

目录

1 选题背景.....	1
2 系统关键定义.....	2
2.1 单词文法描述.....	2
2.2 语句文法描述.....	3
2.3 符号表结构描述.....	5
2.4 错误类型码描述.....	5
2.5 中间代码描述.....	6
2.6 目标代码描述.....	8
3 系统设计与实现.....	10
3.1 编译程序符号表结构.....	10
3.2 编译程序报错功能.....	10
3.3 词法语法分析器（实验一）	11
3.4 语义分析（实验二）	13
3.5 中间代码生成功能（实验三）	17
3.6 汇编代码生成功能（实验四）	19
4 系统测试与评价.....	22
4.1 测试用例.....	22
4.2 正确性测试.....	23
4.3 报错功能测试.....	25
4.4 系统的优点.....	26
4.5 系统的缺点.....	26
5 实验小结或体会.....	27
参考文献	28
附件：源代码	29

1 选题背景

本次课程设计是构造一个高级语言的子集的编译器，目标代码可以是汇编语言也可以是其他形式的机器语言。按照任务书，实现的方案可以有很多种选择。

可以根据自己对编程语言的定义选择实现语言的特定功能。

编译器的语法和词法分析采用课程的课堂实验的结果，重点在语义分析、符号表结构设计、中间代码、目标代码存储结构设计、代码优化等阶段的实现。

课设的任务主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高自己对系统软件编写的兴趣。

总体学习目标：

1. 熟悉编译程序的总体结构
2. 熟悉编译程序各组成部分及其任务
3. 编译过程各阶段所要解决的问题及其采用的方法和技术
4. 掌握关键算法的工作原理

能力要求：

1. 掌握程序变换基本概念、问题描述和处理方法
2. 增强理论结合实际能力
3. 培养“问题→形式化描述→计算机化”的问题求解过程
4. 使学生在系统级上认识算法和系统的设计，培养系统能力

本次实验是基于实验指导提供的代码，进行自己修改完成课题要求。因此会与实验指导有较高重合度。

2 系统关键定义

2.1 单词文法描述

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

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

INT → 整型常量

FLOAT → 浮点型常量

ID → 标识符

ASSIGNOP → =

RELOP → > | >= | < | <= | == | !=

PLUS → +

MINUS → -

STAR → *

DIV → /

AND → &&

OR → ||

NOT → !

TYPE → int | float

RETURN → return

IF → if

ELSE → else

WHILE → while

SEMI → ;

COMMA → ,

SEMI → ;

LP → (

RP →)

LC → {

RC → }

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

2.2 语句文法描述

语法分析采用生成器自动化生成工具 GNU Bison（前身是 YACC），该工具采用了 LALR（1）的自底向上的分析技术，完成语法分析。通常语义分析是采用语法制导的语义分析，所以在语法分析的同时还可以完成部分语义分析的工作，在 Bison 文件中还会包含一些语义分析的工作。Bison 程序的扩展名为.y。

（1）语义值的类型定义

mini-c的文法中，有终结符，如ID表示的标识符，INT表示的整常数，IF表示关键字if，WHILE表示关键字while等；同时也有非终结符，如ExtDefList表示外部定义列表，CompSt表示复合语句等。每个符号（终结符和非终结符）都会有一个属性值，这个值的类型默认为整型。实际运用中，值得类型会有些差异，如ID的属性值类型是一个字符串，INT的属性值类型是整型。

（2）终结符定义

在Flex和Bison联合使用时，需要做的是在parser.y中的%token后面罗列出所有终结符(单词)的种类码标识符，如：

```
%token ID, INT, IF, ELSE, WHILE
```

