

华中科技大学

实践课程报告

题目： 编译原理实践

课程名称： 编译原理实践

专业班级： CS1409

学 号： U20144808

姓 名： 王林

指导教师： 邵志远

报告日期： 2017-01-14

计算机科学与技术学院

目录

1 选题背景.....	1
1.1 任务.....	1
1.2 目标.....	1
1.3 源语言定义	1
2 实验一 词法分析和语法分析.....	3
2.1 单词文法描述.....	3
2.2 语言文法描述.....	4
2.3 词法分析器的设计	4
2.4 语法分析器的设计	5
2.5 语法分析器实现结果展示.....	6
3 语义分析.....	10
3.1 语义表示方法描述.....	10
3.2 符号表结构定义.....	10
3.3 错误类型码定义.....	11
3.4 语义分析实现技术	11
3.5 语义分析结果展示.....	13
4 中间代码生成.....	16
4.1 中间代码格式定义.....	16
4.2 中间代码生成规则定义.....	17
4.3 中间代码生成过程	19
4.4 中间代码生成结果展示	20
5 结束语	22
5.1 实践课程小结	22
5.2 自己的亲身体会.....	22
参考文献	23
附件：源代码	24

1 选题背景

1.1 任务

主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生对系统软件编写的能力。

本实验分为四个阶段，第一阶段：词法、语法分析，第二阶段：语义分析，第三阶段：生成中间代码，第四阶段，生成目标代码以及目标代码的优化。

1.2 目标

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

1.3 源语言定义

源语言选用 C--，BNF 描述为

```
<程序> ::= <分程序>

<分程序> ::= [<变量说明部分>] [<结构体说明部分>] [<函数说明部分>]

<类型> ::= int|float|struct<标识符>

<变量说明部分> ::= <类型> <变量定义> { , <变量定义> } ;

<变量定义> ::= <标识符> { = <立即数> }

<立即数> ::= <integer> | <float>

<函数说明部分> ::= <类型> <函数声明> { <语句> { , <语句> } } ;

<函数声明> ::= <标识符> ( <类型> <标识符> { , <类型> <标识符> } )

<结构体说明部分> ::= <结构体说明> { <结构体说明> }

<结构体说明> ::= struct <标识符> { <变量说明部分> }

<语句> ::= <变量说明> | <赋值语句> | <条件语句> | <当型循环语句> | <函数调用语句> | <复合语句> | <空语句> ;

<赋值语句> ::= <id> = <表达式>

<复合语句> ::= '{ ' <语句> { ; <语句> } '}'

<空语句> ::= ε
```

〈条件〉 表达式	::= 〈表达式〉 〈关系运算符〉 〈表达式〉 〈单目运算符〉〈表达式〉
〈表达式〉	::= [+ -] 〈项〉 { 〈加减运算符〉〈项〉 }
〈项〉	::= 〈因子〉 { 〈乘除运算符〉〈因子〉 }
〈因子〉	::= 〈id〉 〈立即数〉 ‘(’ 〈表达式〉 ‘)’
〈单目运算符〉	::= ! ‘ ’ ~
〈加减运算符〉	::= + -
〈乘除运算符〉	::= * /
〈关系运算符〉	::= = # < <= > >=
〈条件语句〉	::= if 〈条件〉 then 〈语句〉
〈过程调用语句〉	::= 〈id〉(〈立即数〉 〈标识符〉{, 〈立即数〉〈标识符〉})
〈当型循环语句〉	::= while 〈条件〉 ‘{’ 〈语句〉 ‘}

2 实验一 词法分析和语法分析

2.1 单词文法描述

词法分析借助 flex 词法分析器完成，词法分析基于的理论基础是正则表达式和有限状态自动机，本次实验中的正则表达式如下：

表 2.1 正则表达式	
正则表达式	单词
[1-9]+[0-9]* 0	Digit（整数）
[+-]?({digit}*\.?{digit}+ {digit}+\.)	NUMBER（实数）
[A-Za-z_]	Char（符号）
{char}+	WORD（标识符）
[\t\v\n\0]	SPACE（空格）
while	while 关键字
int	int 关键字
float	float 关键字
[&][&]	&&与
[][]	或
[!]	! 非
if	if 关键字
else	else 关键字
for	for 关键字
[+]	+ 运算符
[-]	- 运算符
[*]	* 运算符
[/]	/ 运算符
\((运算符
\)) 运算符
\[[运算符
\]] 运算符
\{	{ 运算符
\}	} 运算符
\\, \\; \\' \\"	界符
int float void	类型
\\= \\< \\> \\!= \\<\\= \\>\\= \\=\\= ({PLUS} {PLUS}) ({SUB} {SUB})	双目运算符
{char} (_ {char}) ({char} {digit} _)*	常数

{IF} {ELSE} {FOR} {INT} {FLOAT} {CHAR} begin end break return	关键字
("/*.*") ("/*"(. \n)*"*/")	注释
{SR} {SL} {LR} {LL} {MR} {ML}	边界
[~] [@] [\$] [。] [,] [;]	错误符号

