華中科技大學

课 程 实验 报 告

题目：concise-C 语言定义及其编译器实现

课程名称：编译原理实验

专业班级：CS1705 班

学　　号：U201714707

姓　　名：练炳诚

指导教师：赵小松

报告日期：2020 年 7 月 21 日

计算机科学与技术学院

目录

1 概述.....1
2 系统描述.....2
2.1 自定义语言概述.....2
2.2 单词文法与语言文法.....3
2.3 符号表结构定义.....4
2.4 错误类型码定义.....5
2.5 中间代码结构定义.....6
2.6 目标代码指令集选择.....7
3 系统设计与实现.....8
3.1 词法分析器.....8
3.2 语法分析器.....11
3.3 符号表管理.....15
3.4 语义检查.....15
3.5 报错功能.....17
3.6 中间代码生成.....18
3.7 代码优化.....22
3.8 汇编代码生成.....22
4 系统测试与评价.....26
4.1 测试用例.....26
4.2 正确性测试.....27
4.3 报错功能测试.....29
4.4 系统的优点.....30
4.5 系统的缺点.....30
5 实验小结或体会.....31
参考文献.....32
附件：源代码.....33

1 概述

本次实验是构造一个高级语言的子集的编译器，目标代码是汇编语言。按照任务书，实现的方案可以有很多种选择。

可以根据自己对编程语言的喜好选择实现。建议大家选用 decaf 语言或 C 语言的简单集合 SC 语言。

实验的任务主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生系统软件研发技术。

2 系统描述

2.1 自定义语言概述

在本次课程实验中，选取实现的语言是标准 C 语言的子集，因此将该语言命名为 concise-C 语言（精简 C 语言），实现的功能见下表。

表 2-1 concise-C 语言功能描述

	关键字	描述
数据类型	int	整形数
	float	单精度浮点数
	char	字符
	double	双精度浮点数
	<int>/<float>//char>/<double>[]	前述 4 种类型的数组
	void	作为函数返回值和指针申明的数据类型
运算类型	+ - * /	加、减、乘、除运算，左结合
	+= -= *= /=	加、减、乘、除复合运算，右结合
	++ --	自增、自减运算，右结合
	=	赋值运算，右结合
	== != > >= < <=	比较运算，返回布尔值，左结合
	&& || !	逻辑与、或、非运算，后者右结合
	-	单目取负，右结合
控制类型	if	条件分支，条件为真跳转至目标语句，否则
	if-else	条件分支，条件为真跳转至分支成功语句，
	for	for 循环
	while	while 循环
	break	循环语句中跳出当前层循环体
	continue	循环语句中停止执行后续语句
	return	返回语句，用于函数返回
其他	//	行注释
	/* */	块注释

在后续的词法和文法设计中，均是基于此表中的功能描述进行的。

2.2 单词文法与语言文法

在单词文法设计中，除了上述表格中定义的关键字外，还有如下单词需要识别：

（1）字面常量，包括正负整数、小数，字符常量，字符常量包括可见字符和不可见字符（如回车、换号符等）；

（2）用于划分语句块的符号：{}，用于数组访问的符号：[]，用于函数参数识别、运算优先级改变的符号：()，标识一条语句结束的符号：;；

（3）由英文字母或下划线开头，非关键字的字符串组成的标识符。

语言文法设计是基于上下文无关文法设计的，文法设计如下：

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

ExtDecList→VarDec | VarDec , ExtDecList

Specifier→int | float |double | char | void

VarDec→ID | VarDec[Exp]

FucDec→ID (VarList) | ID ( )

VarList→ParamDec , VarList | ParamDec

ParamDec→Specifier VarDec

CompSt→{ DefList StmList }

StmList→Stmt StmList | ε

Stmt→Exp ; | CompSt | return Exp ; | if (Exp) Stmt | if (Exp) Stmt else Stmt | while (Exp) Stmt | for(ForDec); | break; | continue;

ForDec→Exp;Exp;Exp | ;Exp;

DefList→Def DefList | ε

Def→Specifier DecList ;

DecList→Dec | Dec , DecList

Dec→VarDec | VarDec = Exp

Exp →Exp =Exp |

Exp&& Exp | Exp || Exp | Exp < Exp | Exp <= Exp | Exp == Exp | Exp != Exp | Exp > Exp | Exp >= Exp

Exp++ | ++Exp | Exp-- | --Exp

| Exp + Exp | Exp += Exp | Exp - Exp | Exp -= Exp

| Exp * Exp | Exp *= Exp | Exp / Exp | Exp /= Exp

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

| ID | int | float | double | char

Args→Exp , Args | Exp

其中，文法开始符号为 program，一段 concise-C 程序由外部定义列表 ExtDefList 组成，外部定义列表要么为空，要么由一系列的外部定义 ExtDef 组成；外部定义要么为外部变量声明列表 ExtDecList，要么为函数定义；外部声明列表由一系列的变量类型 Specifier 和变量声明 VarDec 组成，函数定义由变量类型 Specifier，函数名 ID，参数列表 VarList，函数体 CompSt 组成；变量声明要么是标识符 ID，要么是数组定义 ID[Exp]，参数列表由一系列的参数声明 ParamDec 组成；参数声明由 Specifier 和 VarDec 组成；函数体是由左右花括号及中间的定义列表 DefList 和语句列表 StmList 组成；定义列表和外部声明列表 ExtDecList 类似，由与一系列的 Specifier 和 DecList 组成，语句列表 StmList 由一系列的语句 Stm 组成；语句可以是表达式语句 Exp，也可以是语句块 CompSt，返回语句 return Exp，条件分支语句,while 循环语句或 for 循环语句，break 语句，continue 语句。

对表达式语句，除了表 2-1 中定义的各类运算符相对应的表达式及字面常量作为表达式外，还有函数调用语句 ID(Exp)和数组访问表达式 Exp[Exp]。

2.3 符号表结构定义

为了实现编译过程中的语法分析和目标代码生成时正确访问存储空间，需要设计符号表结构保存各类变量及临时变量的信息。

为支持语义分析，符号表中记录了当前分析位置的可见标识符，即程序运行到某一位置时，可以使用的所有标识符，即符号表中需要记录标识符的名称，由

于 concise-C 语言支持嵌套语句块，不同层次的语句块中可声明同名的变量，且中间代码生成时可能会产生临时变量，故需要为每个变量分配一个独一无二的别名，故符号表中还需记录标识符的别名和层号。在表达式语句中有大量运算，为判断运算操作数的类型是否匹配，还需在符号表中记录该标识符的类型，concise-C 语言不支持强制类型转换和隐式类型提升，故操作数的类型应该一致。由于函数名可能与变量名相等，为区分一个标识符是函数名还是变量，还需要一个标识来区分，0 表示变量，1 表示函数名。最后，在目标代码生成时，需要为每个变量划分存储空间，需要记录每个变量相对数据段的偏移地址 offset。

经过以上分析，设计符号表结构如图 2.1 所示。

变量名	别名	层号	类型	标记	偏移量
a	t1	0	int	0	0
b	t2	0	char	0	4
a	t3	0	int	1	5

图 2.1 符号表结构设计

需要说明的是，对于数组类型的变量，需要记录其内情向量，为节省符号表的空间，没有记录到符号表中，而是记录到语法树的数组 ID 节点内。

2.4 错误类型码定义

在语言分析过程中，需要检查若干类语义错误，只有未发生语义错误时才生成中间代码。对于 concise-C 语言，语义分析的主要工作是进行类型检查，对于循环语句，还要有控制流检查。定义的错误类型见下表。

表 2-2 静态语义错误类型定义

错误类型	描述
变量 x 未定义	语句中引用的变量 x 未定义
变量 x 重复声明	在 x 的声明语句前已经声明过 x（此处是可见声明）
函数 f 未定义	调用函数 f 前未定义该函数

函数 f 重复定义	函数 f 在此前已经定义，concise-C 不支持函数的重载，故每个函数名均不同
参数 P 类型不匹配	函数调用语句中，传入的参数 P 类型与函数定义的参数类型不匹配
参数数量不匹配	函数调用语句中，传入的参数数量和函数定义的参数数量不一致
函数 f 缺少返回值	函数 f 定义中定义了非 void 类型的返回值，但没有 return 语句
函数 f 返回值类型不匹配	函数 f 定义的返回值类型和 return 语句返回值的类型不匹配
赋值语句需要左值	赋值语句或复合赋值语句中缺少左部
自增/自减对象只能是变量	自增/自减表达式中的操作对象不是变量
break 语句不在循环体内	break 控制语句只能在循环体内
continue 语句不再函数体内	continue 控制语句只能在循环体内
数组访问下标类型错误	数组访问的下标只能是整数，即 int 型字面常量或变量
数组访问维数不匹配	数组访问时下标的个数与数组定义维数不匹配（concise-C 语义不支持指针类型，故通过小于维数的下标访问数组获取低维指针是不允许的）
初始化类型不匹配	变量初始化时字面常量的类型与声明类型不匹配
逻辑运算类型错误	逻辑运算（&&、‖、!）表达式中操作数的类型只能是 int 型
比较运算类型不匹配	比较运算（==、！=、>、<、>=、<=）两操作数需要相同的类型
赋值运算类型不匹配	赋值运算（=、+=、-=、*=、/=）中左部和右部需要相同的类型
算术运算类型不匹配	（+、-、*、/）算术运算中两操作数需要相同的类型（concise-C 语义不支持强制类型转换和隐式类型提升）