这样就完成了定义终结符ID、INT、IF、ELSE、WHILE。接着可使用命令：**bison -d parser.y** 对语法分析的Bison文件parser.y进行翻译，当使用参数**-d**时，除了会生成语法分析器的c语言源程序文件parser.tab.c外，还会生成一个头文件parser.tab.h，在该头文件中，将所有的这些终结符作为枚举常量，从258开始，顺序编号。这样在lex.l中，使用宏命令 `#include "parser.tab.h"`，就可以使用这些枚举常量作为终结符（单词）的种类码返回给语法分析程序，语法分析程序接收到这个种类码后，就完成了读取一个单词。

（3）非终结符的属性值类型说明

对于非终结符，如果需要完成语义计算时，会涉及到非终结符的属性值类型，这个类型来源于（1）中联合的某个成员，可使用格式：`%type <union的成员名> 非终结符`。例如parser.y中的：

```
%type <ptr> program ExtDefList
```

这表示非终结符ExtDefList属性值的类型对应联合中成员ptr的类型，在本实验中对一个树结点的指针。

(4) 优先级与结合性定义。

例如对表达式Exp，其部分语法规则有：

$$\text{Exp} \rightarrow \text{Exp} + \text{Exp} \quad | \quad \text{Exp} - \text{Exp} \quad | \quad \text{Exp} * \text{Exp} \quad | \quad \text{Exp} / \text{Exp}$$

在文法介绍时，明确过该文法是二义性的，所以需要通过优先来解决二义性问题。

最后就是条件语句的嵌套时的二义性问题的解决发生，参见参考文献[2]中的解决方法。

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

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

规则后面{}中的是当完成归约时要执行的语义动作。规则左部的Exp的属性值用\$\$表示，右部有2个Exp，位置序号分别是1和3，其属性值分别用\$1和\$3表示。完成建立一个树结点，这里的语义动作是将建立的结点的指针返回赋值给规则左部Exp的属性值，表示完成此次归约后，生成了一棵子树，子树的根结点指针为\$\$，根结点类型是ASSIGNOP，表示是一个赋值表达式。该子树有2棵子树，第一棵是\$1表示的左值表达式的子树，在mini-c中简单化为只要求ID表示的变量作为左值，第二棵对应是\$3的表示的右值表达式的子树，另外yylineno表示行号。

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

在语法分析阶段，一个很重要任务就是生成待编译程序的抽象语法树AST，AST不同于推导树，去掉了一些修饰性的单词部分，简明地把程序的语法结构表示出来，后续的语义分析、中间代码生成都可以通过遍历抽象语法树来完成。

为了创建AST，需要对文法的各个符号规定一些属性值，如表2-1所示列出了终结符绑定词法分析得到的值，非终结符绑定AST中对应的树结点指针。

表2-1 文法符号对应属性

符 号	属 性	
ID	标识符的字符串	
INT	整常数数字	
FLOAT	浮点常数数字	
所有非终结符	抽象语法树的结点指针	

其它终结符	可忽略	
-------	-----	--

通过对AST的遍历并显示语法，能帮助我们分析验证语法分析的结果是否正确，同时熟悉使用遍历算法访问结点的次序，这样在后序的语义分析、中间代码的处理过程中，就能非常方便地使用遍历流程完成其对应的编译阶段工作，同时也能给我们在调试程序中提供方便。

2.3 符号表结构描述

符号表采用顺序表管理符号表。此时的符号表 `symbolTable` 是一个顺序栈，栈顶指针 `index` 初始值为 0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加 1。

符号表大致内容如错误!未找到引用源。:

name	KIND	L	OFFSET	TYPE...
f1	function	0	入口标号	返回类型等
x	<u>Var</u>	1	DX	<u>int</u>
y	<u>Var</u>	1	DX+4	float

图 2-1 符号表结构

`name` 表示变量名或者函数名，此外符号表应该还有一个 `alias` 项，意思是别名，在生成中间代码时很有必要。`L` 表示 `level`，层级的意思，可以用来判断变量的作用域。除此之外，符号表应该还有一个 `Label` 项，用来标明类型。比如 `V`、`F`、`P` 分别表示变量，函数，参数。`Offset` 就是偏移量了，用于标明位置。

