华中科技大学

课 程 实 验 报 告

课程名称：编 译 原 理 实 验

专业班级：CS1806

学　　号：U201814670

姓　　名：李田田

指导教师：万　瑶

报告日期：2021 年 7 月 2 日

计算机科学与技术学院

目录

1. 概述

1.1. 实验环境

本次实验采用 ubuntu 下的 flex 和 bison 进行编译器的设计和实现。

1.2. 任务

本次课程设计是构造一个高级语言的子集的编译器，目标代码可以是汇编语言也可以是其他形式的机器语言。可以根据自己对编程语言的定义选择实现语言的特定功能。课设的任务主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高自己对系统软件编写的兴趣。

2. 实验一 C_blue 语言定义

2.1. 实验内容

自定义一个高级程序设计语言的词法和语法规则。

1) 可以基于某种熟悉的高级程序语言，设计一个自定义的高级语言子语言，完成其词法和语法规则。

2) 也可以选用 Decaf 语言作为源语言。

2.2. 实验目的与要求

(1) 为语言命名；
(2) 标识符、常数、字符串等单词的文法；
(3) 符号集、保留字集、运算符、界符；
(4) 说明语句文法；
(5) 赋值语句文法（简单赋值）；
(6) 表达式求值文法（简单算术运算，包括++， --）；
(7) 分支语句文法；
(8) 循环语句文法；
(9) 输入语句、输出语句文法
(10) 过程或函数调用语句文法。

2.3. 实验设计与实现

2.3.1. 词法单元

定义一个类 c 语言，命名为 C_blue，其词法单元如表 2.1 所示。其中保留字一共 13 个 int、char、float、struct、void、if、else、while、return、read、write、break、continue；运算符有：+、-、*、/、++、-- (|<|>=|<=|==|!=)、 &&、||、 !、

=；界符有'('、')'、'['、']'、'{'、'}'、','、';'、'.'。

表 2.1 词法单元描述表

词法单元	词法单元内容
数据类型	TYPE -> int|char|float
结构体保留字	STURCT -> struct
返回保留字	RETURN -> return
if 保留字	IF -> if
else 保留字	ELSE -> else
while 保留字	WHILE -> while
void 保留字	VOID -> void
退出循环保留字	BREAK -> break
继续保留字	CONTINUE -> continue
输出保留字	WRITE -> write
输入保留字	READ -> read
标识符	ID -> [A-Za-z] [A-Za-z0-9]*
整型	INT -> [0-9]+

词法单元	词法单元内容
字符	CHAR ->\"[A-Za-z0-9]+\"
浮点数	FLOAT -> [0-9]+[.][0-9]+
左括号	LP -> (
右括号	RP ->)
左中括号	LB -> [
右中括号	RB ->]
左花括号	LC -> {
右花括号	RC -> }
结构体访问：点	DOT -> .
分号	SEMI -> ;
逗号	COMMA -> ,
赋值	ASSIGNOP -> =
比较符	RELOP -> >|<|>=|<=|==|!=
加	ADD -> +
减	SUB -> -
乘	MUL -> *

词法单元	词法单元内容
除	DIV -> /
自增	ADDSELF -> ++
自减	SUBSELF -> --
与	AND -> &&
或	OR -> ||
非	NOT -> !
单行注释//	"//".*
多行注释/**/	"/*"([^*]|(*)*[^*/])*(*)*"*/"

2.3.2. 语法规则

对于语法规则的定义见表 2.2 全局变量和函数规则、表 2.3 变量规则、表 2.4 函数和变量的定义规则、表 2.5 语句规则、表 2.6 局部变量规则、表 2.7 表达式规则。

表 2.2 全局变量和函数规则

语法单元	规则
整个程序	Program -> ExtDefLis
全局变量、结构体、函数集合	ExtDefList -> ExtDef ExtDefList|#

语法单元	规则
全局变量、结构体、函数	ExtDef -> Specifier ExtDecList SEMI |StructSpecifier SEMI |Specifier FunDec CompSt |VOID FunDec CompSt
变量定义	ExtDecList -> VarDec |VarDec COMMA ExtDecList

表 2.3 变量规则

语法单元	规则
类型描述符	Specifier -> TYPE
结构体	StructSpecifier -> STRUCT OptTag LC DefList RC|STRUCT OptTag
标识符	OptTag -> ID

表 2.4 函数和变量的定义规则

语法单元	规则
函数头	FunDec -> ID LP VarList RP|ID LP RP
函数形参列表	VarList -> ParamDec COMMA VarList|ParamDec
函数形参	ParamDec -> Specifier VarDec
变量	VarDec -> ID|VarDec LB INT RB

表 2.5 语句规则

语法单元	规则
语句块	CompSt -> LC DefList StmtList RC
语句列表	StmtList -> Stmt StmtList |#
语句	Stmt -> EXP SEMI|CompSt |RETURN EXP SEMI|IF LP EXP RP Stmt |IF LP EXP RP Stmt ELSE Stmt |WHILE LP EXP RP Stmt |READ LP ID RP|WRITE LP ID RP |BREAK SEMI|CONTINUE SEMI

表 2.6 局部变量规则

语法单元	规则
变量定义列表	DefList -> Def DefList|#
一条变量定义	Def -> Specifier DecList SEMI
一条变量定义里面变量列表	DecList -> Dec |Dec COMMA DecList
一个变量	Dec -> VarDec|VarDec ASSIGNOP EXP

表 2.7 表达式规则

语法单元	规则
表达式	EXP -> EXP ASSIGNOP EXP |EXP AND EXP|EXP OR EXP |EXP RELOP EXP |EXP ADD EXP|EXP SUB EXP|EXP MUL EXP|EXP DIV EXP |LP EXP RP|SUB EXP|NOT EXP|ID LP Args RP |ID LP RP|EXP DOT ID|EXP LB EXP RB |EXP ADDSELF|ADDSELF EXP|EXP SUBSELF|SUBSELF EXP |ID|INT|CHAR|FLOAT

语法单元	规则
实参列表	Args -> EXP COMMA Args|EXP

3. 实验二 词法分析器设计与实现

3.1. 实验内容

调研词法分析程序的自动生成工具 LEX 或 FLEX，将实验一定义的单词文法转换为正规式，设计并实现一个能够输出单词序列（二元式）的词法分析器。

3.2. 实验目的与要求

(1) 理解单词的分类和形式化描述；

(2) 掌握自动生成工具 flex 的使用技术；

(3) 定义保留字和操作符、界符的内部码；

(4) 实现一个完整的词法分析器；

(5) 显示词法分析结果（二元组）列表。

3.3. 实验设计与实现

根据 flex 的语法规则，将实验 1 中定义的语法单元编写成.l 文件。

前面的定义部分使用宏定义，将语法单元名和其表达的含义一一对应，例如 TYPE int|char|float；其中有一个%{ 到%}的区间部分，主要包含 c 语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明，会直接被复制到词法分析器源程序 lex.yy.c 中。

在规则部分进行正则表达式和动作的程序编写。因为前面已经将正则表达式定义成了宏，此处只需要写{宏名}即可。而对于动作，也是被包含在{}之中。对

于每个匹配上的宏，输出其对应的宏名和程序读取的单词。因为每当词法分析器识别出一个单词后，将该单词对应的字符串保存在 yytext 中，其长度为 yyleng，供后续使用。所以只需要输出 yytext 即可。如 {TYPE} {printf("TYPE:%s\n",yytext);}

