

# 华中科技大学

## 编译原理实验课程报告

题目：MiniC 语言编译器设计与实现

班 级\_\_\_\_\_

姓 名 黄浩岩\_\_\_\_\_

学 号\_\_\_\_\_

指导教师 刘铭\_\_\_\_\_

报告日期 2020 年 12 月 30 日\_\_\_\_\_

网络空间安全学院

## 要 求

- 1、报告本人独立完成，内容真实，如发现抄袭，成绩无效；如果引用查阅的资料，需将资料出处列入参考文献，其格式请按华中科技大学本科毕业论文规范，并在正文中标注参考文献序号；
- 2、按编译原理实验内容，应包含：语言的文法、语言的词法及语法分析、语义分析、中间代码生成、目标代码生成；
- 3、应说明编译器编写调试时的系统环境、各编译器部分主要采用的设计思路、关键过程、设计的结果（主要源码文件功能、数据结构、函数说明）；同时，对设计实现中遇到的问题、解决过程进行记录和小节；并对完成的编译器过程进行总结，明确指出其优点、不足；
- 4、评分标准：4 个主要实验环节按任务书要求完成；采用的方法合适、设计合理；能体出分析问题、灵活运用知识解决实际问题的能力；报告条理清晰、语句通顺、格式规范；

格式 规范	词法 语法	语义 分析	中间 代码	目标 代码	总分
20	30	30	10	10	100

# 目 录

1 选题背景 .....	1
1.1 总体任务 .....	1
1.2 目标 .....	1
1.3 主要技术 .....	1
2 实验一 词法分析和语法分析 .....	2
2.1 词法的文法描述 .....	2
2.2 语法的文法描述 .....	4
2.3 词法分析器设计 .....	9
2.4 语法分析器设计 .....	12
2.5 词法及语法分析器运行截图 .....	20
2.6 小结 .....	22
3 实验二 语义分析 .....	23
3.1 语义子程序描述 .....	23
3.2 符号表的设计 .....	23
3.3 语义错误类型定义 .....	24
3.4 语义分析实现技术 .....	25
3.5 语义分析结果 .....	25
3.6 小结 .....	28
4 实验三 中间代码生成 .....	30
4.1 中间代码格式定义 .....	30
4.2 中间代码生成规则定义 .....	31
4.3 中间代码生成过程 .....	32
4.4 中间代码生成结果 .....	33
4.5 小结 .....	36
5 实验四 目标代码生成 .....	37
5.1 指令集选择 .....	37
5.2 寄存器分配算法 .....	37
5.3 目标代码生成算法 .....	38
5.4 目标代码生成结果 .....	40
5.5 目标代码运行结果 .....	42
5.6 小结 .....	43
6 总结 .....	44
参考文献 .....	45

# 1 选题背景

## 1.1 总体任务

根据密码学编程时的某个函数（例如计算最大公约数函数）需要完成的程序功能，运用编译原理课程中的知识，设计一个类 C 语言程序设计语言相应的词法及语法，并实现该语言的编译器，将源码翻译为能在模拟器上运行的目标代码。

## 1.2 目标

通过资料查阅、分析设计、编程实现等步骤，逐步培养学生解决工程问题的能力，提高独立思考、灵活运用理论知识以解决实际问题的能力；

通过构造简化编译器的过程，了解编译器各部分的理论知识，找出理论与实践中的差异；灵活运用第三方工具协助解决工程问题；综合运用数据结构、算法、汇编语言、C 语言编程等前序课程的知识 and 技能。

## 1.3 主要技术

采用的具体开发环境、运行环境、完成编译器设计及实现中的关键技术；

## 2 实验一 词法分析和语法分析

实验一任务：

- 1、完成语言的语法描述和单词描述；完成语法规则、词法规则定义；
- 2、构造语法分析程序、词法分析程序；完成抽象语法树 AST 的设计；
- 3、构造合适的源程序样例，参考文献[1]1.1.6 必做内容；
- 4、将样例作为编译程序输入，输出其对应的抽象语法树；
- 5、根据参考文献[1]，完成错误提示功能。

### 2.1 词法的文法描述

#### 1. 语法规则

实验中设计的语言称为 MiniC，采用类 C 语言的文法，并进行了一定简化。其文法如下：

G[Program]:

$\text{Program} \rightarrow \text{ExtDefList}$

$\text{ExtDefList} \rightarrow \varepsilon \mid \text{ExtDef ExtDefList}$

$\text{ExtDef} \rightarrow \text{Specifier ExtDecList}; \mid \text{Specifier FuncDec CompSt} \mid \text{Specifier ArrayDec};$

$\text{Specifier} \rightarrow \text{int} \mid \text{float} \mid \text{char} \mid \text{void}$

$\text{ExtDecList} \rightarrow \text{VarDec} \mid \text{VarDec}, \text{ExtDecList}$

$\text{VarDec} \rightarrow \text{ID}$

$\text{FuncDec} \rightarrow \text{ID} ( \text{VarList} ) \mid \text{ID} ( )$

$\text{VarList} \rightarrow \text{ParamDec} \mid \text{ParamDec}, \text{VarList}$

$\text{ParamDec} \rightarrow \text{Specifier VarDec}$

$\text{ArrayDec} \rightarrow \text{ID} [ \text{Exp} ] \mid \text{ID} [ ]$

$\text{CompSt} \rightarrow \{ \text{DefList StmList} \}$

$\text{StmList} \rightarrow \varepsilon \mid \text{Stmt StmList}$

$\text{Stmt} \rightarrow \text{Exp}; \mid \text{CompSt} \mid \text{return Exp}; \mid \text{return}; \mid \text{break}; \mid \text{continue}; \mid \text{if} ( \text{Exp} ) \text{Stmt} \mid$   
 $\text{if} ( \text{Exp} ) \text{Stmt} \text{else Stmt} \mid \text{while} ( \text{Exp} ) \text{Stmt} \mid \text{for} ( \text{ExpList}; \text{Exp}; \text{ExpList} ) \text{Stmt} \mid \text{for}(; \text{Exp}; )$   
 $\text{Stmt} \mid ;$

$\text{DefList} \rightarrow \varepsilon \mid \text{Def DefList}$

$\text{Def} \rightarrow \text{Specifier DecList}; \mid \text{Specifier ArrayDec};$

$\text{DecList} \rightarrow \text{Dec} \mid \text{Dec}, \text{DecList}$

$$\text{Dec} \rightarrow \text{VarDec} \mid \text{VarDec} = \text{Exp}$$

$$\begin{aligned} \text{Exp} \rightarrow & \text{Exp} = \text{Exp} \mid \text{Exp} \&\& \text{Exp} \mid \text{Exp} \parallel \text{Exp} \mid \text{Exp} < \text{Exp} \mid \text{Exp} \leq \text{Exp} \mid \text{Exp} == \\ & \text{Exp} \mid \text{Exp} != \text{Exp} \mid \text{Exp} > \text{Exp} \mid \text{Exp} \geq \text{Exp} \mid \text{Exp} + \text{Exp} \mid \text{Exp} - \text{Exp} \mid \text{Exp} * \text{Exp} \mid \text{Exp} / \\ & \text{Exp} \mid \text{Exp} += \text{Exp} \mid \text{Exp} -= \text{Exp} \mid \text{Exp} *= \text{Exp} \mid \text{Exp} /= \text{Exp} \mid ( \text{Exp} ) \mid - \text{Exp} \mid ! \text{Exp} \mid ++ \text{Exp} \mid \\ & -- \text{Exp} \mid \text{Exp} ++ \mid \text{Exp} -- \mid \text{ID} ( \text{Args} ) \mid \text{ID} ( ) \mid \text{ID} \mid \text{INT} \mid \text{FLOAT} \mid \text{CHAR} \end{aligned}$$

$$\text{ExpList} \rightarrow \text{Exp} , \text{ExpList} \mid \text{Exp}$$

## 2. 终结符描述

MiniC 中的终结符可分为标识符、关键字、运算符、界符、常量以及注释，共 6 类。每种终结符的文法定义如表 2-1 所示。

表 2-1 MiniC 中终结符的文法定义

终结符类型	单词种类码	正则表达式
{id}	ID (标识符)	[A-Za-z][A-Za-z0-9]*
{char}	CHAR	'[A-Za-z0-9]'
{int}	INT	[0-9]+
{float}	FLOAT	([0-9]*\.[0-9]+) ([0-9]+\.)
"char"	TYPE	
"int"	TYPE	
"float"	TYPE	
"return"	RETURN	
"if"	IF	
"else"	ELSE	
"while"	WHILE	
"for"	FOR	
"void"	VOID	
","	SEMI	
" , "	COMMA	
">"   "<"   ">="   "<="   "=="   "!="	RELOP	
"="	ASSIGNOP	
"+"	PLUS	
"+="	COMADD	
"_="	COMSUB	
"*="	COMSTAR	
"/="	COMDIV	
"++"	SELFADD	
"--"	SELFSUB	
"_"	MINUS	
"*"	STAR	
"/"	DIV	

"&&"	AND	
"  "	OR	
"!"	NOT	
"("	LP	
")"	RP	
"{"	LC	
"}"	RC	
"["	LB	
"]"	RB	
"//[^\n]*"	(单行注释)	
"/*(\\s .)*?*/"	(多行注释)	

### 3. 非终结符描述

MiniC 中的非终结符及其含义如表 2-2 所示

表 2-1 MiniC 中非终结符及其含义

非终结符	含义	非终结符	含义
Program	整个程序	CompSt	复合语句
ExtDefList	外部定义列表	StmList	语句列表
ExtDef	外部定义语句	Stmt	语句
Specifier	变量类型	DefList	定义列表
ExtDecList	变量名列表	Def	定义语句
VarDec	变量名	DecList	变量列表
FuncDec	函数定义	Dec	变量语句
VarList	函数入口参数列表	Exp	表达式
ParamDec	函数参数定义	ExpList	表达式列表
ArrayDec	数组定义语句		

## 2.2 语法的文法描述

### 1. 定义非终结符和终结符的语义值类型

对于语法中的终结符和非终结符，由于 Bison 的需要，要为其定义其语义值类型。

对于非终结符，其均为归约得到，有较多属性，因此其语义值类型选择使用一个结构体表示。此外由于最终要实现一个抽象语法树，因此此处采用结构体指针表示。对于 MiniC，将结点设置为了一个“Node”结构体类型的指针，结构体中包含结点类型、叶节点语义值类型等数据，用“ptr”表示。

对于终结符，根据其分类用不同的语义值类型进行表示。对于数值常量的语义值类型，使用其对应的类型进行定义，如字符常量使用 `char` 类型的“`type_char`”记录、浮点常量使用 `float` 类型的“`type_float`”记录、整数常量使用 `int` 类型的“`type_int`”记录。对于标识符和变量类型（包括“`int`”“`char`”“`float`”），其本质都是字符串，因此两者的语义值类型使用字符数组“`type_id`”定义。此外，由于 Bison 本身需要，对于“`>`”“`<`”“`=`”“`!=`”等比较运算符的语义值类型也使用上述字符数组“`type_id`”定义。而对于“`+`”“`;`”“`&&`”“`while`”“`if`”等运算符、界符、关键字等则不需要特别定义其语义值类型，Bison 则默认定义为 `int` 类型。

## 2. 定义运算符的优先级和结合性

对于终结符中的运算符，需要设定其优先级和结合性，如表 2-3 所示。

表 2-1 MiniC 中运算符及结合性

优先级	结合性	符号
低	左	“+=”， “-=”
	左	“=”
	左	“  ”
	左	“&&”
	左	“>”， “<”， “>=”， “<=”， “==”， “!=”
	左	“+”， “-”
	左	“*”， “/”
高	右	“-”（负号）“! ”， “++”， “--”