需要说明的是，在后续的程序实现中，没有对每种类型的错误进行编码，而是直接打印出具体的出错位置和出错信息。

2.5 中间代码结构定义

实验中选取的中间代码结构为 TAC（Three-Address Code）码，也称四元式，其表示为：

(OP,A,B,C)

(OP,A,-,C)

(OP,-,-,C)

其中，第一种形式对应于双操作数的赋值运算，将 A op B 的结果赋给 C，第二种形式对应于单操作数的赋值运算，如 C = op A 或 C=A，第三种对应于分支语句，分支地址在 C 中。

采用这种中间代码格式可以很方便地在生成 AST 后通过后序遍历语法树产生中间代码序列，对后续基于 DAG 的代码优化可直接复用 TAC 作为 DAG 节点。

2.6 目标代码指令集选择

为了在组原课程设计中搭建的采用 MIPS 指令集的 CPU 上运行目标代码，目标代码指令集采用了 MIPS32 位指令集的子集（组原课设中只完成了 24 条常用 MIPS 指令，且不支持浮点运算）。目标代码中可以产生的 MIPS 指令见下表。

表 2- 生成目标代码可选择的指令

指令	描述
add rd, rs, rt	$rs 和$rt 的内容相加送$rd 中
sub rd, rs, rt	$rs 和$rt 的内容相减送$rd 中
addi rt, rs, immediate	$rs 加上 immediate 符号拓展送$rt 中
lw rt, offset(base)	将内存地址 base+offser 的内容送$rt 中
sw rt, offset(base)	将$rt 的内容送内存 base+offset 中
beq rs, rt, offset	若$rs 和$rt 的内容相等，则跳转至 offset
bne rs, rt, offset	若$rs 和$rt 的内容不相等，则跳转至 offset
blez rs, offset	若$rs 的内容≤0，则跳转至 offset
bltz rs, offset	若$rs 的内容<0，则跳转至 offset
bgez rs, offset	若$rs 的内容≥0，则跳转至 offset
bgtz rs, offset	若$rs 的内容>0，则跳转至 offset

3 系统设计与实现

3.1 词法分析器

使用 flex 工具自动生成词法分析器，flex 程序的代码结构如下：

%{

说明部分

%}

%%

转换规则

%%

辅助过程

其中，说明部分语法与 C 语义类似，主要是进行一些头文件的引入和变量的定义，其中需要定义 YYLVAL 的联合体保存读取到的单词或字面常量。如图 3.1 所示。

```
typedef union {
    int type_int;
    float type_float;
    double type_double;
    char type_char;
    char type_string[32];
    char type_void [5];
    char type_id[32];
} YYLVAL;
```

图 3.1 YYLVAL 联合体定义

其中，type_xx 保存字面常量，type_id 保存变量名称。在规则部分，除了对表 2-1 定义的关键字和运算符进行识别外，还需要识别字面常量和标识符，以及行注释与块注释。

对于字面常量，设计如下正则表达式进行匹配：

int　　([0-9]+)|(0[xX][a-fA-f0-9]+)|([oO][0-7]+)

float　([0-9]+\.[0-9]+)|([0-9]+)

double ([0-9]+\.[0-9]+)|([0-9]+)

char1　　(\'[\x20-\x7F]\')

char2　　(\'\\[ntfrb\\\'\"]\')

对于 int 型字面常量，支持 8 进制、10 进制及 16 进制表示，对于 float 和 double 型字面常量，仅支持 10 进制表示，且不能以小数点开头和结尾，这点与标准 C 语言有所不同。

另外需要说明的是，将字符型常量分为两种类型进行识别，char1 识别可见字符，char2 识别不可见字符：回车、换行、制表等。

对于标识符，设计如下正则表达式进行匹配：

id　　[A-Za-z_][A-Za-z0-9_]*

即以英文字母或下划线开头，外加数字组成的字符串，需要说明的是此处没有通过正则表达式过滤掉表 2-1 中定义的关键字，这一工作放在了静态语义分析过程中进行。

对注释信息，采用如下正则表达式进行匹配：

CommentLine　　("//"[^\n]*)

CommentBlock　("/*"([^*]|(*)*[^*/])*(*)*"*/")

CommentLine 匹配以“//”开头，换行符前一个字符结尾的字符串，表示一个行注释；CommentBlock 匹配以“/*”开头，以“*/”结尾的字符串，表示一个块注释。

在 flex 程序的规则部分，需要将读取到的 id 或字面常量存入上述定义的联合体中相应位置，并返回该符号的类型码，所有识别的符号及其类型码见表 3-1。

表 3-1 识别符号及其类型码

字符	类型码
int 字面常量	INT
float 字面常量	FLOAT
double 字面常量	DOUBLE
char 字面常量	CHAR
int	TYPE
float	TYPE
double	TYPE
char	TYPE

void	TYPE
return	RETURN
if	IF
else	ELSE
while	WHILE
for	FOR
break	BREAK
continue	CONCINUE
标识符 id	ID
;	SEMI
,	COMMA
== != > >= < <=	RELOP
++	PLUS_SELF
--	UMINS_SLEF
+	PULS
-	UMINS
*	STAR
/	DIV
&&	AND
||	OR
!	NOT
(	LP
)	RP
[	LB
]	RB
{	LC
}	RC
.	DOT

在转换规则部分，按此表顺序，对每个单词打印其类型和类型码，并返回其类型码（文法分析程序中用到）。需要说明的是，flex 程序中对单词的识别是有优先级区分的，此表中列举的符号顺序已经规定了符号的优先级，即在将某一个个单词识别为 id 后，就不会再将单词识别为关键字（这类错误将再语义分析中检查）。

3.2 语法分析器

使用 bison 工具自动生成语法分析器，bison 可以根据用户定义的上下文无关文法规则扫描源代码，生成 LALR(1)分析表，bison 程序的格式与 flex 程序类似，说明部分如图 3.2 所示。

```
%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);
%}
```

图 3.2 bison 程序说明部分

图 3.2 中引用了 flex 中的指示扫描位置的指针变量 yytext 和 yyin，以及 flex 中记录当前扫描行号的变量 yyline，即 bison 与 flex 协同产生语法分析器，并可通过规则部分产生 AST。

在自底向上分析文法时，总是从含非终结符语句开始规约的，故需定义终结符，即 3.1 中定义的各类关键字、字面常量及标识符。bison 中的实现如图 3.3 所示。

```
//% token 定义终结符的语义值类型
%token <type_id> ID RELOP TYPE      //指定ID RELOP TYPE的语义值是type_id，有词法分析得到的标识符字符串
%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_float> FLOAT           //指定FLOAT的语义值是type_float，有词法分析得到的数值
%token <type_double> DOUBLE         //指定DOUBLE的语义值是type_double，有词法分析得到的数值
%token <type_char> CHAR             //指定CHAR的语义值是type_char，有词法分析得到的字符
%token <type_string> STRING         //指定STRING的语义值是type_string，有词法分析得到的字符串
%token <type_void> VOID             //指定VOID的语义值是type_void，有词法分析得到的字符串


//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
//左右括号、左右花括号、左右方括号、小数点、分号、逗号
%token LP RP LC RC LB RB DOT SEMI COMMA
//自增、自减、加、减、乘、除、赋值、加等、减等、乘等、除等、与、或、非
%token PRE_PLUS_SELF NEXT_PLUS_SELF PRE_MINUS_SELF NEXT_MINUS_SELF PLUS MINUS STAR DIV ASSIGNOP ASSIGNOP_PLUS ASSIGNOP_MINUS ASSIGNOP_STAR ASSIGNOP_DIV AND
//、if、else、while、return、for
%token IF ELSE WHILE RETURN FOR
//结构体类型
%token <type_struct> STRUCT
//控制语句
%token BREAK CONTINUE
```

图 3.3 终结符定义部分

同时，为消除移进-规约冲突，需要对部分符号定义优先级和结合性，如图 3.4 所示。

```
//赋值类语句具有右结合性，解决shift-reduce冲突
%right ASSIGNOP ASSIGNOP_MINUS ASSIGNOP_PLUS ASSIGNOP_DIV ASSIGNOP_STAR ASSIGNOP_ARRAY ARRAY_ASSIGNOP
//运算类语句具有左结合性，解决shift-reduce冲突
%left PLUS_SELF MINUS_SELF
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT
//定义方括号结合性，解决多维数组的shift-reduce冲突
%right LB
%left RB
//定义小数点的结合性，解决结构体访问的shift-reduce冲突
%left DOT

%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE
```