2.4 错误类型码描述

这次实验并没有设置错误类型码，而是在进行语义检查时直接输出错误。在这次实验中一共能够检查出 15 种错误，下面是这 15 中错误的类型：

1. 变量重复定义
2. 函数参数重复定义
3. 函数名重复定义
4. 返回值类型错误
5. 变量未定义
6. 参数类型不匹配

7. 传入参数过少
8. 传入参数过多
9. 变量不是函数
10. 表达式不能自增
11. 自增表达式需要左值
12. =两边类型不匹配
13. 赋值需左值
14. 函数名未定义
15. break 未出现在合法位置

2.5 中间代码描述

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，建立符号表，并生成以三地址代码 TAC 作为中间语言的中间语言代码序列。

采用三地址代码 TAC 作为中间语言，中间语言代码的定义如表 2-2 所示。

表 2-2 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
LABEL x	定义标号 x	LABEL			X
FUNCTION f:	定义函数 f	FUNCTION			F
x := y	赋值操作	ASSIGN	X		X
x := y + z	加法操作	PLUS	Y	Z	X
x := y - z	减法操作	MINUS	Y	Z	X
x := y * z	乘法操作	STAR	Y	Z	X
x := y / z	除法操作	DIV	Y	Z	X
GOTO x	无条件转移	GOTO			X
IF x [relop] y GOTO z	条件转移	[relop]	X	Y	Z
RETURN x	返回语句	RETURN			X
ARG x	传实参 x	ARG			X
x:=CALL f	调用函数	CALL	F		X
PARAM x	函数形参	PARAM			X
READ x	读入	READ			X
WRITE x	打印	WRITE			X

三地址中间代码 TAC 是一个 4 元组，逻辑上包含 (op、opn1、opn2、result)，其中 op 表示操作类型说明，opn1 和 opn2 表示 2 个操作数，result 表示

运算结果。后续还需要根据个 TAC 序列生成目标代码，所以设计其存储结构时，每一部分要考虑目标代码生成是所需要的信息。

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

(2) 操作数与运算结果：这些部分包含的数据类型有多种，整常量，实常量，还有使用标识符的情况，如变量的别名、变量在其数据区的偏移量和层号、转移语句中的标号等。类型不同，所以考虑使用联合。为了明确联合中的有效成员，将操作数与运算结果设计成结构类型，包含 kind，联合等几个成员，kind 说明联合中的有效，联合成员是整常量，实常量或标识符表示的别名或标号或函数名等。

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

如下是一个中间代码的例子：

```
1. FUNCTION afun :
2.     PARAM v8
3.     PARAM
4. LABEL label1 :
5. LABEL label2 :
6.
7. FUNCTION main :
8.     temp1 := #1
9.     v13 := temp1
10.    temp2 := #1
11.    := temp2
12.    ARG v13
13.    temp3 := CALL afun
14.    ARG v14
15.    temp4 := CALL afun
16.    ARG v14
17.    ARG v13
18.    ARG v15
19.    temp5 := CALL afun
20.    temp7 := temp6 + #1
21.    temp6 := temp7
22.    v14 := v13
```



```

23. LABEL label14 :
24. LABEL label17 :
25.  IF v13 != #0 GOTO label16
26.  GOTO label15
27. LABEL label16 :
28.  GOTO label17
29. LABEL label15 :
30.  temp10 := #1
31.  RETURN temp10
32. LABEL label13 :

```

2.6 目标代码描述

目标语言可选定 MIPS32 指令序列，可以在 SPIM Simulator 上运行，SPIM Simulator 的安装使用参见文献[2]。TAC 指令和 MIPS32 指令的对应关系如表 2-3 所示。其中 $\text{reg}(x)$ 表示变量 x 所分配的寄存器。