## 3. 定义语法规则

根据上述终结符和非终结符的语义值类型定义，以及算符的优先级和结合性，结合 MiniC 的文法规则，定义其语法规则。其中，按照 Bison 的语法要求，使用“`:`”来表示“生成”；规则后面使用大括号“`{}`”括起的语句为该语句归约时需要执行的语义动作；左部的非终结符使用 `$$` 表示，右边的非终结符按照次序使用 `$1`、`$2` 等表示。此外，语法规则中加入了错误处理语句，当归约遇到错误时则以分号（`SEMI`）为标识跳过该语句，继续进行后续的语法分析。

MiniC 的语法规则如下：

<pre>//整个程序：语义分析入口,显示整棵抽象语法树 Program: ExtDefList {displayAST(\$1,0);} ; //外部定义列表：即整棵抽象语法树,每个结点第 1 棵子树对应外部变量声明或函数 ExtDefList: {\$\$=NULL;} //抽象语法树为空   ExtDef ExtDefList {\$\$=mknode(EXT_DEF_LIST,yylineno,2,\$1,\$2);} //其第 1 棵子树对应一个外部变量或函数定义 ; //外部定义语句 ExtDef: Specifier ExtDecList SEMI {\$\$=mknode(EXT_VAR_DEF,yylineno,2,\$1,\$2);} //外部(全局)变量定义语句   Specifier FuncDec CompSt {\$\$=mknode(FUNC_DEF,yylineno,3,\$1,\$2,\$3);} //函数定义</pre>
--



```

    | Specifier ArrayDec SEMI {$$=mknode(ARRAY_DEF,yylineno,2,$1,$2);}
    //数组定义语句
    | error SEMI {$$=NULL;}
    ;
//变量类型: 包括 int,float,char
Specifier: TYPE {$$=mknode(TYPE,yylineno,0); strcpy($$->type_id,$1);
$$->type=(!strcmp($1,"int"?INT:(!strcmp($1,"float"?FLOAT:CHAR));}
    | VOID {$$=mknode(VOID,yylineno,0); strcpy($$->type_id,"void");
$$->type=VOID;}
    ;
//变量名列表: 由一个或多个变量名组成, 多个变量名之间用逗号隔开
//其第 1 棵子树对应一个变量名(ID 类型的结点), 第 2 棵子树对应剩下的外部
变量名
ExtDecList: VarDec {$$=$1;}
    | VarDec COMMA ExtDecList
{$$=mknode(EXT_DEC_LIST,yylineno,2,$1,$3);}
    ;
//变量名: 由一个标识符 ID 组成
VarDec: ID {$$=mknode(ID,yylineno,0);strcpy($$->type_id,$1);} //ID 结点,标识
符字符串存放结点的 type_id 中
    ;
//函数定义
FuncDec: ID LP VarList RP
{$$=mknode(FUNC_DEC,yylineno,1,$3);strcpy($$->type_id,$1);} //有入口参数
    | ID LP RP {$$=mknode(FUNC_DEC,yylineno,0);strcpy($$->type_id,$1);}
//无入口参数
    | error RP {$$=NULL; printf("---函数定义存在错误---\n");}
    ;
//函数入口参数列表: 由一到多个函数参数定义组成, 用逗号隔开
VarList: ParamDec {$$=mknode(PARAM_LIST,yylineno,1,$1);}
    | ParamDec COMMA VarList {$$=mknode(PARAM_LIST,yylineno,2,$1,$3);}
    ;
//函数参数定义:由 1 个变量类型和 1 个变量名组成
ParamDec: Specifier VarDec {$$=mknode(PARAM_DEC,yylineno,2,$1,$2);}
    ;
//数组定义语句
ArrayDec: ID LB Exp RB
{$$=mknode(ARRAY_DEC,yylineno,1,$3);strcpy($$->type_id,$1);} //给定了数组
大小
    | ID LB RB {$$=mknode(ARRAY_DEC,yylineno,0);strcpy($$->type_id,$1);}
//未给定数组大小
    | error RB {$$=NULL;printf("---数组定义存在错误---\n");}
    ;
//复合语句: 由大括号括起,中间有定义列表和语句列表

```

```

CompSt: LC DefList StmList RC {$$=mknode(COMP_STM,yylineno,2,$2,$3);}
    | error RC {$$=NULL; printf("---复合语句存在错误---\n");}
    ;
//语句列表: 由 0 个或多个语句组成
StmList: {$$=NULL;}
    | Stmt StmList {$$=mknode(STM_LIST,yylineno,2,$1,$2);}
    ;
//语句: 可为表达式,复合语句,return 语句,if 语句,if-else 语句,while 语句,for 语句
Stmt: Exp SEMI {$$=mknode(EXP_STMT,yylineno,1,$1);}
    | CompSt {$$=$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 {$$=mknode(RETURN,yylineno,1,$2);}
    | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE
    {$$=mknode(IF_THEN,yylineno,2,$3,$5);} //if 语句
    | IF LP Exp RP Stmt ELSE Stmt
    {$$=mknode(IF_THEN_ELSE,yylineno,3,$3,$5,$7);} //if-else 语句
    | WHILE LP Exp RP Stmt {$$=mknode(WHILE,yylineno,2,$3,$5);}
    | FOR LP ExpList SEMI Exp SEMI ExpList RP Stmt
    {$$=mknode(FOR,yylineno,4,$3,$5,$7,$9);}
    | FOR LP SEMI Exp SEMI RP Stmt {$$=mknode(FOR_E,yylineno,2,$4,$7);}
    | BREAK SEMI
    {$$=mknode(BREAK,yylineno,0);strcpy($$->type_id,"BREAK");}
    | CONTINUE SEMI
    {$$=mknode(CONTINUE,yylineno,0);strcpy($$->type_id,"CONTINUE");}
    | RETURN SEMI
    {$$=mknode(RETURN_E,yylineno,0);strcpy($$->type_id,"RETURN");}
    | SEMI {$$=NULL;}
    ;
//定义列表: 由 0 个或多个定义语句组成
DefList: {$$=NULL;}
    | Def DefList {$$=mknode(DEF_LIST,yylineno,2,$1,$2);}
    ;
//定义语句: 由分号隔开
Def: Specifier DecList SEMI {$$=mknode(VAR_DEF,yylineno,2,$1,$2);} //不同
    变量定义
    | Specifier ArrayDec SEMI {$$=mknode(ARRAY_DEF,yylineno,2,$1,$2);}
    //数组定义
    ;
//变量列表, 由一个或多个变量语句组成, 由逗号隔开
DecList: Dec {$$=mknode(DEC_LIST,yylineno,1,$1);}
    | Dec COMMA DecList {$$=mknode(DEC_LIST,yylineno,2,$1,$3);}
    ;
//变量语句: 1 个变量名或者 1 个变量赋值语句
Dec: VarDec {$$=$1;}
    | VarDec ASSIGNOP Exp

```

```

{$$=mknode(ASSIGNOP,yylineno,2,$1,$3);strcpy($$->type_id,"ASSIGNOP");}
;
//表达式
Exp: Exp ASSIGNOP Exp
{$$=mknode(ASSIGNOP,yylineno,2,$1,$3);strcpy($$->type_id,"ASSIGNOP");}
//type_id 记录运算符
| Exp AND Exp
{$$=mknode(AND,yylineno,2,$1,$3);strcpy($$->type_id,"AND");}
| Exp OR Exp {$$=mknode(OR,yylineno,2,$1,$3);strcpy($$->type_id,"OR");}
| Exp RELOP Exp
{$$=mknode(RELOP,yylineno,2,$1,$3);strcpy($$->type_id,$2);} //词法分析关系运
算符号自身值保存在$2 中
| Exp PLUS Exp
{$$=mknode(PLUS,yylineno,2,$1,$3);strcpy($$->type_id,"PLUS");}
| Exp MINUS Exp
{$$=mknode(MINUS,yylineno,2,$1,$3);strcpy($$->type_id,"MINUS");}
| Exp STAR Exp
{$$=mknode(STAR,yylineno,2,$1,$3);strcpy($$->type_id,"STAR");}
| Exp DIV Exp {$$=mknode(DIV,yylineno,2,$1,$3);strcpy($$->type_id,"DIV");}
| Exp COMADD Exp
{$$=mknode(COMADD,yylineno,2,$1,$3);strcpy($$->type_id,"COMADD");}
| Exp COMSUB Exp
{$$=mknode(COMSUB,yylineno,2,$1,$3);strcpy($$->type_id,"COMSUB");}
| Exp COMSTAR Exp
{$$=mknode(COMSTAR,yylineno,2,$1,$3);strcpy($$->type_id,"COMSTAR");}
| Exp COMDIV Exp
{$$=mknode(COMDIV,yylineno,2,$1,$3);strcpy($$->type_id,"COMDIV");}
| LP Exp RP {$$=$2;} //遇到左右括号,直接忽略括号,表达式的值就为括号里
面的表达式值
| MINUS Exp %prec UMINUS
{$$=mknode(UMINUS,yylineno,1,$2);strcpy($$->type_id,"UMINUS");} //负号
| NOT Exp {$$=mknode(NOT,yylineno,1,$2);strcpy($$->type_id,"NOT");}
| SELFADD Exp
{$$=mknode(SELFADD_L,yylineno,1,$2);strcpy($$->type_id,"SELFADD");}
| SELFSUB Exp
{$$=mknode(SELFSUB_L,yylineno,1,$2);strcpy($$->type_id,"SELFSUB");}
| Exp SELFADD
{$$=mknode(SELFADD_R,yylineno,1,$1);strcpy($$->type_id,"SELFADD");}
| Exp SELFSUB
{$$=mknode(SELFSUB_R,yylineno,1,$1);strcpy($$->type_id,"SELFSUB");}
| ID LP ExpList RP
{$$=mknode(FUNC_CALL,yylineno,1,$3);strcpy($$->type_id,$1);} //有参数函数
调用
| ID LP RP {$$=mknode(FUNC_CALL,yylineno,0);strcpy($$->type_id,$1);} //

```

无参数函数调用

```
| ID {$$=mknode(ID,yylineno,0);strcpy($$->type_id,$1);}
| INT {$$=mknode(INT,yylineno,0);$$->type_int=$1;$$->type=INT;}
| FLOAT
{$$=mknode(FLOAT,yylineno,0);$$->type_float=$1;$$->type=FLOAT;}
| CHAR {$$=mknode(CHAR,yylineno,0); $$->type_char=$1;$$->type=CHAR;}
;
//实参列表
ExpList: Exp COMMA ExpList {$$=mknode(EXP_LIST,yylineno,2,$1,$3);}
| Exp {$$=mknode(EXP_LIST,yylineno,1,$1);}
;
```

## 2.3 词法分析器设计

词法分析器部分，通过编写 lex.l 文件结合 Flex 工具作为词法分析器，对 MiniC 语言进行词法分析。Flex 程序主要由声明部分、规则部分和 C 代码部分三部分构成，每个部分使用%%划分。

### 1. 声明部分

声明部分“%{ }%”部分代码会被拷贝到最后生成的 C 文件 lex.yy.c 的头部，其中，使用“#include”语句列出编译所需的头文件，此处包括 Bison 工具生成的“parser.tab.h”文件，以及编译过程中自定义数据结构部分的头文件“def.h”，和调用的字符串处理库“string.h”。使用“#define YY\_USER\_ACTION”定义完成语法单元定位的宏，用于在每个动作之前执行。然后定义联合体 YYLVAL，用于表示非终结符和终结符的语义值类型。