用户子程序部分，这部分代码会原封不动的被复制到词法分析器源程序lex.yy.c中。在此处定义main函数，调用运行yylex()函数。

ID的正则表达式为[A-Za-z][A-Za-z0-9]*，只可以出现字母和数字，且不能以数字开头。FLOAT的正则表达式为[0-9]+[.][0-9]+ ,只能以小数的形式出现。CHAR的正则表达式为\"[A-Za-z0-9]+\"，定义为双引号里面的字符。注释部分有单行注释和多行注释。如果代码中出现与定义的正则表达式都不匹配的字符，则将其报错输出。

3.4. 实验结果

编写脚本文件.sh，文件中为编译和运行命令。flex 编译.l 文件：flex c_blue.l；将生成的 lex.yy.c 文件编译为运行文件：cc lex.yy.c -o c_blue.out -lfl；用运行程序运行编写的测试代码。

在实验过程中发现相同的输入可以被多种不同的模式匹配，于是出现了二义性。flex 使用以下两个原则进行解决：尽可能匹配匹配多的字符串；关键字的匹配模式先于标识符的匹配，所以关键字会被正确的匹配。

所以在编写时，要将保留字写在 ID 的前面。

测试代码及对应的输出结果如下：

表 3.1 测试代码与结果对比

测试代码	输出结果	

struct test{
int a;
char t;
};
int ok ()
{
char str1[15];
char str2[15];
int ret;
ret=0;
//ggggg
return !ret;
/*dddddd
dddd
*/
}
void main(){
int a=0;
float b=1.0;
char c="a";
if(a==0)
c="b";
else
return 0;
read(a);
write(b);
while(a==1){
a++;
a--;
b=1;
a||b;
a&&b;
a/b;
a*b;
test.a=a;
if(a==b){
break;
}
else{
 	
continue;
}
	STRUCT: struct
	ID: test
	LC:{
	TYPE:int
	ID: a
	SEMI:;
	TYPE:char
	ID: t
	SEMI:;
	RC:}
	SEMI:;
	TYPE:int
	ID: ok
	LP:(
	RP:)
	LC:{
	TYPE:char
	ID: str1
	LB:[
	INT:15
	RB:]
	SEMI:;
	TYPE:char
	ID: str2
	LB:[
	INT:15
	RB:]
	SEMI:;
	TYPE:int
	ID: ret
	SEMI:;
	ID: ret
	ASSIGNOP:=
	INT:0
	SEMI:;
	RETURN:return
	NOT:!
	ID: ret
	SEMI:;
	RC:}
	VOID:void
	ID: main
		READ:read
		LP:(
		ID: a
		RP:)
		SEMI:;
		WRITE:write
		LP:(
		ID: b
		RP:)
		SEMI:;
		WHILE:while
		LP:(
		ID: a
		RELOP:==
		INT:1
		RP:)
		LC:{
		ID: a
		ADDSELF:++
		SEMI:;
		ID: a
		SUBSELF:--
		SEMI:;
		ID: b
		ASSIGNOP:=
		INT:1
		SEMI:;
		ID: a
		OR:||
		ID: b
		SEMI:;
		ID: a
		AND:&&
		ID: b
		SEMI:;
		ID: a
		DIV:/
		ID: b
		SEMI:;
		ID: a
		MUL:*
		ID: b

测试代码	输出结果
} }	LP:(RP:) LC:{ TYPE:int ID: a ASSIGNOP:= INT:0 SEMI:; TYPE:float ID: b ASSIGNOP:= FLOAT:1.0 SEMI:; TYPE:char ID: c ASSIGNOP:= CHAR:"a" SEMI:; IF:if LP:(ID: a RELOP:== INT:0 RP:) ID: c ASSIGNOP:= CHAR:"b" SEMI:; ELSE:else RETURN:return INT:0 SEMI:; SEMI:; ID: test DOT:. ID: a ASSIGNOP:= ID: a SEMI:; IF:if LP:(ID: a RELOP:== ID: b RP:) LC:{ BREAK:break SEMI:; RC:} ELSE:else LC:{ CONTINUE:continue SEMI:; RC:} RC:} RC:}

4. 实验三 语法分析器设计与实现

4.1. 实验内容

选择语法分析器实现方法，可以选择自上而下的方法，也可以使用自底向上的方法。

4.2. 实验目的与要求

与实验(二)结果关联，实现一个自底向上的 LR 分析器

(1) 调研语法生成工具 Bison；

(2) 根据实验（一）定义的语法规则，编写 bison 源程序；

(3) 用 bison 生成语法分析程序；

(4) 显示语法分析结果-语法树。

4.3. 实验设计与实现

本实验开始的第一步，是要将实验 2 完成的.l 文件进行修正，把正则表达式后面的动作部分除了原有的输出 ,后面再加上 return 宏名 ;其目的是方便与 bison 联合编程。

根据实验 1 编写的自定义语言的语法规则，按照 bison 的要求格式将语法规则编写成.y 文件。bison 的语法规则与 flex 相似，不过在这个实验，声明部分会有很多变动。

4.3.1. 声明部分：

因为对于语法分析，需要进行精准的定位报错，所以此处会用到位置信息。

(1)行号

在flex中定义了一个内部变量yylineno，当在flex文件的开头加上%option yylineno后，就可以直接使用这个内部变量了，并且不需要去维护yylineno的值，在词法分析过程中，每次遇到一个回车，yylineno会自动加一。同时在bison文件的声明部分加上extern int yylineno，就可以共用yylineno的值。

(2)列号

利用一个结构体YYLTYPE，该结构体不用定义，bison中含有。bison中的每一个语法单元（终结符和非终结符）对应一个YYLTYPE类型的位置信息，YYLTYPE定义形式为：

```
typedef struct {
    int first_line;
    int first_column;
    int last_line;
    int last_column;
} YYLTYPE;
```

其中first_line和first_column表示该语法单元第一个单词出现的行号和列号，last_line和last_column表示该语法单元最后一个单词出现的行号和列号。

(3)引用位置信息

为了能正确引用位置信息，需要使用flex的内置变量yylloc，yylloc表示当前词法单元所在的位置信息。首先需要在flex文件的声明部分加上：

```
int yycolumn=1;
#define YY_USER_ACTION yylloc.first_line = yylloc.last_line =
```

yylineno;\yylloc.first_column = yylloc.last_column; \yylloc.last_column = yycolumn + yyleng - 1; \ yycolumn += yyleng;

在flex文件规则部分的正则式‘\n’后面将yycolumn赋值为1 ,表示一个新的行，列数从1开始。同时在bison文件的开头加上%locations，这样在bison中，就可以使用yylloc.first_line和yylloc.first_column表示当前单词位置的行号和列号，准确地标识错误的位置。

(4)错误类型

因为想要更准确地给出错误的性质，在.y文件的开头加上%define parse.error verbose。这样编写语法报错函数yyerror时，定义一个char* fmt的参数，报错时，只需要输出行、列和该字符串fmt即可得到报错的位置及类型。

4.3.2. 辅助定义部分：

(1)联合编码

在 Flex 和 Bison 联合使用时 ,需要在 parser.y 中的%token 后面罗列出所有终结符(单词)的种类码标识符 ,终结符也就是实验 2 中定义的宏 ,以便于联合使用。