表 2-3 中间代码与 MIPS32 指令对应关系

中间代码	MIPS32 指令
LABEL x	x:
$x := \#k$	li $\text{reg}(x), k$
$x := y$	move $\text{reg}(x), \text{reg}(y)$
$x := y + z$	add $\text{reg}(x), \text{reg}(y), \text{reg}(z)$
$x := y - z$	sub $\text{reg}(x), \text{reg}(y), \text{reg}(z)$
$x := y * z$	mul $\text{reg}(x), \text{reg}(y), \text{reg}(z)$
$x := y / z$	div $\text{reg}(y), \text{reg}(z)$ mflo $\text{reg}(x)$
GOTO x	j x
RETURN x	move \$v0, $\text{reg}(x)$ jr \$ra
IF $x == y$ GOTO z	beq $\text{reg}(x), \text{reg}(y), z$
IF $x != y$ GOTO z	bne $\text{reg}(x), \text{reg}(y), z$
IF $x > y$ GOTO z	bgt $\text{reg}(x), \text{reg}(y), z$
IF $x \geq y$ GOTO z	bge $\text{reg}(x), \text{reg}(y), z$
IF $x < y$ GOTO z	ble $\text{reg}(x), \text{reg}(y), z$

IF $x \leq y$ GOTO z	blt reg(x),reg(y),z
$X := \text{CALL } f$	jal f move reg(x), \$v0

另外一个比较重要的方面就是寄存器的分配问题，如果寄存器使用不当，那么就很容易是程序出错，比如你进入函数调用时，如果寄存器使用不当，很可能会丢失中间结果，在这里使用朴素的寄存器分配方案。具体实现在详细设计中。

下面是一段目标代码的例子：

```

1. main:
2.     addi $sp, $sp, -36
3.     li $t3, 1
4.     sw $t3, 16($sp)
5.     lw $t1, 16($sp)
6.     move $t3, $t1
7.     sw $t3, 12($sp)
8.     li $t3, 1
9.     sw $t3, 32($sp)
10.    lw $t1, 32($sp)
11.    move $t3, $t1

```

3 系统设计与实现

3.1 编译程序符号表结构

根据符号表结构描述，符号表采用顺序表来管理。此时的符号表 `symbolTable` 是一个顺序栈，栈顶指针 `index` 初始值为 0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加 1。同时，本表也需要记录各种名字的特征信息。名字包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括：上述名字的种类、具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

具体符号表结构如下图所示：

VarId	Alias	Level	Type	Label	Offset
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
fibonacci	v6	0	int	F	0
a	v7	1	int	P	12
	temp1	1	int	T	16
	temp2	1	int	T	16
	temp3	1	int	T	16
	temp4	1	int	T	16
	temp5	1	int	T	20
	temp6	1	int	T	24
	temp7	1	int	T	28
	temp8	1	int	T	32
	temp9	1	int	T	36
	temp10	1	int	T	40

图 3-1 符号表结构

3.2 编译程序报错功能

根据上文的错误类型，本程序一共能够报 15 种错误类型。报错能够指出位置以及错误原因，示例如下：

```

      arun      v7      0      int      F      28
      a      v8      1      float     P      12
      b      v11     1      float     V      12
at line 20 , y Variable undefined
at line 20 , Two sides' type unmatched
at line 21 , Paramter unmatched
at line 22 , Paramter is not enough
at line 23 , Paramter is superabundant
at line 24 , x is not a function name
at line 25 , expression cannot self plus
at line 26 , Left must be left-value(+=)
at line 27 , Two sides' type unmatched
at line 28 , Left must be left-value(=)
at line 29 , ab function name undefined
VarId Alias Level Type Label Offset
1 0 0 0 0 0

```

图 2-2 程序正常报错

3.3 词法语法分析器（实验一）

依据mini-c语言的定义，在此给出各单词的种类码和相应符号说明：

INT → 整型常量

FLOAT → 浮点型常量

ID → 标识符

ASSIGNOP → =

RELOP → > | >= | < | <= | == | !=

PLUS → +

MINUS → -

STAR → *

DIV → /

AND → &&

OR → ||

NOT → !