图 3.4 部分符号的优先级及结合性定义

在规则部分，实现较为简单，及将 2.2 中定义的文法按 yacc 语法格式表示，对每个产生式，可产生一个结点，并定义结点类型。如下所示：

```
//程序入口
program: ExtDefList                { semantic_Analysis0($1);/*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       {$$=mknode(EXT_VAR_DEF,$1,$2,NULL,yylineno);}      //该结点对应一个外部变量声明
          | Specifier VarDec ASSIGNOP Exp SEMI {$$=mknode(EXT_VAR_DEF_ASSIGN,$1,$2,$4,yylineno);}//对应一个外部变量的声明赋值
        | Specifier FuncDec CompSt         {$$=mknode(FUNC_DEF,$1,$2,$3,yylineno);}            //该结点对应一个函数定义
        | error SEMI                       {$$=NULL; }
        ;
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);}
            ;
Specifier:                                                                                                  TYPE
{$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);if(!strcmp($1,"void"))$$->type=VOID;if(!strcmp($1,"int"))$$->type=INT;if(!strcmp($1,"float"))$$->type=FLOAT;if(!strcmp($1,"double"))$$->type=DOUBLE;if(!strcmp($1,"char"))$$->type=CHAR;if(!strcmp($1,"string"))$$->type=STRING;}
                ;
VarDec: ID              {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}     //ID 结点，标识符符号串存放结点的 type_id
        | VarDec LB Exp RB     {$$=mknode(ARRAY_DEC,$1,$3,NULL,yylineno);}
        ;
FuncDec: ID LP VarList RP      {$$=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;}          //复合语句结点直接作为语句结点，不再生成新的结点
    | RETURN Exp SEMI     {$$=mknode(RETURN,$2,NULL,NULL,yylineno);}
      | RETURN SEMI           {$$=mknode(RETURN,NULL,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 {$$=mknode(WHILE,$3,$5,NULL,yylineno);}
    | FOR LP ForDec RP Stmt {$$=mknode(FOR,$3,$5,NULL,yylineno);}
      | BREAK SEMI {$$=mknode(BREAK,NULL,NULL,NULL,yylineno);}
      | CONTINUE SEMI {$$=mknode(CONTINUE,NULL,NULL,NULL,yylineno);}
    ;
ForDec: Exp SEMI Exp SEMI Exp {$$=mknode(FOR_DEC,$1,$3,$5,yylineno);}
      | SEMI Exp SEMI {$$=mknode(FOR_DEC,$2,NULL,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,"=");}
   ;
Exp:      Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"=");}//$$结点 type_id 空置
未用，正好存放运算符
      | Exp AND Exp    {$$=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($$->type_id,"PLUS");}
      | Exp PLUS ASSIGNOP Exp {$$=mknode(ASSIGNOP_PLUS,$1,$4,NULL,yylineno);strcpy($$->type_id,"+=");}
      | Exp PLUS_SELF    {$$=mknode(NEXT_PLUS_SELF,$1,NULL,NULL,yylineno);strcpy($$->type_id,"++");}
      | PLUS_SELF Exp    {$$=mknode(PRE_PLUS_SELF,$2,NULL,NULL,yylineno);strcpy($$->type_id,"++");}
      | Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type_id,"MINUS");}
      | Exp MINUS ASSIGNOP Exp {$$=mknode(ASSIGNOP_MINUS,$1,$4,NULL,yylineno);strcpy($$->type_id,"-=");}
      | Exp MINUS_SELF    {$$=mknode(NEXT_MINUS_SELF,$1,NULL,NULL,yylineno);strcpy($$->type_id,"--");}
      | MINUS_SELF Exp    {$$=mknode(PRE_MINUS_SELF,$2,NULL,NULL,yylineno);strcpy($$->type_id,"--");}
      | Exp STAR Exp    {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type_id,"STAR");}
      | Exp STAR ASSIGNOP Exp {$$=mknode(ASSIGNOP_STAR,$1,$4,NULL,yylineno);strcpy($$->type_id,"*=");}
      | Exp DIV Exp     {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type_id,"DIV");}
      | Exp DIV ASSIGNOP Exp {$$=mknode(ASSIGNOP_DIV,$1,$4,NULL,yylineno);strcpy($$->type_id,"\\=");}
      | LP Exp RP         {$$=$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,"!");}
      | 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);}
      | Exp LB Exp RB {$$=mknode(EXP_ARRAY,$1,$3,NULL,yylineno)}
      | Exp DOT ID {$$=mknode(EXP_ELE,NULL,NULL,NULL,yylineno);}
      | 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;}
      | DOUBLE               {$$=mknode(DOUBLE,NULL,NULL,NULL,yylineno);$$->type_double=$1;$$->type=DOUBLE;}
      | CHAR               {$$=mknode(CHAR,NULL,NULL,NULL,yylineno);$$->type_char=$1;$$->type=CHAR;}
      |                                                                                          STRING
{$$=mknode(STRING,NULL,NULL,NULL,yylineno);strcpy($$->type_string,$1);$$->type=STRING;}
       ;
Args:    Exp COMMA Args       {$$=mknode(ARGS,$1,$3,NULL,yylineno);}
   |     Exp
```

每次选择一条产生式进行规约时，就根据这条语句的类型调用 makenode 函

数创建一个结点（生成 AST），根据设计的文法，每个结点最多有 3 个子结点，即每条产生式右部最多 3 个非终结符，产生结点的类型由语句具体类型决定。

在生成 AST 时，定义的语法树结点如图 3.5 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
    char struct_name[32];
    union {
        char type_id[32]; //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; //由浮点常数生成的叶结点
        char type_char;
        char type_string[32];
    };
    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];              //该结点对应语句执行后的下一条语句位置标号
    struct codenode *code;       //该结点中间代码链表头指针
    char op[10];
    int type;   //结点对应值的类型
    int pos;    //语法单位所在位置行号
    int offset; //偏移量
    int width;  //占数据字节数
    int array_par[30];//数组每一维大小
    int array_size;//数组维数
    int num;//参数数量
    char isWhile;//是否为循环体节点
    char Ebreak[15], Econtinue[15];//对循环语句翻译时，break语句、continue语句转义目标的标号
};
```

图 3.5 AST 结点定义

其中各字段的含义见注释。

使用 flex 指令执行 flex 程序即可生成词法分析器，使用 bison -d 指令即可生成语法的 LALR(1)分析器，bison 和 flex 协作关系如图 3.6 所示。

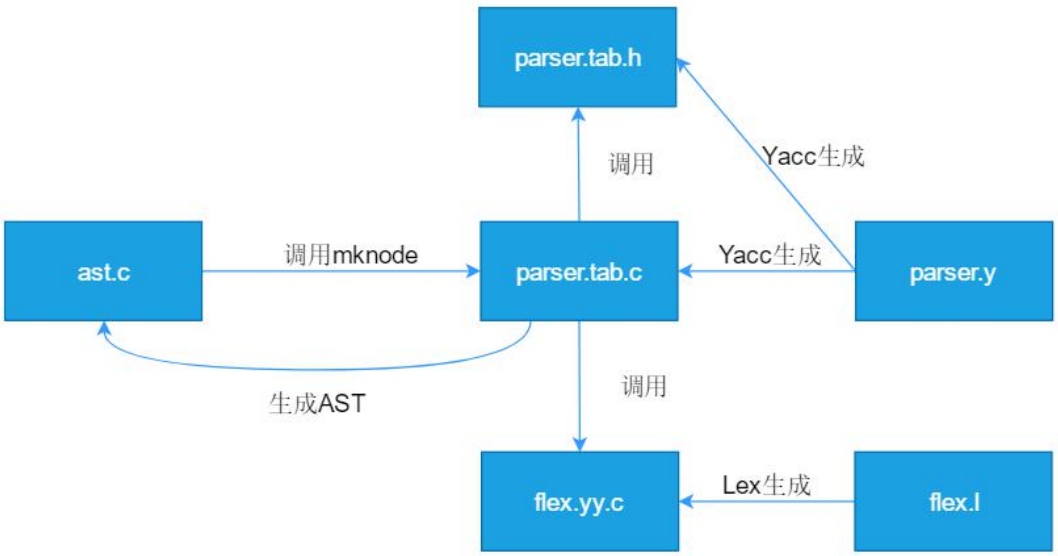

图 3.6 bison 和 flex 协作关系图

其中，ast.c 是遍历 AST 的 C 程序，bison 中调用 mknode 结点产生语法树节点构建语法树，最终返回根节点给 ast.c，ast.c 从根节点开始先序遍历语法树进行打印。

3.3 符号表管理

由于 concise-C 语义支持嵌套语句块，故若采用全局符号表的管理方式较为复杂，此处采用局部符号表的形式，每个语句块各自维护一个符号表，当离开该语句块时，删除该语句块的符号表。

为实现这一功能，需要定义局部符号表的存储结构，在 2.3 中符号表定义为每行定长的表格，故可采用结构数组的形式表示符号表，每个结构题内保存一行的信息，维护一个索引变量，每添加一条记录，索引变量加 1。进入一个语句块前，将当前符号表索引变量压入栈区，退出符号表时，栈顶元素出栈，可恢复符号表，其代码实现如图 3.7 所示。

