大难度。

3 系统设计与实现

3.1 词法语法分析器（实验一）

3.1.1 词法分析系统设计

实验中，词法分析器采用词法生成器自动化生成工具GNU Flex，该工具要求以正则表达式（正规式）的形式给出词法规则， Flex自动生成给定的词法规则的词法分析程序。于是，设计能准确识别各类单词的正则表达式就是关键。

Flex中的单词种类码有：INT、FLOAT、.....、WHILE，每一个对应一个整数值作为其单词的种类码，实现时不需要自己指定这个值，当词法分析程序生成工具Flex和语法分析程序生成器Bison联合使用时，将这些单词符号作为%token的形式在Bison的文件(文件扩展名为.y)中罗列出来，就可生成扩展名为.h的头文件，以枚举常量的形式给这些单词种类码进行自动编号。这些标识符在Flex文件(文件扩展名为.l)中，每个表示一个（或一类）单词的种类码，在Bison文件(文件扩展名为.y)中，每个代表一个终结符。

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

定义部分

%%

规则部分

%%

用户子程序部分

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

第二个部分为规则部分，一条规则的组成为：

正规表达式　　动作

表示词法分析器一旦识别出正规表达式所对应的单词，就执行动作所对应的操作，返回单词的种类码。词法分析器识别出一个单词后，将该单词对应的字符串保存在yytext中，其长度为yyleng。

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

依据三个部分代码功能的不同，可为mini-c语言编写对应的词法分析程序lex.l，再通过Flex进行翻译即可得到所需的词法分析c语言源程序。

3.1.2 词法分析实现

在编写mini-c语言的词法分析程序lex.l时，需要分别编写Flex程序的三个部分。

1. 定义部分

分别在定义部分首先引用通过Bison生成的包含关键字枚举值的头文件parser.tab.h，同时引用字符串处理函数库string.h和类型值的联合定义。最后对正规式宏名进行定义如下：

```
id          [_A-Za-z][_A-Za-z0-9]*
int         [+-]?[0-9]+
float       [+-]?([0-9]*\.[0-9]+)|[+-]?([0-9]+\.)
char        ['][\\]?.?[']
```

2. 规则部分

此部分中词法分析器一旦识别出正规表达式对应的单词就返回单词的种类码，同时在返回单词的种类码之前通过代码将识别结果以二元组的方式输出打印，方便调试或观察结果。

3. 用户子程序部分

Flex与Bison合用时，无需编写此部分代码。

在对三个部分代码编写完成后，对该文件使用Flex进行翻译，命令形式为：flex lex.l，即可得到词法分析器的c语言源程序文件lex.yy.c。

3.1.3 语法分析系统设计

语法分析采用生成器自动化生成工具GNU Bison（前身是YACC），该工具采用了LALR（1）的自底向上的分析技术，完成语法分析。通常语义分析是采用语法制导的语义分析，所以在语法分析的同时还可以完成部分语义分析的工作，在Bison文件中还会包含一些语义分析的工作。Bison程序的扩展名为.y，本次实验中通过Bison来编写mini-c语言的语法分析Bison程序并实现语法分析。parser.y的格式为：

%{

声明部分

%}

辅助定义部分

%%

规则部分

%%

用户函数部分

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

辅助定义部分在实验中要用到的几个主要内容有：

（1）语义值的类型定义

（2）终结符定义

（3）非终结符的属性值类型说明

（4）优先级与结合性定义

对于规则部分，由于Bison采用的是LR分析法，需要在每条规则后给出相应的语义动作。

通过上述给出的所有规则的语义动作，当一个程序使用LR分析法完成语法分析后，如果正确则可生成一棵抽象语法树。

在使用Bison的过程中，要完成报错和容错：使用Bison得到的语法分析程序，对mini-c程序进行编译时，一旦有语法错误，需要准确、及时的报错，这个由yyerror函数负责完成；编译过程中，待编译的mini-c源程序可能会有多个错

误，这时，需要有容错的功能，跳过错误的代码段，继续向后进行语法分析，一次尽可能多的报出源程序的语法错误，减少交互次数，提高编译效率。

3.1.4 语法分析实现

在编写mini-c语言的语法分析程序parser.y时，需要分别编写Bison程序的四个部分。

1. **声明部分**

需要引入用于字符串处理的string.h库、定义了节点结构等信息的用户头文件def.h。

2. **辅助定义部分**

辅助定义部分中具体实现的内容有：

（1）语义值的类型定义，mini-c的文法中，每个符号（终结符和非终结符）都会有一个属性值，这个值的类型默认为整型。实际运用中，ID的属性值类型是一个字符串，INT的属性值类型是整型，FLOAT的属性值是浮点数，CHAR的属性值是一个字符。语法分析时需要建立抽象语法树，这时ExtDefList的属性值类型会是树结点（结构类型）的指针。用联合将多种类型统一为：

```
%union {
    int       type_int;
    float     type_float;
    char      type_char;
    char      type_id[32];
    struct node *ptr;
}
```

（2）终结符定义，在Flex和Bison联合使用时，parser.y要想使用lex.l中识别出的单词的种类码，就需要在parser.y中的%token后面罗列出所有终结符(单词)的种类码标识符，mini-c中的终结符定义如下：

```
%token <type_int> INT                    //指定INT的语义值是type_int，有词法分析得到的数值
%token <type_id> ID RELOP TYPE    //指定ID,RELOP  的语义值是type_id，
```

有词法分析得到的标识符字符串

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

%token <type_char> CHAR　　//指定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 ASSIGNPLUS ASSIGNMINUS ASSIGNSTAR ASSIGNDIV AND OR NOT IF ELSE WHILE RETURN BREAK

%token SELFADD SELFDEC

（3）非终结符的属性值类型说明，对于非终结符，如果需要完成语义计算时，会涉及到非终结符的属性值类型，这个类型来源于（1）中联合的某个成员，可使用格式：%type <union的成员名>　非终结符。mini-c的非终结符语义值类型定义如下：

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

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

（4）优先级与结合性定义。对Bison文件进行翻译，得到语法分析分析程序的源程序时，通常会出现报错，大部分是移进和归约(shift/reduce)，归约和归约(reduce /reduce)的冲突类的错误。为了改正这些错误，一般通过设定单词优先级，参照2.2节中系统关键定义的mini-c运算符优先级表，定义优先级与结合性如下：

%left ASSIGNOP ASSIGNPLUS ASSIGNMINUS ASSIGNSTAR ASSIGNDIV

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT FSELFADD FSELFDEC

%left BSELFADD BSELFDEC

3. **规则部分**

由于使用Bison采用的是LR分析法，需要在每条规则后给出相应的语义动作,例如对规则：Exp →Exp =Exp，在parser.y中为：

Exp: Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno); }

规则后面{}中的是当完成归约时要执行的语义动作。规则左部的Exp的属性值用$$表示，右部有2个Exp，位置序号分别是1和3，其属性值分别用$1和$3表示。在编写代码的过程中，参照mini-c的定义编写对应规则即可。

4. **用户函数部分**

需要为联合编译产生的可执行程序提供入口函数以及分析对象。同时在语法分析出错时，通过调用编写的用户函数yyerror来实现报错并继续分析，即语法分析器在报错的同时也具有容错的功能。

为实现抽象语法树的构建，def.h中定义了mknode函数，完成建立一个树结点，这里的语义动作是将建立的结点的指针返回赋值给规则左部Exp的属性值，表示完成此次归约后，生成了一棵子树，子树的根结点指针为$$，根结点类型是ASSIGNOP，表示是一个赋值表达式。该子树有2棵子树，第一棵是$1表示的左值表达式的子树，在mini-c中简单化为只要求ID表示的变量作为左值，第二棵对应是$3的表示的右值表达式的子树，另外yylineno表示行号。

通过上述给出的所有规则的语义动作，当一个程序使用LR分析法完成语法分析后，如果正确则可生成一棵抽象语法树。

3.2 语义分析（实验二）

3.2.1 系统设计

基于实验一词法语法分析生成的AST抽象语法树，通过编程能够在对AST进行遍历时完成语义分析。

语义分析主要基于对符号表的管理实现，参照2.3中符号表结构的定义编程实现对符号表的管理，值得注意的是在保证表结构的同时需要处理好同名变量

的换名问题等。

在能够正确维护符号表后，每当需要对符号表进行操作时检查是否触发定义重名变量等语义错误，从而实现静态语义分析。在本次实验中需要实现的静态语义分析具体包括：

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

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

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

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

3.2.2 系统实现

在编写语义分析程序的过程中，程序入口函数为semantic_Analysis0，输入参数为AST的根结点。入口函数则调用递归函数semantic_Analysis对AST进行递归分析。

在semantic_Analysis函数中，对于函数定义、变量定义、参数定义三种情况，调用函数fillSymbolTable尝试更新符号表，若更新失败说明函数、变量或参数重复定义。

在semantic_Analysis函数中，对于INT、FLOAT等类型的值，函数调用的返回值，以及加减乘除的运算结果值，调用函数fill_Temp尝试更新符号表的临时变量。