TYPE → int | float

RETURN → return

IF → if

ELSE → else

WHILE → while

SEMI → ;

COMMA → ,

SEMI → ;

LP → (

RP →)

LC → {

RC → }

除此之外，还定义了以下符号。

非终结符：

1. program ExtDefList ExtDef Specifier ExtDecList FuncDec CompSt VarList
2. VarDec ParamDec Stmt StmtList DefList Def DecList Dec Exp Args

终结符：

1. ELF_PLUS SELF_MINUS BREAK PLUSASS MINUSASS SELF_PLUS SELF_MINUS

然后需要定义运算符的优先级与结合性，如下表所示：

表 3-1 优先级与结合性

优先级	结合性	符号
高	右结合	UMINUS NOT
	左结合	COMMENT
	左结合	STAR DIV
	左结合	PLUS MINUS SELF_PLUS SELF_MINUS
	左结合	RELOP
	左结合	AND
	左结合	OR
低	左结合	ASSIGNOP PLUSASS MINUSASS

然后根据上文单词文法描述，需要我们在 lex.l 中自己添加新定义的单词，比如 char, break 等等。在此处，char 使用正则表达式来进行匹配，'[A-Za-z]'，在这里采用的是匹配字母。再比如操作符需要进行以下操作：

```
"+=" {return PLUSASS;}
```

同时，在 parser.y 中也需要添加自己新定义的单词。

char 类型需要添加在 union 中，按照上表所示的结合性，将 parser.y 补充完整。

也还需要进行以下操作，生成一个树节点。

1. | Exp SELF_PLUS {\$%=mknnode(SELF_PLUS,\$1,NULL,NULL,yylineno);strcpy(\$\$->type_id,"SELF_PLUS");}