2.2 语言文法描述

语法分析借助语法分析器 bison 完成，采用自底向上的分析方法，书写文法的产生式，通过语法分析程序分析，产生式为：

表 2.2 文法产生式

产生式左部	产生式右部
Program	xtDefList
ExtDefList	ExtDef ϵ
ExtDef	Specifier Specifier SEMI Specifier FunDec error SEMI
ExtDecList	VarDec VarDec error SEMI
Specifier	TYPE StructSpecifier
StructSpecifier	STRUCT OptTag LC DefList RC STRUCT Tag
OptTag	ϵ ID
Tag	ID
VarDec	ID VarDec LB INT RB
FunDec	ID LP VarList RP ID LP RP
VarList	ParamDec ParamDec
ParamDec	Specifier VarDec
CompSt	LC DefList StmtList RC error RC
StmtList	ϵ Stmt StmtList
Stmt	Exp SEMI CompSt RETURN Exp SEMI IF LP Exp RP Stmt IF LP Exp RP Stmt ELSE Stmt WHILE LP Exp RP Stmt error SEMI
DefList	ϵ Def DefList
Def	Specifier DecList SEMI Specifier SEMI
DecList	Dec Dec COMMA DecList
Dec	VarDec VarDec ASSIGN Exp error SEMI
Exp	Exp ASSIGN Exp Exp AND Exp Exp OR Exp Exp RELOP Exp Exp PLUS 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 LB Exp RB Exp DOT ID ID INT FLOAT
Args	Exp Exp

上表即 c—的文法产生式，借助 bison，实现语法分析器。

2.3 词法分析器的设计

借助词法分析工具 flex，创建 lexical.l 文件，写入正则表达式，经过 flex 的编译，生成 lex.yy.c 文件，再经过编译，即可生成此法分析器可执行程序。