在这部分之后，使用“%option yylineno”记录符号的所在行号。

最后是辅助定义部分，使用模式宏定义用于对识别规则中出现的正规式进行辅助匹配。此处定义了整数、浮点数、字符和标识符 ID 的模式宏定义。

lex.l 的声明部分如下：

```
/*声明部分*/
%{
    /*头文件声明*/
    #include <string.h>
    #include "parser.tab.h"
    #include "def.h"
    int yycolumn = 1;
    #define YY_USER_ACTION    yylloc.first_line=yylloc.last_line=yylineno; \
    yylloc.first_column=yycolumn; \
    yylloc.last_column=yycolumn+yylength-1; \
    yycolumn+=yylength;
    /*语义值定义*/
    typedef union{
        int type_int;
        float type_float;
        char type_char;
```

```

        char type_id[32];
        struct Node *ptr;
    }YYLVAL;
    #define YYSTYPE YYLVAL
%}
%option yylineno
/*辅助定义*/
id [A-Za-z][A-Za-z0-9]*
int [0-9]+
float ([0-9]*\.[0-9]+)|([0-9]+\.)
char \"[0-9aa-zA-z]\"

```

## 2. 规则部分

规则部分列出了语法规则中所有的终结符（词法单元）的匹配模式，即用于匹配的正则表达式和相应的响应动作。其格式为 `pattern { action }`，其中，`pattern` 是需要匹配的内容，此处为语法规则中的非终结符或其相应的正则表达式；`action` 为匹配到相应字符后执行的内容。使用 “.” 匹配其他未列出的其他符号，用于错误处理。

利用 Flex 工具生成的词法分析器将按照该部分列出的模式依次进行匹配，尽可能匹配最长的输入串，识别出一个词法单元后，会将其相关信息记录在 Flex 提供的一些变量中：`yytext` 记录当前词法单元的内容，`yylineno` 和 `yycolumn` 分别记录了当前匹配的词法单元的行号和列号，`yylen` 记录了词法单元的长度，`yyval` 类型是上述定义的联合体 `YYLAL`，用于记录词法单元的属性值等等。此外这些值 Bison 也可以使用，因此也具有数据传递效果。

lex.l 中的模式匹配部分如下：

```

/* 规则部分：模式+动作 */
{int} {yylval.type_int=atoi(yytext);return INT;}
{float} {yylval.type_float=atof(yytext); return FLOAT;}
{char} {yylval.type_char=yytext[1];return CHAR;}
"int" {strcpy(yylval.type_id,yytext); return TYPE;}
"float" {strcpy(yylval.type_id,yytext); return TYPE;}
"char" {strcpy(yylval.type_id,yytext); return TYPE;}
"return" {return RETURN;}
"if" {return IF;}
"else" {return ELSE;}
"while" {return WHILE;}
"for" {return FOR;}
"break" {return BREAK;}
"continue" {return CONTINUE;}
"void" {return VOID;}
{id} {strcpy(yylval.type_id,yytext); return ID;}
";" {return SEMI;}
"," {return COMMA;}
">"|"<"|">="|"<="|"=="|!=" {strcpy(yylval.type_id,yytext); return RELOP;}

```

```

"=" {return ASSIGNOP;}
"+" {return PLUS;}
"-" {return MINUS;}
"+=" {return COMADD;}
"_" {return COMSUB;}
"*=" {return COMSTAR;}
"/=" {return COMDIV;}
"++" {return SELFADD;}
"--" {return SELFSUB;}
"*" {return STAR;}
"/" {return DIV;}
"&&" {return AND;}
"||" {return OR;}
"!" {return NOT;}
"(" {return LP;}
")" {return RP;}
"[" {return LB;}
"]" {return RB;}
"{" {return LC;}
"}" {return RC;}
[\n] {yycolumn=1;}
[ \r\t] {};
\\[^\n]*   {};          //匹配单行注释的正则表达式
\\*(s|.)?*\n {};        //匹配多行注释的正则表达式
.          {printf("Error:      Mysterious      character\""%s\"      at
line %d,column %d\n",yytext,yylineno,yycolumn);} //错误处理

```

### 3. C 代码部分

该部分是用户自定义函数的代码部分，最后会被 Flex 拷贝到词法分析器的 C 代码 lex.yy.c 中，以方便用户在词法分析器中执行自定义的函数。由于需要 Flex 工具结合 Bison 工具共同使用，此处的 main 函数则不需要执行，因此注释起来。随后跟着的 yywrap 函数为 Flex 工具需要自定义的函数，Flex 在解析一个文件完毕后会调用该函数，返回值 1 表示结束 0 表示继续。此处定义该函数返回 1 表示解析完一个文件后就结束词法分析。

lex.l 中的用户自定义代码部分如下：

```

/*
void main(int argc,char *argv[]) {
    yylex();
    return;
}
*/
int yywrap() {
    return 1;
}

```

## 2.4 语法分析器设计

语法分析器部分，通过编写 `parser.y` 文件结合 `Bison` 工具作为语法分析器，对 `MiniC` 语言进行语法分析。`Bison` 程序与 `Flex` 程序相同，由声明部分、规则部分和 `C` 代码部分三部分构成，每个部分同样使用 `%%` 划分。

### 1. 声明部分

声明部分也包含了会被拷贝到目标分析程序 `parser.tab.c` 开头的 `C` 代码，通过 `%{}` 括起，此处主要是包含了文件以下内容中调用的函数库包括 `"stdio.h"` `"string.h"` `"math.h"` 以及自定义函数的头文件 `"def.h"`。此外，使用 `"extern"` 来声明 `C` 代码中使用的外部变量 `yylineno`（词法单元行号）、`yytext`（词法单元内容）、`yyin`（字符输入文件指针），以及 `C` 代码中的函数声明 `yyerror`（错误处理）和 `displayAST`（输出抽象语法树的结点）。

除上述外，开头 `"%error-verbose"` 用于指示 `Bison` 生成错误信息，`"%location"` 用于记录行号。`"%union"` 声明了各种语义值类型，定义与 `lex.l` 文件声明部分的 `YYLVAL` 保持一致。

紧接着，是对语法中非终结符和终结符的语义值类型的定义。

`Bison` 中使用 `"%type"` 来定义非终结符语义值类型，其定义形式为：`%type <union 成员名> 非终结符`。其中，`union` 成员名的类型则表示了该规则符号的语义值类型，在语法分析时会将该符号的对应语义值存到其 `union` 变量的对应成员中。对于 `MiniC` 中非终结符的语义值类型，`union` 成员名均使用 `"ptr"`，其为 `"struct Node"` 类型的指针，表示在抽象语法树中的一个 `"Node"` 结点。

`Bison` 中使用 `"%token"` 来定义终结符语义值类型，其定义形式为：`%token <union 成员名> 终结符`。对于 `MiniC` 中终结符的语义值类型，根据终结符的分类，`char`、`int`、`float` 的数值常量的语义值类型对应 `union` 中的 `"type_char"` `"type_int"` `"type_float"`；标识符 `ID`、比较运算符 `RELOP` 和变量类型 `TYPE` 对应 `union` 中的 `"type_id"`，其值以字符串形式存到 `"type_id"` 中。而对于其他非终结符，如 `"(" "+"` `"while"` 等符号或关键字，则不具体指定类型，以默认的 `int` 类型。

随后使用 `"%left"` 或 `"%right"` 来定义终结符中运算符的结合性，且排在前面行的运算符优先级低、排在后面行运算符优先级高；“使用 `%nonassoc`”来定义无结合性的符号。

`parser.y` 中声明部分如下：

```
%error-verbose //指示 bison 生成错误信息
%locations      //记录行号

//声明部分
%{
#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 displayAST(struct Node* , int);
%}

//声明各种类型, 默认为 int
%union {
    int type_int;
    float type_float;
    char type_char;
    char type_id[32];
    struct Node* ptr;
};

//%type 指定非终结符的语义值类型 %type <union 的成员名> 非终结符名
%type<ptr> Program ExtDefList ExtDef Specifier ExtDecList FuncDec ArrayDec
CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp
ExpList
//%token 指定终结符的语义值类型
%token<type_int> INT
%token<type_id> ID RELOP TYPE
%token<type_float> FLOAT
%token<type_char> CHAR
%token LP RP LC RC SEMI COMMA LB RB
%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE
RETURN COMADD COMSUB FOR COMSTAR COMDIV BREAK CONTINUE
FOR_E RETURN_E VOID
//结合性, 优先级由低到高
%left COMADD COMSUB COMSTAR COMDIV
%left ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT SELFADD SELFSUB
//无结合性
%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE

```

## 2. 规则部分

规则部分包含了通过简单的 BNF 定义的语法规则, 即产生式。其中 Bison 使用 “:” 来表示 “生成”, 使用 “;” 来表示规则的结束, 规则后面使用大括号 “{}” 括起的语句为该语句归约时需要执行的语义动作。

具体的文法规则的代码和相关说明已在 “语法的文法描述部分” 的 “定义语法规则” 处进行过详细描述, 在此不再赘述。



### 3. C 代码部分

该部分也是用户自定义的函数代码，会被 Bison 拷贝到 parser.tab.c 文件中,该部分函数使用的变量及函数声明需要在前面声明部分声明。此处主要定义了编译器的主函数 main，主要功能就是调用 Flex 提供的 yyparse 函数进行分析。其次是 yyerror 函数，用于进行错误处理。

parser.y 中 C 代码部分如下：

```
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: ",
    yyloc.first_line,yyloc.first_column);
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, ".\n");
}
```

### 4. 抽象语法树的显示及建立

除了上述编写的 Bison 程序 parser.y 之外，在语法分析阶段，要生成并显示抽象语法树，该部分代码在文件 def.h 和 ast.c 中。

抽象语法树将词法分析之后得到的词法单元按照文法以相应规则提取出来，利用二叉树的数据结构将这些词法单元组织起来，并记录相关的属性值。

MiniC 中抽象语法树的结点定义如下：

```
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 NodeKind kind; //结点类型
    union {
        char type_id[33]; //由标识符生成的叶结点
        int type_int; //由整常数生成的叶结点
        char type_char; //由字符型生成的叶节点
        float type_float; //由浮点常数生成的叶结点
    };
    struct Node *ptr[3]; //子树指针,由 kind 确定有多少棵子树
    int level; //层号
};
```

```

int place; //表示结点对应的变量或运算结果临时变量在符号表的位置序号
char eTrue[15],eFalse[15]; //对布尔表达式的翻译时,真假转移目标的标号
char sNext[15]; //该结点对应语句执行后的下一条语句位置标号
struct CodeNode *code; //该结点中间代码链表头指针
char op[10];
int type; //结点对应值的类型
int pos; //语法单位所在位置行号
int offset; //偏移量
int width; //各种数据占用的字节数
};

```

抽象语法树的结点建立利用函数 `mknode`。该函数在 Bison 程序 `parser.y` 中语法规则中频繁出现,每当编译器语法分析得到对应的规则时,就会利用该函数在抽象语法树的对应位置建立一个结点。其中使用“...”来传递不定长参数。

`mknode` 函数具体代码如下:

```

struct Node *mknode(int kind,int pos,int num,...) {
    struct Node *T = (struct Node*)malloc(sizeof(struct Node));
    int i=0;
    T->kind = kind; //类型
    T->pos = pos;    //位置
    va_list pArgs;  //变长参数
    va_start(pArgs,num);
    for(i=0;i<num;++i) T->ptr[i]=va_arg(pArgs, struct Node *);
    while(i<4) T->ptr[i++]=NULL;
    va_end(pArgs);
    return T;
}

```

显示抽象语法树使用函数 `displayAST`。该函数为语法规则中第一条语句归约时调用,即在整个程序完成分析和抽象语法树建立后,以先根遍历整个抽象语法树,并对其结点的信息进行输出。

`displayAST` 函数具体代码如下:

```

void displayAST(struct Node* T,int indent) {
    if(T){
        switch (T->kind){
            case EXT_DEF_LIST: //外部定义列表
                displayAST(T->ptr[0],indent); //显示该外部定义列表中的第
一个
                displayAST(T->ptr[1],indent); //显示外部定义列表中的其它
外部定义
                break;
            case EXT_VAR_DEF: //外部变量定义
                printf("%*c%s\n",indent,'',"外部变量定义: ");
                displayAST(T->ptr[0],indent+3); //显示外部变量类型
                printf("%*c%s\n",indent+3,'',"变量名: ");
                displayAST(T->ptr[1],indent+3); //显示变量列表