```
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[32];  //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'
    int offset;    //外部变量和局部变量在其静态数据区或活动记录中的偏移量
                    //或函数活动记录大小，目标代码生成时使用
};
//符号表，是一个顺序栈，index初值为0
struct symboltable
{
    struct symbol symbols[MAXLENGTH];
    int index;
} symbolTable;

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
    int TX[30];
    int top;
} symbol_scope_TX;
```

图 3.7 符号表代码实现

其中，符号表的最大记录数 MAXLENGTH 是一个宏定义常量，指定为 1000，即符号表最多记录 1000 条记录。

3.4 语义检查

在生成 AST 后，对 AST 进行遍历即可进行语义检查，此处主要是进行静态语义错误的检查，相关的语义错误在 2.4 中已经定义。

检查的内容主要为类型检查和控制流检查，流程是：

（1）首先进行基于 L-属性文法的语义计算，自上而下，深度优先从左至右遍历语法树，计算各结点的综合属性和继承属性；

（2）每个结点完成属性计算时，即可根据计算得到的属性进行语义检查，此时需用到符号表中记录的标识符及其属性。

（3）若检测出语义错误，记录错误信息，继续检查；

（4）若无语义错误，则产生中间代码，否则打印错误信息。

在进行语义计算时，需要计算的属性主要是图 3.5 中的 offset、width、type，前两者中，offset 是继承属性，用于目标代码生成时存储地址的分配，width 是综合属性，表示这个结点所占据的字节大小，type 是继承属性，表示该结点的数据类型或操作类型。其他需要计算的属性有：isWhile，继承属性，为 1 表示该结点是循环体内的结点，为 0 则不是；EBreak 和 EContinue 分别表示 break 语句和 continue 语句的分支标号；Etrue 和 Eflase 表示条件分支成功和分支失败的转移标号；Snext 表示下一条语句标号；num 记录函数体结点的参数数量，用于在符号表中获取函数参数；array_par 和 array_size 记录数组结点每一维的大小和维数，相当于符号表中的内情向量。

对 width 属性的计算方式是：对于变量结点 char、int/float、double，其值分别为 1、4、8，对数组型变量，其值为该值乘上数组大小。对于非变量结点，其值是其所有子结点的 width 值的和。

对 type 属性的计算方式是：继承其父节点或兄弟结点的 type 属性；对 2.2 中定义的 Specifier 结点，其值由变量类型决定。

对 offset 属性的计算方式是：对于子结点，其值等于父结点的 offset；对于父结点，其值是子结点的 width。

对于 isWhile 属性的计算方式是：在循环体文法的产生式右部的所有结点及其子结点均为 1，其余情况是 0.

对 array_par 和 array_size 属性的计算方式是：当遍历到数组声明结点时，递归地遍历其语法树，例如数组 a[7][8][9]的语法树如图 3.8 所示。

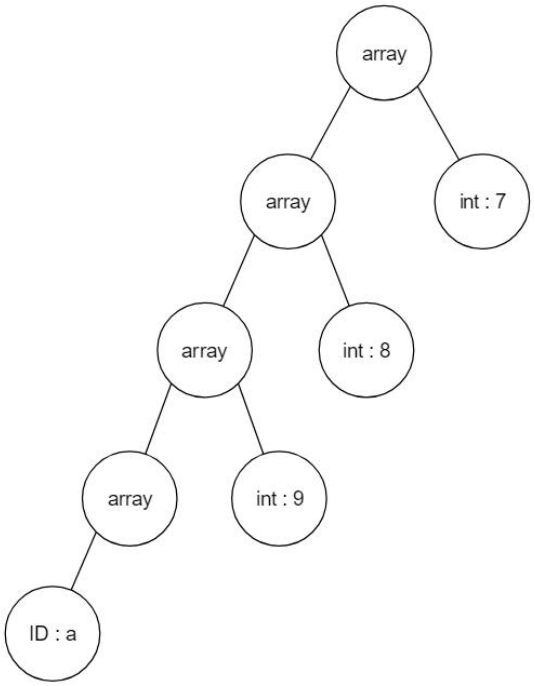

图 3.8 数组 a[7][8][9]的语法树

根据递归深度 depth 计算得到 array_size = depth - 1，arrary_par 数组记录每一维的大小。

此外，在生成中间代码时需要查找符号表中结点对应的变量的属性，故需在语法树结点中添加 place 属性，记录该变量在符号表中的下标，以便访问。

对于 EBreak、EContinue、Etrue、Efalse、Snext 这些属性，将在中间代码生成时进行计算。

基于以上计算规则，从语法树根节点进行遍历完成属性值的计算。

3.5 报错功能

在静态语义分析过程中，遇到语法错误需要进行报错，且分析程序继续执行，这意味这需要保存报错信息，在扫描完源程序后再一并显示出来。

采用一个简单的方式实现这一功能：将报错信息用输出到一个文本缓冲区内，并维护一个错误计数器，当扫描完毕后，若计数器值为 0，则可以进行中间代码生成等后序工作；若计数值不为 0，则根据计数值打印出文本缓冲区的内容。

文本缓冲区可以简单地用二维字符数组实现，第一维的大小为 256，表示一条错误信息最多占用 256 字节，第二维大小为 1000，表示最多存储 1000 条错误

信息。

3.6 中间代码生成

中间代码生成时，需要产生 TAC 结点，并进行合适的组织，这时需要对 3.4 中未完成的分支相关的属性进行计算。

在组织 TAC 结点序列时，考虑到生成目标代码需要对 TAC 序列进行顺序访问，以及代码优化时可能会删除部分 TAC 结点，故采用双向循环链表的数据结构保存 TAC 序列。在遍历 AST 产生 TAC 序列时，定义一个 genIR(op,opn1,opn2,result)函数产生 TAC 结点，其中 op 是操作类型的类型码，在 2.2 中定义，opn1、opn2、result 均为 opn 类型的结点，定义如下图所示。