在semantic_Analysis函数中，对于标识符的使用以及函数的调用，通过函数searchSymbolTable对符号表进行查找，若查找失败则说明变量未定义、函数未定义、类型不匹配等错误发生。

在每个复合语句结束前打印输出当前符号表，此时的符号表应能够正确反映出复合语句中的临时变量以及换名变量等信息。

通过在递归分析AST的过程中维护符号表，语义分析程序能够大体识别出待分析程序中存在的15种错误，系统实现过程中每种错误类型的具体内容以及产生条件如下所示：

1. **变量重复定义：**在处理局部变量与外部变量时，若尝试将新的变量插入符号表失败，说明变量重复定义；

2. **参数名重复定义：**在处理参数列表时，若尝试将参数名插入符号表失败，说明参数名重复定义；

3. **参数类型不匹配：**在处理函数调用时，依顺序比较输入参数与函数参数列表，若某一个输入参数与列表中参数类型不匹配，则抛出该错误；

4. **函数调用参数太多：**在处理函数调用时，依顺序比较输入参数与函数参数列表，若输入参数匹配完毕但函数参数列表还有剩余，则抛出该错误；

5. **函数调用参数太少：**在处理函数调用时，依顺序比较输入参数与函数参数列表，若函数参数列表匹配完毕但输入参数还有剩余，则抛出该错误；

6. **变量未定义：**在处理标识符时，首先在符号表中查找该标识符，若在符号表中查找失败，则说明变量未定义；

7. **操作函数名，类型不匹配：**在处理标识符时，首先在符号表中查找该标识符，若在符号表中查找成功但符号类型为函数，则说明操作类型不匹配；

8. **赋值语句需左值：**在处理赋值符号时，若操作数不是标识符，则说明该值非左值，提示赋值语句错误；

9. **函数未定义：**在处理函数调用时，首先在符号表中查找该函数名，若在符号表中查找失败，则说明函数未定义；

10. **调用符号不是函数：**在处理函数调用时，首先在符号表中查找该函数名，若在符号表中查找成功但符号类型不为函数，则说明调用的符号不是函数；

11. **返回值类型错误：**在处理函数返回值时，首先在符号表中查找该函数的返回值类型，随后将其与待处理的返回值类型进行比较，若类型不一致则说明返回值类型错误；

12. **循环外使用continue：**在处理while循环时，每次进入循环时将标记位加一，记录进入了一个while循环，在退出while循环时将标记位减一，标记位初始为0。若在标记位为0时处理到continue表达式，则说明在循环外不正确的使用了continue；

13. **循环外使用break：**在处理while循环时，每次进入循环时将标记位加一，记录进入了一个while循环，在退出while循环时将标记位减一，标记位初始为0。若在标记位为0时处理到break表达式，则说明在循环外不正确的使用了break；

14. **自增语句需要对左值表达式进行操作：**在处理前后缀自增语句时，若操作数不是标识符，则说明该值非左值，提示赋值语句错误；

15. **自减语句需要对左值表达式进行操作：**在处理前后缀自减语句时，若操作数不是标识符，则说明该值非左值，提示赋值语句错误。

语义分析在递归分析AST的过程中若检测到上述语义错误，对错误类型进行输出并跳过该错误继续进行分析。

3.3 中间代码生成功能（实验三）

3.3.1 系统设计

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3.1所示。

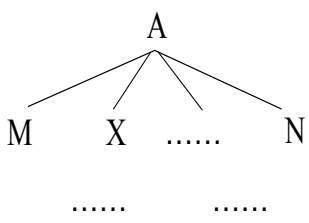

图 3.1 翻译模式规则部分的 AST

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。

3.3.2 系统实现

为了完成中间代码的生成，对于AST中的结点，需要在头文件def.h中设置以下属性，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。

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，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，具体执行操作的函数分为：

1. **genIR**：用于生成一条TAC代码的结点组成的双向循环链表，返回头指针
2. **genLabel**：用于生成一条标号语句，返回头指针
3. **genGoto**：用于生成GOTO语句，返回头指针

再按语句的语义，通过**merge()**函数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。

最后通过**prnIR()**函数将组成好的中间代码段输出。

3.4 目标代码生成功能（实验四）

3.4.1 系统设计

参照2.6的目标代码设计，目标代码采用MIPS32指令序列，中间代码TAC指令需要如表3.1所示进行到目标代码的转换。

表 3.1 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

此外为使得生成的目标代码能够正确运行，还需要对各个变量x进行寄存器的分配。

3.4.2 系统实现

为实现后续与组原实验的对接，需要根据组原实验中CPU支持的指令集修改中间代码与MIPS指令的对应关系。由于组原实验中的CPU仅支持基本的24条MIPS指令和额外的四条CCMB指令，需要对中间代码和MIPS32指令的对应关系进行如下修改或精简：

1. 由于基本指令中不包含读取立即数指令li，需要使用addi reg(x), $0, k指令进行替换；

2. 由于基本指令中不包含寄存器赋值指令move，需要使用add reg(x), $0, reg(y)指令进行替换；

3. 使用简单的斐波拉契计算程序进行测试，无需mul、div指令；

4. 对于条件判断指令，组原CPU仅支持也仅需要beq和bne两条指令，其余的bgt、bge、ble、blt均可进行精简；

5. 在调用函数前需要通过中间代码ARG进行输入参数的保存，仅考虑输入参数个数为1的情况，将输入参数保存在$a0寄存器中，对应指令为add $a0,$zero,%s；

6. 在调用函数时需要通过中间代码PARAM进行输入参数的传递，仅考虑输入参数个数为1的情况，从$a0寄存器中取出输入参数并保存在形参对应的寄存器中，对应指令为add %s,$zero,$a0；

7. 将中间代码CALL与RETURN对应MIPS32的目标代码中的move指令通过add指令进行替换。

修改并精简采用的指令序列后，中间代码TAC指令到目标代码的对应关系如表3.2所示。

表 3.2 修改后的中间代码与 MIPS32 指令对应关系

中间代码	MIPS32 指令
LABEL x	x：
x :=#k	addi reg(x), $0, k
x := y	add reg(x), $0, reg(y)
x := y + z	add reg(x), reg(y) , reg(z)
x := y - z	sub reg(x), reg(y) , reg(z)
GOTO x	j x
RETURN x	add $v0,$zero,reg(x) jr $ra
IF x==y GOTO z	beq reg(x),reg(y),z
IF x!=y GOTO z	bne reg(x),reg(y),z
X:=CALL f	jal f add reg(x),$zero,$v0
ARG x	add $a0,$zero,reg(x)
PARAM x	add reg(x),$a0,$zero

对于对应关系中形如reg(x)的指令部分，需要进行寄存器分配。仅考虑程序所涉及变量和临时变量较少且逻辑相对简单（不考虑在进入复合语句时寄存器

值的保存问题）的情形，直接为v1~v7分配寄存器s1~s7，为临时变量temp1~temp9分配寄存器t1~t9。在寄存器分配结束后，参照表3.2仿照pnIR函数编写pnTAR函数，实现中间代码到目标代码的替换即可。

3.5 组原实验对接（选做）

1. 对编译输出的目标代码稍微进行手工修改，令其能够使用组原实验中的数码管输出功能展示斐波拉契序列的计算结果。

2. 通过MARS程序将手工修改后的MIPS目标代码翻译为16进制文件。

3. 在16进制文件头部加上组原CPU所处的模拟平台logism的数据标识信息“raw 2.0”后，将十六进制文件倒入组原CPU的指令存储器中。

4. 开启组原CPU的工作时钟，观察实验对接结果，即斐波拉契序列能否正常计算并正确输出。

4 系统测试与评价

4.1 测试用例

4.1.1 词法语法分析测试用例

为实现对词法语法分析程序完整功能的测试，测试用例中需要包含外部变量声明、局部变量声明、函数声明、条件判断语句、复合语句、循环语句、返回语句等；同时为了验证分析程序对运算符优先级及结合性的判断，需要对形如x=a+b/c*z、++x+y的运算语句进行测试；另外测试程序应当包含两类注释和不合语法规范的字符串以测试词法语法分析是否能正常跳过注释并对错误语句报错。具体测试用例如下：

```
// my comment test 1
/* my comment test 2 */

char a_char;
int b_int;
float c_float;
void my_func(char a, int b, float c){
    // local var assign
    int test = 10;
    int x,y,z;
    //  算术运算
    x=y+z;
    x=a+b/c*z;
    x=b/c*z+a;


    if(x&&y){
        x++;
        ++x;

        x+y++;// equal x+(y++)
        ++x+y;// equal (++x)+y
        x+++y;
    }
```

```
        else{
            int test_assign = b+x-x/y*z;
        }
        while(x||y){
            y--;
            --y;
        }

        return 's';
        return '\n';
        return 'sss';
    }
```