用 bison-d c_blue.y 会生成一个 c_blue.tab.h 的头文件 ,头文件将会有如图 4.1 所示的编码部分，将该头文件加入.l 文件的声明部分。如果 bison 编译时有-v 参数，会生成自动机文件.output，通过该文件可以看到是如何进行规约的，在消除规则的偏移和规约冲突时该文件非常有用。

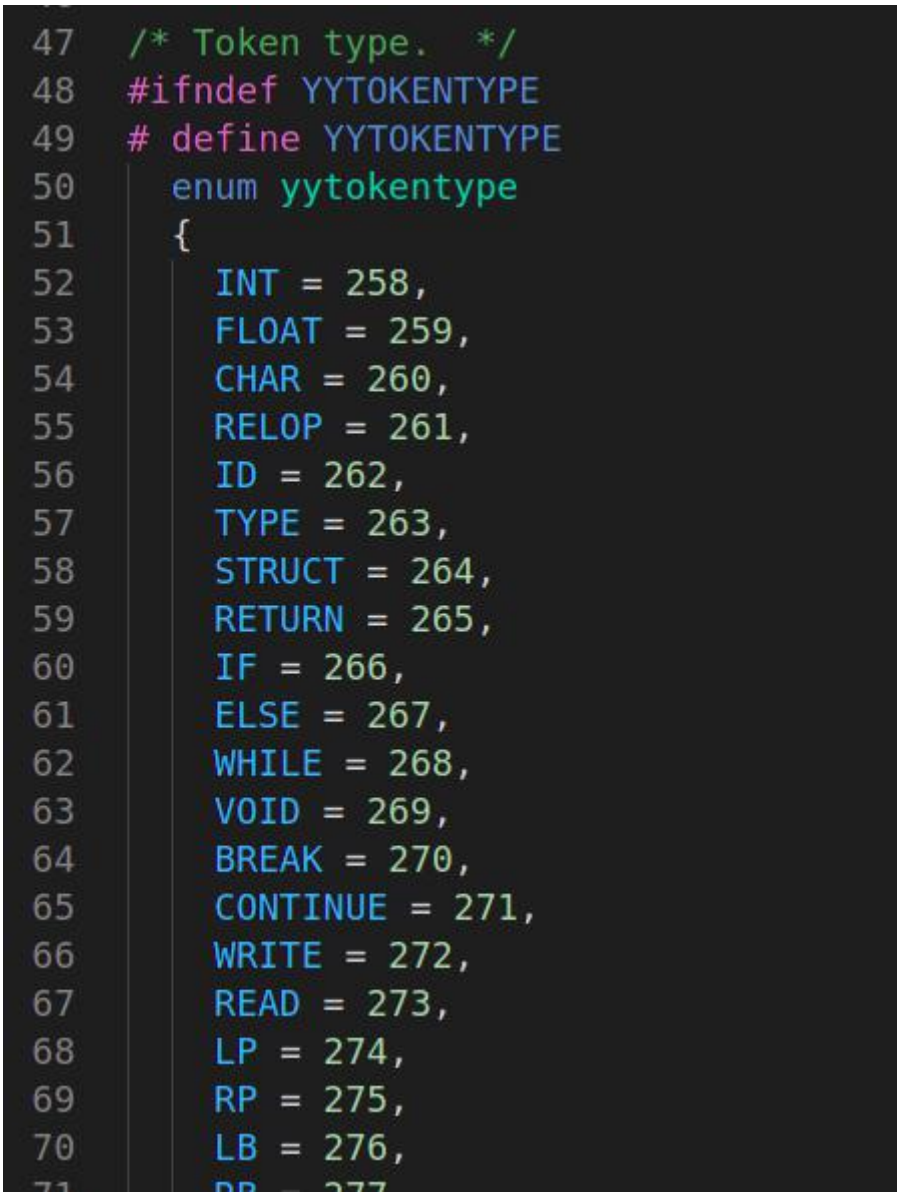

图 4.1 终结符的编号

(2)类型说明

对于.y 文件的辅助定义部分，对其进行一个联合体的声明，该联合体为：%union {

```
int type_int;
float type_float;
char type_char[32];
char type_id[32];
struct Node *ptr;
};
```

其中的含义分别表示整型数据的值、浮点型数据的值、字符型数据的值、标

志符 id、语法树的节点，每一个非终结符对应一个节点。

用 %token <type_int> INT 来说明 INT 对应联合中的成员 type_int；用 %token <type_float> FLOAT 来说明 FLOAT 对应联合中的成员 type_float；用%token <type_char> CHAR 来说明 CHAR 对应联合中的成员 type_char；用 %token <type_id> RELOP ID TYPE 来说明 RELOP ID TYPE 对应联合中的成员 type_id。

用%type <ptr> program ExtDefList.....表示所有非终结符属性值的类型对应联合中成员 ptr 的类型，对应一个树节点的指针。

(3)优先级和结合性

在编写程序时，终结符越靠后优先级越大。

对于结合性，本语法的结合性为‘=’、‘-（负号）’、‘！’为右结合，其余的运算符都为左结合。

优先级一次为=、||、&&、比较符、（+、-）、（*、/）、（++、--）、（括号、中括号）、负号、！。括号内的同优先级。

消除偏移规约冲突：%nonassoc 的含义是没有结合性。它一般与%prec 结合使用表示该操作有同样的优先级。SUB EXP %prec USUB： ‘-’原本声明时的优先级要小于'*'，使用%pre USUB，可以让'-'拥有 USUB 的优先级，从而使得表达出"负号"的效果。IF LP EXP RP Stmt %prec LOWER_THEN_ELSE 将只有 if 的语句标记为 LOWER_THEN_ELSE，其优先级高于 ELSE。

4.3.3. 规则部分：

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

(1) 编写规则

编写规则部分，先将语法规则编写出来，先暂且不管每个规则后面的动作。在这部分要注意将空规则放到该语法单元的规则最前面，后面跟动作{$$=NULL；}，其中$$表示该语法单元，该动作表示什么都不做。

函数 mknode ,完成建立一个树结点。例如 ExtDefList:ExtDef ExtDefList {$$ = mknode(2,EXT_DEF_LIST,yylineno,$1,$2);};这里的语义动作是将建立的结点 EXT_DEF_LIST 的指针返回赋值给规则左部 ExtDefList，表示完成此次归约后，生成了一棵子树，子树的根结点指针为$$，根结点类型是 EXT_DEF_LIST。该子树有 2 棵子树，第一棵是$1 表示的 ExtDef 的子树；第二棵对应是$2 的表示的 ExtDefList 的子树，另外 yylineno 表示赋值语句的行号。

按照此种步骤和用法即可完成剩余动作的编写。

(2) 容错机制

此外，为了实现报错后仍然能继续对之后的代码进行语法分析，这里添加了一些语法规则用来实现容错机制。

ExtDef: error SEMI {$$ = NULL; };表示有错时跳过分号，继续分析。

FunDec: error RP {$$ = NULL;};表示有错时跳过右括号，继续分析。

Stmt: error SEMI {$$ = NULL; };表示有错时跳过分号，继续分析。

DefList: error SEMI {$$ = NULL; };表示有错时跳过分号，继续分析。

EXP: error RP {$$ = NULL;};表示有错时跳过右括号，继续分析。

需要注意的一点是 ,再编译时 ,可能会因为这些容错机制造成移进规约冲突，这种不会对后面的结果产生影响。

4.3.4. 语法树

对于语法树，首先构造树节点的结构体 Node，其包含变量和含义如下：

```
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
    int kind;
    union {
        int type_int;
        float type_float;
        char type_char[32];
        char type_id[32];
    };
    struct Node *ptr[4];              //由 kind 确定有多少棵子树
    int place;                        //存放（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];             //结点对应语句 S 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
    int type;                           //用以标识表达式结点的类型
    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 num;                            //计数器，可以用来统计形参个数
};
```