上面是对自增操作符生成一个树节点，使用了 mknnode 这个函数，第一个参

数表明操作符的类型，第二个参数表示左边第一棵子树，\$1 代表了 Exp 表达式。类似地，添加其他的单词也需要进行这种操作。

这样做完之后，根据给出的 ast.c 文件，使用 flex 和 bison 分别编译 lex.l 和 parser.y，就可以生成一棵语法树了。

3.4 语义分析（实验二）

1. 符号表管理

语义分析一个重要的工作是符号表的管理，3.1 符号表的结构已经很详细地讲述了符号表的管理方式以及存储的信息。

2. 语法树的结构

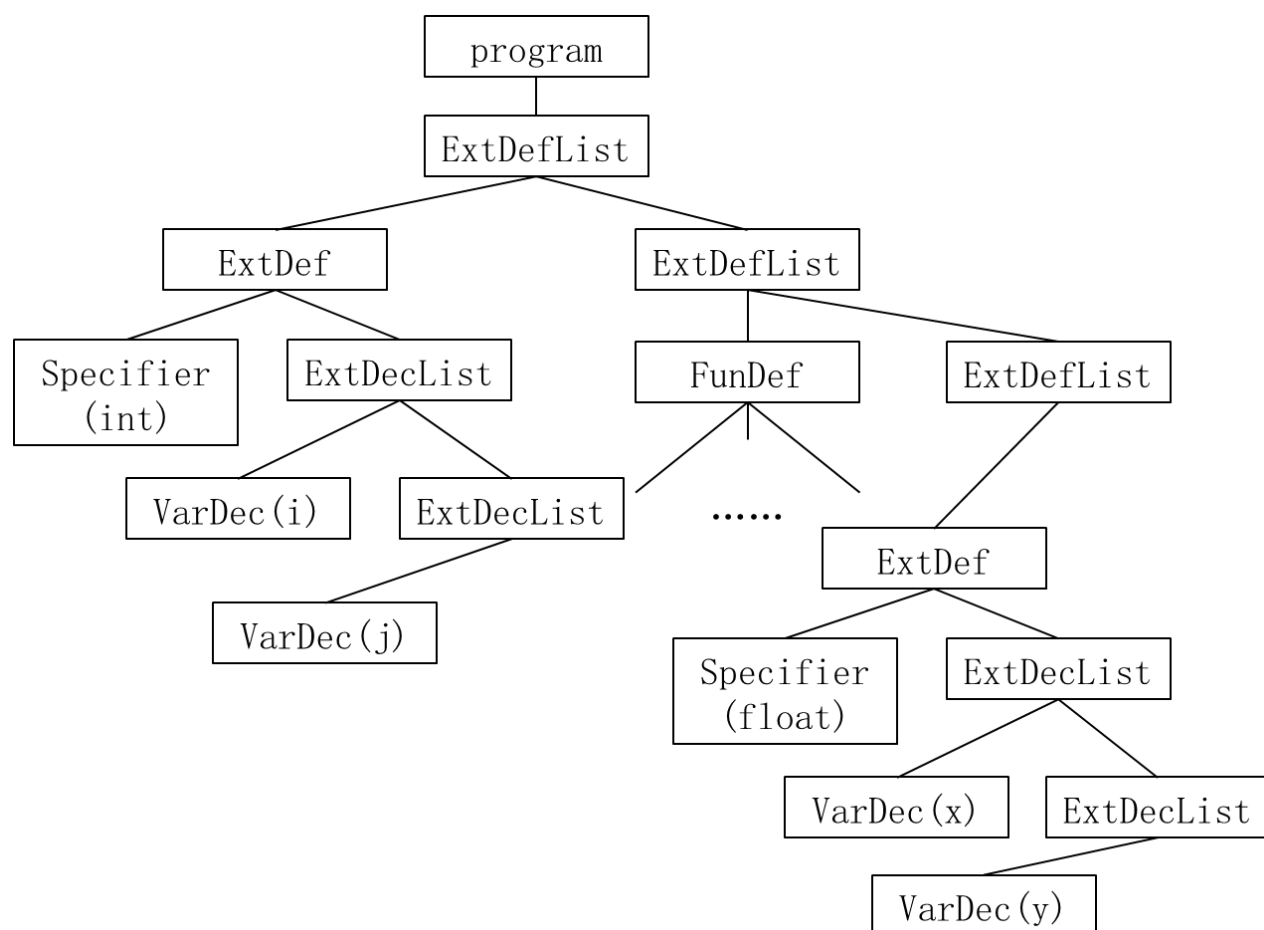


图 3-3 语法树

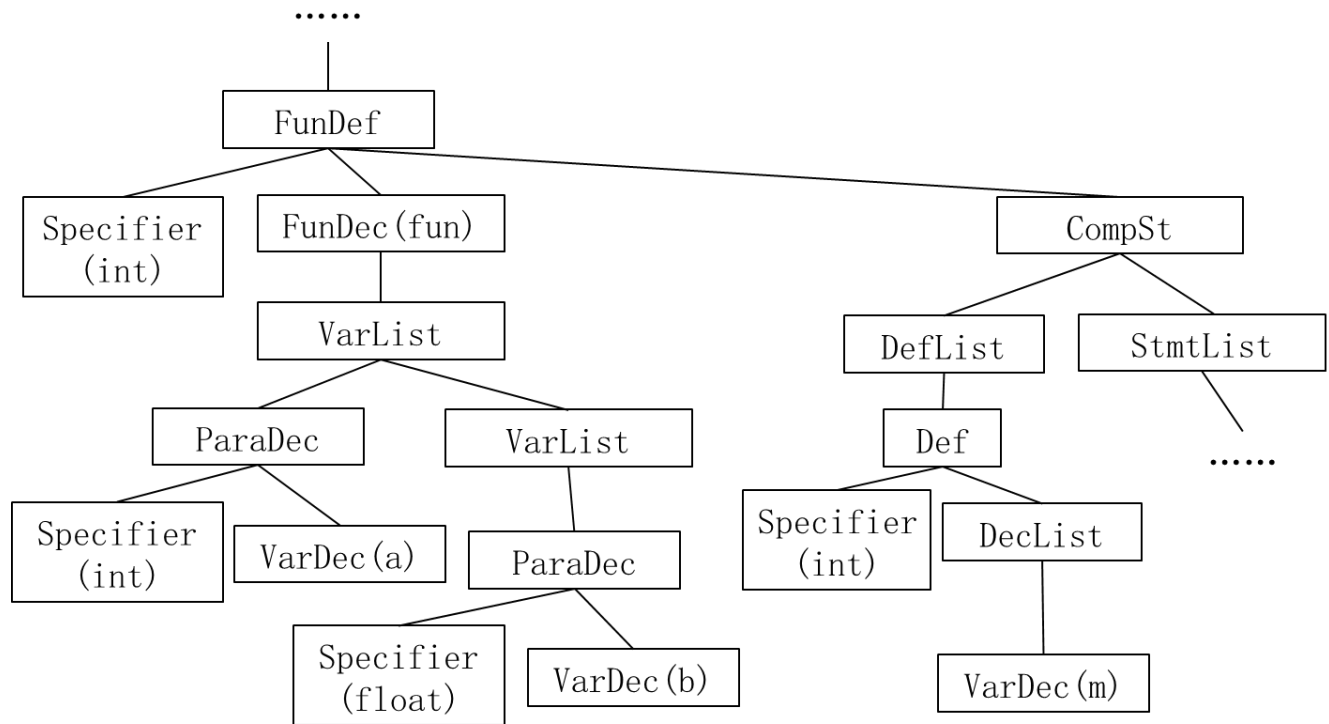


图 3-4 语法树

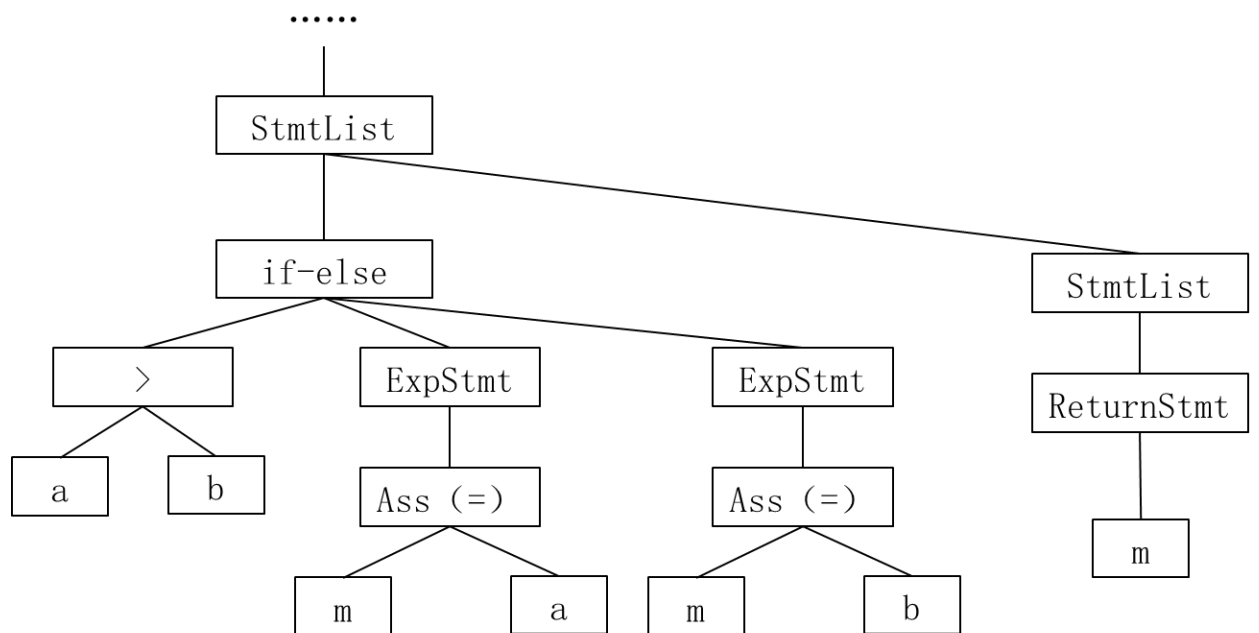


图 3-5 语法树

这个语法树的构建还需要考虑的就是偏移量的确定，因此节点还有`.width`和`.type`属性，这个在上文已经说明了，根据子树的这两个属性，`parent`可以继承`width`属性，这样就可以确定偏移量了。

构建完了语法树之后，就可以对抽象语法树先根遍历生成语法树了，先根遍历 AST 算法的框架很简单，采用递归算法实现，设 `T` 为根结点指针。