4.1.2 符号表构建及语义分析测试用例

为实现对符号表构建及使用符号表进行语义分析的检测，测试代码中需要包含语义分析程序能够检测出的15种错误类型。具体测试用例如下：

```
    int i,j;
    // 1.重复定义
    int i,j;
    int fun(int a, float b)
    {
        int m;
        if (a>b) m=a;
        else m=b;

        return m;
    }

    // 2.函数名重复
    int fun(){}
    // 3.参数定义重复
    int fun2(int a, float a){}

    int fun3(int myint, float myfloat)
    {
        int i;
        int a,b,c;
        float x,y,z;
        fun(a,x);
        // 4.参数类型不符
        fun(a,b);
        // 5.参数太多
```

```
    fun(a,x,c);
    // 6.参数太少
    fun(b);
    // 7.变量未定义
    p = 10;
    // 8.函数名，类型不匹配
    fun = 10;
    x = fun;
    // 9.赋值语句需要左值
    10 = i;
    // 10.函数未定义
    fun4(x);
    // 11.x不是函数
    x(y);
    // 12.返回值类型错误
    return 2.0;
    return 0;
}

int fun4(){
    int x;
    // 13.continue error
    continue;
    while(1){
        continue;
        break;
    }
    // 14.break error
    break;
    // 15.++/--
    (x+y)++;
    ++(x+y);
    (x+y)--;
    --(x+y);

    return 0;
}

float x,y;
```

4.1.3 中间代码和目标代码生成测试用例

使用斐波拉契序列输出程序对中间代码和目标代码的生成进行测试，测试

用例如下：

```
int fibo(int a)
{
    int add_1,add_2,sum;
    if(a == 0 || a == 1)
        return 1;
    add_2 = 1;
    sum = 1;
    a = a - 1;
    while(a)
    {
        add_1 = add_2;
        add_2 = sum;
        sum = add_1 + add_2;
        a = a - 1;
    }
    return sum;
}
int main()
{
    int i = 0, n;
    while(i != 9){
        n = fibo(i);
        i = i + 1;
    }
    return i;
}
```

4.1.4 组原实验对接测试用例

为4.1.3目标代码生成测试的结果中添加数码管输出指令及停机指令，得到能够在组原实验中正确执行并输出停机的汇编指令：

```
.text
j main

fibo :
  add $s2,$zero,$a0
  addi $t1,$zero,0
  beq $s2,$t1,label3
  j label4
label4:
  addi $t2,$zero,1
```

```
    beq $s2,$t2,label3
    j label2
label3:
    addi $t3,$zero,1
    add $v0,$zero,$t3
    jr $ra
label2:
    addi $t4,$zero,1
    add $s4,$zero,$t4
    addi $t5,$zero,1
    add $s5,$zero,$t5
    addi $t6,$zero,1
    sub $t7,$s2,$t6
    add $s2,$zero,$t7
label01:
    bne $s2,0,label9
    j label8
label9:
    add $s3,$zero,$s4
    add $s4,$zero,$s5
    add $t8,$s3,$s4
    add $s5,$zero,$t8
    addi $t9,$zero,1
    sub $t0,$s2,$t9
    add $s2,$zero,$t0
    j label01
label8:
    add $v0,$zero,$s5
    jr $ra
label1:

main :
    addi $t1,$zero,0
    add $s7,$zero,$t1
label71:
    addi $t2,$zero,9
    bne $s7,$t2,label61
    j label51
label61:
    add $a0,$zero,$s7
    jal fibo
    add $t3,$zero,$v0
    add $s0,$zero,$t3
    ########################################
```

```
    add       $a0,$0,$s0
    addi      $v0,$0,34                # display hex
    syscall                            # we are out of here.
    ########################################
    addi $t4,$zero,1
    add $t5,$s7,$t4
    add $s7,$zero,$t5
    j label71
label51:
    add $v0,$zero,$s7
    ########################################
    addi      $v0,$zero,10               # pause
    syscall                            # we are out of here.
    ########################################
    jr $ra
label41:
```

将修改后的MIPS指令通过MARS转换为十六进制数据，添加logisim平台指定的数据头部后送入组原CPU，即可观察执行结果。

4.2 测试结果

4.2.1 词法语法分析正确性测试结果

观察词法语法分析程序对测试代码的分析结果，发现：

分析程序对char、int、float三种类型的外部变量定义能够进行正确解析，如图4.1所示。

```
外部变量定义:
  类型:  char
  变量名:
    ID:  a_char
外部变量定义:
  类型:  int
  变量名:
    ID:  b_int
外部变量定义:
  类型:  float
  变量名:
    ID:  c_float
```

图4.1 正确解析外部变量

分析程序对函数定义能够进行正确解析，如图4.2所示。

```
函数定义:
  类型:  void
  函数名: my_func
  函数形参:
     类型: float, 参数名: a
     类型: int, 参数名: b
     类型: float, 参数名: c
  复合语句:
     复合语句的变量定义:
        LOCAL VAR_NAME:
           类型:  int
           VAR_NAME:
                test ASSIGNOP
                     INT: 10
        LOCAL VAR_NAME:
           类型:  int
           VAR_NAME:
                x
                y
                z
     复合语句的语句部分:
        表达式语句:
           ASSIGNOP
```

图4.2 正确解析函数定义

分析程序对条件判断语句能够进行正确解析，如图4.3所示。

```
条件语句(IF_THEN_ELSE):
   条件:
      >
         ID:  a
         ID:  b
   IF子句:
      表达式语句:
         ASSIGNOP
            ID:  m
            ID:  a
   ELSE子句:
      表达式语句:
         ASSIGNOP
            ID:  m
            ID:  b
```

图4.3 正确解析条件判断语句

分析程序对循环语句能够进行正确解析，如图4.4所示。

```
循环语句:
    循环条件:
        OR
            ID:  x
            ID:  y
    循环体:
        复合语句:
            复合语句的变量定义:
            复合语句的语句部分:
                表达式语句:
                    BSELFDEC
                        ID:  y
                表达式语句:
                    FSELFDEC
                        ID:  y
```

图4.4 正确解析循环语句

分析程序能够正确依照优先级与结合性对表达式进行解释，分析程序对表达式int test_assign = b+x-x/y*z的解析结果如图4.5所示

```
复合语句:
    复合语句的变量定义:
        LOCAL VAR_NAME:
            类型:  int
            VAR_NAME:
                test_assign ASSIGNOP
                            MINUS
                              PLUS
                                 ID:  b
                                 ID:  x
                              STAR
                                 DIV
                                    ID:  x
                                    ID:  y
                                 ID:  z
```

图4.5 解析int test_assign = b+x-x/y*z

分析程序对表达式x+y++的解析结果如图4.6所示。

```
表达式语句:
    PLUS
        ID:  x
        BSELFADD
            ID:  y
```

图4.6 解析x+y++

分析程序能够通过词法分析对注释正确解析并忽略，分析到的二元组如图4.7所示。

```
<COMMENT_TYPE1, // my comment test 1>
<COMMENT_TYPE2, /* my comment test 2 */>
```

图4.7 程序解析两种类型的注释

分析程序能够对未知字符通过词法分析报错，对于语句return 'sss'，程序分析到的二元组与报错信息如图4.8所示。

```
Error type A :Mysterious character "'" at Line 54
<ID, sss>

Error type A :Mysterious character "'" at Line 54
```