```

```

        break;
case FUNC_DEF: //函数定义语句
    printf("%*c%s\n",indent,'',"函数定义: ");
    displayAST(T->ptr[0],indent+3); //显示函数返回类型
    displayAST(T->ptr[1],indent+3); //显示函数名和参数
    displayAST(T->ptr[2],indent+3); //显示函数体
    break;
case ARRAY_DEF://数组定义语句
    printf("%*c%s\n",indent,'',"数组定义: ");
    displayAST(T->ptr[0],indent+3);
    displayAST(T->ptr[1],indent+3);
    break;
case FUNC_DEC: //函数定义
    printf("%*c%s%s\n",indent,'',"函数名: ",T->type_id);
    if(T->ptr[0]){
        printf("%*c%s\n",indent,'',"函数形参: ");//显示函数参数列表
        displayAST(T->ptr[0],indent+3);
    }
    else printf("%*c%s\n",indent,'',"无形参");
    break;
case ARRAY_DEC: //数组定义
    printf("%*c%s%s\n",indent,'',"数组名: ",T->type_id);
    printf("%*c%s\n",indent,'',"数组大小: ");
    displayAST(T->ptr[0],indent+3);
    break;
case EXT_DEC_LIST: //变量名列表
    displayAST(T->ptr[0],indent+3); //依次显示外部变量名
    if(T->ptr[1]->ptr[0]==NULL)//后续还有相同的，仅显示语法树
        displayAST(T->ptr[1],indent+3);
    else
        displayAST(T->ptr[1],indent);
    break;
case PARAM_LIST: //函数入口参数列表
    displayAST(T->ptr[0],indent);//依次显示全部参数类型和名称
    displayAST(T->ptr[1],indent);
    break;
case PARAM_DEC: //函数参数定义
    displayAST(T->ptr[0],indent);
    displayAST(T->ptr[1],indent);
    break;
case VAR_DEF: //变量定义
    displayAST(T->ptr[0],indent+3); //显示变量类型
    displayAST(T->ptr[1],indent+3); //显示该定义的全部变量名

```

此处

```

        break;
case DEC_LIST: //变量列表
    printf("%*c%s\n",indent,'',"变量名: ");
    displayAST(T->ptr[0],indent+3);
    displayAST(T->ptr[1],indent);
    break;
case DEF_LIST: //定义列表
    printf("%*c%s\n",indent+3,'',"局部变量名: ");
    while(T){
        displayAST(T->ptr[0],indent+3);
        T=T->ptr[1];
    }
    break;
case COMP_STM: //复合语句
    printf("%*c%s\n",indent,'',"复合语句: ");
    printf("%*c%s\n",indent+3,'',"复合语句的变量定义: ");
    displayAST(T->ptr[0],indent+3); //显示定义部分
    printf("%*c%s\n",indent+3,'',"复合语句的语句部分: ");
    displayAST(T->ptr[1],indent+3); //显示语句部分
    break;
case STM_LIST: //语句列表
    displayAST(T->ptr[0],indent+3); //显示第一条语句
    displayAST(T->ptr[1],indent); //显示剩下语句
    break;
case EXP_STMT: //表达式
    printf("%*c%s\n",indent,'',"表达式语句: ");
    displayAST(T->ptr[0],indent+3);
    break;
case IF_THEN: //if 语句
    printf("%*c%s\n",indent,'',"if 条件语句: ");
    printf("%*c%s\n",indent,'',"条件: ");
    displayAST(T->ptr[0],indent+3); //显示条件
    printf("%*c%s\n",indent,'',"if 子句: ");
    displayAST(T->ptr[1],indent+3); //显示 if 子句
    break;
case IF_THEN_ELSE: //if-else if 语句
    printf("%*c%s\n",indent,'',"if-else 条件语句: ");
    printf("%*c%s\n",indent,'',"条件: ");
    displayAST(T->ptr[0],indent+3); //显示条件
    printf("%*c%s\n",indent,'',"if 子句: ");
    displayAST(T->ptr[1],indent+3);
    printf("%*c%s\n",indent,'',"else 子句: ");
    displayAST(T->ptr[2],indent+3);
    break;

```

```

case WHILE: //while 语句
    printf("%*c%s\n",indent,'',"while 循环语句: ");
    printf("%*c%s\n",indent+3,'',"循环条件: ");
    displayAST(T->ptr[0],indent+3);    //显示循环条件
    printf("%*c%s\n",indent+3,'',"循环体: ");
    displayAST(T->ptr[1],indent+3);    //显示循环体
    break;
case FOR: //for 语句
    printf("%*c%s\n",indent,'',"for 循环语句: ");
    printf("%*c%s\n",indent+3,'',"表达式 1: ");
    displayAST(T->ptr[0],indent+3);
    printf("%*c%s\n",indent+3,'',"表达式 2: ");
    displayAST(T->ptr[1],indent+3);
    printf("%*c%s\n",indent+3,'',"表达式 3: ");
    displayAST(T->ptr[2],indent+3);
    printf("%*c%s\n",indent+3,'',"循环体: ");
    displayAST(T->ptr[3],indent+3);
    break;
case FOR_E:
    printf("%*c%s\n",indent,'',"for 循环语句: ");
    printf("%*c%s\n",indent+3,'',"循环条件: ");
    displayAST(T->ptr[0],indent+3);
    printf("%*c%s\n",indent+3,'',"循环体: ");
    displayAST(T->ptr[1],indent+3);
    break;
case FUNC_CALL:    //函数调用语句
    printf("%*c%s\n",indent,'',"函数调用: ");
    printf("%*c%s%s\n",indent+3,'',"函数名: ",T->type_id);
    printf("%*c%s\n",indent+3,'',"实参: ");
    displayAST(T->ptr[0],indent+3);
    break;
case EXP_LIST:    //实参列表
    while(T){
        displayAST(T->ptr[0],indent+3);
        T=T->ptr[1];
    }
    break;
case ID:    //标识符
    printf("%*c 标识符:  %s\n",indent,' ',T->type_id);//控制新的一
行输出的空格数, indent 代替%*c 中*
    break;
case INT:
    printf("%*cINT:  %d\n",indent,' ',T->type_int);
    break;

```

```

case FLOAT:
    printf("%*cFLOAT:  %f\n",indent,' ',T->type_float);
    break;
case CHAR:
    printf("%*cCHAR:  %c\n",indent,' ',T->type_char);
case ARRAY:
    printf("%*c 数组名称:  %s\n",indent,' ',T->type_id);
    break;
case TYPE:
    if(T->type==INT)
        printf("%*c%s\n",indent,' ','类型:  int");
    else if(T->type==FLOAT)
        printf("%*c%s\n",indent,' ','类型:  float");
    else if(T->type==CHAR)
        printf("%*c%s\n",indent,' ','类型:  char");
    break;
case ASSIGNOP:
case OR:
case SELFADD_L:
case SELFSUB_L:
case SELFADD_R:
case SELFSUB_R:
case AND:
case RELOP:
case PLUS:
case MINUS:
case STAR:
case DIV:
case COMADD:
case COMSUB:
case COMSTAR:
case COMDIV:
    printf("%*c%s\n",indent,' ',T->type_id);
    displayAST(T->ptr[0],indent+3);
    displayAST(T->ptr[1],indent+3);
    break;
case NOT:
case UMINUS:
    printf("%*c%s\n",indent,' ',T->type_id);
    displayAST(T->ptr[0],indent+3);
    break;
case RETURN:
    printf("%*c%s\n",indent,' ','return 语句:  ");
    displayAST(T->ptr[0],indent+3);

```

```

        break;
    case BREAK:
    case CONTINUE:
    case RETURN_E:
    case VOID:
        printf("%*c%s\n",indent,' ',T->type_id);
        break;
    }
}
}

```

## 2.5 词法及语法分析器运行截图

联合使用 Flex 和 Bison 构造词法分析器和语法分析器,构建以下 Makefile 文件用于构造可执行文件。

```

miniC: lex.l parser.y ast.c def.h
    bison -d -v parser.y
    flex lex.l
    cc -o miniC.yy.c parser.tab.c ast.c

clean:
    rm -rf lex.yy.c
    rm -rf parser.tab.h
    rm -rf parser.tab.c
    rm -rf parser.output
    rm -rf miniC

```

定义用于 MiniC 分析的测试源文件 test.c--如下:

```

int a,b,c;
float m,n;
char c1,c2;//增加 char 类型
char a[10];//增加数组的定义
int fibo(int a)
/*注释部分自动去掉*/
{
    int i;
    if(a == 1 || a == 2){
        return 1;
    }
    return fibo(a-1)+fibo(a-2);
}
void add(int a,int b) //增加了 void 函数类型
{
    int i=a+b;
    return; //增加了无返回值 return
}

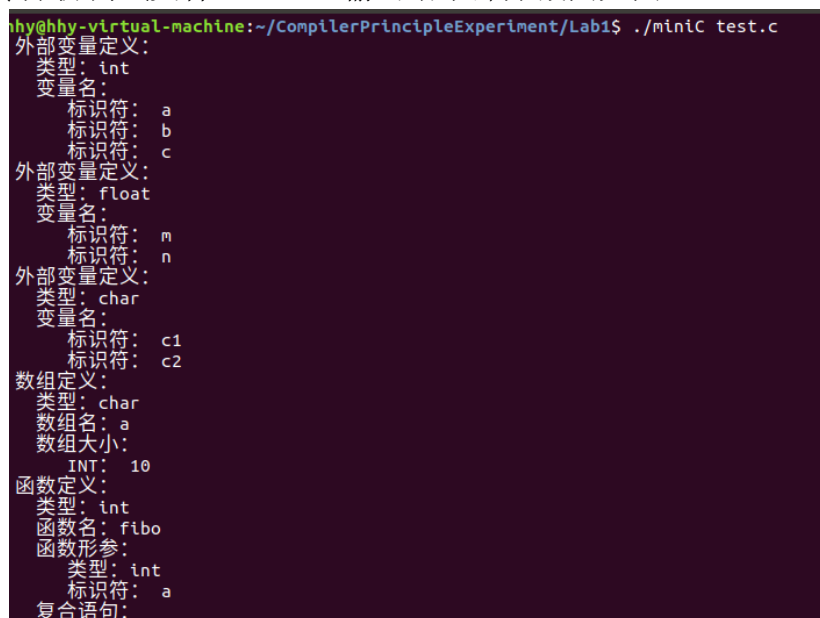
```

```

int main()//注释部分自动去掉
{
    int m,n,i;
    char c;
    float ar[20];
    c='a';
    i++;
    i+=1;
    for(i=0,m=1;i<2;i++){    //增加了 for 循环
        m*=5;
        break; //增加了 break 语句
    }
    i=-m;
    add(i,m);
    --i;//增加了自增和自减
    m+=i+15;//增加了复合赋值运算
    while(i <= m){
        n=fibo(i);
        i=i+1;
    }
    return 1;
}

```

在终端使用“make”指令编译 MiniC 语法词法分析器，然后使用“./miniC test.c--”指令分析测试文件 test.c--，输出结果部分截图如图 2-1 至 2-3。



```

hhy@hhy-virtual-machine:~/CompilerPrincipleExperiment/Lab1$ ./miniC test.c
外部变量定义:
  类型: int
  变量名:
    标识符: a
    标识符: b
    标识符: c
外部变量定义:
  类型: float
  变量名:
    标识符: m
    标识符: n
外部变量定义:
  类型: char
  变量名:
    标识符: c1
    标识符: c2
数组定义:
  类型: char
  数组名: a
  数组大小:
    INT: 10
函数定义:
  类型: int
  函数名: fibo
  函数形参:
    类型: int
    标识符: a
复合语句:

```

图 2-1 语法及法分析器运行结果截图 1



```

复合语句的变量定义:
  局部变量名:
    类型: int
    变量名:
      标识符: i
复合语句的语句部分:
  if条件语句:
    条件:
      OR
      ==
      标识符: a
      INT: 1
      ==
      标识符: a
      INT: 2
  if子句:
    复合语句:
      复合语句的变量定义:
      复合语句的语句部分:
        return语句:
          INT: 1
    return语句:
      PLUS
      函数调用:
        函数名: fibo
        实参:
          MINUS
          标识符: a
          INT: 1

```

图 2-2 语法及法分析器运行结果截图 2

```

COMADD
  标识符: m
PLUS
  标识符: i
  INT: 15
while循环语句:
  循环条件:
    <=
    标识符: i
    标识符: m
  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:
          ASSIGNOP
          标识符: n
          函数调用:
            函数名: fibo
            实参:
              标识符: i
        表达式语句:
          ASSIGNOP
          标识符: i
          PLUS
          标识符: i
          INT: 1
    return语句:
      INT: 1

```

图 2-3 语法及法分析器运行结果截图 3

## 2.6 小结

本次实验完成了对 MiniC 语言词法分析器和语法分析器的设计和构建。学习使用 Flex 工具和 Bison 工具编写相应的程序来构建词法、语法分析器，并对其语法有了一定的认识。同时在实现过程中添加了新的语法规则，如 char 变量、数组、自增、for 循环等，更深入的了解对于类 C 语言如何构造其相应文法。也学习了如何构建抽象语法树，并利用树结点存储相应的词法单元的属性值。

## 3 实验二 语义分析

实验二任务：

- 1、设计符号表存储结构；
- 2、修改语法分析程序中的语义子程序，增加合适的语义信息，填写符号表信息；
- 3、根据符号表信息，能完成静态语义检查，并对样例程序进行分析，识别语义错误；参考文献[1]2.1.6 必做样例；
- 4、可以根据需要输出符号表信息，其中包含符号的类型、作用域层次等信息。

### 3.1 语义子程序描述

对源程序进行词法分析和语法分析后，若源程序没有词法错误和语法错误，便可以建立出程序对应的抽象语法树。在建立语法树的过程值，即可同时进行语义分析。

具体来讲，语法分析构建抽象语法树的同时，会进行语义分析：提取程序中的符号来建立符号表，同时设定该符号对应的相关属性，如符号名、别名、类型、层次、符号表中的位置、所占字节数等。对于不同的符号（如变量名、函数名、常量等）会有不同的属性。而在构建语法树的过程中，符号的属性也会在抽象语法树结点对应的符号间进行传递，用于确定其他符号相应的属性。

与此同时，在进行语义分析的过程中，也会根据语义规则检查源程序，按照定义的语义错误类型对源程序进行识别，将发现的语义错误进行输出。

在程序进行语义分析后，便能得到整个源程序在不同层次的符号表，用于后续的中间代码生成。

### 3.2 符号表的设计

编译的过程中，符号表用于记录源程序中出现的各种符号及其相关属性，以方便对程序中的类型检查、分析等。因此，在设计符号表之前，要先对“符号”进行定义，确定符号表中的符号所具有的属性。在 MiniC 中，对于一个符号记录包括符号名、符号的层号、类型、形参个数、别名、符号标记和偏移量等属性。其中，符号的层号用于确定符号是外部变量或函数还是函数内部或复合语句中定义的变量；形参个数是针对函数符号设置的属性；别名是用于解决不同层次中符号同名而使用的，确保每个符号对应的别名唯一；符号标记用于区分函数、变量、函数形参、临时变量等不同的符号，偏移量用来记录变量在其静态数据区或活动记录的偏移量，或者函数的活动记录大小。具体定义如下：

```
//符号结构体
struct Symbol {
    char name[33]; //变量或函数名
    int level;     //层号
    int type;      //变量类型或函数返回值类型
    int paramnum;  //形参个数
    char alias[10]; //别名
```

```

char flag;      //符号标记
char offset;    //变量的偏移量 或 函数活动记录大小
};

```

定义好“符号”结构体后，即可对“符号表”的结构进行定义。对于符号表，在 MiniC 中采用了较为简单的单表组织结构，使用一个数组实现的顺序栈结构来存储符号，并使用一个变量 `index` 来作为指向栈顶的指针。每当有一个新的符号出现时，便将该符号及其属性压入栈中。

此外，每到达一个新的作用域，比如进入函数体或者进入复合语句，都要有一个相对独立的符号表来存储该作用域层次的可以访问符号，而离开该作用域的时候将其中的符号删除，用于实现符号在不同作用域中不同的可见性。此处借助一个“作用域栈”来实现，每当进入一个新的作用域，会将当前符号表的栈顶指针 `index` 值压入到作用域栈中，而新作用域的符号的偏移值属性以 `index` 为基准从 0 开始开始重新设定；而当离开该作用域的时候，会将存储到作用域栈中的 `index` 值出栈重新赋值给符号表的栈顶指针，这样栈顶指针就恢复到了进入该作用域前的位置，使得在作用域中创建的符号从符号表中出栈，从而消除其可见性。MiniC 中符号表和作用域栈的定义如下：

```

//符号表,是一个顺序栈,index 初值为 0
struct SymbolTable {
    struct Symbol symbols[MAXLENGTH];
    int index; //符号表索引
} symbolTable;

//作用域栈
struct SymbolScopeArray {
    int scopeArray[30];
    int top;
} symbolScopeStack;

```

### 3.3 语义错误类型定义

在构建符号表的过程中，通过符号的属性可对源程序进行语义层面的检查。通过定义错误类型从而对语言的语义规则加以限制，源程序中不满足语言语义规则的语句将会触发语义错误。

在 MiniC 中，根据文法的定义和实际需求定义了如下语义错误类型：

- (1) 变量出现重复定义
- (2) 函数形参名重复定义
- (3) 函数缺少返回值
- (4) 返回值与函数类型不匹配或返回值不能为空
- (4) 不在循环体内部，不能使用 `break` 或 `continue` 语句
- (5) 变量未定义或类型不匹配
- (6) 赋值号左边只有一个右值表达式，而赋值变量需要左值
- (7) 赋值语句的赋值号两边表达式类型不匹配
- (8) 不支持 `CHAR` 类型算术计算
- (9) 操作数左右值不匹配

- (10) 函数在使用时未定义或标识符不为函数
- (11) 函数调用时实参与形参数目或类型不匹配

### 3.4 语义分析实现技术

语义分析在语法分析的过程中同时进行，在生成抽象语法树的过程中，通过语法规则确定需要提取的符号以及符号的相关属性，并将符号加入到符号表中；同时使用作用域栈来控制符号在不同作用域中的可见性。

在 MiniC 中，使用以 `semanticAnalysis()` 为代表的一系列函数完成源程序的语义分析及符号表的生成。`semanticAnalysis()` 的函数主体是一个 `switch` 语句，根据抽象语法树结点的类型 (`kind`) 及结点的其它属性 (包括 `type_id`、`type` 等) 来提取符号及其属性信息，并根据当前符号表的状态获取结点新的属性 (如 `width`、`offset` 等)。

而在遍历抽象语法树的过程中，结点间的属性会进行传递：比如在变量定义的相关结点中，“类型”终结符结点的 `type` 属性会向其非终结符父结点传递，而父结点又会将 `type` 类型传递到其他的变量结点；又比如非终结符结点的宽度 (`width`) 属性即为其全部叶子结点 `width` 属性之和再加上其本身在语义执行过程中需要的宽度大小；再比如结点的偏移 (`offset`) 属性是表示符号相对作用域的起始位置的偏移量，在同一作用域里是逐渐增加的，每遇到一个需要记录的符号 (包括变量、函数名等标识符以及运算过程中的临时变量)，偏移量都会加上生成符号的宽度。

根据以上规则，在变量的过程中即可完成符号表的生成。由于结点属性的判断分析比较复杂，因此在 Mini 的源程序中对这部分进行了划分，对于不同的抽象语法树结点类型，其对应的操作均封装成了独立的函数，如 `ext_def_list()`、`ext_var_def()`、`fun_def()` 等放置在了 C 文件 `semanticCases.c` 中，而对于一些规则较为接近的表达式结点，如 `ID`、`ASSIGNOP`、`PLUS`、`FUNC_CALL` 等则统一封装进函数 `Exp()` 进行处理。从而使得函数 `semanticAnalysis()` 的主体部分不至于过于冗长。

而根据抽象语法树的结点对符号表的操作，主要是通过函数 `fillSymbolTable()`、`fillTemp()` 等来实现的：

`fillSymbolTable()` 是在语义分析遇到符号时调用，首先判断符号是否已经在同作用域的符号表中 (因为同一作用域符号不能重复定义)，然后再将新的符号及其相关属性添加到符号表中。

`fillTemp()` 是生成临时变量并加入到符号表中，对于表达式的相关运算，如加减乘除、取负、自增等运算，计算的中间结果均以临时变量的形式加入到符号表中，以方便后续使用；值得一提的是，由于 MiniC 要求变量的定义与运算要分离，因此在变量的运算语句部分采用了顺序结构共享单元的方式存储临时变量，即临时变量的作用域仅在当前语句，因此对下一计算语句进行语义分析时生成的变量可以覆盖上一语句的临时变量，从而节省符号表的空间。

此外使用了 `newAlias()` 函数来顺序生成符号的别名，`newTemp()` 函数为生成的临时变量命名等等。当语义分析遇到定义的语义错误时，也会调用函数 `semanticError()` 对错误信息进行输出，并中断当前结点的语义分析。

通过以上函数为主体的语义分析程序最终即可完成源程序的语义分析及符号表的构建。

### 3.5 语义分析结果

使用两个源文件对 MiniC 的语义分析进行测试。test1.c—源文件用于测试定义的语义错误类型，其代码如下，可以看到其中定义了 MiniC 中可以检测到的绝大部分语义错误。

```
int func(int i, float i) {    //函数形参名重复定义
    return;                  //返回值不能为空
}
int func(){                  //函数重复定义
    return 5;
}
void func1(){
    break;                  //不在循环体内部不能使用 break
}
int main() {
    int i = 0;
    int i;                  //变量重复定义
    float k;
    char c;
    c = -'a'; //CHAR 类型不能进行算术计算
    j = i + 1; //变量未定义
    inc(i);                //函数未定义
    i = 3.7; //赋值号两边表达式类型不匹配
    10 = i;                //赋值号左边需要左值
    10 + k;                //操作数左右值不匹配
    func(1, 2); //函数调用实参与形参数目不匹配
    func(j); //函数调用实参与形参类型不匹配
    return k; //返回值与函数类型不匹配
}
```

test1.c--运行结果如下图 3-1 和 3-2 所示，可以看到源程序的相应语法错误均被检测了出来。

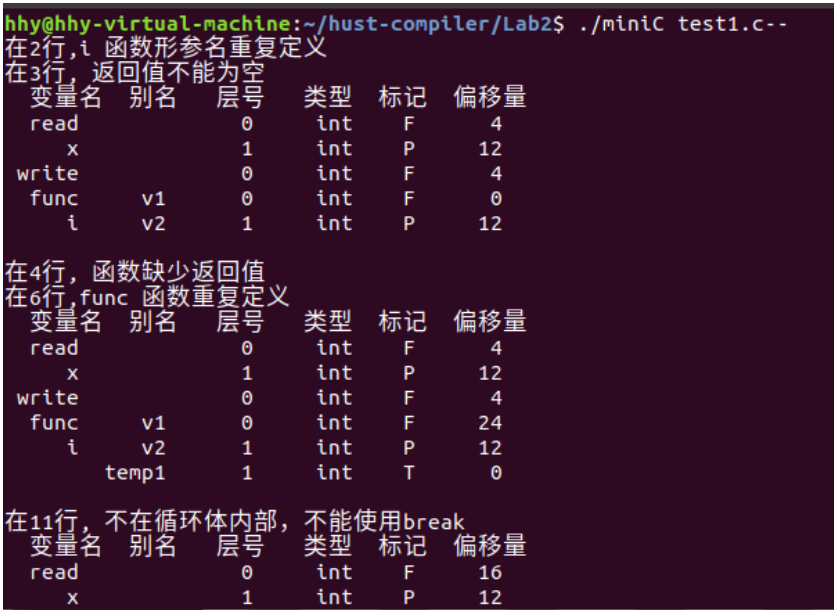


图 3-1 语义错误测试 1

write		0	int	F	4
func	v1	0	int	F	24
i	v2	1	int	P	12
func1	v5	0	void	F	0

在16行,i 变量重复定义  
 在19行, 不支持CHAR类型算数计算  
 在20行,j 变量未定义  
 在20行, 赋值语句类型不匹配  
 在21行,inc 函数未定义  
 在22行, 赋值语句类型不匹配  
 在23行, 赋值语句需要左值  
 在26行,j 变量未定义  
 在26行, 函数参数类型不匹配  
 在27行, 返回值类型错误