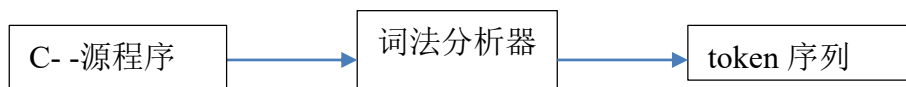


图 2.1 词法分析器设计图

如果有错，就报出错误信息，如果没错，继续分析。

2.4 语法分析器的设计

借助语法分析工具 `bison`,将产生式规则写入 `bison.y` 文件中，调用词法分析程序，对词法分析的结果进一步语法分析，分析是否满足某种语法规则。

设计图为：

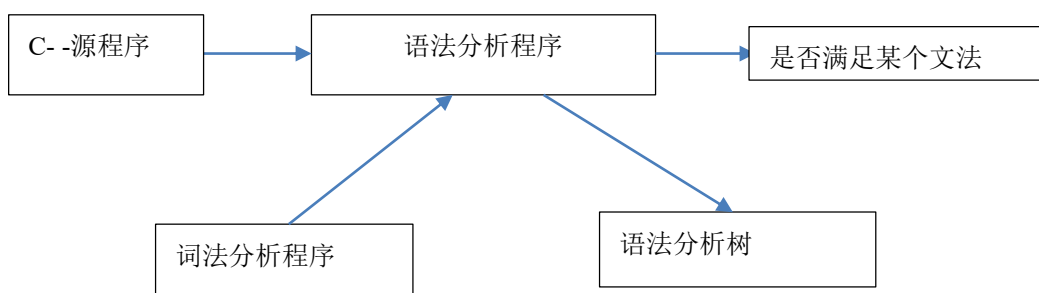


图 2.2 语法分析器设计图

如果满足，就继续分析，同时生成一个新的语法节点，并加入到语法树中，如果有错，报错，然后做回复错误处理，继续分析。

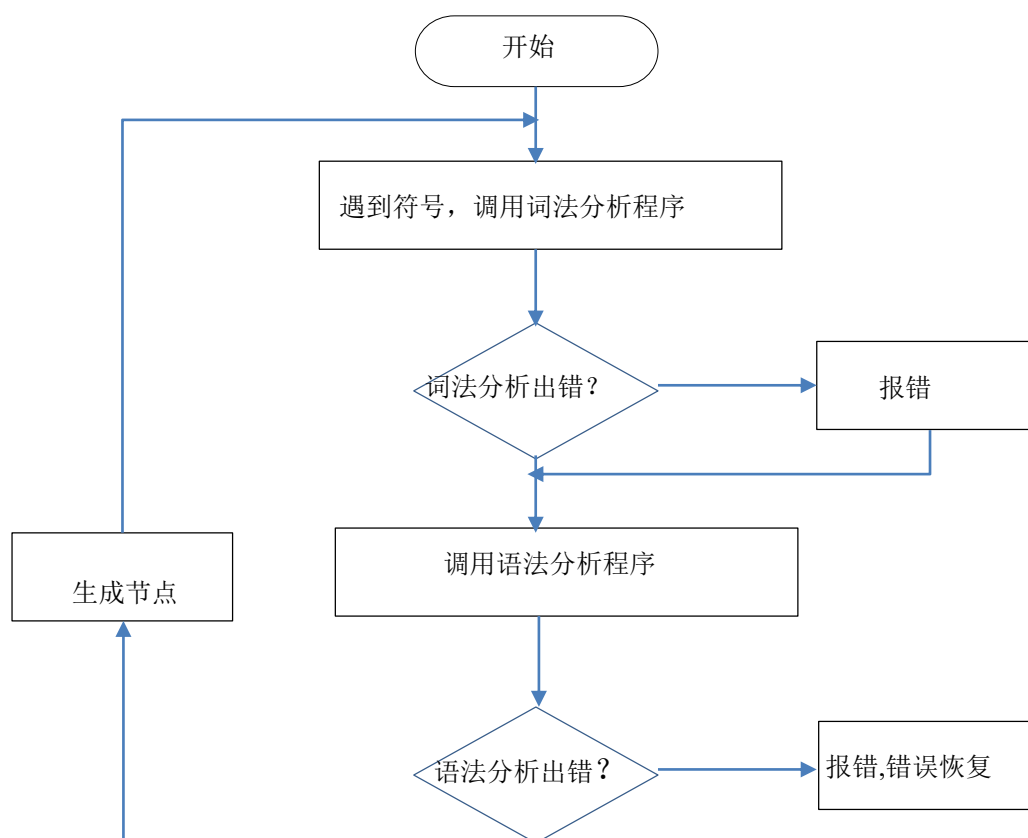


图 2.3 语法分析器设计流程图

语法分析设计图和流程图如上所示，按照流程图设计出语法分析程序对源程序进行分析。

2.5 语法分析器实现结果展示

对于 C- -源程序，先进行词法分析，调用词法分析程序，观察打印输出的结果。

```

└─[0] <git:(master 10dac0d) > ./a <test.c
Line 6: NOTE /*****
*****
> File Name: test.c
> Author: strawberrylin
> Github: https://github.com/strawberrylin
> Created Time: 2016年11月01日 星期二 01时12分59秒
*****
Line 8: Headinclude #include<stdio.h>
Line 9: Keyword void
Line 9: Identifier main
Line 9: KuoHu ()
Line 9: Bounder {
Line 10: Keyword int
Line 10: Identifier a
Line 10: Bounder [
Line 10: Num 100
Line 10: Bounder ]
Line 10: Dot ;
Line 11: Keyword int
Line 11: Identifier a
Line 11: Dot ,
Line 11: Identifier b
Line 11: Dot ,
Line 11: Identifier c
Line 11: Dot ;
Line 12: Keyword int
Line 12: Identifier j
Line 12: Operator =
Error type A at line 12 : Mysterious character : '~'
Line 12: Identifier i
Line 12: Dot ;
Line 13: Keyword float
Line 13: Identifier d
Line 13: Operator =
Line 13: Num 1
Line 13: Num 3
Line 13: Dot ;
Line 14: Keyword float
Line 14: Identifier f
Line 14: Operator =
Line 14: Num 5
Line 14: Num 123
Line 14: Dot ;
Line 15: Identifier func
Line 15: KuoHu ()
Error type A at line 15 : Mysterious character : '~'
Error type A at line 15 : Mysterious character : '~'
Error type A at line 15 : Mysterious character : '~'
Line 16: Identifier a
Line 16: Operator =

```

图 2.4 词法分析结果图

如图是词法分析的结果，首先识别出注释，然后识别出关键字，识别出标识符，并且输出单词所在的行号。如果词法分析遇到错误，则打印出对应的错误和错误所在行号。

对于 C-程序，进行语法分析，调用语法分析程序，输出语法分析的结果，打印出语法分析树。

```

└─[0] <git:(master 10dac0d*) > ./test <testfile.c
>print the syntax tree:
Program(1)
  ExtDefList(1)
    ExtDef(1)
      Specifier(1)
        TYPE:int
      FunDec(1)
        IDENTIFIER:main
        RP(1)
        LP(1)
        CompSt(1)
          RP(1)
          DefList(2)
            Def(2)
              Specifier(2)
                TYPE:int
              Declist(2)
                Dec(2)
                  VarDec(2)
                    IDENTIFIER:a
                  COMMA(2)
                  Declist(2)
                    Dec(2)
                      VarDec(2)
                        IDENTIFIER:b
                SEMI(2)
            StmtList(3)
              Stmt(3)
                IF(3)
                RP(3)
                exp(3)
                  exp(3)
                    IDENTIFIER:a
                  RELOP(3)
                  exp(3)
                    IDENTIFIER:b
                LP(3)
                Stmt(4)
                  exp(4)
                    exp(4)
                      IDENTIFIER:a
                    ASSIGN(4)
                    exp(4)
                      exp(4)
                        IDENTIFIER:a
                      SUB(4)
                      exp(4)
                        IDENTIFIER:b
                  SEMI(4)
            LP(5)
syntax tree over

```

图 2.5 语法分析结果图

如图 2.5 所示，输出结果为针对某一个 c-源程序进行语法分析的结果，打印出对应的语法树，没有语法错误，没有报错。

修改 c-源程序，删除';'，再次分析结果为：

```
—[strawberrytin@strawberrytin-X550JK] - [~/Gitrepository/Co  
—[0] <git:(master 10dac0d*) > ./lab <testfile.c  
syntax error: Line 4 : column 2 :error: "}"  
syntax error: Line 5 : column 1 :error: ""
```

图 2.6 语法分析结果图

如图，报错，并继续分析下一行的错误。

经过测试，语法分析程序和此法分析程序测试通过。

3 语义分析

3.1 语义表示方法描述

目前被广泛使用的用于语义分析的理论工具叫做属性文法，他的核心思想是，为上下文无关文法中的每一个终结符或非终结符赋予一个或多个属性值。

以属性文法为基础可以衍生出一种非常强大的翻译模式，称之为语法制导翻译。在语法制导翻译中，人们把属性文法中的属性规则用计算属性值的语义动作来表示，并用花括号括起来，它们可被插入到产生式右部的任何合适的位置上，这是一种语法分析和语义动作交替的表示方法。