```
struct opn
{
    int kind; //标识操作的类型
    int type; //标识操作数的类型
    union {
        int const_int;      //整常数值，立即数
        float const_float; //浮点常数值，立即数
        double const_double; //浮点常数值，立即数
        char const_char;    //字符常数值，立即数
        char const_string[32];    //字符串常数值，立即数
        char id[32];        //变量或临时变量的别名或标号字符串
    };
    int level;  //变量的层号，0表示是全局变量，数据保存在静态数据区
    int offset; //变量单元偏移量，或函数在符号表的定义位置序号，目标代码生成时用
};
```

图 3.9 opn 结点定义

操作数结点中保存了操作数的类型和操作类型，以及操作数对应的字面常量或标识符，以及其层号和偏移量，这将在目标代码生成中用到。生成的 TAC 序列的指针将保存在语法树结点的 code 成员中。

在 3.4 中提到，Snext 属性将在中间代码生成时进行计算，计算规则是：若为复合语句类型的结点（即由一对花括号括起来的语句序列，如函数体、if-else 语句块、while 循环体等），则 Snext 为该语句块后紧邻的语句块的起始标号，这时需要定义 genLabe()函数为每个语句块的起始位置或分支目标地址产生一个 Label。Snext 作为继承属性向其子结点进行传递，其他情况下该值为空串。

产生 TAC 结点后，涉及多个 TAC 结点的连接操作，故需要定义 merge(TAC1, TAC2, ... TACn)函数，接受边长的 TAC 结点指针作为参数，将这些结点按序连接为双向链表。产生中间代码的规则见下表。

表3-2 语句类结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作
COMP_STM T1　说明子树（可空） T2　语句子树（可空）	访问到 T：T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code || T2.code
IF_THEN T1　条件子树 T2　if 子句子树	访问到 T：T1.Etrue=newLabel, T1.Efalse= T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code || T1.Etrue ||T2.code
IF_THEN_ELSE T1　条件子树 T2　if 子句子树 T3　else 子句子树	访问到 T：　T1.Etrue=newLabel,T1.Efalse=T.Snext T1.Snext= T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code || T1.Etrue || T2.code || goto T.Enext || T1.Efalse || T3.code
WHILE T1　条件子树 T2　while 子句子树	访问到 T：T1.Etrue=newLabel, T1.Efalse=T.Snext, T2.Snext= newLabel, T2.EBreak=T.Snext,T2.EContinue=T2.Snext; 访问 T 的所有子树后： T.code=T2.Snext || T1.code || T1.Etrue || T2.code || goto T2.Snext

STM_LIST
T1　语句 1 子树
T2　语句 2 子树（可空）
	访问到 T：if(T2 非空)　{T1.Snext=newLabel , T2.Snext=T.Snext;}
	else T1.Snext=S.next;
	访问 T 的所有子树后：
	if (T2 为空)　T.code=T1.code
	else　T.code=T1.Snext || T1.Snext || T2.code

EXP_STM T1　表达式子树	访问到 T：T1.Snext=T.Snext; 访问 T 的所有子树后：T.code=T1.code
RETURN T1　表达式子树（可空）	访问到 T：T1.Snext=T.Snext; 访问 T 的所有子树后： if (T1 非空)　T.code=T1.code|| return T1.alias else　T.code=T1.code|| return

FOR T1 表达式子树（可空） T2 条件子树 T3 表达式子树（可空） T4 for 语句子树	访问到 T：T2.Etrue = newLabel,T2.Efalse = T.Snext, T2.Snext = newLabel,T4.Snext = newLabel, T4.EBreak = T.Snext,T4.EContinue = newLabel; 访问 T 的所有子树后： T. code = T1.code || T3.Snext || T2.code || T2.Etrue || T4.code || T4.EContinue || T3.code || T4.Snext

在上表中，用到了 Etrue、Efalse、Ebreak、EContinue 四个属性，涉及到的语法结构有 if、if-else、while 循环、for 循环，对于 EBreak 和 EContinue 属性，涉及到的语法结构是 while 循环和 for 循环，对于 Etrue 和 Efalse 属性，计算规则是：条件为真则产生一个 Label 置于条件为真时执行的语句序列的前面，条件为假时，if-else 额外产生一个 Label 至于条件为假时执行的语句序列的前面，其余均继承 T.Snext。

对于 EBreak 和 EContinue 属性，计算规则是：对于 while 循环，EBreak 继承 T.Snext，EContinue 等于在条件语句前产生一个 Label；对于 for 循环，EBrak 继承 T.Snext，EContinue 等于在 for 循环体中的 T3 表达式前产生一个 Label。

对于各类表达式语句，直接按 2.5 中定义的 TAC 结构产生对应的 TAC 结点即可。可分为 3 种类型，如 x = y op z 产生的 TAC 结点，其 op 为 op 类型码，opn1 为 y，opn2 为 z，result 为 x；x op y 产生的 TAC 结点，其 op 为 op 类型码，opn1 为 y，result 为 x，opn2 为空；goto x 产生的 TAC 结点，其 result 为 x，op 为 goto，其余为空。

在表达式语句的中间代码中，需要产生临时变量来记录一些运算结果，需要定义 newTemp()函数产生一个唯一的临时变量名，将中间结果按临时变量的形式暂存在符号表中。

另外，关于数组的访问，例如对于数组 a[7][8][9]，访问 a[1][2][3]时，为了方便目标代码生成时计算其地址，故需产生一系列代码计算其地址，计算方式是：

（1）首先遍历数组访问结点，找到数组名，按名查找符号表，找到数组定义的信息，并通过内情向量获取其维数和每一维的大小；

（2）再次遍历数组访问结点，获取每一维的访问下标，并与其大小进行比较，若大于该维度的大小，则报错（动态语义检查）；

（3）若在范围内，则产生计算地址的中间代码，初始地址临时变量为 temp1，其值为最低维的访问下标；

（4）依次访问高纬的下标，临时变量为 tempx，新的地址 tempz = tempx* 低维度的大小 + temp1，temp1 = tempz。

遍历完毕后，tempz 中保存了该数组元素的地址，将这些中间代码连接起来，就形成了数组访问的中间代码。

经过一次语法树遍历后，产生的 TAC 链表头指针保存在了语法树的根结点中，逻辑上中间代码已产生，若需要打印出中间代码，则可按下表格式打印出中间代码。

表 3-3 中间代码转换规则

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		y
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	x
RETURN x	返回语句	RETURN			x
ARG x	传实参 x	ARG			x
x:=CALL f	调用函数(有返回值)	CALL	f		x
CALL f	调用函数(无返回值)	CALL	f		
PARAM x	函数形参	PARAM			x

3.7 代码优化

代码优化可采用窥孔优化、局部优化、循环优化、全局优化等方式，此处采用局部优化。

在局部优化中，常见的优化方法有基于流图的优化，和基于 DAG 图的优化，DAG 图优化需要建立 DAG 数据结构，对中间代码进行重新组织，较为麻烦，此处采用基于流图的优化方式。

首先是划分基本块，划分基本块的工作就是找到基本块的开始语句和技术语句，划分基本块的规则如下：

⑴ 依据入口语句定义，确定程序所有的入口语句；

⑵ 对每一个入口语句，确定对应的基本块。这些基本块是由入口语句向后直到

① 转移语句(包括该语句在内)；

② 或到停止语句；

③ 或到下一个入口语句(不包括该语句)之间的代码段。

⑶ 凡不属于任何一个基本块的语句都是无用语句，将其全部删除。

在 3.6 中将中间代码组织成双向循环链表的数据结构，此处可方便地表示基本块，每个基本块内存储两个指针 begin 和 end，从 TAC 链表头结点开始扫描，按上述规则，每扫描到开始语句，则产生一个基本块，记录到 begin 指针中；每扫描到结束语句，记录到 end 指针中，将该基本块存储到基本块数组内。

扫描完 TAC 序列即完成了基本块的划分，同时删除了不在任何基本块中无用代码，达到了代码优化的目的。

为进一步优化代码，由于有的基本块是不可达的，故可建立基本块间的流图，对流图进行遍历，未遍历到的基本块结点则可删除，建立流图并删除不可达基本块的步骤如下：

（1）扫描基本块结点，若其无分支指令，则下一个基本块是其邻居，若有分支指令，则标记该基本块有分支，记录分支 Label 和分支失败的 Label；

（2）再次扫描基本块结点，对每个基本块，若有 Label 记录，则遍历基本块，含有该 Label 的基本块是其邻居，（1）、（2）步完成了流图的建立；

（3）对产生的流图进行 dfs 遍历，对每个遍历到的结点标记为 1；

（4）再次扫描基本块结点，将标记为 0 的结点删除。

按以上步骤可完成流图的建立和不可达块的删除，具体实现代码见附件。代码优化步骤完成了无用代码的删除。

3.8 汇编代码生成

汇编代码选取 MIPS 指令集实现，采用了朴素寄存器分配算法，即每个要访问的变量先存储到数据段中，再读取到寄存器，这样最多同时只要用到 3 个寄存器。目标代码生成中，需要获取每个变量的偏移地址以正确划分程序的栈帧结构。

对于函数，在符号表的 offset 字段中已经记录了其栈帧大小，对于变量,offset 字段中记录了在程序中相对数据段的偏移地址。

除了函数调用 TAC 结点，其余 TAC 结点直接按下表规则翻译即可。

表 3-4 目标代码翻译规则

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) blt $t1,$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

对于函数调用，首先要分配其栈帧，在符号表中通过函数名查找，获取 offset 字段的值，将$sp 减去该值，然后是在符号表中查找该函数的所有参数，将参数压栈，最后调用 jal 指令转移到函数起始地址执行。函数执行返回后，需要恢复$sp 的值。

具体实现见附件 2。

4 系统测试与评价

4.1 测试用例

对编译器的测试分为正确性测试和报错测试两个测试方面。

在正确性测试中，要测试到所有设计的文法，如外部定义、函数定义、各类变量的声明及赋值、数组访问、if-else 结构、while 循环结构、for 循环结构、break 和 continue 循环控制、函数返回等。设计正确性测试用例如下；

```
int a, b, c;//外部声明测试及行注释测试
float m, n;
int d[10][10];//数组定义测试
/*
*块注释测试*/
int fibo(int a) {//函数定义测试
    if (a == 1 || a == 2)
        return 1;
    if (a>=3){/if 结构测试
        c = d[1][2];//数组访问测试
        return fibo(a-1) + fibo(a-2);//函数调用测试
    }
    return 0;//返回测试
}
int main(int argc) {
    int i;//内部变量声明测试
    for(i = 1;i<10;i++) {//for 循环测试
        if(i>9) {//if-else 结构测试
            fibo(i);//函数调用测试
            break;//break 测试
        }
        else{
            fibo(i);
            continue;//continue 测试
        }
        i++;//if-else 结构测试
    }
    return 1;
}
```

针对 2.4 中定义的各类错误类型，设计报错测试用例如下：

```
int a, b, c;
float m,n;
int b[10][10],d[10][10];//变量重发声明

int fibo(int a){
      test1(c);//函数未定义
       if (a == 1 || a == 2)
            return 1;
       return fibo(a - 1) + fibo(a - 2);
       if (a > 10)
            return fibo(a - 1, a - 2);//函数参数过多
       if (b > 100)
             return fibo(a - 1) + fibo(a - 2);
       else{
             b--;
       }
      fibo(tt);//变量未定义
      c = a[8]; //数组访问非法
      m = 0.8;
      n = 0.9;
      c = d[m][n];//数组访问下标不是 int 型
       break; // break 语句不在循环体内
      return 0;
}
void testf(){
      return 'a';//返回值类型不对
}

int   testff(){
       int i;
       i++;
}//缺少返回值

int main(int argc){
       int m=10, n, i;
       test1 = 0;//test1 未定义
       i = 1;
            n = m(); //m 不是函数
            (n+1) = m; //赋值语句需要左值
            --(n+1);   //自增自减对象只能是变量
            a = m; // a 未定义
            continue; //continue 语句不在循环体内
       while (i <= m){
              n = fibo(i);
              i = i + 1;
            if(i>10){
                    break;
              }
            else
                  continue;
       }
      n = fibo; //  函数调用缺少参数
       return 1;
}
```

4.2 正确性测试

正确测试代码的测试结构如图 4.1 所示。