图4.8 程序报错未知字符

4.2.2 符号表构建及语义报错测试结果

符号表构建及语义分析报错结果如下：

```
在3行,i 变量重复定义
在3行,j 变量重复定义
变量名   别 名   层 号    类   型   标记   偏移量
      i      v1       0      int     V       0
      j      v2       0      int     V       4
    fun      v5       0      int     F       0
      a      v6       1      int     P      12
      b      v7       1    float     P      16
      m      v8       1      int     V      24
在14行,fun 函数重复定义
变量名   别 名   层 号    类   型   标记   偏移量
      i      v1       0      int     V       0
      j      v2       0      int     V       4
    fun      v5       0      int     F      28
      a      v6       1      int     P      12
      b      v7       1    float     P      16
在16行,a 参数名重复定义
变量名   别 名   层 号    类   型   标记   偏移量
      i      v1       0      int     V      12
      j      v2       0      int     V       4
    fun      v5       0      int     F      28
      a      v6       1      int     P      12
```

```
        b       v7        1     float      P       16
     fun2      v01        0       int      F        0
        a      v11        1       int      P       12
在25行, 参数类型不匹配
在27行, 函数调用参数太多
在29行, 函数调用参数太少
在31行,p 变量未定义
在33行,fun 是函数名，类型不匹配
在34行,fun 是函数名，类型不匹配
在36行, 赋值语句需要左值
在38行,fun4 函数未定义
在40行,x 不是一个函数
在42行,返回值类型错误
变量名   别 名   层 号    类   型   标记   偏移量
        i       v1        0       int      V       12
        j       v2        0       int      V        4
      fun       v5        0       int      F       28
        a       v6        1       int      P       12
        b       v7        1     float      P       16
     fun2      v01        0       int      F       24
        a      v11        1       int      P       12
     fun3      v31        0       int      F        0
    myint      v41        1       int      P       12
  myfloat      v51        1     float      P       16
        i      v61        1       int      V       24
        a      v71        1       int      V       28
        b      v81        1       int      V       32
        c      v91        1       int      V       36
        x      v02        1     float      V       40
        y      v12        1     float      V       48
        z      v22        1     float      V       56
            temp1        1       int      T       64
            temp2        1       int      T       64
            temp3        1       int      T       64
            temp4        1       int      T       64
            temp5        1       int      T       64
            temp6        1       int      T       64
            temp7        1     float      T       64
            temp8        1       int      T       64
在49行,CONTINUE 在循环体外不应使用continue
变量名   别 名   层 号    类   型   标记   偏移量
        i       v1        0       int      V       12
        j       v2        0       int      V        4
      fun       v5        0       int      F       28
```

```
        a       v6        1       int      P       12
        b       v7        1     float      P       16
     fun2      v01        0       int      F       24
        a      v11        1       int      P       12
     fun3      v31        0       int      F       68
    myint      v41        1       int      P       12
  myfloat      v51        1     float      P       16
     fun4      v32        0       int      F        0
        x      v42        1       int      V       12
在55行,BREAK 在循环体外不应使用break
在57行, 自增语句需要对左值进行操作
在58行, 自增语句需要对左值进行操作
在59行, 自减语句需要对左值进行操作
在60行, 自减语句需要对左值进行操作
变量名  别 名  层 号   类  型  标记   偏移量
        i       v1        0       int      V       12
        j       v2        0       int      V        4
      fun       v5        0       int      F       28
        a       v6        1       int      P       12
        b       v7        1     float      P       16
     fun2      v01        0       int      F       24
        a      v11        1       int      P       12
     fun3      v31        0       int      F       68
    myint      v41        1       int      P       12
  myfloat      v51        1     float      P       16
     fun4      v32        0       int      F        0
        x      v42        1       int      V       12
            temp9        1       int      T       16
```

观察分析结果，发现在生成符号表的过程中程序能够正确识别变量名类型、正确的对重名变量进行换名、正确计算偏移量、在从复合语句返回时正确输出符号表并删除在复合语句中定义的符号。同时语义分析能够正确依据生成的符号表对15种不同的语义错误进行识别与报错，达到了语义分析的效果。

4.2.3 中间代码和目标代码生成测试结果

遍历AST语法树后将斐波拉契输出测试程序翻译为中间代码如下：

```
FUNCTION fibo :
   PARAM v2
   temp1 := #0
   IF v2 == temp1 GOTO label3
   GOTO label4
```

```
LABEL label4 :
   temp2 := #1
   IF v2 == temp2 GOTO label3
   GOTO label2
LABEL label3 :
   temp3 := #1
   RETURN temp3
LABEL label2 :
   temp4 := #1
   v4 := temp4
   temp5 := #1
   v5 := temp5
   temp6 := #1
   temp7 := v2 - temp6
   v2 := temp7
LABEL label01 :
   IF v2 != #0 GOTO label9
   GOTO label8
LABEL label9 :
   v3 := v4
   v4 := v5
   temp8 := v3 + v4
   v5 := temp8
   temp9 := #1
   temp01 := v2 - temp9
   v2 := temp01
   GOTO label01
LABEL label8 :
   RETURN v5
LABEL label1 :

FUNCTION main :
   temp11 := #0
   v7 := temp11
LABEL label71 :
   temp21 := #9
   IF v7 != temp21 GOTO label61
   GOTO label51
LABEL label61 :
   ARG v7
   temp31 := CALL fibo
   v8 := temp31
   temp41 := #1
   temp51 := v7 + temp41
```

```
    v7 := temp51
    GOTO label71
LABEL label51 :
    RETURN v7
LABEL label41 :
```

依据3.4系统实现中的中间代码到MIPS32指令的替换关系，翻译得到MIPS目标代码如下：

```
.text
j main

fibo :
    add $s2,$zero,$a0
    addi $t1,$zero,0
    beq $s2,$t1,label3
    j label4
label4:
    addi $t2,$zero,1
    beq $s2,$t2,label3
    j label2
label3:
    addi $t3,$zero,1
    add $v0,$zero,$t3
    jr $ra
label2:
    addi $t4,$zero,1
    add $s4,$zero,$t4
    addi $t5,$zero,1
    add $s5,$zero,$t5
    addi $t6,$zero,1
    sub $t7,$s2,$t6
    add $s2,$zero,$t7
label01:
    bne $s2,0,label9
    j label8
label9:
    add $s3,$zero,$s4
    add $s4,$zero,$s5
    add $t8,$s3,$s4
    add $s5,$zero,$t8
    addi $t9,$zero,1
    sub $t0,$s2,$t9
    add $s2,$zero,$t0
    j label01
```

```
label8:
    add $v0,$zero,$s5
    jr $ra
label1:

main :
    addi $t1,$zero,0
    add $s7,$zero,$t1
label71:
    addi $t2,$zero,9
    bne $s7,$t2,label61
    j label51
label61:
    add $a0,$zero,$s7
    jal fibo
    add $t3,$zero,$v0
    add $s0,$zero,$t3
    addi $t4,$zero,1
    add $t5,$s7,$t4
    add $s7,$zero,$t5
    j label71
label51:
    add $v0,$zero,$s7
    jr $ra
label41:
```

使用MARS模拟器打开目标代码并编译，在每次调用函数fibo计算斐波拉契数后设置断点，观察程序运行时斐波拉契数的计算结果，计算结果保存在s0寄存器中，如图4.9所示。