采用语法分析和语义动作交替的表示方法，对符号的属性值进行分析。

3.2 符号表结构定义

语法树节点定义为：

```
struct syntax_node{           //节点结构
int line;                     //行号
char *name;                   //语法单元的名字
int id_no;                    //id 号
SYN_KIND syn_kind;           //语法单元的类型
char *syntax_value;           //语义值
char *idtype;                 //int or float
struct args *args;            //函数参数链表
struct scope *scope;          //作用域
struct syntax_node *l;        //左子树指针
struct syntax_node *r;        //右子树指针
union{                        //constant 共用联合体存放值
    int int_dex;
    float float_dex;
};
};
```

符号表采用的数据结构是线性链表，定义为：

```
struct symbal_node{
SYM_KIND sym_kind;           //variable, array, function, struct
int line;
int id_no;                   //id 号
char *name;                  //符号名, 与 syntax_value 对应
char *idtype;                // idtype, int or float
int size;
int num_arg;                 //函数有参数个数
struct symbal_node *next;    //指向下一个节点的指针
};
```

符号节点包含的属性值有行号，符号类型，符号名，符号类型名，函数参数，指向下一节点的指针等...,符号节点和语法树节点部分属性对应一致。

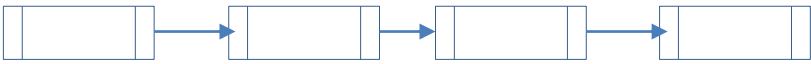


图 3.1 符号表结构示意图

采用符号表存储符号，每个节点存储一个符号，同时存储符号的属性值，在链表中增删操作时，复杂度都是 $O(1)$ ，链表中查询时，时间复杂度是 $O(n)$ ，符号表采用链表实现，结构简单，编程容易实现。

3.3 错误类型码定义

表 3.1 语义错误码表

错误码	错误类型
1	变量使用时未定义
2	函数调用时未定义
3	变量重复定义
4	函数重复定义
5	赋值号两边的表达式类型不匹配
6	赋值号右边出现一个只有右值的表达式
7	操作数类型不匹配或操作数类型与操作符不匹配
8	return 值类型与函数定义的类型不一致
9	函数调用时实参与形参不一致
10	对非数组类型变量使用 [...] 操作
11	对普通变量使用 (...) 或 () 函数调用
12	数组访问操作符 [...] 中出现非整数
13	对非结构体变量使用 . 操作
14	访问结构体中未定义的成员
15	结构体中成员名重复
16	结构体的名字与前面定义过的结构体或变量的名字重复
17	直接使用未定义过的结构体来定义变量
18	函数重复声明

语义分析阶段，可能分析出的语义错误如上表所示。

3.4 语义分析实现技术

1. 符号表绑定在块作用域的入口，实现作用域的嵌套

利用语法分析的结果，在语法分析没有出错的情况下，构造语法树，然后在每个作用域的入口节点上绑上当前作用域的符号表，这个符号表是静态存在的符号表，随语法树一直存在，将上一级作用的符号表压栈，栈是动态的，进入新的作用域时将上一级的符号表压栈，退出下一级的作用域时，符号表退栈，作为当前作用域，实现作用域的动态变化，即实现了作用域的嵌套，符号表与语法树的

关系如图：

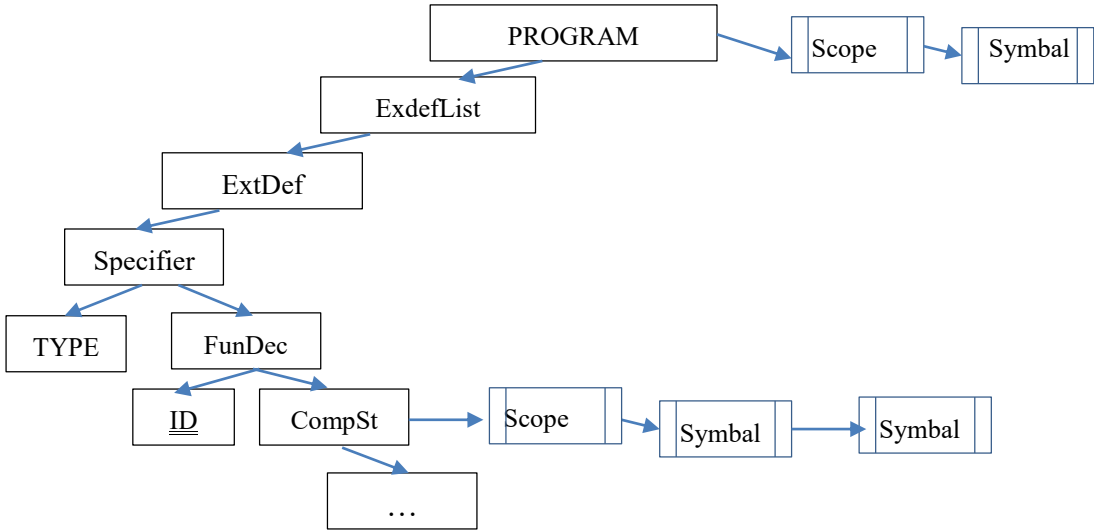


图 3.2 语法树绑定符号表结构示意图

动态的符号表栈如图所示：

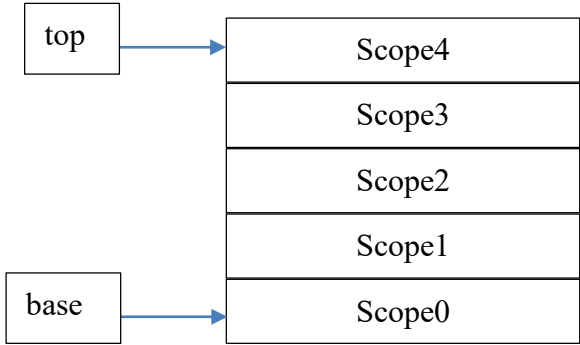


图 3.3 语法树动态符号符号表栈结构示意图

采用静态符号表保存符号信息，方便后面的查询和相关操作，使用动态的符号表实现作用域的嵌套。(Scope 代表作用域，Symbal 代表符号表)