每进一层递归，层数就加一，对每个节点，只需要分析相应的情况，对节点的属性进行输出，对每个子树进行递归遍历即可得到语法树。例如：

```
根节点 EXT_DEF_LIST:
display(T->ptr[0],indent);
//显示该外部定义（外部变量和函数）列表中的第一个
```

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

4.4. 实验结果

对于语法分析器，根据 flex 和 bison 的使用说明，主要是将 flex 编译后生成的 lex.yy.c 文件，通过调用 yylex 进行词法分析后，将得到的每个单词的种类码和自身值提供给语法分析程序。

语法分析程序调用 yyparser 进行语法分析，如果正确构造语法分析树，否则进行报错。

4.4.1. 语法分析测试结果

对于语法分析和报错容错处理见表 4.1，因为代码太长，这里只选取了部分测试代码作为说明。红字部分为语法分析报错。

表 4.1 语法分析测试

部分测试代码	输出结果

struct test{
int a;
char t;
};
int ok ()
{
int;
char str1[15];
char str2[15];
int ret;
while(){
 	
}
ret=0;
//ggggg
return !ret;
/*dddddd
	TYPE:int
	ID: ok
	LP:(
	RP:)
	LC:{
	TYPE:int
	SEMI:;
	Grammar Error at Line 7 Column 7:syntax error, unexpected SEMI, expecting ID.
	TYPE:char
	ID: str1
	LB:[
	INT:15
	RB:]
	SEMI:;
	TYPE:int
	ID: ret
	SEMI:;
	WHILE:while
	LP:(
	RP:)
	Grammar Error at Line 11 Column 10:syntax error, unexpected RP.
	LC:{
	RC:}
	ID: ret
	ASSIGNOP:=
	INT:0
	SEMI:;
	RETURN:return
	NOT:!

部分测试代码	输出结果
dddd */ }	TYPE:char ID: str2 LB:[INT:15 RB:] SEMI:; ID: ret SEMI:; RC:}

4.4.2. 生成语法树结果

对于语法树的测试表 4.2，左边为测试代码，右边为生成的语法树。

表 4.2 生成语法树结果

测试代码	输出结果

struct test{
int a;
char t;
};
int ok ()
{
// int;
char str1[15];
char str2[15];
int ret;
// while(){
 	
// }
ret=0;
//ggggg
return !ret;
/*dddddd
dddd
*/
}
void main(){
int a=0;
float b=1.0;
char c="a";
if(a==0)
	结构体定义：(4)
	结构定义：(4)
	ID： test
	局部变量定义：(2)
	类型： int
	变量名：
	ID： a
	局部变量定义：(3)
	类型： char
	变量名：
	ID： t
	函数定义：(20)

测试代码	输出结果

c="b";
else
return 0;
read(a);
write(b);
while(a==1){
a++;
a--;
b=1;
a||b;
a&&b;
a/b;
a*b;
test.a=a;
if(a==b){
break;
}
else{
continue;
}
}
 	
}
	类型： int
	
	ID： ok
	
	复合语句：(20)
	
	复合语句的变量定义部分：
	
	局部变量定义：(8)
	
	类型： char
	
	变量名：
	
	数组：(8)
	
	ID： str1
	
	INT：15
	
	局部变量定义：(9)
	
	类型： char
	
	变量名：
	
	数组：(9)
	
	ID： str2
	
	INT：15
	
	局部变量定义：(10)
	
	类型： int
	
	变量名：
	
	ID： ret
	
	复合语句的语句部分：

测试代码	输出结果
	表达式语句：(14) ASSIGNOP:= ID： ret INT：0 返回语句：(16) NOT:! ID： ret 函数定义：(48) 类型：void ID： main 复合语句：(48) 复合语句的变量定义部分： 局部变量定义：(22) 类型： int 变量名： ASSIGNOP:= ID： a INT：0 局部变量定义：(23) 类型： float 变量名： ASSIGNOP:= ID： b

测试代码	输出结果
	FLAOT：1.000000 局部变量定义：(24) 类型： char 变量名： ASSIGNOP:= ID： c CHAR: "a" 复合语句的语句部分： 条件语句(IF_THEN_ELSE)：(28) 条件： == ID： a INT：0 IF 子句：(28) 表达式语句：(26) ASSIGNOP:= ID： c CHAR: "b" ELSE 子句：(28) 返回语句：(28) INT：0 READ 语句：(29) READ 变量：(29)

测试代码	输出结果
	ID： a WRITE 语句：(30) WRITE 变量：(30) ID： b 循环语句：(46) 循环条件： == ID： a INT：1 循环体：(46) 复合语句：(46) 复合语句的变量定义部分： 复合语句的语句部分： 表达式语句：(32) ADDSELF(E++):++ ID： a 表达式语句：(33) SUBSELF(E--):-- ID： a 表达式语句：(34) ASSIGNOP:= ID： b INT：1

测试代码	输出结果
	表达式语句：(35) OR:|| ID： a ID： b 表达式语句：(36) AND:&& ID： a ID： b 表达式语句：(37) DIV:/ ID： a ID： b 表达式语句：(38) MUL:* ID： a ID： b 表达式语句：(39) ASSIGNOP:= 结构体访问： ID： test 访问成员变量：a ID： a 条件语句(IF_THEN_ELSE)：(45)

测试代码	输出结果
	条件： == ID： a ID： b IF 子句：(45) 复合语句：(42) 复合语句的变量定义部分： 复合语句的语句部分： BREAK 语句：(41) ELSE 子句：(45) 复合语句：(45) 复合语句的变量定义部分： 复合语句的语句部分： CONTINUE 语句：(44)

5. 实验四 符号表管理和属性计算

5.1. 实验内容

设计符号表数据结构和关键管理功能；完成相关属性计算。

5.2. 实验目的与要求

(1) 选择合适的数据结构实现符号表；

(2) 实现符号表上的操作，包括创建符号表、插入表项、查询表项、修改表项、删除表项、释放符号表空间等等；

(3) 计算符号的属性值；

(4) 输出符号表和符号表动态变化过程。

5.3. 实验设计与实现

5.3.1. 实验实现说明

本次实验采用顺序表实现符号表。本次产生的符号表的属性有索引（符号在符号表的位置）、名字、层号、类型、标记（符号标记，函数：'F' 全局变量：'V' 参数：'P' 临时变量：'T'，结构体：S，结构体变量：M，数组：A）、参数个数（主要记录符号为函数的时候的形参的个数）、数组大小（记录声明的数组大小，方便语义分析的报错）、所属结构（标记该符号所属结构体的名字）。