```
E:\>cd E:\0编译原理\实验\lab4

E:\0编译原理\实验\lab4>parser testsematics4.c
.globl main0
.text
main0:
    addi $fp, $0, 0x10010000
    move $t0,$sp
    addi $sp, $sp, -24
    sw $ra,0($sp)
    jal main
    lw $ra,0($sp)
    addi $sp,$sp,24
    li $v0,10
    syscall

fibo:
    li $t3, 1
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    beq $t1,$t2,label3
    j label4

label4:
    li $t3, 2
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    beq $t1,$t2,label3
    j label2

label3:
```

图 4.1 正确测试结果

将输出的目标代码用 MARS 仿真器翻译成机器码，如图 4.2 所示。

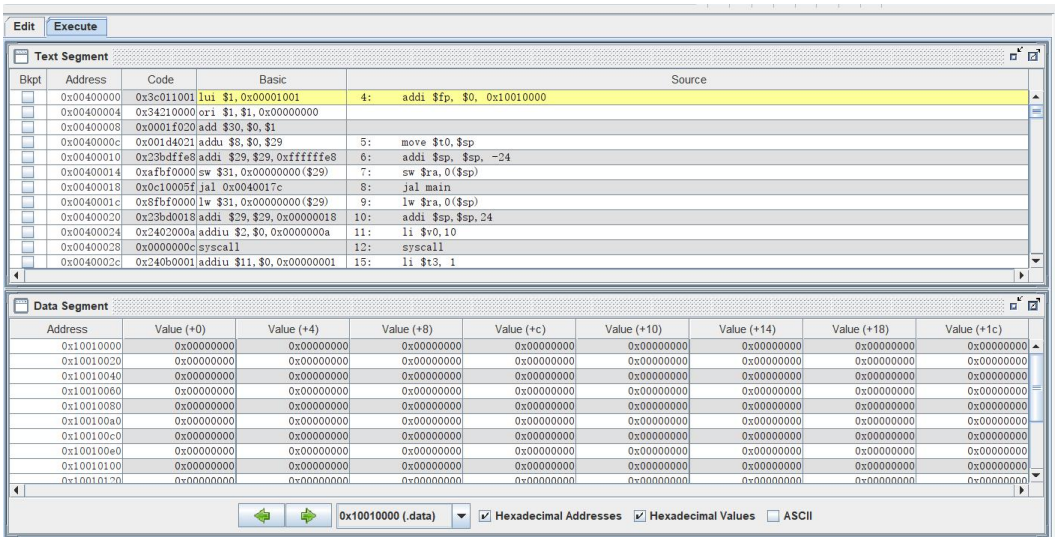

图 4.2 目标代码在 MARS 仿真器中翻译

在 Logisim 仿真器中，把机器码载入搭建的 CPU 的指令寄存器中，运行结果如图 4.3 所示。

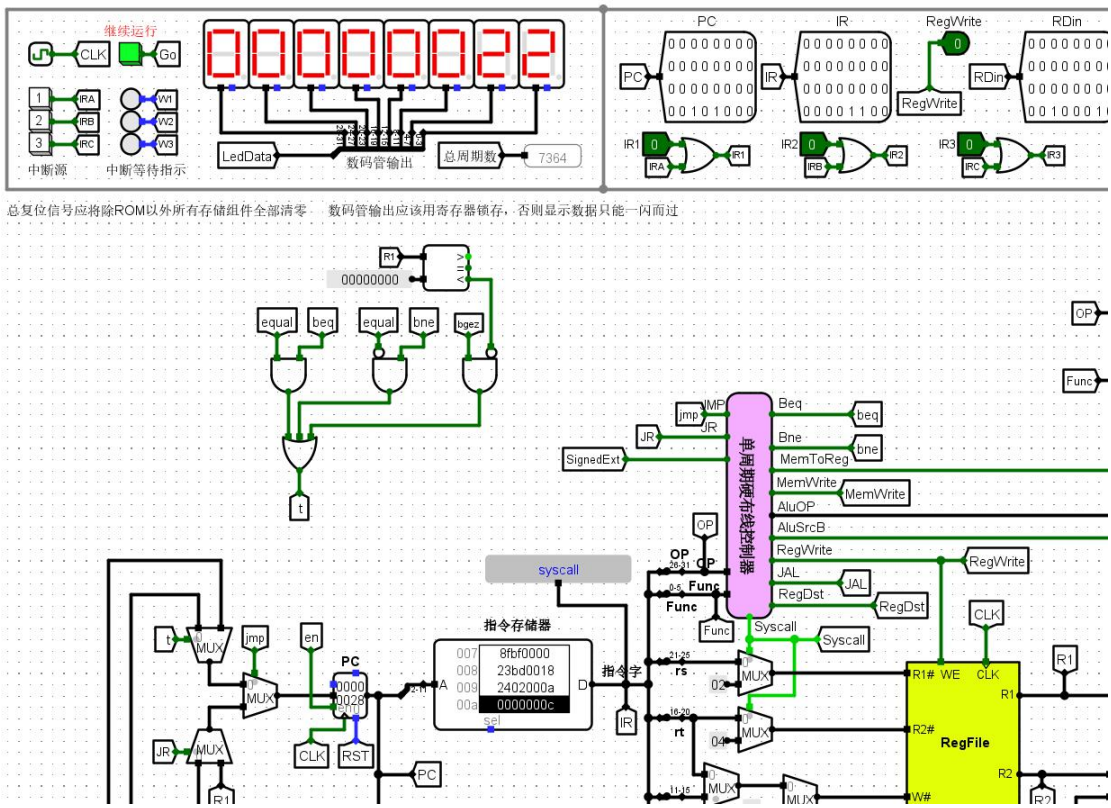

图 4.3 目标代码运行结果

结果显示了 fibonacci 数列第 10 项的 16 进制值 0x22，即 34，结果正确。

4.3 报错功能测试

编译报错测试代码，结果如图 4.4 所示，

```
静态语义错误信息：
error: in line 3,b 变量重复声明
error: in line 6,test1 函数未定义
error: in line 11, 函数调用参数太多
error: in line 17,tt 变量未定义
error: in line 17, 参数类型不匹配
error: in line 18,a 错误的数组访问
error: in line 21, 数组访问的下标只能是正整数
error: in line 22,break语句前没有匹配的循环体
error: in line 26,返回值类型错误
error: in line 32, 函数缺少返回值
error: in line 36,test1 变量未定义
error: in line 38,m 不是一个函数
error: in line 39, 赋值语句需要左值
error: in line 40, 自增自减对象只能是int变量
error: in line 42,continue语句前没有匹配的循环体
error: in line 44, 参数类型不匹配
error: in line 52,fibo 是函数名，类型不匹配
```

图 4.4 报错功能测试结果

编译器能够识别出定义的所有错误类型，并指出错误的行号。

4.4 系统的优点

（1）定义的语言支持了多维数组的访问、for 循环、break 和 continue 分支控制语句，程序功能更加强大；

（2）通过划分基本块以及构建流图的方式，删除了无用代码；

（3）编译器产生的目标代码能够在 MIPS32 的 CPU 上运行一段测试程序，具有实用性。

4.5 系统的缺点

（1）语言没有支持指针、结构体等高级特性；

（2）寄存器分配采用了朴素分配算法，导致目标代码产生了大量 lw、sw 指令，没有充分利用寄存器组，目标代码运行速度较慢。

5 实验小结或体会

在本学期编译原理实验中，通过 4 次实验，学会了编译过程中的词法分析、文法分析、语义分析、中间代码生成、代码优化和目标代码生成的基本步骤，实验难度较大，代码量很多。

第一次实验的难度在于学习 flex 和 yacc 程序的编写，设计合适的文法来消除移进—规约冲突和规约—规约冲突，以及生成抽象语法树 AST 为后续实验打好基础；第二次实验的难度在于进行静态语义分析及属性值的计算，需要考虑到每个类型结点的属性值的计算规则，设计完备的属性计算规则；第三次实验的难度在于遍历语法树产生 TAC 结点时，TAC 的组织方式，以及代码优化，由于时间限制，代码优化仅删除了无用代码；第四次实验的难度在于通过 offset 属性，正确地设置函数的栈帧和遍历的存储空间分配，考虑到寄存器分配算法是 NP 难问题，故简单地采用了朴素寄存器分配算法。

由于疫情影响，导致在家里度过了整个学期，不能像往常一样方便的和同学及老师进行交流，这也为实验添加了难度。由于实验是环环相扣的，前面的没做好就会为后面的实现埋下隐患，我就在进行后面的实验过程中反复修改了实验咦定义的词法和文法，一开始设计了结构体的文法规则，但后面进行语义分析时实在是很难处理结构体，遂放弃了结构体的实现。

在此还要多谢指导老师在每次检查时的严格要求，使我们在后续实验中避免了很多的隐患。

总之，这门实验课虽然很难，但收获也很多，提高了编程能力，也对课上学习的理论知识亲手进行了实现，加深了我对理论知识的理解，最终能够实现一个简单的编译器，也很有成就感。

参考文献

[1] 王生元等. 编译原理(第三版). 北京：清华大学出版社，20016

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

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

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C 语言程序设计. 北京：科学出版社，2008

附件 1：代码优化源代码

1. 基本块定义：