2. 函数参数处理

函数参数的处理类似作用域的处理，当遍历语法树遍历到函数节点时，用链表保存函数的参数，同时将链表绑在函数节点上，从而实现函数参数的处理，当需要查询函数参数时，只需遍历到对应的函数节点，然后遍历函数的参数链表即可。

3. 符号属性值的传递

对于每一个符号，属性值包括符号名，符号所在行，符号类型，符号类型名，符号的值等等，这些属性里面有的属性经过词法分析后不一定知道。通过多趟遍历语法树完善语法树个节点的属性值。比如符号的类型，通过其父节点传递下来，比如说符号的值，通过直接点传递上来；通过这种方式，完善符号的属性值，在进行语义分析时，更加方便。

4. 向下递归的进行语义分析

列举一个符号类型是否匹配判断的函数处理过程：

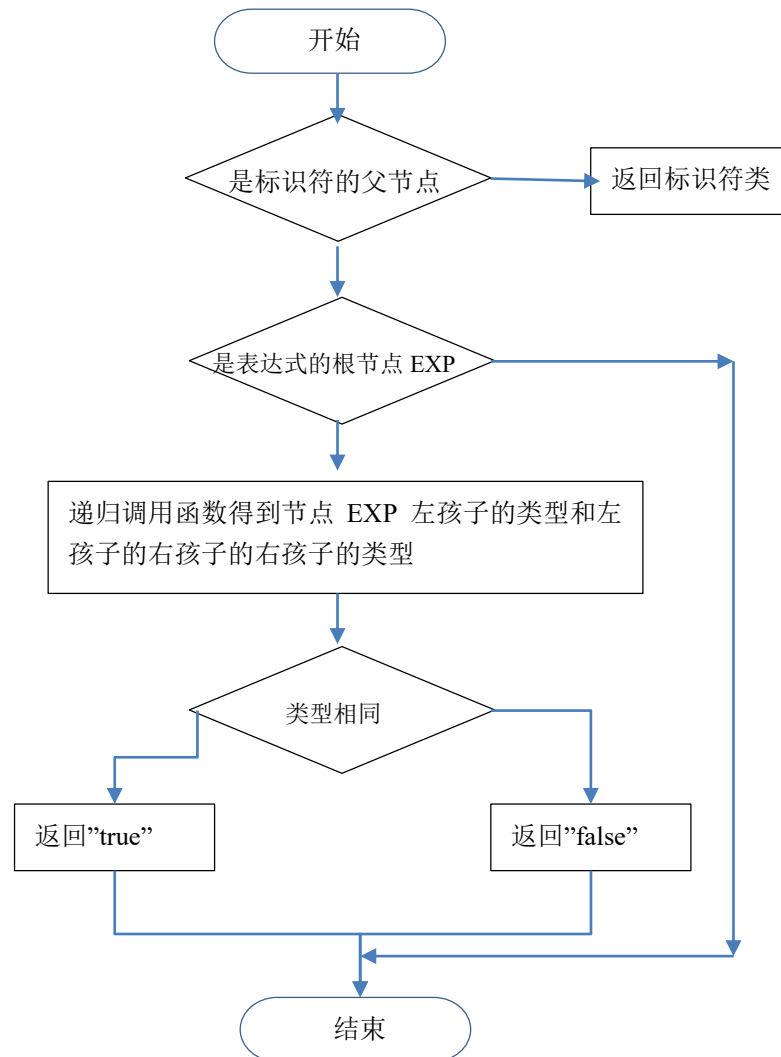


图 3.4 `getType` 函数流程示意图

如图 3.4 所示，在获取表示符的类型时，使用递归函数的设计方法，使标识符的属性自底向上传递，从而判断两个操作数的类型是否一致，如果不一致，则报错。

3.5 语义分析结果展示

测试用例 1：进行语义分析的源程序为：

```
int test(int m,int n){
    int a,b,c;
    a = b + c;;
}
int main(){
    int m,n;
    test(m,n);
    return 0;
```



```

}
调用语义分析程序分析的结果为:
start the symbal table func:
gloable scope
a new block scope :
name      idtype      lineno      kind      id_no      args
a          int         8           1         2
b          int         8           1         3
c          int         8           1         4
quit the scope

a new block scope :
name      idtype      lineno      kind      id_no      args
m          int         12          1         6
n          int         12          1         7
quit the scope

name      idtype      lineno      kind      id_no      args
test      int         7           3         1         2
main      int         11          3         5         0
quit the program
MakeSymbal over

```

图 3.5 语义分析效果图 1

如图 3.5，语义分析，test 块作用域下有 a、b、c 三个局部变量，类型都是整形，main 块作用域下有两个变量 m、n，整形；在全局作用域下，有两个函数：test、main,都是整形。

测试用例 2：源程序为：

```

int test(int m,int n){
    int a;
    float b, c;
    a = b + c;;
}