实验所用到的函数包括：prn_symbol：显示符号表；searchmyTable：查找符号表；fillmyTable：添加符号到符号表，先进行查找是否有重复再进行填写；

semantic_Analysis：生成符号表的过程，采用对生成的语法树的先根遍历进行生成；semantic_Analysis0：在符号表加入 read 和 write 函数，其形参个数都为 1 个，并对生成符号表函数进行调用。该函数会在.y 文件的 program 语法单元的规则的动作里进行调用。semantic_error：对报错信息进行收集。

5.3.2. 符号表生成

在符号表的生成过程主要有语法树 AST 的先根遍历和作用域。

(1) 层号与符号添加

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量 LEV 来管理，LEV 的初始值为 0。

在处理外部变量名，以及函数名时，对应符号的层号值都是 0；处理函数形式参数时，固定形参名在填写符号表时，层号为 1。

每次遇到一个语句块 COMP_STM，LEV 增加 1，表示准备进入一个新的作用域。为了管理这个作用域中的变量，使用栈 symbol_scope_TX，记录该作用域变量在符号表中的起点位置，即将符号表 symbolTable 的栈顶位置 symbolTable.index 保存在栈 symbol_scope_TX 中。

每次遍历完一个语句块 COMP_STM 的所有子树，准备回到其父结点时，这时该复合语句语义分析完成，需要从符号表中删除该复合语句的变量，方法是首先 symbol_scope_TX 退栈，取出该复合语句作用域的起点，再根据这个值修改 symbolTable.index，同时 LEV 减一，很简单地完成了符号表的符号删除操作。

每次进行添加符号时，先进行符号表的查找操作，在 symbolTable 中，从栈顶向栈底方向查询是否有相同的符号定义。如果找到了该符号，且除了名字外，

标记、层数、所属结构都一致，则不可添加；否则可以添加。

(2) 结构体、数组、全局变量说明

全局变量：要注意的一点是，在查找符号表时，因为我定义的语法设置了全局变量，所以除了查找该层号的符号之外，还需要比较要添加的符号是否与全局变量重复，如果重复不能添加，而且会给予报错。

结构体：在遍历 AST 时，如果遇到结构体，会在添加符号表时将符号表属性的 struct_name 设为结构体的名字，以便于在符号表中显示。

数组：在 AST 中检测到数组时，除了将其相关属性放入符号表中，还要将数组的长度也加入符号表中，为语义分析提供方便。

5.4. 实验结果

在本次测试中，根据测试代码，生成的符号表见表 5.1。在表格中，左边红色字体为输出符号表的编号位置，右边编号对应的图片为生成的符号表。这样表示测试结果较为清晰。

表 5.1 测试代码与符号表

测试代码	输出结果

测试代码	输出结果

A1
struct test{
int a;
char t;
};
int ok(int a,int b){
int ret;
int str1[15];
char str1[15];
a=1;
ret=1;
str1[5]=0;
if(ret < 0)
{
int t=1;
ret=ret+1;
}**A2**
else if(ret > 0)
{
ret=ret-1;
}**A3**
//ggggg
return !ret;
/*dddddd
dddd
*/
}**A4**
void main(){
int x;
int a=0;
float b=1.0;
char c="a";
ok(1,2);
if(a==0)
c="b";
A5
read(a);
write(b);
while(a==1){
a++;
++a;
a--;
	A1:
	符号表
	索引 名字 层号 类型 标记 参数个数 数组大小 所属结构
	0 read 0 int F 1 null null
	1 x 1 int P null null null
	2 write 0 int F 1 null null
	3 y 1 int P null null null
	A2:
	符号表
	索引 名字 层号 类型 标记 参数个数 数组大小 所属结构
	0 read 0 int F 1 null null
	1 x 1 int P null null null
	2 write 0 int F 1 null null
	3 y 1 int P null null null
	4 test 0 struct S null null null
	5 a 0 int M null null test
	6 t 0 char M null null test
	7 ok 0 int F 2 null null
	8 a 1 int P null null null
	9 b 1 int P null null null
	10 ret 1 int T null null null
	11 str1 1 int A null 15 null
	12 str1 1 char A null 15 null
	13 t 2 int T null null null
	A3:
	符号表
	索引 名字 层号 类型 标记 参数个数 数组大小 所属结构
	0 read 0 int F 1 null null
	1 x 1 int P null null null
	2 write 0 int F 1 null null
	3 y 1 int P null null null
	4 test 0 struct S null null null
	5 a 0 int M null null test
	6 t 0 char M null null test
	7 ok 0 int F 2 null null
	8 a 1 int P null null null
	9 b 1 int P null null null
	10 ret 1 int T null null null
	11 str1 1 int A null 15 null
	12 str1 1 char A null 15 null
	A4:

测试代码	输出结果

b=1;
a||b;
a&&b;
a/b;
a*b;
test.a=a;
if(a==b){
break;
}A6
else{
continue;
}A7
}A8
}A9
	符号表
	索引 名字 层号 类型 标记 参数个数 数组大小 所属结构
	0 read 0 int F 1 null null
	1 x 1 int P null null null
	2 write 0 int F 1 null null
	3 y 1 int P null null null
	4 test 0 struct S null null null
	5 a 0 int M null null test
	6 t 0 char M null null test
	7 ok 0 int F 2 null null
	8 a 1 int P null null null
	9 b 1 int P null null null
	10 ret 1 int T null null null
	11 str1 1 int A null 15 null
	12 str1 1 char A null 15 null
	A5:
	符号表
	索引 名字 层号 类型 标记 参数个数 数组大小 所属结构
	0 read 0 int F 1 null null
	1 x 1 int P null null null
	2 write 0 int F 1 null null
	3 y 1 int P null null null
	4 test 0 struct S null null null
	5 a 0 int M null null test
	6 t 0 char M null null test
	7 ok 0 int F 2 null null
	8 main 0 void F 0 null null
	9 x 1 int T null null null
	10 a 1 int T null null null
	11 b 1 float T null null null
	12 c 1 char T null null null
	A6:
	符号表
	索引 名字 层号 类型 标记 参数个数 数组大小 所属结构
	0 read 0 int F 1 null null
	1 x 1 int P null null null
	2 write 0 int F 1 null null
	3 y 1 int P null null null
	4 test 0 struct S null null null
	5 a 0 int M null null test
	6 t 0 char M null null test
	7 ok 0 int F 2 null null
	8 main 0 void F 0 null null
	9 x 1 int T null null null
	10 a 1 int T null null null
	11 b 1 float T null null null
	12 c 1 char T null null null
	A7:

测试代码	输出结果
	符号表 索引 名字 层号 类型 标记 参数个数 数组大小 所属结构 0 read 0 int F 1 null null 1 x 1 int P null null null 2 write 0 int F 1 null null 3 y 1 int P null null null 4 test 0 struct S null null null 5 a 0 int M null null test 6 t 0 char M null null test 7 ok 0 int F 2 null null 8 main 0 void F 0 null null 9 x 1 int T null null null 10 a 1 int T null null null 11 b 1 float T null null null 12 c 1 char T null null null **A8:** ***符号表*** 索引 名字 层号 类型 标记 参数个数 数组大小 所属结构 0 read 0 int F 1 null null 1 x 1 int P null null null 2 write 0 int F 1 null null 3 y 1 int P null null null 4 test 0 struct S null null null 5 a 0 int M null null test 6 t 0 char M null null test 7 ok 0 int F 2 null null 8 main 0 void F 0 null null 9 x 1 int T null null null 10 a 1 int T null null null 11 b 1 float T null null null 12 c 1 char T null null null **A9:** ***符号表*** 索引 名字 层号 类型 标记 参数个数 数组大小 所属结构 0 read 0 int F 1 null null 1 x 1 int P null null null 2 write 0 int F 1 null null 3 y 1 int P null null null 4 test 0 struct S null null null 5 a 0 int M null null test 6 t 0 char M null null test 7 ok 0 int F 2 null null 8 main 0 void F 0 null null