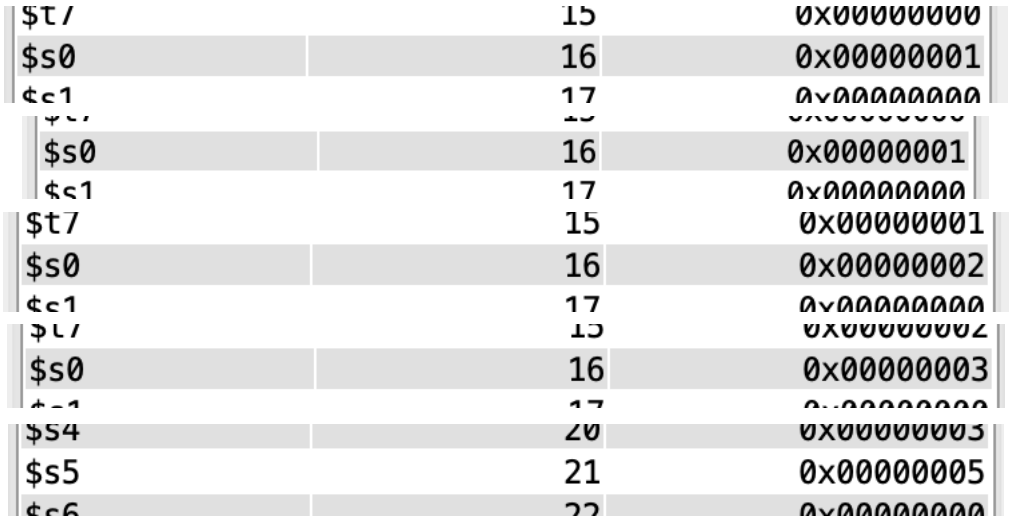

图4.9 MARS模拟器计算斐波拉契数

结果说明通过运行生成的汇编代码，斐波拉契数列能够被正确计算并输出，验证了目标代码翻译过程的正确性。

4.2.4 组原实验对接测试结果

如4.1.4中所述处理目标代码，将处理后的代码转换为十六进制数并倒入组原CPU，CPU执行结果如图4.10所示。

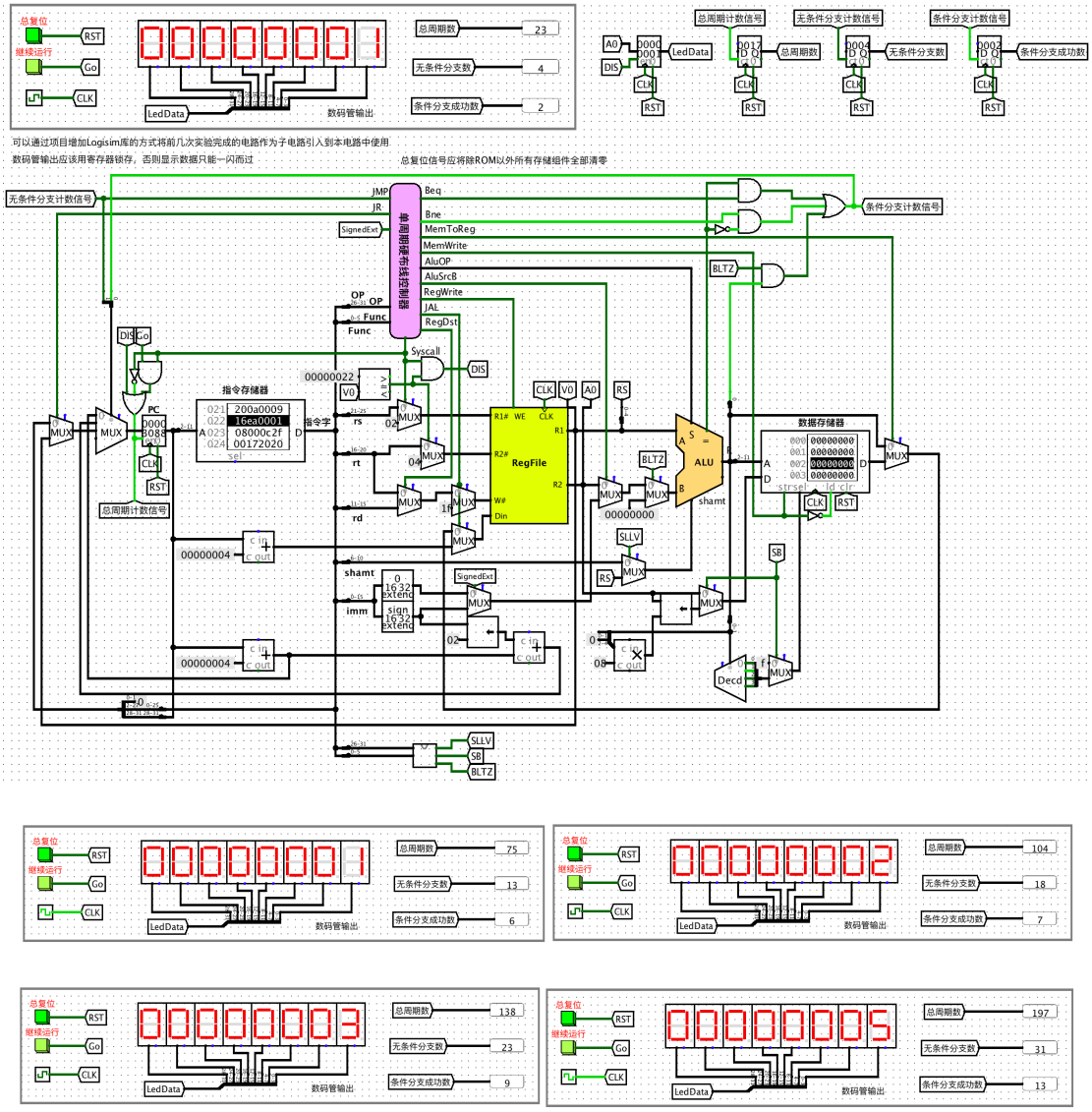

图4.10 组原CPU正确执行斐波拉契数列计算程序

结果说明通过修改后的目标代码，斐波拉契数列能够被组原CPU正确运算并输出。进一步验证了目标代码翻译的正确性，且目标代码能在支持指令集较少的CPU上正确运行。

4.3 系统的优点

1. 系统所进行的词法分析的同时能够以二元组形式输出分析结果，便于定位错误产生的位置。

2. 系统所进行的语义分析较为全面，能够分析检测出15种以上的语义错误并打印出对应的错误提示，从而避免由于源代码编写的错误而翻译出错误的目标代码。

3. 系统所生成的目标代码需要的支持指令集极其精简，目标代码能够在实现得更加简易的CPU上被运行。

4.4 系统的缺点

1. 系统所能够分析支持的变量类型比较有限，本系统的词法分析仅支持整型、单精度浮点、字符类型的解析，对于双精度浮点、数组、结构联合等数据类型不能进行相应的解析工作。

2. 系统所支持的循环语句的类型比较单一，本系统的词法分析及语法分析仅支持while循环，对于do-while循环不能进行相应的解析。

3. 在系统生成目标代码的工作中，寄存器分配所采用的原则过于简单，在一块复合语句中涉及了过多的变量或临时变量时系统为新变量分配的寄存器将会覆盖旧变量对应的寄存器，从而造成值的丢失。另外由于组原实验并未支持PUSH和PULL两条指令，在设计目标代码的过程中并未考虑在进行函数调用时的现场保护，故而当函数的调用比较复杂（如递归或多层嵌套）时，函数的返回值可能丢失，即系统仅支持将比较小巧的mini-c语言程序正确翻译为目标代码。

5 实验小结与体会

5.1 实验小结

在本次实验中，我借助Flex、Bison等工具制作了一个mini-c语言编译器。

在实验过程中主要通过Flex程序来进行词法分析，通过Bison程序来进行语法分析，通过Flex和Bison的联合编译实现对输入文件同时进行词法与语法分析。在进行语法分析的过程中，同时对抽象语法树（AST）进行构建，创建语法树节点这部分的代码被编写在ast.c中并与Flex程序和Bison程序联合编译，在AST构建完成后通过ast.c中的打印函数对AST进行输出以核对语法树的构建是否正确。

在正确构建抽象语法树后，通过遍历AST来进行语义分析及中间代码的生成。为进行语义分析，首先需要进行符号表的构建与管理，在遍历AST的过程中不断通过对符号表进行查找与更新来进行程序语义的判断；在进行中间代码生成的过程中，需要在递归遍历AST的过程中不断对当前节点生成标号语句，并在返回语法树上层时对子节点的标号语句进行合并，形成双向循环链表，最后通过打印函数输出遍历AST生成的中间代码。

对于目标代码的生成，事先选取合适的目标代码指令集并约定中间代码到目标代码的对应关系即可，同时在进行目标代码的转换时需要设计对寄存器的分配方式，以保证在目标代码执行的过程中每个变量或临时变量都能够正常使用寄存器而不互相干扰。

最后将生成的目标代码送入模拟CPU等环境，即可验证mini-c编译器对源码的编译结果是否正确。

5.2 实验体会

经过本次实验的几个阶段，我有了许多的收获与体会：

1. 通过本次实验我对于编译器工作的原理有了更深入的理解，能够准确清晰的描述出源代码通过编译器经过了哪几个阶段如何变成了目标汇编代码。

2. 回顾整个编译器的架构，我体会到了如何才能完整设计一个系统，在保

证其低耦合度的同时使其中的各个数据结构能够相互配合，并最终使得系统能够正常工作。更近一步的，系统最好还能够保证一定的可拓展性。

3. 通过自己动手制作编译器，切实体会到了课堂上所学的理论知识如何最终变成能够被实际使用的一种语言的编译器，这有助于加深我对课堂知识的理解，也更能够激发我学习的热情。

4. 本次实验依旧有很多地方存在不足并需要改进，比如词法语法分析所支持的数据类型、运算符等均还有提升空间，比如在目标代码的生成过程中对寄存器的分配还有更加优秀的方式，又比如在生成中间代码的过程中可以考虑更换一种代码格式以配合LLVM等成熟的软件。系统中的这些不足之处都将成为日后我更深入学习编译原理这门课程时的落脚点，有助于持续激发我的学习动力。

本次实验对我来说是一次十分宝贵的经验，这些经验在我未来的学习生活中一定会给我带来很多的帮助。

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

1. Flex 词法分析 lex.l

```
%{
#include "parser.tab.h"
#include "string.h"
#include "def.h"
int yycolumn=1;
int old_status;
#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;
    float    type_float;
    char     type_char;
    char     type_id[32];
    struct node *ptr;
} YYLVAL;
#define YYSTYPE YYLVAL
%}

%option yylineno
/*   Q:
    数据类型至少包括 char 类型、int 类型和 float 类型
    基本运算至少包括算术运算、比较运算、逻辑运算、自增自减运算和复合赋值运算
    控制语句至少包括 if 语句和 while 语句
    其它选项：数组、结构，for 循环等等 */
/* DEFINE */
/* keywords */
id                [_A-Za-z][_A-Za-z0-9]*
int               [+-]?[0-9]+
float             [+-]?([0-9]*\.[0-9]+)|[+-]?([0-9]+\.)
char              ['][\\]?.?[']
/* STRING          \"(\\.|[^"\n])*$ */
/* COMMENT_TYPE2 */
A [/]
B [*]
C [^*/]

%%
{int}             {printf("<INT, %s>\n",yytext); yylval.type_int=atoi(yytext); return INT;}
{float}           {printf("<FLOAT, %s>\n",yytext); yylval.type_float=atof(yytext); return FLOAT;}
```

```
{char}              {printf("<CHAR, %s>\n",yytext); yylval.type_char=yytext[1]; return CHAR;}
"int"               {printf("<TYPE, %s>\n",yytext); strcpy(yylval.type_id,    yytext);return TYPE;}
"float"             {printf("<TYPE, %s>\n",yytext); strcpy(yylval.type_id,    yytext);return TYPE;}
"char"              {printf("<TYPE, %s>\n",yytext); strcpy(yylval.type_id,    yytext);return TYPE;}
"void"              {printf("<TYPE, %s>\n",yytext); strcpy(yylval.type_id,    yytext);return TYPE;}
"return"            {printf("<RETURN, %s>\n",yytext); return RETURN;}
"if"                {printf("<IF, %s>\n",yytext); return IF;}
"else"              {printf("<ELSE, %s>\n",yytext); return ELSE;}
"while"             {printf("<WHILE, %s>\n",yytext); return WHILE;}
"break"             {return BREAK;}
"continue"          {return CONTINUE;}
{id}                {printf("<ID, %s>\n",yytext); 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*/}
";"                 {printf("<SEMI, %s>\n",yytext); return SEMI;}
","                 {printf("<COMMA, %s>\n",yytext); return COMMA;}
">"|"<"|">="|"<="|"=="|"!="        {printf("<RELOP,        %s>\n",yytext);        strcpy(yylval.type_id, yytext);;return RELOP;}
"="                 {printf("<ASSIGNOP, %s>\n",yytext); return ASSIGNOP;}
"+"                 {printf("<PLUS, %s>\n",yytext); return PLUS;}
"-"                 {printf("<MINUS, %s>\n",yytext); return MINUS;}
"*"                 {printf("<STAR, %s>\n",yytext); return STAR;}
"/"                 {printf("<DIV, %s>\n",yytext); return DIV;}
"++"        {printf("<SELFADD, %s>\n",yytext); return SELFADD;}
"--"        {printf("<SELFDEC, %s>\n",yytext); return SELFDEC;}
"&&"                {printf("<AND, %s>\n",yytext); return AND;}
"||"              {printf("<OR, %s>\n",yytext); return OR;}
"!"                 {printf("<NOT, %s>\n",yytext); return NOT;}
"("                 {printf("<LP, %s>\n",yytext); return LP;}
")"                 {printf("<RP, %s>\n",yytext); return RP;}
"{"                 {printf("<LC, %s>\n",yytext); return LC;}
"}"                 {printf("<RC, %s>\n",yytext); return RC;}
[\n]                  {}
[ \r\t]             {}

"//".*              {printf("<COMMENT_TYPE1, %s>\n",yytext); }
"/*"{A}*({C}{A}*|{B}|{C})*"*/" {printf("<COMMENT_TYPE2, %s>\n",yytext); }
.                   {printf("\nError type A :Mysterious character \"%s\" at Line %d\n",yytext,yylineno);}
%%

int yywrap()
{
return 1;
}
```