```

```

start the symbal table func:
gloable scope
a new block scope :
name      idtype      lineno      kind      id_no      args
a          int         8           1         2
b          float        9           1         3
c          float        9           1         4
quit the scope

name      idtype      lineno      kind      id_no      args
test      int         7           3         1         2
quit the program
Error Type 5 at line 10 : Type mismatched for assignments;

```

图 3.6 语义分析效果图 2

如图 3.6，源程序中存在明显错误，b+c 和 a 的类型不一致，而予以分析的结果为：Type mismatched for assignments;分析的结果正确，测试通过。

4 中间代码生成

4.1 中间代码格式定义

本实验中，中间代码采用三地址码，即 $(:=, t1, t2, t3)$ 的形式，中间代码的形式以及操作规范如下表所示：

表 4.1 中间代码的形式以及操作规范

#	语法	描述
1	LABEL x:	定义标号 x
2	FUNCTION f:	定义函数 f
3	x:=y	赋值操作
4	x:=y+z	加法操作
5	x:=y-z	减法操作
6	x:=y*z	乘法操作
7	x:=y/z	除法操作
8	x:=&y	取 y 地址赋值给 x
9	x:=*y	取 y 地址的内容值给 x
10	*x:=y	给 x 地址的内容赋值为 y
11	GOTO x	无条件跳转 x
12	IF x relop y GOTO z	条件转移
13	RETURN x	退出当前函数并返回 x
14	DEC x size	内存空间申请
15	ARG x	传实参 x
16	x:= CALL f	函数调用
17	PARAM x	函数参数声明
18	READ x	从控制台读取 x 的值
19	WRITE x	向控制台打印 x

如表 4.1 所示，中间代码的形式以及操作规范的表示。对前面阶段生成的语法树遍历，对于每一条语句进行转换操作。

本次实验中，采用线性中间代码的形式，这种结构优点在于表示简单、处理高效。所以本次试验中，中间代码的存储通过线性链表实现，具体程序定义如下：

```
//中间代码结构定义
typedef struct Operand_ * Operand;

struct Operand_ {
    enum{CONSTANT,VARIABLE,ADDRSS} kind;
    union{
        int var_no;
        int valuei;
        float valuef;
    };
};
```

```

    } u;
};

struct InterCode{
enum{ASSIGNOP,ADDOP,SUBOP,MULOP,DIVOP,RELOPS,GOTOOP,LABLE,FUNDEC,FUN
CALL,VARLIST}op_kind;
    union{
        struct {Operand right,left;} assign;
        struct {Operand result,op1,op2;} binop;
        struct {Operand right,left;char * op;int lable_true;} relop;
        int goto_lable;
        char *funcname;
        int arg_no;
    }u;
};

struct InterCodes{
    struct InterCode *code;
    struct InterCodes *prev,*next;
};

```

将中间代码存储在线性链表中，比较灵活，可以有效的进行增删操作。

4.2 中间代码生成规则定义

基本表达式生成中间代码的规则如下表：

表 4.2 基本表达式中间代码生成规则表

表达式	生成规则
INT	value = get_value(INT) return [temp:=#value]
ID	t1 = new_temp() return [temp := variable.name]
Exp1 Assign Exp2 (Exp1->ID)	t1 = new_temp() code1 = translate_Exp(Exp2, sym_table, ID) code2 = [variable.name :=t1]+ [temp:=variable.name] return code1 + code2
Exp1 PLUS Exp2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp1, sym_table, ID) code2 = translate_Exp(Exp2, sym_table, ID) code3 = [temp:= t1 + t2] return code1 + code2+code3
MINUS Exp1	t1 = new_temp() code1 = translate_Exp(Exp1, sym_table, t1)

(续)

续上页表 4. 2

表达式	生成规则
MINUS Exp1	code2 = [temp:=#0-t1] return code1 +code2
Exp1 RELOP Exp2	lable1 = new_temp()
NOT Exp1	lable2 = new_temp()
Exp1 AND Exp2	code0 = [temp :=#0]
Exp1 PR Exp2	code1 = translate_Exp(Exp, lable1, lable2, sym_table) code2 = [LABEL lable1]+[temp := #1] return code0+code1code2+[LABEL lable2]

语句的中间代码生成规则如下表:

表 4. 3 语句中间代码生成规则表

表达式	生成规则
Exp SEMI	return translate_Exp(Exp, sym_table, NULL)
CompSt	return translate_Exp(CompSt, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [RETURN t1] return code1+code2
IF LP Exp RP Stmt1	lable1 = new_lable() lable2 = new_lable() code1=translate_Cond(Exp, lable1, lable2, sym_table) code2 = translate_Stmt(Stmt1, sym_table) return code1+[LABEL lable1]+code2+[LABEL lable2]
IF LP Exp RP Stmt1 ELSE Stmt2	lable1 = new_lable() lable2 = new_lable() lable3 = new_lable() code1=translate_Cond(Exp, lable1, lable2, sym_table) code2 = translate_Stmt(Stmt1, sym_table) code3 = translate_Stmt(Stmt2, sym_table) return code1+[LABEL lable1]+code2+[GOTO lable3]+[LABEL lable2]+code3+[LABEL lable3]
WHILE LP Exp RP Stmt1	lable1 = new_lable() lable2 = new_lable() lable3 = new_lable() code1=translate_Cond(Exp, lable2, lable3, sym_table) code2 = translate_Stmt(Stmt1, sym_table) return [LABEL lable1]+code1+[LABEL lable2]+code2+[GOTO lable1]+[LABEL lable3]