变量名	别名	层号	类型	标记	偏移量
read		0	int	F	16
x		1	int	P	12
write		0	int	F	4
func	v1	0	int	F	24
i	v2	1	int	P	12
func1	v5	0	void	F	12
main	v6	0	int	F	0
i	v7	1	int	V	12

图 3-2 语义错误测试 2

test2.c—源文件用于检测正常的语义分析和符号表的生成，其具体代码如下所示：

```
int a,b,c;
float m,n;
int add(int i) {
    return i;
}
int main() {
    int m, n, i = 1;
    char ch;
    float f;
    m = 10;
    n = add(i);
    while(i <= m) i++;
    return 0;
}
```

test2.c—源文件运行结果如下图所示，可以看到每进入一个新的作用域，符号的偏移量便会从 0 开始计算（函数的起始偏移量为 12 用于存放活动记录），不同的符号的类型会占用不同的大小，会影响后续符号的偏移量值，不同的符号也对应着不同的标记（F 为函数，P 为函数参数，V 为变量，T 为临时变量），不同作用域的符号会有着不同的层次。而由于采用顺序结构共享单元，后面的临时变量的偏移量均为 37。一共输出三个符号表，分别对应了函数 add、函数 main 和全局的三个符号表。

```
hhy@hhy-virtual-machine:~/hust-compiler/Lab2$ ./miniC test2.c--
```

变量名	别名	层号	类型	标记	偏移量
read		0	int	F	4
x		1	int	P	12
write		0	int	F	4
a	v1	0	int	V	0
b	v2	0	int	V	4
c	v3	0	int	V	8
m	v4	0	float	V	12
n	v5	0	float	V	20
add	v6	0	int	F	0
i	v7	1	int	P	12

图 3-3 生成的函数 add 符号表

变量名	别名	层号	类型	标记	偏移量
read		0	int	F	4
x		1	int	P	12
write		0	int	F	4
a	v1	0	int	V	0
b	v2	0	int	V	4
c	v3	0	int	V	8
m	v4	0	float	V	12
n	v5	0	float	V	20
add	v6	0	int	F	16
i	v7	1	int	P	12
main	v8	0	int	F	0
m	v9	1	int	V	12
n	v10	1	int	V	16
i	v11	1	int	V	20
	temp1	1	int	T	24
ch	v12	1	char	V	28
f	v13	1	float	V	29
	temp2	1	int	T	37
	temp3	1	int	T	37
	temp4	1	int	T	37
	temp5	1	int	T	37

图 3-4 生成的函数 main 的符号表

变量名	别名	层号	类型	标记	偏移量
read		0	int	F	4
x		1	int	P	12
write		0	int	F	4
a	v1	0	int	V	0
b	v2	0	int	V	4
c	v3	0	int	V	8
m	v4	0	float	V	12
n	v5	0	float	V	20
add	v6	0	int	F	16
i	v7	1	int	P	12
main	v8	0	int	F	41

图 3-5 生成的全局符号表

### 3.6 小结

本次实验相较于上次实验难度有了较大的提升，关键是要理解如何在变量抽象语法树的时候根据结点的属性信息提取符号的属性，其中如何在变量语法树的过

程中将属性正确的进行传递是关键而困难的。同时对于不同的作用域，要利用作用域栈配合符号表来达到符号表指针的快速复原。

在进行语义分析的过程中，也会发现初始定义的文法规则会对后续语义分析有着很大的影响。在实验一开始的时候为了使 MiniC 更加接近 C 语言，定义了诸如数组、for 循环等文法规则，但在实际进行语义分析生成符号表并考虑到后续的中代码生成时会较为困难，便将其进行了取舍，而对于无返回值的 `return`，`void` 类型函数，`break`、`continue` 语句等定义的文法则予以保留（实验一提交的报告为未删减文法的版本，即文法包含 for 循环等可输出正确语法树；而后续的三次实验则对上述文法进行了删除）。

该部分实验延续了上次实验的词法分析和语法分析，完成了对 MiniC 进行语义分析及语言错误检查和符号表生成的设计与构建。加深了对符号表的理解。



## 4 实验三 中间代码生成

实验三任务：

- 1、设计中间代码生成需要的数据结构；
- 2、完成对 AST 遍历并生成 TAC 序列，并输出；参考文献[1]3.1.6；

### 4.1 中间代码格式定义

MiniC 的中间代码采用三地址码 TAC。中间代码的格式定义如表 4-1 所示。

其中，对于一些复合赋值操作如赋值加(+=)、赋值减(-=)、自增(++)、取负(-)等操作，都会转换为赋值和加减乘除的基本运算的中间代码。

表 4-1 中间代码格式定义

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	ASSIGNOP	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
PARAM x	函数形参	PARAM			x
x:=CALL f CALL f	调用函数（有/无返回值）	CALL	f		x
ARG x	函数实参	ARG			x
READ x	读入	READ			x
WRITE x	打印	WRITE			x

## 4.2 中间代码生成规则定义

### 1. 基本表达式的中间代码生成规则

(1) 对于常数类型的表达式，包括 INT、FLOAT 和 CHAR 类型的常数，则需生成对应的数字或字符即可。

(2) 对于标识符 ID，则需生成该标识符（可能为变量名、函数名以及临时变量名）在符号表中记录的别名。

以上两种表达式基本上对应着中间代码生成过程中最小的代码生成单元，对应算术运算、循环语句等复杂的中间代码，都是基于以上两种符号的中间代码生成的。

(3) 对于赋值运算表达式  $\text{Exp ASSIGNOP Exp}$  和算术运算表达式（包括加减乘除），则按照表 4-1 中对应的中间代码进行生成，运算结果存到表达式对应的临时变量中。

(4) 对应符号赋值操作，包括加法赋值、减法赋值、乘法赋值和除法赋值， $\text{Exp1 Op Exp2}$ ，则将其转换成  $\text{Exp1} := \text{Exp1 Op Exp2}$  ( $\text{Op} \in \{+, -, *, /\}$ )，即转换成相应的加减乘除操作生成中间代码。

(5) 对于自增/自减操作  $\text{Exp SELFADD}$  或者  $\text{Exp SELFSUB}$ ，则转换为  $\text{Exp} := \text{Exp Op 1}$  ( $\text{Op} \in \{+, -\}$ )，即转换为加法或减法操作对应的中间代码、

(6) 对应取负操作， $\text{UMINUS Exp}$ ，则将其转换为  $\text{Exp1} := 0 - \text{Exp}$ ，即转换为 0 作被减数的减法操作，生成中间代码，其中  $\text{Exp1}$  为取负表达式对应的临时变量。

(7) 对于逻辑表达式，包括逻辑与、逻辑或、逻辑非以及比较运算，则会将其转换成对应的条件分支语句，其分支语句的真出口为表达式的值为 1 的赋值操作中间代码，假出口为表达式的值为 0 的赋值操作中间代码。

### 2. 语句的中间代码生成规则

(1) 对于表达式语句，则直接转换成相应的表达式进行中间代码生成，对于一些必要的语句，也需要在语句结尾生成标号。

(2) 对于复合语句，则直接对其中的语句进行中间代码生成。

(3) 对于返回语句，包括带返回值的 ( $\text{RETURN Exp};$ ) 和不带返回值的 ( $\text{RETURN};$ )，都会生成表 4-1 所示的 RETURN 中间代码。而对于带返回值的 RETURN 语句，则需要先对返回值的表达式生成中间代码，然后将返回值作为中间代码中的运算结果。

(4) 对于 if 条件语句，先要 if 的条件表达式进行 bool 表达式分析生成中间代码；然后生成 if 子句标号的中间代码，对应着 if 条件表达式的真出口；再进一步生成 if 子句的中间代码。

(5) 对于 if-else 条件语句，同样先要对 if 条件表达式进行 bool 表达式分析并生成中间代码；然后生成 if 条件表达式真出口对应的 if 子句标号的中间代码，接着对 if 子句进行中间代码生成；在 if 子句结束后需要生成一个跳到 if-else 语句出口标号的无条件跳转语句；接着生成 if 条件表达式假出口对应的 else 子句标号的中间代码；最后对 else 子句进行中间代码生成。

(6) 对于 while 循环语句，先要在循环条件表达式前生成循环条件标号，作为循环体的出口；然后对循环条件表达式进行 bool 表达式分析并生成中间代码；接着生成循环条件真出口对应的循环体标号的中间代码；然后对循环体进行中间代码生成；最后生成一条跳转到循环条件标号的中间代码。其中，在对 while 语句进

行语义分析时，会将循环条件的标号和循环语句出口（即循环条件的假出口）标号予以记录，用于 `continue` 语句和 `break` 语句的中间代码生成。

（7）对于 `break` 语句和 `continue` 语句，这两个语句仅在 `while` 语句的循环体中使用。`break` 语句会生成跳转到 `while` 循环体出口的标号的无条件跳转中间代码，而 `continue` 语句会生成跳转到 `while` 循环条件的标号的无条件跳转中间代码。

### 3. 函数的中间代码生成规则

（1）函数定义：在 MiniC 中，输入函数 `read()` 和输出函数 `write()` 作为编译自带的函数会直接生成相应的目标代码，而不会生成中间代码。只会生成用户自己定义的函数的中间代码。首先对于自定义函数会生成定义函数的中间代码 `FUNCTION f`；然后对于函数的每个形参生成形参的中间代码 `PARAM x`；然后对函数体进行中间代码生成；最后在函数结尾生成一个函数出口标号的中间代码。

（2）函数调用：首先会对函数的实参生成目标代码 `ARG`，其中若实参是表达式则在此之前需要对表达式进行中间代码生成，对表达式的值在生成函数实参的中间代码；然后生成调用函数的中间代码，若函数有返回值则为 `x:=CALL f`，`x` 为函数返回值对应的临时变量，若无返回值（`void` 类型）则为 `CALL f`。

### 4.bool 表达式的中间代码生成规则

此处指的 `bool` 表达式即为 `if` 语句、`if-else` 语句和 `while` 语句中的条件表达式，这些表达式并非直接计算其值，而是设定表达式的真假出口，用于配合上面语句使用，在对逻辑运算符和比较运算符的计算中也采用了该方法。以下中间代码生成规则仅考虑表达式本身，而对于其子表达式应参照上述规则率先完成中间代码生成。

（1）对于常数类型，则判断其常数值是否为 0，若为 0 则生成跳转到语句假出口的中间代码，若为 1 则生成跳转到真出口的中间代码。

（2）对于标识符，即变量名，则依次生成一条条件转移中间代码：`IF x != 0 GOTO eTrue` 和一条无条件跳转中间代码：`GOTO eFalse`，即若变量值不为 0 则跳至真出口，变量为 0 则跳至假出口。