```
//基本块节点
struct block
{
    char name[10];//基本块名字，形如：BLOCK1、BLOCK2...
    struct codenode *begin;//基本块代码链表头指针
    struct codenode *end;//基本块代码链表尾指针
    struct block_node next[2];//其 next 流节点信息，用于代码优化
    int num;//next 节点个数
    char label[32];//记录基本块中的 label
    int hasLabel;//标记 label 是否存在
    int isSearched;//搜索标记，用于遍历基本块时判断是否已搜索
    int isFunction;//函数基本块标记，若基本块内有 FUNCTION 则置 1，用于代码优化
};
//基本块链表
struct block_list
{
    int valid;//有效位，若基本块无有效语句，则 valid = 0
    struct block *BLOCK;
};
```

2. 基本块生成

```
//生成一个基本块节点，指向该基本块代码链表的起始和结尾
struct block *genBlock(char *name, struct codenode *begin, struct codenode *end)
{
    struct block * BLOCK = (struct block *)malloc(sizeof(struct block));
    strcpy(BLOCK->name, name);
    BLOCK->begin = begin;
    BLOCK->end = end;
}
```

3. 基本块链表生成

```
//扫描中间代码产生基本块，填充基本块列表
void fill_block_list(struct codenode *head)
{
    struct codenode *h = head,*begin,*end;
    int begin_flag,end_flag,branch_flag,return_goto_flag;
    begin_flag = end_flag = branch_flag = return_goto_flag =  0;
    do
```

```
    {
        if(return_goto_flag)
        {
            begin = h;
            begin_flag = 1;
            return_goto_flag = 0;
            if (h->op >= JLT && h->op <= NEQ || h->op == RETURN || h-> op == GOTO)//
若该语句是特殊语句，则结束
            {
                end = h;
                end_flag = 1;
                if (h->op >= JLT && h->op <= NEQ)
                {
                    branch_flag = 1;
                }
                else
                {
                    return_goto_flag = 1;
                }

            }
        }
        else if (h->op == LABEL && begin_flag == 0)
        {
            begin = h;
            begin_flag = 1;
        }
        else if (h->op == LABEL && begin_flag == 1)
        {
            end = h->prior;
            begin_flag = 1;
            end_flag = 1;
            BLOCK_LIST[block_list_index].BLOCK = genBlock(newBlockName(), begin,
end);
            BLOCK_LIST[block_list_index].valid = 1;
            block_list_index++;
            begin_flag = end_flag = 0;
            begin = h;
            begin_flag = 1;
            h = h->next;
            continue;
        }
        else if (h->op == FUNCTION)
        {
```

```
            begin = h;
            begin_flag = 1;
        }
        else if (h->op >= JLT && h->op <= NEQ)
        {
            end = h;
            end_flag = branch_flag = 1;
        }
        else if(h->op == RETURN || h->op == GOTO)
        {
            end = h;
            end_flag = 1;
            return_goto_flag = 1;
        }
        if (begin_flag && end_flag)
        {
            BLOCK_LIST[block_list_index].BLOCK = genBlock(newBlockName(), begin,
end);
            BLOCK_LIST[block_list_index].valid = 1;
            block_list_index++;
            begin_flag = end_flag = 0;
        }
        if (branch_flag)
        {
            begin = end->next;
            begin_flag = 1;
            branch_flag = 0;
        }
        h=h->next;
    } while (h != head);
}
```

4. 流图生成

```
//基本块优化
void blockOptimization(void)
{
    struct codenode *c;
    int i,j,branch_flag;
    struct block *BLOCK = NULL;
    //第一遍扫描基本块数组，初始化基本块的部分信息
    for ( i = 0; i < block_list_index; i++)
    {
        BLOCK = BLOCK_LIST[i].BLOCK;
        branch_flag = 0;//初始分支标记为 0
```

```
//初始化基本块信息
BLOCK->num = 0;
BLOCK->hasLabel = 0;
BLOCK->isFunction = 0;
BLOCK->isSearched = 0;
BLOCK->next[0].pos = BLOCK->next[1].pos = -1;
c = BLOCK->begin;
do
{
    if (c->op == FUNCTION)
        BLOCK->isFunction = 1;
    if (c->op == LABEL)
    {
        BLOCK->hasLabel = 1;
        strcpy(BLOCK->label, c->result.id);//基本块标记为其 LABEL 代码节点
    }
    if (c->op ==RETURN)
    {
       BLOCK->num = 0;//返回语句是基本块的结束语句，表示无 next 流
       branch_flag = 1;
       break;
    }
    if (c->op == GOTO)
    {
        BLOCK->num = 1;//GOTO 语句是基本块的结束语句，有一个 next 流
        strcpy(BLOCK->next[0].label, c->result.id);//记录分支目标的 LABEL
        branch_flag = 1;
        break;
    }
    if (c->op >= JLT && c->op <= NEQ)
    {
        BLOCK->num = 2;//条件转移语句是基本块的结束语句，有条件为真和条件为假的两个 next 流，此处仅记录数量和条件为真的转移 LABEL
        //prnIR(c,c->next);
        strcpy(BLOCK->next[0].label, c->result.id);//记录分支目标的 LABEL
        branch_flag = 1;
        break;
    }
    c = c->next;
} while (c != BLOCK->begin);
if (branch_flag == 0)//基本块无分支，若不是最后一个基本块，下一个基本块是其 next 流
{
    if (i < block_list_index - 1)//不是最后一个基本块
```

```
                {
                    BLOCK->num = 1;
                    BLOCK->next[0].pos = i + 1;//记录 next 流的基本块标号
                }
            }
        }
        //第二遍扫描基本块数组，
        for ( i = 0; i < block_list_index; i++)
        {
            BLOCK = BLOCK_LIST[i].BLOCK;
            if (BLOCK->num == 1)//该基本块有一个 next 流
            {
                if (BLOCK->next[0].pos != -1)//已经记录了 next 流基本块的下标
                {
                    if (BLOCK_LIST[BLOCK->next[0].pos].BLOCK->hasLabel)// 如果 next 流
基本块有标签

strcpy(BLOCK->next[0].label,BLOCK_LIST[BLOCK->next[0].pos].BLOCK->label);//将 next 流
的基本块标签传递到基本块信息节点
                    continue;
                }
                for (j = 0; j < block_list_index; j++)//搜索 next 流基本块，并记录下标
                {
                    if (!strcmp(BLOCK->next[0].label, BLOCK_LIST[j].BLOCK->label))// 两个
label 相等，即找到目标基本块
                    {
                        BLOCK->next[0].pos = j;//记录下标
                    }
                }
            }
            else if (BLOCK->num == 2)//该基本块有两个 next 流，此处仅为条件分支语句的基
本块
            {
                BLOCK->next[1].pos = i + 1;//条件分支语句的条件为假语句，在基本块数组中
即下一个元素，下标为 i+1
                if (i < block_list_index - 1)//如果不是最后一个基本块
                    if (BLOCK_LIST[i + 1].BLOCK->hasLabel)//如果该基本块有 label
                        strcpy(BLOCK->next[1].label, BLOCK_LIST[i + 1].BLOCK->label);//
将 next 流中条件为假的分支地址的 label 传递到基本块信息节点
                for (j = 0; j < block_list_index; j++)//搜索 next 流基本块，并记录下标
                {
                    if (!strcmp(BLOCK->next[0].label, BLOCK_LIST[j].BLOCK->label))// 两个
label 相等，即找到目标基本块
                    {
```

```
                    BLOCK->next[0].pos = j;//记录下标
                }
            }
        }
    }
    printf("\n 流图的 dfs 遍历：\n");
    //第三遍扫描基本块数组，对其进行深度优先遍历，对遍历到的节点进行标记，注意起始节点一定是函数基本块节点
    for (i = 0; i < block_list_index; i++)
    {
        if (BLOCK_LIST[i].BLOCK->isFunction)
            dfsBlock(BLOCK_LIST[i].BLOCK);
    }
    printf("\n");
    //第四遍扫描基本块数组，将没遍历到的基本块的 valid 置为 0
    for (i = 0; i < block_list_index; i++)
    {
        if (BLOCK_LIST[i].BLOCK->isSearched == 0)//若该基本块未被搜索
        {
            BLOCK_LIST[i].valid = 0;//置 valid=0，逻辑删除该基本块
            optimization_msg(BLOCK_LIST[i].BLOCK->name, "基本块不可达，删除");
        }
    }
}
```

5. 流图遍历

```
void dfsBlock(struct block *BLOCK)
{
    if (BLOCK->isSearched)//基本块已扫描过，返回
        return;
        BLOCK->isSearched = 1;
    if (BLOCK->num == 1)//有一个 next 流
    {
        if (BLOCK->name && BLOCK_LIST[BLOCK->next[0].pos].BLOCK->name){
            printf("%s----->%s\n",BLOCK->name,
BLOCK_LIST[BLOCK->next[0].pos].BLOCK->name);
            dfsBlock(BLOCK_LIST[BLOCK->next[0].pos].BLOCK);}
    }
    else if(BLOCK->num == 2)//有两个 next 流
    {
        if (BLOCK->name && BLOCK_LIST[BLOCK->next[0].pos].BLOCK->name){
          printf("%s----->%s\n",BLOCK->name,
BLOCK_LIST[BLOCK->next[0].pos].BLOCK->name);
        dfsBlock(BLOCK_LIST[BLOCK->next[0].pos].BLOCK);}
```