函数调用的中间代码生成规则如下表：

表 4.4 函数调用中间代码生成规则表

表达式	生成规则
ID LP RP	return [temp:=CALL function.name]
ID LP Args RP	code1=translate_Args(Args, sym_table, arg_list) for i=1 to n code2=code2+[ARG arg_list[i]] return code1+code2+[temp=CALL function.name]

按照上表所给的规则，遍历实验前面阶段生成的语法树，即可生成对应的中间代码。

4.3 中间代码生成过程

中间代码中的生成流程如下：

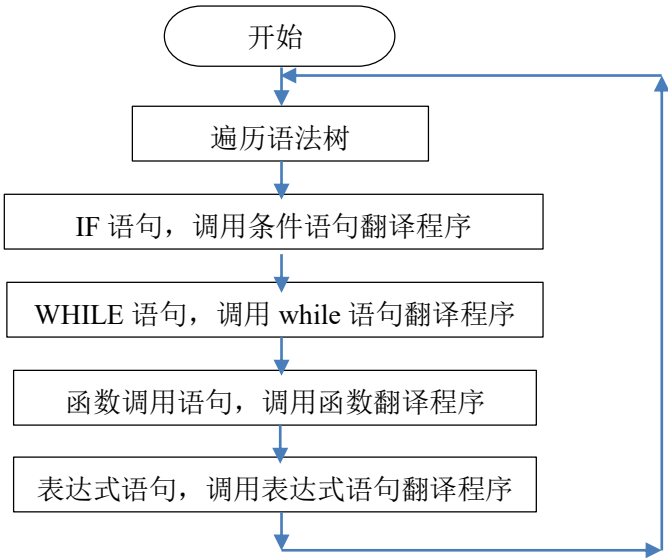


图 4.1 中间代码生成过程流程图

1. IF 语句-条件表达式的翻译

当遍历到 IF 节点时，首先判断是否有 ELSE 语句，如果有采取 IF 语句翻译的有 ELSE 的翻译模式，将产生的中间代码存入中间代码线性链表中如果 IF 语句没有对应的 ELSE 语句，在采用没有 ELSE 的翻译规则，然后将翻译的中间代码存入中间代码线性链表中。

2. 表达式语句翻译

遍历到 Exp 节点时，如果右孩子是 SEMI 节点，则直接调用表达式翻译程序进行翻译。

3. 函数语句翻译

函数语句包括函数定义和函数调用语句，如果遍历到 Expp 节点，即左孩子是 ID 的 Exp 节点，且左孩子的右孩子是 LP，则是函数调用的翻译；如果遍历到 FunFec 节点，则是函数定义，调用函数定义的翻译程序。

其他语句的翻译类似，同样是遍历到对应的节点，然后调用对应的翻译程序进行翻译，产生中间代码，然后存入中间代码线性链表中。

4.4 中间代码生成结果展示

测试用例 1:

```
#include<stdio.h>
```

```
int main(){  
    int a,b,c;  
    b = 1;  
    c = 2;  
    a = b + c;  
    a = b - c;  
    a = b * c;  
    a = b / c;  
    return 0;  
}
```