（3）对于比较运算表达式，则依次生成一条条件转移到真出口的中间代码：`IF x [relop] y GOTO eTrue`，和一条无条件跳至假出口的中间代码 `GOTO eFalse`。

（4）对于逻辑与表达式 `x AND y`，表达式 `x` 的真出口为表达式 `y`，需要在 `y` 前生成 `x` 真出口标号的中间代码，而 `x` 的假出口即为该语句的假出口。

（5）对于逻辑或表达式 `x OR y`，表达式 `x` 的假出口为表达式 `y`，需要在 `y` 前生成 `x` 假出口标号的中间代码，而 `x` 的真出口即为该语句的真出口。

（6）对于逻辑非表达式 `!x`，表达式 `x` 的真出口即为该语句的假出口，表达式 `x` 的假出口即为该语句的真出口，只需改变出口指向，无需额外生成中间代码。

## 4.3 中间代码生成过程

### 1. 相关结构体定义及说明

#### （1）操作数结构体 `Operand`

操作数结构体即表示中间代码中的一个操作数，其成员包括操作数的类型 `kind`（一般包括标识符 `ID`、整型常数 `INT` 等），和其对应的在抽象语法树中的结点类型 `type`、用于记录常量值的 `union`，变量的层号和符号表的偏移。具体结构如下：

```
struct Operand {  
    int kind; //标识操作数结点的类型(标识符 ID、常数 INT 等)
```

```

int type; //标识操作数的对应的语法树结点类型
union {
    int const_int; //整常数值,立即数
    float const_float; //浮点常数值,立即数
    char const_char; //字符常数值,立即数
    char id[33]; //变量或临时变量的别名或标号字符串
};
int level; //变量的层号,0 表示是全局变量,数据保存在静态数据区
int offset; //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
};

```

## (2) 中间代码 TAC 结点结构体 CodeNode

对于每一条中间代码,均以中间代码结点标识,结点中记录了中间代码的运算符种类以及 2 个操作数和运算结果。此外,对于同一抽象语法树结点的多条中间代码,其 TAC 的结点采用双向循环链表的数据结构进行连接,因此 CodeNode 的成员中还包含双向链表的前后指针。具体结构如下:

```

struct CodeNode { //采用双向循环链表存放中间语言代码
    int op; //TAC 代码的运算符种类
    struct Operand opn1, opn2, result; //2 个操作数和运算结果
    struct CodeNode *prior, *next; //双向链表前后指针
};

```

## (3) 抽象语法树中的相关成员

在第一次实验进行抽象语法树构建时定义了抽象语法树结点的结构体 Node (具体见报告 14 页),其中的的一些成员变量与中间代码生成有关:成员 place 记录该结点操作数在符号表中的位置序号;code 记录该语法树结点对于的中间代码链表的头指针;eTrue、eFalse 记录在 bool 表达式的真假出口的标号;sNext 继续该语法树结点的中间代码后的下一条中间代码的位置。

## 2.相关函数定义

genLabel()用于生成一个标号的中间代码, genGoto()用于生成一条无条件跳转 GOTO 的中间代码, genIR()用于生成一条中间代码(结点), merge()用于合并多条中间代码的双向循环链表。

## 3.生成过程

中间代码生成是在语义分析阶段完成的,在语义分析遍历每个抽象语法树结点的过程中,在完成符号的提取并加入到符号表后,接下来便进行该语法树结点的中间代码生成,具体的中间代码生成规则如上文所述,生成的方式为使用上述的相关函数。生成的中间代码的双向循环链表的头指针存放在语法树结点的 code 成员中,对于多条中间代码要使用 merge()函数合并为一个双向链表。

## 4.4 中间代码生成结果

使用两个源文件对 MiniC 的中间代码生成部分进行测试。

test1.c—源文件仅包含 main()函数,涉及一些简单的条件分支等语句,具体代码如下:

```

int main() {

```

```

int n;
n = read();
if(n > 0) write(1);
else if(n < 0) write(-1); //取负
else write(0);
return 0;
}

```

其生成的对应中间代码如图 4-1 所示，其中的变量名均以其在符号表中的别名表示，临时变量为 temp。

中间代码：

```

FUNCTION main :
  temp1 := CALL read
  v2 := temp1
  temp2 := #0
  IF v2 > temp2 GOTO label4
  GOTO label5
LABEL label4 :
  temp3 := #1
  ARG temp3
  CALL write
  GOTO label3
LABEL label5 :
  temp5 := #0
  IF v2 < temp5 GOTO label6
  GOTO label7
LABEL label6 :
  temp7 := #1
  temp6 := #0 - temp7
  ARG temp6
  CALL write
  GOTO label3
LABEL label7 :
  temp9 := #0
  ARG temp9
  CALL write
LABEL label3 :
  temp11 := #0
  RETURN temp11
LABEL label1 :

```

图 4-1 test1.c—生成的中间代码

test2.c—文件中定义了两个函数，涉及了加法赋值运算(+=)，while 循环语句、break 语句、自增运算(++ )以及逻辑非(!)运算等。其具体代码如下，生成的中间代码如图 4-2 和图 4-3 所示。

```

int fact(int n) {
  if(n == 1) return n;
  else return (n * fact(n-1));
}
int main(){
  int m, result;
  m = read();
  while(m <= 1) { //while 循环语句
    m += 2; //加法赋值运算
    if(m == 0) break; //break 语句
  }
  if(m > 1) result = fact(m);
}

```

```

else {
    m++; //自增运算
    result = !m; //取非运算
}
write(result);
return 0;
}

```

中间代码:

```

FUNCTION fact :
    PARAM v2
    temp1 := #1
    IF v2 == temp1 GOTO label2
    GOTO label3
LABEL label2 :
    RETURN v2
    GOTO label1
LABEL label3 :
    temp2 := #1
    temp3 := v2 - temp2
    ARG temp3
    temp4 := CALL fact
    temp5 := v2 * temp4
    RETURN temp5
LABEL label1 :

FUNCTION main :
    temp6 := CALL read
    v4 := temp6
LABEL label8 :
    temp7 := #1
    IF v4 <= temp7 GOTO label7
    GOTO label6
LABEL label7 :
    temp8 := #2
    v4 := v4 + temp8

```

图 4-2 test2.c—生成的中间代码 1

```

temp9 := #0
IF v4 == temp9 GOTO label10
GOTO label8
LABEL label10 :
    GOTO label6
    GOTO label8
LABEL label6 :
    temp10 := #1
    IF v4 > temp10 GOTO label12
    GOTO label13
LABEL label12 :
    ARG v4
    temp11 := CALL fact
    v5 := temp11
    GOTO label11
LABEL label13 :
    temp12 := v4
    v4 := temp12 + 1
    IF v4 == #0 GOTO label15
    temp13 := #0
    GOTO label16
LABEL label15 :
    temp13 := #1
LABEL label16 :
    v5 := temp13
LABEL label11 :
    ARG v5
    CALL write
    temp15 := #0
    RETURN temp15
LABEL label4 :

```

图 4-3 test2.c—生成的中间代码 2

## 4.5 小结

本次实验基于上一次实验的语义分析过程，由于每个语法树结点的中间代码生成是紧接着其符号生成的，因此该部分是耦合在语义分析过程的代码中的，主要是在符号表生成后根据当前结点的类型按照相应的规则生成对应的中间代码。该部分的一些简单表达式语句只需对照生成相应的中间代码即可，主要难点是对和逻辑有关的 `bool` 表达式、比较表达式和条件语句、循环语句的中间代码生成。需要根据理论课上的学习，对结点的真假出口等进行设置，在合适的位置生成相应的标号以及无条件跳转语句，从而使程序流跳转到指定运行位置。此外，对于符合赋值运算和自增自减运算也需要按照给定的中间代码格式将其拆分成多个中间代码。

实验中并没有考虑中间代码优化，因此对于一些循环语句和条件语句相互嵌套的情况，可能会生成一定的冗余，比如在上述图 4-3 中就有连续两条 `GOTO` 语句的情况，而第二条 `GOTO` 语句不可达，属于冗余的代码。

该部分实验完成了 MiniC 的中间代码生成部分，学习并实践了如何根据抽象语法树结点的相关属性生成对应的中间代码，加深了对该部分的理解。

## 5 实验四 目标代码生成

实验四任务：

- 1、将 TAC 指令序列转换成目标代码；
- 2、完成样例的分析转换；参考文献[1]4. 1. 6；
- 3、能在模拟器上运行并得到结果；例如 **SPIM Simulator**；

### 5.1 指令集选择

目标代码语言选定的为 MIPS32 指令序列。中间代码 TAC 指令和 MIPS32 指令的对应关系如表 5-1 所示。

表 5-1 中间代码与 MIPS32 指令对应关系

中间代码	MIPS32 指令
LABEL x :	x:
x := #k	li reg(x),k
x := y	move reg(x), reg(y)
x := y + z	add reg(x), reg(y) , reg(z)
x := y - z	sub reg(x), reg(y) , reg(z)
x := y * z	mult reg(y) , reg(z) mflo reg(x)
x := y / z	div reg(y) , reg(z) mflo reg(x)
x := y y := x + 1 (右自增)	sw reg(y), x addi reg(y), reg(x), 1
x := y y := x - 1 (右自减)	sw reg(y), x addi reg(x), reg(x), -1
x := 0 - z	li reg(y), 0 sub reg(x), reg(y), reg(z)
GOTO x	j x
RETURN x	move \$v0, reg(x) jr \$ra
RETURN	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
CALL f	jal f

### 5.2 寄存器分配算法



MiniC 中寄存器分配算法采用的为朴素的寄存器分配算法，即每翻译一条中间代码时，都会先把该中间代码中涉及的变量从内存中加载到寄存器中，运算结束再将运算结果存回内存中。

该分配算法实现起来较为简单，基本上只需要在相应的运算指令前后加上存取指令（lw、sw）即可；不过寄存器的利用率过低，需要大量访问内存，而且使用的寄存器较少，部分寄存器一直处于闲置状态。在实际操作中，除了像栈顶指针存放在\$sp 寄存器、函数返回地址存放在\$ra 寄存器、函数返回值存放在\$vo 寄存器等按照 MIPS 规则使用的寄存器外，MiniC 在执行运算操作中仅使用了\$t1、\$t2 寄存器用于存放两个操作数、\$t3 寄存器存放运算结果。

### 5.3 目标代码生成算法

目标代码主要是根据抽象语法树结点 Node 结构体中 code 成员存放的中间代码 TAC 的双向循环链表进行生成的。通过判断每个中间代码的运算符种类（op），来转换成对应的 MIPS 指令。按照朴素寄存器分配算法进行的中间代码翻译如表 4-2 所示。