```
            if (BLOCK->name && BLOCK_LIST[BLOCK->next[1].pos].BLOCK->name){
                printf("%s----->%s\n",BLOCK->name,
BLOCK_LIST[BLOCK->next[1].pos].BLOCK->name);
            dfsBlock(BLOCK_LIST[BLOCK->next[1].pos].BLOCK);}
        }
}
```

6. 基本块输出

```
//输出基本块
void prnBlock(void)
{
    int i = 0;
    while (i < block_list_index)
    {
        if (BLOCK_LIST[i].valid)
        {
            printf("------------------%s------------------\n",BLOCK_LIST[i].BLOCK->name);
            prnIR(BLOCK_LIST[i].BLOCK->begin, BLOCK_LIST[i].BLOCK->end->next);
            printf("------------------end-----------------------\n\n");
        }
        i++;
    }
}
```

附件 2：目标代码生成源代码

```
void objectCode(struct codenode *head, struct codenode *tail){
    char opnstr1[32],opnstr2[32],resultstr[32];
    struct codenode *h=head,*hh,*hhh;
    int i,global_flag1,global_flag2,global_flag3;
    printf(".globl main0\n");
    printf(".text\n");
    printf("main0:\n");
    printf("       addi $fp, $0, 0x10010000\n");//mars 仿真器下 data 段的起始地址
    for ( i = 0; i < symbolTable.index; i++)
    {
        if (!strcmp("main",symbolTable.symbols[i].name))
        {
            break;
        }

    }
    printf("       move $t0,$sp\n");
    printf("       addi $sp, $sp, -%d\n", symbolTable.symbols[i].offset);
    printf("       sw $ra,0($sp)\n");
    printf("       jal main\n");
    printf("       lw $ra,0($sp)\n");
    printf("       addi $sp,$sp,%d\n",symbolTable.symbols[i].offset);
    printf("       li $v0,10\n");
    printf("       syscall\n");
    do
    {
        switch (h->op)
        {

          case ASSIGNOP:
                        if (h->opn1.kind==INT)//只考虑整形运算
                            printf("       li $t3, %d\n", h->opn1.const_int);
                        else
                        {
                            if(h->opn1.level == 0)
                            {
                                printf("       addi $t2, $fp, %d\n", h->opn1.offset);
                                printf("       lw $t1, 0($t2)\n");
                            }
                            else
```

```
                    printf("        lw $t1, %d($sp)\n", h->opn1.offset);
                printf("        move $t3, $t1\n");
            }
            if(h->result.level == 0)
            {
                printf("        addi $t2, $fp, %d\n", h->result.offset);
                printf("        lw $t1, 0($t2)\n");
            }
            else
                printf("        sw $t3, %d($sp)\n", h->result.offset);
            break;
case ASSIGNOP_ARRAY:
            printf("        lw $t1,    %d($sp)\n",h->opn2.offset);
            if(h->opn1.level == 0)
            {
                printf("        addi $t2, $fp, %d\n",h->opn1.offset);
                printf("        add $t1, $t1, $t2\n");
            }
            else
                printf("        add $t1, $t1, %d($sp)\n",h->opn1.offset);
            printf("        lw $t2, 0($t1)\n",h->opn1.offset);
            if (h->result.level == 0)
            {
                printf("        sw $t2, %d($fp)\n",h->result.offset);
            }
            else
                printf("        sw $t2, %d($sp)\n", h->result.offset);
            break;
case ARRAY_ASSIGNOP:
            printf("        lw $t2,    %d($sp)\n",h->opn1.offset);
            printf("        lw $t1,    %d($sp)\n",h->opn2.offset);
            if(h->result.level == 0)
                printf("        add $t1, $t1, -%d($fp)\n",h->result.offset);
            else
                printf("        add $t1, $t1, %d($sp)\n",h->result.offset);
            printf("        sw $t2, %d($t1)\n",h->opn1.offset);
            break;
case PLUS:
case MINUS:
case STAR:
case DIV:
            printf("        lw $t1, %d($sp)\n", h->opn1.offset);
            printf("        lw $t2, %d($sp)\n", h->opn2.offset);
            if (h->op == PLUS)
```

```
                printf("    add $t3,$t1,$t2\n");
            else if (h->op == MINUS)
                printf("    sub $t3,$t1,$t2\n");
            else if (h->op == STAR)
                printf("    mul $t3,$t1,$t2\n");
            else
            {
                printf("    div $t1, $t2\n");
                printf("    mflo $t3\n");
            }
            printf("    sw $t3, %d($sp)\n", h->result.offset);
            break;
        case PRE_PLUS_SELF:
        case PRE_MINUS_SELF:
            printf("    lw $t1, %d($sp)\n",h->opn1.offset);
            printf("        addi    $t1,    $t1,    %c1\n",h->op    ==
PRE_PLUS_SELF?'+':'-');
            printf("    sw $t1, %d($sp)\n",h->opn1.offset);
            printf("    move $t3, $t1\n");
            printf("    sw $t3, %d($sp)\n", h->result.offset);
            break;
        case NEXT_PLUS_SELF:
        case NEXT_MINUS_SELF:
            printf("    lw $t1, %d($sp)\n",h->opn1.offset);
            printf("    move $t3, $t1\n");
            printf("    sw $t3, %d($sp)\n", h->result.offset);
            printf("        addi    $t1,    $t1,    %c1\n",h->op    ==
NEXT_PLUS_SELF?'+':'-');
            printf("    sw $t1, %d($sp)\n",h->opn1.offset);
            break;
        case ASSIGNOP_PLUS:
        case ASSIGNOP_MINUS:
        case ASSIGNOP_STAR:
        case ASSIGNOP_DIV:
            printf("    lw $t1, %d($sp)\n",h->opn2.offset);
            printf("    lw $t2, %d($sp)\n",h->opn1.offset);
            if (h->op != ASSIGNOP_DIV)
                printf("                %s      $t3,    $t2,
$t1\n",h->op==ASSIGNOP_PLUS?"add":h->op==ASSIGNOP_MINUS?"sub":"mul");
            else
            {
                printf("    div $1, $2\n");
                printf("    mflo $3\n");
            }
```

```
                printf("    sw $t3, %d($sp)\n",h->opn1.offset);
                break;
        case GOTO:
                printf("    j %s\n",h->result.id);
                break;
        case RETURN:
                printf("    lw $v0,%d($sp)\n",h->result.offset);
                printf("    jr $ra\n");
                break;
        case JLE:
        case JLT:
        case JGE:
        case JGT:
        case EQ:
        case NEQ:
                printf("    lw $t1, %d($sp)\n", h->opn1.offset);
                printf("    lw $t2, %d($sp)\n", h->opn2.offset);
                if (h->op == JLE)
                    printf("    ble $t1,$t2,%s\n", h->result.id);
                else if (h->op == JLT)
                    printf("    blt $t1,$t2,%s\n", h->result.id);
                else if (h->op == JGE)
                    printf("    bge $t1,$t2,%s\n", h->result.id);
                else if (h->op == JGT)
                    printf("    bgt $t1,$t2,%s\n", h->result.id);
                else if (h->op == EQ)
                    printf("    beq $t1,$t2,%s\n", h->result.id);
                else
                    printf("    bne $t1,$t2,%s\n", h->result.id);
                break;
        case FUNCTION:
                printf("\n%s:\n", h->result.id);
                // if (!strcmp(h->result.id,"main"))
                //                 printf("    addi    $sp,    $sp,
-%d\n",symbolTable.symbols[h->result.offset].offset);
                break;
        case PARAM:
                break;
        case LABEL:
                printf("\n%s:\n",h->result.id);
                break;
        case ARG:
                break;
        case CALL:
```

```
for(hh=h,i=0;i<symbolTable.symbols[h->opn1.offset].paramnum;i++)
                    hh=hh->prior;
                printf("    move $t0,$sp\n");
                printf("                addi    $sp,    $sp,    -%d\n",
symbolTable.symbols[h->opn1.offset].offset);
                printf("    sw $ra,0($sp)\n");
                i=h->opn1.offset+1;
                while (symbolTable.symbols[i].flag=='P')
                {
                    printf("    lw $t1, %d($t0)\n", hh->result.offset);
                    printf("    move $t3,$t1\n");
                    printf("                sw      $t3,%d($sp)\n",
symbolTable.symbols[i].offset);
                    hh=hh->next;
                    i++;
                }
                printf("    jal %s\n",h->opn1.id);
                printf("    lw $ra,0($sp)\n");
                printf("                                    addi
$sp,$sp,%d\n",symbolTable.symbols[h->opn1.offset].offset);
                printf("    sw $v0,%d($sp)\n", h->result.offset);
                break;
            }
        h=h->next;
        } while (h != tail);
}
```