2. Bison 语法分析 parser.y

```
%error-verbose
%locations
%{
#include "stdio.h"
#include "math.h"
#include "string.h"
#include "def.h"
extern int yylineno;
extern char *yytext;
extern FILE *yyin;
void yyerror(const char* fmt, ...);
void display(struct node *,int);
%}

%union {
    int       type_int;
    float     type_float;
    char      type_char;
    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                    //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              //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 BREAK CONTINUE
%token SELFADD SELFDEC
```

```
%left ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT FSELFADD FSELFDEC
%left BSELFADD BSELFDEC

%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: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");}//$$ 结点 type_id 空置未用，正好存放运算符
| Exp AND Exp {$$=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");}

        | SELFADD Exp %prec FSELFADD {$$=mknode(FSELFADD,$2,NULL,NULL,yylineno);strcpy($$->type_id,"FSELFADD");}
        | Exp SELFADD {$$=mknode(BSELFADD,$1,NULL,NULL,yylineno);strcpy($$->type_id,"BSELFADD");}
        | SELFDEC Exp %prec FSELFDEC {$$=mknode(FSELFDEC,$2,NULL,NULL,yylineno);strcpy($$->type_id,"FSELFDEC");}
        | Exp SELFDEC {$$=mknode(BSELFDEC,$1,NULL,NULL,yylineno);strcpy($$->type_id,"BSELFDEC");}

        | 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($$->type_id,"NOT");}
        | ID LP Args RP {$$=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;}
        | FLOAT {$$=mknode(FLOAT,NULL,NULL,NULL,yylineno);$$->type_float=$1;$$->type=FLOAT;}
        | CHAR {$$=mknode(CHAR,NULL,NULL,NULL,yylineno);$$->type_char=$1;$$->type=CHAR;}
        | BREAK {$$=mknode(BREAK,NULL,NULL,NULL,yylineno);strcpy($$->type_id,"BREAK");$$->type=BREAK;}
        | CONTINUE {$$=mknode(CONTINUE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,"CONTINUE");$$->type=CONTINUE;}
        ;
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 0;
	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");
}
```

3. AST 语法树生成 ast.c

```
#include "def.h"
struct node * mknode(int kind,struct node *first,struct node *second, struct node *third,int pos ) {
   struct node *T=(struct node *)malloc(sizeof(struct node));
   T->kind=kind;
   T->ptr[0]=first;
   T->ptr[1]=second;
   T->ptr[2]=third;
   T->pos=pos;
   return T;
}

void display(struct node *T,int indent)   {//对抽象语法树的先根遍历
   int i=1;
   struct node *T0;
   if (T)
      {
      switch (T->kind) {
      case EXT_DEF_LIST:   display(T->ptr[0],indent);      //显示该外部定义列表中的第一个
                           display(T->ptr[1],indent);      //显示该外部定义列表中的其它外
部定义
                           break;
      case EXT_VAR_DEF:     printf("%*c 外部变量定义：\n",indent,' ');
                           display(T->ptr[0],indent+3);            //显示外部变量类型
                           printf("%*c 变量名：\n",indent+3,' ');
                           display(T->ptr[1],indent+6);            //显示变量列表
                           break;
      case TYPE:           printf("%*c 类型：  %s\n",indent,' ',T->type_id);
                           break;
      case EXT_DEC_LIST:   display(T->ptr[0],indent);       //依次显示外部变量名，
                           display(T->ptr[1],indent);       //后续还有相同的，仅显示语法树
此处理代码可以和类似代码合并
                           break;
      case FUNC_DEF:        printf("%*c 函数定义：\n",indent,' ');
                           display(T->ptr[0],indent+3);         //显示函数返回类型
                           display(T->ptr[1],indent+3);         //显示函数名和参数
                           display(T->ptr[2],indent+3);         //显示函数体
                           break;
      case FUNC_DEC:        printf("%*c 函数名：%s\n",indent,' ',T->type_id);
                           if (T->ptr[0]) {
                                    printf("%*c 函数形参：\n",indent,' ');
                                    display(T->ptr[0],indent+3);   //显示函数参数列表
                                    }
```

```
                    else printf("%*c 无参函数\n",indent+3,' ');
                    break;
case  PARAM_LIST:    display(T->ptr[0],indent);        //依次显示全部参数类型和名称，
                    display(T->ptr[1],indent);
                    break;
case PARAM_DEC:      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：\n",indent,' ');
                    display(T->ptr[0],indent+3);
                    break;
case RETURN:         printf("%*c 返回语句：\n",indent,' ');
                    display(T->ptr[0],indent+3);
                    break;
case COMP_STM:       printf("%*c 复合语句：\n",indent,' ');
                    printf("%*c 复合语句的变量定义：\n",indent+3,' ');
                    display(T->ptr[0],indent+6);          //显示定义部分
                    printf("%*c 复合语句的语句部分：\n",indent+3,' ');
                    display(T->ptr[1],indent+6);          //显示语句部分
                    break;
case STM_LIST:       display(T->ptr[0],indent);          //显示第一条语句
                    display(T->ptr[1],indent);            //显示剩下语句
                    break;
case WHILE:          printf("%*c 循环语句：\n",indent,' ');
                    printf("%*c 循环条件：\n",indent+3,' ');
                    display(T->ptr[0],indent+6);          //显示循环条件
                    printf("%*c 循环体：\n",indent+3,' ');
                    display(T->ptr[1],indent+6);          //显示循环体
                    break;
case IF_THEN:        printf("%*c 条件语句(IF_THEN)：\n",indent,' ');
                    printf("%*c 条件：\n",indent+3,' ');
                    display(T->ptr[0],indent+6);          //显示条件
                    printf("%*cIF 子句：\n",indent+3,' ');
                    display(T->ptr[1],indent+6);          //显示 if 子句
                    break;
case IF_THEN_ELSE:   printf("%*c 条件语句(IF_THEN_ELSE)：\n",indent,' ');
                    printf("%*c 条件：\n",indent+3,' ');
                    display(T->ptr[0],indent+6);          //显示条件
                    printf("%*cIF 子句：\n",indent+3,' ');
                    display(T->ptr[1],indent+6);          //显示 if 子句
                    printf("%*cELSE 子句：\n",indent+3,' ');
                    display(T->ptr[2],indent+6);          //显示 else 子句
                    break;
case DEF_LIST:       display(T->ptr[0],indent);      //显示该局部变量定义列表中的第一
```

```
个
                        display(T->ptr[1],indent);      //显示其它局部变量定义
                        break;
    case VAR_DEF:       printf("%*cLOCAL VAR_NAME：\n",indent,' ');
                        display(T->ptr[0],indent+3);    //显示变量类型
                        display(T->ptr[1],indent+3);    //显示该定义的全部变量名
                        break;
    case DEC_LIST:      printf("%*cVAR_NAME：\n",indent,' ');
                        T0=T;
                        while (T0) {
                            if (T0->ptr[0]->kind==ID)
                                printf("%*c %s\n",indent+3,' ',T0->ptr[0]->type_id);
                            else if (T0->ptr[0]->kind==ASSIGNOP)
                                {
                                printf("%*c     %s     ASSIGNOP\n     ",indent+3,'
',T0->ptr[0]->ptr[0]->type_id);
                            //显示初始化表达式

display(T0->ptr[0]->ptr[1],indent+strlen(T0->ptr[0]->ptr[0]->type_id)+4);
                                }
                            T0=T0->ptr[1];
                            }
                        break;
    case ID:            printf("%*cID：  %s\n",indent,' ',T->type_id);
                        break;
    case INT:           printf("%*cINT：%d\n",indent,' ',T->type_int);
                        break;
    case FLOAT:         printf("%*cFLAOT：%f\n",indent,' ',T->type_float);
                        break;
    case CHAR:          printf("%*cCHAR：%c\n",indent,' ',T->type_char);
                        break;
    case ASSIGNOP:
    case AND:
    case OR:
    case RELOP:
    case PLUS:
    case MINUS:
    case STAR:
    case DIV:
    case BSELFADD:
    case BSELFDEC:
    case FSELFADD:
    case FSELFDEC:
                        printf("%*c%s\n",indent,' ',T->type_id);
```

```
                        display(T->ptr[0],indent+3);
                        display(T->ptr[1],indent+3);
                        break;
    case NOT:
    case UMINUS:        printf("%*c%s\n",indent,' ',T->type_id);
                        display(T->ptr[0],indent+3);
                        break;
    case FUNC_CALL: printf("%*c 函数调用：\n",indent,' ');
                        printf("%*c 函数名：%s\n",indent+3,' ',T->type_id);
                        display(T->ptr[0],indent+3);
                        break;
    case ARGS:          i=1;
                        while(T) {    //ARGS 表示实际参数表达式序列结点，其第一棵子树为
其一个实际参数表达式，第二棵子树为剩下的。
                            struct node *T0=T->ptr[0];
                            printf("%*c 第%d 个实际参数表达式：\n",indent,' ',i++);
                            display(T0,indent+3);
                            T=T->ptr[1];
                            }
//                          printf("%*c 第%d 个实际参数表达式：\n",indent,' ',i);
  //                        display(T,indent+3);
                        printf("\n");
                        break;
        }
    }
}
```