表 5-2 朴素寄存器分配下中间代码到目标代码的翻译

类型	中间代码	MIPS32 指令
赋值立即数	$x := \#k$	li \$t3,k sw \$t3, x 的偏移量(\$sp)
赋值变量	$x := y$	lw \$t1, y 的偏移量(\$sp) move \$t3,\$t1 sw \$t3, x 的偏移量(\$sp)
加法运算	$x := y + z$	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)
减法运算	$x := y - z$	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)
乘法运算	$x := y * z$	lw \$t1, y 的偏移量(\$sp) lw \$t2, z 的偏移量(\$sp) mult \$t1,\$t2 mflo \$t3 sw \$t3, x 的偏移量(\$sp)
除法运算	$x := y / z$	lw \$t1, y 的偏移量(\$sp) lw \$t2, z 的偏移量(\$sp) div \$t1,\$t2 mflo \$t3 sw \$t3, x 的偏移量(\$sp)
加法赋值	$x := x + y$	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) add \$t3 , \$t1, \$t2

		sw \$t3, x 的偏移量(\$sp)
减法赋值	$x := x - y$	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) sub \$t3, \$t1, \$t2 sw \$t3, x 的偏移量(\$sp)
乘法赋值	$x := x * y$	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) mult \$t1, \$t2 mflo \$t3 sw \$t3, x 的偏移量(\$sp)
除法赋值	$x := x / y$	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) div \$t1, \$t2 mflo \$t3 sw \$t3, x 的偏移量(\$sp)
右自增	$x := y$ $y := x + 1$	lw \$t1, y 的偏移量(\$sp) sw \$t1, x 的偏移量(\$sp) addi \$t1, \$t1, 1 sw \$t1, y 的偏移量(\$sp)
右自减	$x := y$ $x := x - 1$	lw \$t1, y 的偏移量(\$sp) sw \$t1, x 的偏移量(\$sp) addi \$t1, \$t1, -1 sw \$t1, y 的偏移量(\$sp)
取负操作	$x := 0 - z$	li \$t1, 0 lw \$t2, z 的偏移量(\$sp) sub \$t3, \$t1, \$t2 sw \$t3, x 的偏移量(\$sp)
无条件跳转	GOTO x	j x
函数返回	RETURN x RETURN	move \$v0, x 的偏移量(\$sp);有返回值时 jr \$ra
等于跳转	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
不等于跳转	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
大于跳转	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
大于等于跳转	IF $x \geq y$ GOTO z	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) bge \$t1, \$t2, z
小于跳转	IF $x < y$ GOTO z	lw \$t1, x 的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp)

		ble \$t1,\$t2,z
小于等于跳转	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
调用函数	x :=CALL f CALL f	move \$t0, \$sp addi \$sp,\$sp, -活动记录大小 sw \$ra, 0(\$sp) ;加载所有实参 lw \$t1, 实参的偏移量(\$t0) move \$t3, \$t1 sw \$t3, 形参的偏移量(\$sp) ... jal f lw \$ra, 0(\$sp) addi \$sp, \$sp, 活动记录大小 sw \$v0, x 的偏移量(\$sp) ;有返回值时

上表基本上罗列了全部 MiniC 中涉及的中间代码的对应翻译规则。值得一提的是，对应 read()和 write()函数，会作为系统自带函数，其目标代码会直接放置在编译后的目标代码头部。此外，对于函数，在定义部分的实参的中间代码 PARAM 不参与目标代码生成，而在调用部分时实参的中间代码 ARG 也不参与目标代码生成，而是在遇到中间代码 CALL 时将实参加载到函数定义的形参中去。

## 5.4 目标代码生成结果

使用三个源文件对 MiniC 的目标代码生成部分进行测试。

test1.c—和 test2.c—为实验要求中给定的测试文件，仅涉及一些简单语句，此处就不罗列具体的代码了。test1.c—生成的部分目标代码如图 5-1 所示，test2.c—生成的部分目标代码如图 5-2 所示，完整的输出见文件 test1.asm 和 test2.asm。

```
hhy@hhy-virtual-machine:~/hust-compiler/Lab4$ ./miniC test1.c--
.data
_Prompt: .asciiz "Enter an integer: "
_ret: .asciiz "\n"
.globl main
.text
li $t7,0x40
jal main
li $v0,10
syscall
read:
li $v0,4
la $a0,_Prompt
syscall
li $v0,5
syscall
jr $ra
write:
li $v0,1
syscall
li $v0,4
la $a0,_ret
syscall
move $v0,$0
jr $ra
```

图 5-1 test1.c—生成的部分目标代码

```

fact:
    li $t3,1
    sw $t3, 16($sp)
    lw $t1,12($sp)
    lw $t2,16($sp)
    beq $t1,$t2,label2
    j label3
label2:
    lw $v0,12($sp)
    jr $ra
    j label1
label3:
    li $t3,1
    sw $t3, 16($sp)
    lw $t1,12($sp)
    lw $t2,16($sp)
    sub $t3,$t1,$t2
    sw $t3,20($sp)
    move $t0,$sp
    addi $sp,$sp,-32
    sw $ra,0($sp)
    lw $t1,20($t0)
    move $t3,$t1
    sw $t3,12($sp)
    jal fact

```

图 5-2 test2.c—生成的部分目标代码

此外，使用了自定义的测试文件 test3.c—来测试程序中诸如加法赋值（+=）、自增、continue 语句、逻辑运算等较为复杂的运算的目标代码生成情况。test3.c—的代码具体如下：

```

int s;
int min(int a, int b){
    if(a < b) return a;
    return b;
}
int main() {
    int a, b, c = 5;
    int i = 1;
    a = read();    // 输入 8
    b = read();    // 输入 4
    b = min(b, c);
    s = 0;
    while(i <= a && i < 10){
        if(i == b) {
            s += -i;
            i += 1;
            continue;
        }
        s += i;
        i++;
    }
    b = 1 && 0;
    write(s);    // 输出 28

```

```
    write(b);    // 输出 0
    return 0;
}
```

test3.c—生成的部分中间代码如图 5-3 所示，完整输出见文件 test3.asm。

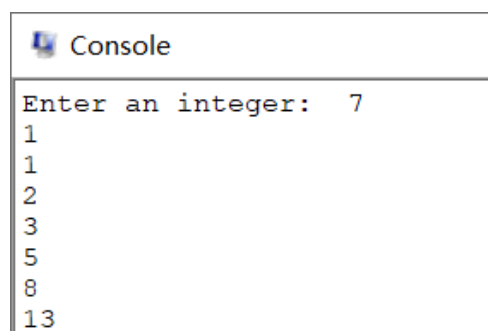
```
min:
    lw $t1,12($sp)
    lw $t2,16($sp)
    blt $t1,$t2,label3
    j label2
label3:
    lw $v0,12($sp)
    jr $ra
label2:
    lw $v0,16($sp)
    jr $ra
label1:

main:
    addi $sp,$sp,-32
    li $t3,5
    sw $t3, 24($sp)
    lw $t1,24($sp)
    move $t3,$t1
    sw $t3, 20($sp)
    li $t3,1
    sw $t3, 32($sp)
    lw $t1,32($sp)
    move $t3,$t1
    sw $t3, 28($sp)
```

图 5-3 test3.c—生成的部分目标代码

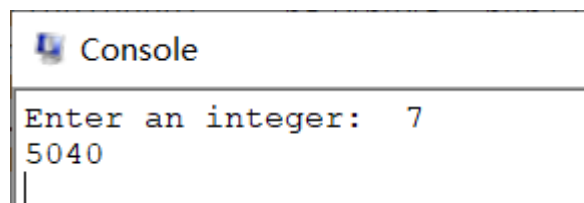
## 5.5 目标代码运行结果

使用 SPIM simulator 工具运行生成的目标代码：将目标代码（asm 文件）装入程序后，点击允许即可在控制台输入并查看输出。test1.c--、test2.c—和 test3.c—生成的目标代码的运行结果如图 5-4、图 5-5 和图 5-6 所示。



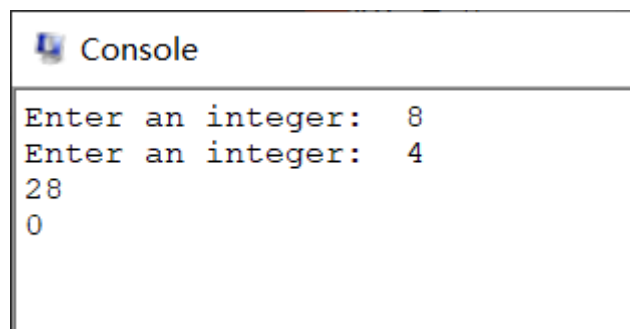
The screenshot shows the SPIM simulator's console window. The title bar says "Console". The text inside shows a prompt "Enter an integer:" followed by the number 7. Below this, a list of numbers is displayed: 1, 1, 2, 3, 5, 8, and 13, each on a new line.

图 5-4 test1.c—目标代码的运行结果



The screenshot shows the SPIM simulator's console window. The title bar says "Console". The text inside shows a prompt "Enter an integer:" followed by the number 7. Below this, the number 5040 is displayed on a new line.

图 5-5 test2.c—目标代码的运行结果



```
Console
Enter an integer: 8
Enter an integer: 4
28
0
```

图 5-5 test3.c-目标代码的运行结果

## 5.6 小结

本次实验基于上一次实验的生成的中间代码，并学习了解了 MIPS32 的指令集，通过建立中间代码和 MIPS 指令的对应关系，从而确定目标代码的生成规则。不过在此部分只涉及了简单的整数运算，并没有考虑之前语法中的 float 类型和 char 类型的目标代码生成。

在本部分实验中，也巩固学习了相关的寄存器分配算法，涉及朴素的寄存器分配方法、局部寄存器分配方法、以及利用待用信息和合约信息的寄存器分配方法。但由于时间等原因，在 MiniC 中选择了实现最为简单的朴素寄存器分配方法，实际上对于寄存器的分配仍有较大的改进空间。

在进行中间代码到目标代码的翻译过程中，比较困难的即为函数调用的翻译，需要了解 MIPS 关于函数的相关指令和寄存器，与中间代码也有所区别，对实参的中间代码并不进行翻译，而是才函数调用时将实参的值赋值给形参对应的变量，以达到参数传递效果，同时利用 MIPS 给定的 \$v0 和 \$ra 寄存器完成返回值的传递和返回地址的恢复。翻译的过程中，对变量的存取主要是靠操作数中的 offset 属性，而该属性关联着该操作数在符号表中的偏移量，需要保证之前符号表中计算的偏移量的准确性，才能保证目标代码能够正确执行。

此次试验后，能够生成了目标代码及运行，学习并实践了如何将目标代码进行处理，生成相应的目标代码。

## 6 总结

整个编译原理实验，完成了词法、语法分析建立抽象语法树，语义分析生成符号表，生成中间代码和生成目标代码四个部分。实现了一个简单的类 C 语言的编译器 MiniC。在整个过程中，也不断加深了我对编译原理理论课中学到的相关知识的理解，已经如何用代码进行具体实现。

实验中，我学习了解了如何使用 flex 和 bison 进行单词和文法的定义识别，并学习实践了抽象语法树的数据结构以及其结点的生成。在语义分析部分，学习实践了如何通过当前结点的文法属性来设置当前语法树结点的属性值，并提取相应的符号到符号表中，并了解了如何使用作用域栈来解决变量的作用域的问题。同时根据属性结点的数据可以对语言进行语义上的约束，进行语义层面的错误检查。在目标代码生成阶段，学习实践了根据结点的类型生成对应的三地址码，并结合理论课上的内容进行了 bool 表达式的控制流翻译。最后目标代码阶段学习了 MIPS 指令集以及寄存器分配算法，将中间代码转换为相应的可执行的 MIPS 指令。

整体而言，此次编译原理实验的总体重心是在符号表的生成以及中间代码和目标代码的生成部分，而这些部分在理论课上老师讲的并不详尽，而在实验过程中也是不断学习。同时，此次实验的工程量也比较大，刚开始的词法分析和语法分析阶段个人还想让自己的编译器更接近简单的 C 语言，还加入了 for 循环、数组等文法，但由于后续的实现，最后也进行了舍弃。此外，此次实验做起来也比较困难，个人也是花费了很多时间才查阅资料，以及阅读附录代码和 GitHub 相关编译器的代码上，提高了我的代码阅读和分析能力，最后能够整体理解整个编译器的运行原理，并最后也加入了自己设计的一些简单的运算等。不过这次实验的时间比较紧迫，临近期末考试以及同时有多门实验需要完成，MiniC 在部分地方仍有改进完善的空间，但整个实验还是比较充实，有很大收获。

## 参考文献

- [1] 许畅 等编著. 《编译原理实践与指导教程》.机械工业出版社
- [2] John Levine著, 陆军 译. 《Flex与Bison》.东南大学出版社
- [3] 王生原等编著. 《编译原理 (第3版)》.清华大学出版社