6. 实验五 静态语义分析

6.1. 实验内容

(1) 定义错误类型；

(2) 熟悉语义规则的表达形式，完成类型检查；

(3) 完成静态语义分析。

6.2. 实验目的与要求

要求根据实验定义的语言，检查出如下所述类型的静态语义错误：

(1) 使用未定义的变量；

(2) 调用未定义或未声明的函数；

(3) 在同一作用域，名称的重复定义（如变量名、函数名、结构类型名以及结构体成员名等）。为更清楚说明语义错误，这里也可以拆分成几种类型的错误，如变量重复定义、函数重复定义、结构体成员名重复等；

(4) 对非函数名采用函数调用形式；

(5) 对函数名采用非函数调用形式访问；

(6) 函数调用时参数个数不匹配，如实参表达式个数太多、或实参表达式个数太少；

(7) 函数调用时实参和形参类型不匹配；

(8) 对非数组变量采用下标变量的形式访问；

(9) 数组变量的下标不是整型表达式；

(10) 对非结构变量采用成员选择运算符“.”；

(11) 结构成员不存在；

(12) 赋值号左边不是左值表达式；

(13) 对非左值表达式进行自增、自减运算；

(14) 对结构体变量进行自增、自减运算；

(15) 类型不匹配。如数组名与结构变量名间的运算，需要指出类型不匹配错误；有些需要根据定义的语言的语义自行进行界定，比如：32+'A'，10*12.3，如果使用强类型规则，则需要报错，如果按 C 语言的弱类型规则，则是允许这类运算的，但需要在后续阶段需要进行类型转换，类型统一后再进行对应运算；

(16) 函数返回值类型与函数定义的返回值类型不匹配；

(17) 函数没有返回语句（当函数返回值类型不是 void 时）；

(18) break 语句不在循环语句或 switch 语句中；

(19) continue 语句不在循环语句中；

6.3. 实验设计与实现

语义分析这部分完成的是静态语义分析，主要包括：

(1) 控制流检查。如 break、continue 语句必须出现在循环语句当中。

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

(3) 名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明。

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

在实验四生成符号表的函数的基础上，对遍历 AST 的过程进行添加语义分析。对于每次进行变量访问时，进行符号表的查找操作。在 AST 的遍历过程中，遇到变量的访问时，在 symbolTable 中，从栈顶向栈底方向查询是否有相同的符号定义，如果全部查询完后没有找到，就是该符号没有定义；如果相同符号在符号表中有多处定义，按查找的方向可知，符合就近优先的原则。

如果查找到符号后，就进一步进行语义分析，如函数调用时，根据函数名在符号表找到的是一个变量，不是函数，需要报错；函数调用时，根据函数名找到这个函数，需要判断参数个数、类型是否匹配；根据变量名查找的是一个函数。等等，需要做出各种检查。

在访问结构体里面的变量时，会将该变量前面的结构体名字与在符号表中查找到的该变量的所属结构进行比对，如果不一样，则进行报错，一样才可以继续访问。

使用 semantic_error 集中收集报错进行输出。

6.4. 实验结果

在实验测试时，测试结果见表 6.1，表格左边对应的编号的错误如下：

(1)使用未定义的变量；

(2)调用未定义或未声明的函数；

(3)在同一作用域，名称的重复定义。如变量重复定义、函数重复定义、结构体成员名重复等；

(4)对非函数名采用函数调用形式；

(5)对函数名采用非函数调用形式访问；

(6)函数调用时参数个数不匹配；

(7)函数调用时实参和形参类型不匹配；

(8)对非数组变量采用下标变量的形式访问；

(9)数组变量的下标不是整型表达式；

(10)对非结构变量采用成员选择运算符“.”；

(11)结构成员不存在；

(12)赋值号左边不是左值表达式；

(13)对非左值表达式进行自增、自减运算；

(14)对结构体变量进行自增、自减运算；

(15)类型不匹配。

(16)函数返回值类型与函数定义的返回值类型不匹配；

(17)函数没有返回语句（当函数返回值类型不是 void 时）；

(18)break 语句不在循环语句中；

(19)continue 语句不在循环语句中；

(20)数组下标出界

表 6.1 语义分析测试

测试代码

测试代码

```
int ok(int a,int b){
    int ret;
    int str1[15];
    char str2[15];
    int ret;//(3)在同一作用域，名称的重复定义。
    a=1;
    ret=1;
    str1[5]=0;
    str1[20]=0;//(20)数组下标出界
    next=1;//(1)使用未定义的变量;
    test();//(2)调用未定义或未声明的函数;
    ret();//(4)对非函数名采用函数调用形式;
    ok=1;//(5)对函数名采用非函数调用形式访问;
    ok(1);//(6)函数调用时参数个数不匹配;
    ok(str1,str2);//(7)函数调用时实参和形参类型不匹配;
    a[1]=0;//(8)对非数组变量采用下标变量的形式访问;
    str1[1.5]=0;//(9)数组变量的下标不是整型表达式;
    a.b=1;//(10)对非结构变量采用成员选择运算符“.”;
    test.x=1;//(11)结构成员不存在;
    1=0;//(12)赋值号左边不是左值表达式;
    2++;//(13)对非左值表达式进行自增、自减运算;
    test++;//(14)对结构体变量进行自增、自减运算;
    str1=1;//(15)类型不匹配。
    if(ret < 0)
    {
        int t=1;
        ret=ret+1;
    }
    else if(ret > 0)
    {
        ret=ret-1;
    }
    return str1;//(16)函数返回值类型与函数定义的返回值类型不匹配;
}
void main(){
    int x;
    int a=0;
    float b=1.0;
    char c="a";
    ok(1,2);
    break;//(18)break语句不在循环语句中;
    continue;//(19)continue语句不在循环语句中;
    return 0;//(17)函数返回类型与返回语句不匹配;
}
```

测试代码

测试结果：

错误（9）在编译时就会报错：

```
ID: str1
LB:[
FLOAT:1.500000
Grammar Error at Line 21 Column 9:syntax error, unexpected FLOAT, expecting INT.
RB:]
```

错误 1-8、10-20：

```
error! 第9行, ret 局部变量重复定义
error! 第13行, str1 数组index错误
error! 第14行, next 变量未定义
error! 第15行, test 不是函数
error! 第16行, ret 不是函数
error! 第17行, ok 变量未定义
error! 第18行,  参数数量不匹配
error! 第19行,  函数调用参数类型不匹配
error! 第20行, a 不是数组
error! 第22行, a 不是结构体
error! 第23行, 结构体不含成员变量 x
error! 第24行,  赋值表达式需要左值
error! 第25行,  自增自减表达式需要左值
error! 第26行, test 不是左值
error! 第27行,  类型不匹配
```

```
error! 第40行, x 局部变量重复定义
error! 第45行,  break语句要在循环语句中
error! 第46行,  continue语句要在循环语句中
error! 第47行,  返回值类型错误
```

```
error! 第48行, void 函数不应该有返回值
```