此项测试针对表达式的翻译，生成的测试结果为：

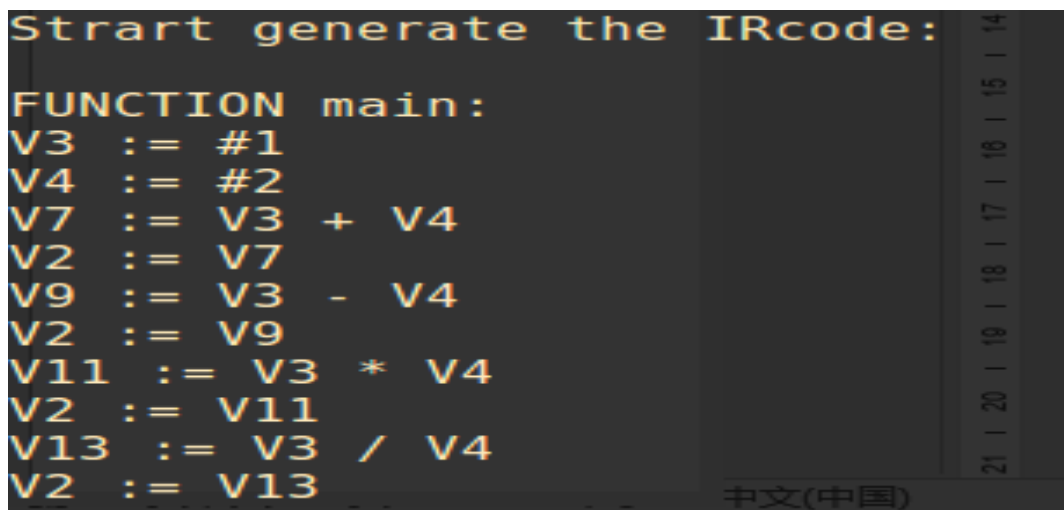


图 4.2 中间代码生成测试效果图 1

如图 4.2 所示，生成的中间代码和源程序对比，逻辑正确，中间代码能够正确表示源程序，测试通过。

测试用例 2

```
#include<stdio.h>
```

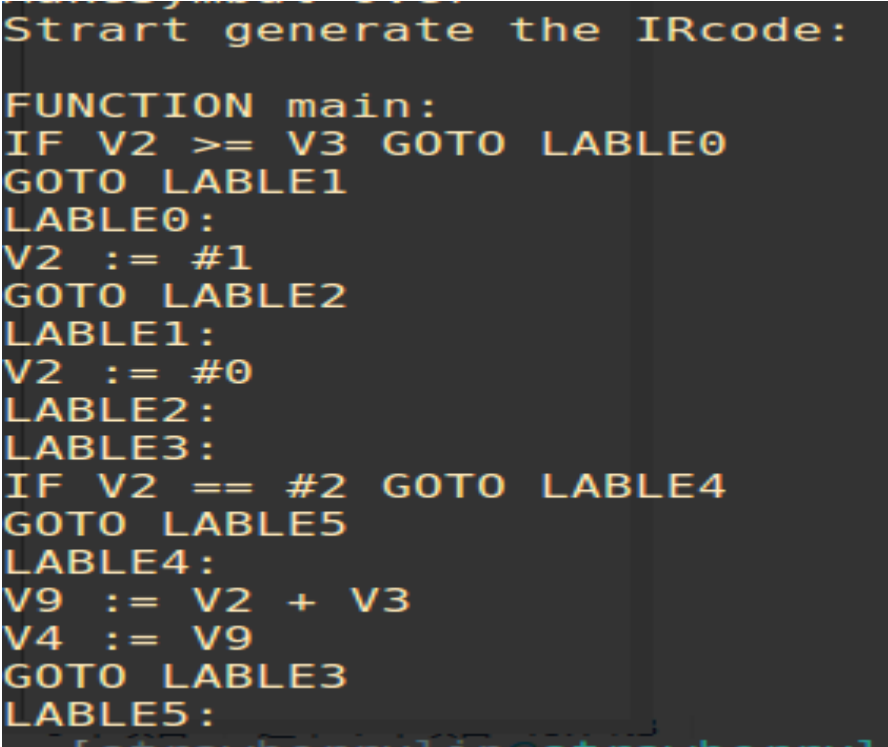
```
int main(){  
    int a,b,c;  
    if(a >= b) a = 1;  
    else a = 0;  
    while(a == 0){  
        c = a + b;  
    }
```

```

    }
    return 0;
}

```

此项测试针对 IF 条件语句以及 WHILE 语句的翻译，并且观察是否支持多重嵌套。测试结果如图：



```

Strart generate the IRcode:
FUNCTION main:
IF V2 >= V3 GOTO LABEL0
GOTO LABEL1
LABEL0:
V2 := #1
GOTO LABEL2
LABEL1:
V2 := #0
LABEL2:
LABEL3:
IF V2 == #2 GOTO LABEL4
GOTO LABEL5
LABEL4:
V9 := V2 + V3
V4 := V9
GOTO LABEL3
LABEL5:

```

图 4.3 中间代码生成测试效果图 2

如图所示，成功生成中间代码，元源程序对比，if-else 语句正确翻译，逻辑正确，while 语句正确翻译，逻辑正确。并且支持多重嵌套，则测试通过。

5 结束语

5.1 实践课程小结

本次实验分为四个阶段，我独立完成了前面三个阶段，第四阶段由于时间原因只完成部分工作。

前面三个阶段实验：词法分析和语法分析，语义分析，中间代码生成实验，第一阶段的实验主要借助工具完成，借助 flex 词法分析器可 bison 语法分析器来实现自己的词法分析器和语法分析器，比较方便，而且生成的词法分析器和语法分析器比较高效，比较漂亮。第二阶段实验语义分析需要借助第一阶段的实验生成的语法树来完成，通过遍历语法树将程序中的符号加入符号表中，然后根据符号表和语法树来进行语义的分析，结合上下文进行分析，检测出程序中的语义错误；第三阶段完成中间代码的生成，借助第一阶段的语法树和第二阶段的符号表，对每一种语句进行翻译，生成中间代码并存入线性链表中。

至此，前三个阶段的实验任务基本完成，生成的编译程序能够对 c- 程序进行编译，词法分析可以检测程序中的词法错误，语法分析检测程序中的语法错误，在没有词法和语法错误的情况下进行语义分析，语义分析结合上下文进行分析，检查程序中的语义错误，在没有语义错误的情况下，进行中间代码生成，依照一定的翻译规则，对每一条语句进行翻译，生成中间代码并存储。

本次实验：完成了词法分析程序，语法分析程序，语义分析程序，中间代码生成程序。最后的编译程序正常工作且结果正确。

5.2 自己的亲身体会

本次实验我只完成了前面 3 个阶段，因为时间不够。本学期花在编译原理实践的时间应该是最多的，实验比较难，尤其是开始阶段根本不知道做什么，不过在后面逐渐对实验目标更加明确以及方法了解得更多后，实验的效率也在提升。

通过本次实验，对吱声的代码编写能力有一定提升，尤其是对多种数据结构的应用以及部分算法的实现。通过本次编译实验，对编译器的了解更加深刻，更加了解程序的编译过程，对源代码到目标代码的过程有了更深的了解，有助于在平时的代码编写过程中编写更加高质量的代码。

总的来说，收获很多。

参考文献

- [1] 吕映芝等. 编译原理(第二版). 北京: 清华大学出版社, 2005
- [2] 胡伦俊等. 编译原理(第二版). 北京: 电子工业出版社, 2005
- [3] 王元珍等. 80X86 汇编语言程序设计. 武汉: 华中科技大学出版社, 2005
- [4] 王雷等. 编译原理课程设计. 北京: 机械工业出版社, 2005
- [5] 曹计昌等. C 语言程序设计. 北京: 科学出版社, 2008

附件：源代码