4. 符号表生成及语义分析，中间代码及目标代码翻译 ana.c（部分）

```
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;
      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: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==INT?4:8;   //将一个变量的宽度向下传递
                  ext_var_list(T->ptr[1]);                  //处理外部变量说明中的标识符序列
                  T->width=(T->type==INT?4:8)* T->ptr[1]->num; //计算这个外部变量说明的宽度
                  T->code=NULL;                     //这里假定外部变量不支持初始化
                  break;
      case FUNC_DEF:          //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==INT?4:8;   //参数宽度
            result.kind=ID;     strcpy(result.id, symbolTable.symbols[rtn].alias);
            result.offset=T->offset;
            T->code=genIR(PARAM,opn1,opn2,result);        //生成：FUNCTION 函数名
            break;
    case COMP_STM:
            LEV++;
            //设置层号加 1，并且保存该层局部变量在符号表中的起始位置在
symbol_scope_TX
            symbol_scope_TX.TX[symbol_scope_TX.top++]=symbolTable.index;
            T->width=0;
            T->code=NULL;
            if (T->ptr[0]) {
                T->ptr[0]->offset=T->offset;
                semantic_Analysis(T->ptr[0]);   //处理该层的局部变量 DEF_LIST
                T->width+=T->ptr[0]->width;
                T->code=T->ptr[0]->code;
                }
            if (T->ptr[1]){
                T->ptr[1]->offset=T->offset+T->width;
                strcpy(T->ptr[1]->Snext,T->Snext);   //S.next 属性向下传递
                semantic_Analysis(T->ptr[1]);           //处理复合语句的语句序列
                T->width+=T->ptr[1]->width;
                T->code=merge(2,T->code,T->ptr[1]->code);
                }
             prn_symbol();          //c 在退出一个符合语句前显示的符号表
             LEV--;      //出复合语句，层号减 1
             symbolTable.index=symbol_scope_TX.TX[--symbol_scope_TX.top]; //删除该作
用域中的符号
             break;
    case DEF_LIST:
            T->code=NULL;
            if (T->ptr[0]){
                T->ptr[0]->offset=T->offset;
                semantic_Analysis(T->ptr[0]);    //处理一个局部变量定义
                T->code=T->ptr[0]->code;
                T->width=T->ptr[0]->width;
                }
```

```
            if (T->ptr[1]) {
                T->ptr[1]->offset=T->offset+T->ptr[0]->width;
                semantic_Analysis(T->ptr[1]);    //处理剩下的局部变量定义
                T->code=merge(2,T->code,T->ptr[1]->code);
                T->width+=T->ptr[1]->width;
                }
                break;
    case VAR_DEF://处理一个局部变量定义,将第一个孩子(TYPE 结点)中的类型送到第二个孩子的类型域
                 //类似于上面的外部变量 EXT_VAR_DEF，换了一种处理方法
                T->code=NULL;
                T->ptr[1]->type=!strcmp(T->ptr[0]->type_id,"int")?INT: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==INT?4:8;  //一个变量宽度
                while (T0) {  //处理所以 DEC_LIST 结点
                    num++;
                    T0->ptr[0]->type=T0->type;  //类型属性向下传递
                    if (T0->ptr[1]) T0->ptr[1]->type=T0->type;
                    T0->ptr[0]->offset=T0->offset;  //类型属性向下传递
                    if (T0->ptr[1]) T0->ptr[1]->offset=T0->offset+width;
                    if (T0->ptr[0]->kind==ID){

rtn=fillSymbolTable(T0->ptr[0]->type_id,newAlias(),LEV,T0->ptr[0]->type,'V',T->offset+T->width);//此处偏移量未计算，暂时为 0
                        if (rtn==-1)
                            semantic_error(T0->ptr[0]->pos,T0->ptr[0]->type_id, "变量重复定义");
                        else T0->ptr[0]->place=rtn;
                        T->width+=width;
                        }
                    else if (T0->ptr[0]->kind==ASSIGNOP){

rtn=fillSymbolTable(T0->ptr[0]->ptr[0]->type_id,newAlias(),LEV,T0->ptr[0]->type,'V',T->offset+T->width);//此处偏移量未计算，暂时为 0
                        if (rtn==-1)

semantic_error(T0->ptr[0]->ptr[0]->pos,T0->ptr[0]->ptr[0]->type_id, "变量重复定义");
                        else {
                            T0->ptr[0]->place=rtn;
```

```
                                        T0->ptr[0]->ptr[1]->offset=T->offset+T->width+width;
                                        Exp(T0->ptr[0]->ptr[1]);
                                        opn1.kind=ID;
strcpy(opn1.id,symbolTable.symbols[T0->ptr[0]->ptr[1]->place].alias);
                                        result.kind=ID;
strcpy(result.id,symbolTable.symbols[T0->ptr[0]->place].alias);

T->code=merge(3,T->code,T0->ptr[0]->ptr[1]->code,genIR(ASSIGNOP,opn1,opn2,result));
                                        }
                                    T->width+=width+T0->ptr[0]->ptr[1]->width;
                                    }
                            T0=T0->ptr[1];
                            }
                        break;
    case STM_LIST:
                        if (!T->ptr[0]) { T->code=NULL; T->width=0; break;}     //空语句序列
                        if (T->ptr[1]) //2 条以上语句连接，生成新标号作为第一条语句结束后到达
的位置
                            strcpy(T->ptr[0]->Snext,newLabel());
                        else        //语句序列仅有一条语句，S.next 属性向下传递
                            strcpy(T->ptr[0]->Snext,T->Snext);
                        T->ptr[0]->offset=T->offset;
                        semantic_Analysis(T->ptr[0]);
                        T->code=T->ptr[0]->code;
                        T->width=T->ptr[0]->width;
                        if (T->ptr[1]){         //2 条以上语句连接,S.next 属性向下传递
                            strcpy(T->ptr[1]->Snext,T->Snext);
                            T->ptr[1]->offset=T->offset;    //顺序结构共享单元方式
//                          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);
                            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->ptr[1]->offset=T->offset;
                        boolExp(T->ptr[0]);
                        T->width=T->ptr[0]->width;
                        strcpy(T->ptr[1]->Snext,T->Snext);
                        semantic_Analysis(T->ptr[1]);          //if 子句
                        if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;
                        T->code=merge(3,T->ptr[0]->code,
genLabel(T->ptr[0]->Etrue),T->ptr[1]->code);
                        break;  //控制语句都还没有处理 offset 和 width 属性
        case IF_THEN_ELSE:
                        strcpy(T->ptr[0]->Etrue,newLabel());  //设置条件语句真假转移位置
                        strcpy(T->ptr[0]->Efalse,newLabel());
                        T->ptr[0]->offset=T->ptr[1]->offset=T->ptr[2]->offset=T->offset;
                        boolExp(T->ptr[0]);          //条件，要单独按短路代码处理
                        T->width=T->ptr[0]->width;
                        strcpy(T->ptr[1]->Snext,T->Snext);
                        semantic_Analysis(T->ptr[1]);          //if 子句
                        if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;
                        strcpy(T->ptr[2]->Snext,T->Snext);
                        semantic_Analysis(T->ptr[2]);          //else 子句
                        if (T->width<T->ptr[2]->width) T->width=T->ptr[2]->width;

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

genGoto(T->Snext),genLabel(T->ptr[0]->Efalse),T->ptr[2]->code);
                        break;
        case WHILE: while_en+=1;
                        strcpy(T->ptr[0]->Etrue,newLabel());  //子结点继承属性的计算
                        strcpy(T->ptr[0]->Efalse,T->Snext);
                        T->ptr[0]->offset=T->ptr[1]->offset=T->offset;
                        boolExp(T->ptr[0]);          //循环条件，要单独按短路代码处理
                        T->width=T->ptr[0]->width;
                        strcpy(T->ptr[1]->Snext,newLabel());
                        semantic_Analysis(T->ptr[1]);          //循环体
                        if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;
                        T->code=merge(5,genLabel(T->ptr[1]->Snext),T->ptr[0]->code, \
                        genLabel(T->ptr[0]->Etrue),T->ptr[1]->code,genGoto(T->ptr[1]->Snext));
                        while_en-=1;
                        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 AND:
    case OR:
    case RELOP:
    case PLUS:
    case MINUS:
    case STAR:
    case DIV:
    case NOT:
    case UMINUS:
    case FUNC_CALL:
    case CONTINUE:
    case BREAK:
    case FSELFADD:
    case FSELFDEC:
```

```
        case BSELFADD:
        case BSELFDEC:
                        Exp(T);             //处理基本表达式
                        break;
        }
        }
}

void semantic_Analysis0(struct node *T) {
        symbolTable.index=0;
        // fillSymbolTable("read","",0,INT,'F',4);
        // symbolTable.symbols[0].paramnum=0;//read 的形参个数
        // fillSymbolTable("write","",0,INT,'F',4);
        // symbolTable.symbols[2].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);

        printf("----------------\n");


        prnTAR(T->code);
        // objectCode(T->code);
  }
//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 PLUS:
            case MINUS:
            case STAR:
            case DIV: printf("    %s := %s %c %s\n",resultstr,opnstr1, \
                        h->op==PLUS?'+':h->op==MINUS?'-':h->op==STAR?'*':'\\',opnstr2);
                        break;
            case FUNCTION: printf("\nFUNCTION %s :\n",h->result.id);
                           break;
            case PARAM:       printf("    PARAM %s\n",h->result.id);
                           break;
            case LABEL:       printf("LABEL %s :\n",h->result.id);
                           break;
            case GOTO:        printf("    GOTO %s\n",h->result.id);
                           break;
            case JLE:          printf("    IF %s <= %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                           break;
            case JLT:          printf("    IF %s < %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                           break;
            case JGE:          printf("    IF %s >= %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                           break;
            case JGT:          printf("    IF %s > %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                           break;
            case EQ:           printf("    IF %s == %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                           break;
            case NEQ:          printf("    IF %s != %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                           break;
            case ARG:          printf("    ARG %s\n",h->result.id);
                           break;
            case CALL:         printf("    %s := CALL %s\n",resultstr, opnstr1);
                           break;
            case RETURN:      if (h->result.kind)
                                    printf("    RETURN %s\n",resultstr);
                              else
                                    printf("    RETURN\n");
                              break;

        }
    h=h->next;
    } while (h!=head);
}
```

```
//输出目标代码
void prnTAR(struct codenode *head){
    printf(".text\n");
    printf("j main\n");
    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);

        //trans v0->$v0
        //trans temp0->$t0
        char temp_t1[32] = "$t";
        char temp_t2[32] = "$t";
        char temp_t3[32] = "$t";
        char temp_v1[32] = "$";
        char temp_v2[32] = "$";
        char temp_v3[32] = "$";
        if(strstr(resultstr,"temp")!=NULL){resultstr[5]='\0';strcpy(resultstr,
strcat(temp_t1,resultstr+4));}
        else                                    if(resultstr[0]=='v'){resultstr[0]='s';strcpy(resultstr,
strcat(temp_v1,resultstr));if(resultstr[2]=='8')resultstr[2]='0';}
        if(strstr(opnstr1,"temp")!=NULL){opnstr1[5]='\0';strcpy(opnstr1,
strcat(temp_t2,opnstr1+4));}
        else                                    if(opnstr1[0]=='v'){opnstr1[0]='s';strcpy(opnstr1,
strcat(temp_v2,opnstr1));if(opnstr1[2]=='8')opnstr1[2]='0';}
        if(strstr(opnstr2,"temp")!=NULL){opnstr2[5]='\0';strcpy(opnstr2,
strcat(temp_t3,opnstr2+4));}
        else                                    if(opnstr2[0]=='v'){opnstr2[0]='s';strcpy(opnstr2,
strcat(temp_v3,opnstr2));if(opnstr2[2]=='8')opnstr2[2]='0';}

        char para[32] = "";
```

```
        char temp_t4[32] = "$t";
        char temp_v4[32] = "$s";

        switch (h->op) {
            case ASSIGNOP:  //printf("    %s := %s\n",resultstr,opnstr1);
                            if(opnstr1[0] == '#')printf("
addi %s,$zero,%s\n",resultstr,opnstr1+1);
                            else printf("    add %s,$zero,%s\n",resultstr,opnstr1);
                            break;
            case PLUS:
            case MINUS:
            case STAR://printf("    %s := %s %c %s\n",resultstr,opnstr1, \
                    //h->op==PLUS?'+':h->op==MINUS?'-':'*',opnstr2);

                    printf("    %s %s,%s,%s\n", \

h->op==PLUS?"add":h->op==MINUS?"sub":"mul",resultstr,opnstr1,opnstr2);
                    break;
            case DIV: //printf("    %s := %s %c %s\n",resultstr,opnstr1, \
                    //'\\',opnstr2);

                    printf("    div %s,%s\nmflo %s\n", \
                    opnstr1,opnstr2,resultstr);
                    break;
            case FUNCTION: //printf("\nFUNCTION %s :\n",h->result.id);
                        printf("\n%s :\n",h->result.id);
                        break;
            case PARAM:     //printf("    PARAM %s\n",h->result.id);

                        if(strstr(h->result.id,"temp")!=NULL){strcpy(para,
strcat(temp_t4,h->result.id+4));para[3]='\0';}
                        else if(h->result.id[0]=='v'){strcpy(para,
strcat(temp_v4,h->result.id+1));para[3]='\0';}

                        printf("    add %s,$zero,$a0\n",para);
                        break;
            case LABEL:     //printf("LABEL %s :\n",h->result.id);
                        printf("%s:\n",h->result.id);//
                        break;
            case GOTO:      //printf("    GOTO %s\n",h->result.id);
                        printf("    j %s\n",h->result.id);
                        break;
            case JLE:       //printf("    IF %s <= %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                        printf("    blt %s,%s,%s\n",opnstr1,opnstr2,resultstr);
```

```
                    break;
            case JLT:       //printf("    IF %s < %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                    printf("    ble %s,%s,%s\n",opnstr1,opnstr2,resultstr);
                    break;
            case JGE:       //printf("    IF %s >= %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                    printf("    bge %s,%s,%s\n",opnstr1,opnstr2,resultstr);
                    break;
            case JGT:       //printf("    IF %s > %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                    printf("    bgt %s,%s,%s\n",opnstr1,opnstr2,resultstr);
                    break;
            case EQ:        //printf("    IF %s == %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                    printf("    beq %s,%s,%s\n",opnstr1,opnstr2,resultstr);
                    break;
            case NEQ:       //printf("    IF %s != %s GOTO %s\n",opnstr1,opnstr2,resultstr);
                    printf("
bne %s,%s,%s\n",opnstr1,opnstr2[0]=='#'?(opnstr2+1):opnstr2,resultstr);
                    break;
            case ARG:       //printf("    ARG %s\n",h->result.id);

                    if(strstr(h->result.id,"temp")!=NULL){strcpy(para,
strcat(temp_t4,h->result.id+4));para[3]='\0';}
                    else                    if(h->result.id[0]=='v'){strcpy(para,
strcat(temp_v4,h->result.id+1));para[3]='\0';}

                    printf("    add $a0,$zero,%s\n",para);
                    break;
            case CALL:      //printf("    %s := CALL %s\n",resultstr, opnstr1);
                    printf("    jal %s\n    add %s,$zero,$v0\n",opnstr1, resultstr);
                    break;
            case RETURN:    if (h->result.kind){
                        //printf("    RETURN %s\n",resultstr);
                        printf("    add $v0,$zero,%s\n    jr $ra\n",resultstr);
                     }
                    else{
                        // printf("    RETURN\n");
                        printf("    jr $ra\n");
                     }
                    break;

        }
    h=h->next;
    } while (h!=head);
}
```