7. 实验六 中间代码生成

7.1. 实验内容

定义中间代码的形式，生成中间代码并显示。

通过前面对 AST 遍历，完成了语义分析后，如果没有语法语义错误，就可以再次对 AST 进行遍历，计算相关的属性值，利用符号表，生成以三地址代码 TAC 作为中间语言的中间语言代码序列。

7.2. 实验目的与要求

(1) 掌握中间代码形式设计技术;

(2) 熟悉中间代码生成规则;

(3) 熟悉逻辑表达式的生成技术;

(4) 完成各种可执行语句的中间代码生成。

7.3. 实验设计与实现

7.3.1. 中间语言的定义

用三地址代码 TAC 作为中间语言，中间语言代码的定义如表 7.1 所示。

表 7.1 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
LABEL x	定义标号 x	LABEL			x
FUNCTION f:	定义函数 f	FUNCTION			f

x := y	赋值操作	ASSIGN	x		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

三地址中间代码 TAC 是一个 4 元组，逻辑上包含（op、opn1、opn2、result），其中 op 表示操作类型说明，opn1 和 opn2 表示 2 个操作数，result 表示运算结果。后续还需要根据 TAC 序列生成目标代码，所以设计其存储结构时，每一部分要考虑目标代码生成时所需要的信息。

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

(2) 操作数与运算结果：这些部分包含多种类型：整常量、实常量、标识符（如变量的别名、变量在其数据区的偏移量（外部变量给出的是在静态数据区的偏移量，局部变量、临时变量给出的是在活动记录空间的偏移量）、转移语句中的标

号等。类型互不相同，所以考虑使用联合。为了明确联合中的有效成员，将操作数与运算结果设计成结构类型，包含 kind，联合等几个成员，kind 说明联合成员属于哪种类型，是整常量、或是实常量、或是标识符表示的别名、或是标号、或是函数名等。

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

7.3.2. 翻译

函数：Newtemp：生成一临时变量，登记到符号表中，以 temp+序号的形式组成的符号串作为别名，符号名称用空串的形式登记到符号表中。newLabel：生成一个标号，标号命名形式为 LABEL+序号。genIR：生成一条 TAC 的中间代码语句。genLabel：生成标号语 TAC 的中间代码语句。Merge：将多个 TAC 语句序列顺序连接在一起。

在翻译时主要加入了数组的访问和结构体的访问，并补充了相应的代码。

数组和结构体的访问的中间代码采用偏移位置进行生成。采用取数组的数组名的地址加上下标的偏移量*数据类型的长度；结构体采用找到结构体的地址，

各种语句类型的翻译见表 7.2、表 7.3、表 7.4、表 7.5。

表 7.2 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 说明部分子树 T2 语句部分子树	访问到 T：T2.Snext=T.Snext 访问 T 的所有子树后： T.code=T1.code || T2.code

当 前 结 点 类 型	翻 译 动 作
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　if 子句子树	访问到 T：T1.Etrue=newLabel, T1.Efalse=T.Snext, T2.Snext= newLabel; 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

表 7.3 基本表达式类结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作

当 前 结 点 类 型	翻 译 动 作
INT 其它如 FLOAT 类的结点按类似方法处理	t_i=newtemp, t_i 在符号表的入口赋值给 T.place。后续可通过 T.alias 读取该值。 T.code 为： t_i = INT 的值
ID	ID 在符号表中的入口赋值给 T.place，代码为空
ASSIGNOP T1 左值表达式子树 T2 左值表达式子树	访问到 T： T. palce=T1.place 访问 T 的所有子树后 :T.code=T1.code || T2.code || T1.alias= T2.alias
OP 算术运算符。 T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place 访问 T 的所有子树后： T.code=T1.code || T2.code || t_i=T1.alias OP T2.alias
UMINUS T1 操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place 访问 T 的所有子树后：T.code=T1.code || t_i=-T1.alias
RELOP 关系运算符 T1 第一操作数子树 T2 第二操作数子	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel，Label2=newLabel。 访问 T 的所有子树后： T.code=T1.code || T2.code || if T1.alias RELOP T2.alias goto label1 || t_i=0 || goto label2 || label1： || t_i=1 ||

当 前 结 点 类 型	翻 译 动 作
树	label2：
AND T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel。 访问 T 的所有子树后： T.code=T1.code || T2.code || t_i=T1.alias * T2.alias || if t_i==0 goto label1 || t_i=1 || label1：
OR T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel，Label2=newLabel。 访问 T 的所有子树后： T.code=T1.code || T2.code || t_i=0 || if T1.alias ==0 goto label1 || t_i=1 || goto label2 || label1： || if T2.alias ==0 goto label2 || t_i=1 || label2：
NOT T1 操作数子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place， Label1=newLabel，Label2=newLabel。 访问 T 的子树后： T.code= if T1.alias ==0 goto label1 || t_i=0 || goto label2 || label1：|| t_i=1 || label2：
FUNC_CALL T1 实参列表子树	t_i=newtemp, t_i 在符号表的入口赋值给 T.place T.code=T1.code; 访问 T 的子树，从上至下依次对每个 ARGS 实参结点 T0，完成实参处理。 T.code= T. code || ARG T01.alias

当 前 结 点 类 型	翻 译 动 作
	这里 T01 表示 T0 的第一个孩子，访问 T 的子树后： T.code= T.code || t_i=CALL 函数名

表 7.4 控制语句布尔表达式语句结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作
INT 其它如 FLOAT 类的结点按类似方法处理	if (T.Etrue==””) 按基本表达式处理 else if (INT 的值) T.code= goto T.Etrue else T.code= goto T.Efalse
ID	ID 在符号表中的入口赋值给 T.place if (T.Etrue==””) 按基本表达式处理 else T.code= if T.alias!=0 goto T.Etrue || goto T.Efalse
ASSIGNOP T1 左值表达式子树 T2 左值表达式子树	T.palce=T1.place 访问 T 的所有子树后： T.code=T1.code || T2.code || T1.alias= T2.alias if (T.true!=””) T.code=T.code || if T1.alias!=0 goto T.Etrue || goto T.Efalse
OP 算术运算符。 T1 第一操作数子树 T2 第二操作数子树	t_i=newtemp, t_i 入口赋值给当前结点 T.place 访问 T 的所有子树后： T.code =T1.code || T2.code || t_i=T1.alias OP T2.alias if (T.true!=””) T.code = T.code || if t_i!=0 goto T.Etrue || goto T.Efalse
UMINUS T1 操作数子树	t_i=newtemp, t_i 入口赋值给当前结点 place 访问 T 的所有子树后：

当 前 结 点 类 型	翻 译 动 作
	T.code = T1.code || t_i=- T1.alias if (T.true!=””) T.code=T.code || if t_i!=0 goto T.Etrue || goto T.Efalse

RELOP 关系运算符
T1 第一操作数子树
T2 第二操作数子树
	if (T.true==””) t_i=newtemp, t_i 入口赋值给 T.place，
	Label1=newLabel，
	Label2=newLabel。
	访问 T 的所有子树后：
	T.code = T1.code || T2.code
	if (T.true!=””)
	T.code=T.code || if T1.alias RELOP T2.alias goto T.Etrue
	|| goto T.Efalse
	else T.code=T.code || if T1.alias RELOP T2.alias goto label1
	|| t_i=0 || goto label2 || label1 ：|| t_i=1 || label2：

AND
T1 第一操作数子树
T2 第二操作数子树
	if (T.Etrue==””) 按基本表达式处理
	else T1.Etrue= newLabel, T2.Etrue= T.Etrue, T1.Efalse= T2.Efalse =T.Efalse;
	T.code=T1.code || T1.Etrue || T2.code

OR T1 第一操作数子树 T2 第二操作数子树	if (T.Etrue==””) 按基本表达式处理 else T1.Etrue=T2.Etrue= T.Etrue,T1.Efalse= newLabel, T2..Efalse =T.Efalse; T.code=T1.code || T1.Efalse || T2.code
NOT T1 操作数子树	f (T.Etrue==””) 按基本表达式处理 else T1.Etrue=T.Efalse, T1.Efalse =T.Etrue; T.code=T1.code：
FUNC_CALL	t_i=newtemp, t_i 入口赋值给 T.place

当 前 结 点 类 型	翻 译 动 作
T1　实参列表子树	T.code=T1.code; 访问 T 的子树，从上至下依次对每个 ARGS 结点 T0，完成实参处理。 T.code= T. code || ARG T01.alias 这里 T01 表示 T0 的第一个孩子，访问 T 的子树后： T.code= T.code || ti=CALL 函数名 f (T.Etrue!="") T.code=T.code || if t_i!=0 goto T.Etrue || goto T.Efalse

表 7.5 其它类结点的中间代码生成

当 前 结 点 类 型	翻 译 动 作
FUNC_DEF T1　返回值类型 T2　函数名与参数 T3　函数体	访问 T 时：T3.Snext=newLabel 访问 T 的所有子树后: T.code=T2.code || T3.code || T3.Snext
FUNC_DEC T1　参数列表 (可空)	访问 T 的所有子树后: T.code=FUNCTION 函数名 if　(T1 非空) T.code=T.code || T1.code
PARAM_LIST T1 形参说明子树 T2 形参列表子树(可空)	访问 T 的所有子树后: T.code=T1.code if　(T2 非空)　T.code=T.code || T2.code
PARAM_DEC T1 形参类型 T2 形参名	访问 T 的所有子树后: T.code=PARAM T2.alias

当 前 结 点 类 型	翻 译 动 作
ARGS T1 实参子树 T2 实参列表子树（可空）	访问 T 的所有子树后： if（T2==NULL）T.code= T1.code else T.code= T1.code || T2.code

7.4. 实验结果

对于测试代码生成的中间代码见表 7.6 所示 ,尤其是数组和结构体所生成的中间代码的形式。

表 7.6 测试代码和中间代码

测试代码	输出结果

测试代码	输出结果

struct test{
int a;
char t;
};
int main(){
int e[10];
int x;
int a;
float b;
char c;
e[1]=1;
b=1.0;
c="a";
if(a==0)
c="b";
else
return 0;
read(a);
write(b);
while(a==1){
a=a+1;
a++;
++a;
a--;
b=1.0;
a||b;
a&&b;
a/b;
a*b;
test.a=1;
if(a==b){
break;
}
else{
continue;
}
}
return 0;
}
	FUNCTION main :
	temp1 := #1
	v5 offset 8 := temp1
	temp2 := #1.000000
	v8 := temp2
	temp3 := #T
	v9 := temp3
	temp4 := #0
	IF v7 == temp4 GOTO label6
	GOTO label7
	LABEL label6 :
	temp5 := #$
	v9 := temp5
	GOTO label5
	LABEL label7 :
	temp6 := #0
	RETURN temp6
	LABEL label5 :
	LABEL label8 :
	LABEL label9 :
	LABEL label12 :
	temp7 := #1
	IF v7 == temp7 GOTO label11
	GOTO label10
	LABEL label11 :
	temp8 := #1
	temp9 := v7 + temp8
	v7 := temp9
	temp10 := #1
	temp11 := v7 + temp10
	v7 := temp11
	temp12 := #1
	temp13 := v7 + temp12
	v7 := temp13
	temp14 := #1
	temp15 := v7 - temp14
	v7 := temp15
	temp16 := #1.000000
	v8 := temp16
	temp17 := v7 \ v8
	temp18 := v7 * v8
	temp19 := #1

测试代码	输出结果
	v1 offset 12 := temp19 IF v7 == v8 GOTO label23 GOTO label24 LABEL label23 : GOTO label10 GOTO label12 LABEL label24 : GOTO label12 GOTO label12 LABEL label10 : temp20 := #0 RETURN temp20

8. 总结

8.1. 实验完成情况

完成内容：

(1) 自定义语言的词法和语法规则。

(2) 设计实现词法分析器。

(3) 设计实现自定义的语法分析器。

(4) 符号表管理和属性计算。

(5) 静态语义分析。

(6) 生成中间代码。

所完成的六个实验，均已实现自定义语言的全部功能。除了完成了实验书上制指定的内容之外，部分实验还加入了自己所构想的一些要求，如加入了全局变量；对于已经完成的实验都实现了结构体和数组；符号表还加入了所属结构；静态语义分析比实验书所要求的还要多一些等等。

8.2. 实验感想

这学期的编译原理实验总体写下来，难度确实比较大。尤其是在理论课的进度没有实验课块的时候，因为对于整体的理论知识缺少整体的架构，导致很多东西理解起来特别花费时间和精力，这也导致了编译原理实验的每一次小实验对我来说都是一个挑战。

老师的指导教程给的很详细，看到了老师也耗费了很大的精力。所以即便实

验时举步维艰，我也在很努力地看完指导文档，并努力地跟这文档编写程序，并加入自己的思考。

刚开始实验时，主要难点在于如何正确理解 flex 和 bison 之间的来回调用和联合，除了使用文档，自己也查阅了很多相关资料和例子。最初对自己所要写的的样子很模糊，以至于不知道该如何下手。但写完了实验 1、2、3 之后，便逐渐地形成了自己所明白、所想写的编译器的样子。前面，主要是定义自己的语言的文法的时候需要很清晰的逻辑，必须地时刻记得自己在写什么；其次，是关于语法树的生成，这也是后面实验的基础。

完成了前三个实验之后，实验 4、5 又是一道坎。符号表的生成时，必须要清楚那些地方需要添加符号进去，变量的哪些属性需要在节点之间传递。语义分析只需要在生成符号表时加入一些错误判断即可。

实验六是最难的部分了，代码量也比较大，也是编译器比较核心的东西。写这个实验的时候的折磨程度也是一流的。虽然后面对编译原理进行整体复习的时候发现其中还是有很多技巧和奥妙的。

编译原理实验是一个具有挑战性的实验，除了锻炼编码能力之外，也很锻炼我的理解能力。此次实验我学会了如何将复杂问题拆分为简单问题，从简单问题入手逐步解决困难问题。

8.3. 展望

经过这个实验才明白原来这看似不经意的一个个编译器的按键背后都隐藏了这么多原理和辛劳，也希望编译原理被更多人所学习。

最后，感谢编译原理实验锻炼了我在编程和学习能力的抗压能力，也提高了

我的编程水平和学习新东西的速度，让我以后能在这条路上走得更远，谢谢编译原理，谢谢老师。