- (1) 如果 T 为空，遍历结束返回
- (2) 根据 T->kind，即结点类型，可知道该结点有多少棵子树，依次递归访问各子树。

对局部变量说明语句 `int a, b;` 对应的 AST 如图：

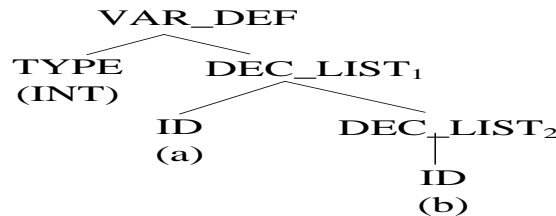


图 3-6 AST 图

当第一次访问到 VAR_DEF 结点时，按遍历次序，接着访问 TYPE 结点，确定 TYPE 结点的数据类型为 INT，回到 VAR_DEF 后，该说明语句中的变量列表中个变量的类型确定了，可将此类型属性向下传给结点 DEC_LIST1。接着类型由 DEC_LIST1 传到 a 这个 ID 结点，这时就明确了 a 是一个整型变量，查符号表，如果在当前作用域（根据层号）没有定义，就根据 a 填写一个整型的变量 a 到符号表中，否则报错，变量重复定义。再接着数据类型 INT 由 DEC_LIST1 传到 DEC_LIST2 结点，直到整型变量 b 完成查表和填写到符号表中。

上述例子的属性计算仅考虑语义分析这部分的需求，但在整个编译过程中，需要同时完成的属性计算还很多，比如访问 VAR_DEF 结点时，首先到此结点，由前面的计算结果，已经得到这个说明语句的变量在活动记录中的地址偏移量（offset），这时访问过 TYPE 结点后，得到该类型变量的宽度值

（width），这样 a 的地址偏移量就为 offset，b 的地址偏移量为 offset+width；最后回到 VAR_TYPE 结点时，其说明语句中变量的总宽度计算出为 2*width。所以再遇到 VAR_DEF 之后的其它变量说明的结点时，地址偏移量为 offset+2*width。由此给计算出一个函数中所有变量在活动记录中的地址偏移量。在遍历过程中，会涉及到较多的属性计算，需要分清楚哪些是在语义分析中必须的，哪些是后续中间代码生成需要的，语义分析只用做语义分析的事，避免重复计算属性。

比如：

```
1. case EXT_VAR_DEF:    printf("%*c 外部变量定义: \n",indent,' ');
2.                      display(T->ptr[0],indent+3);           //显示外部变量类
                        型
3.                      printf("%*c 变量名: \n",indent+3,' ');
4.                      display(T->ptr[1],indent+6);           //显示变量列表
```


这样做完之后就可以生成一棵语法树了，如：

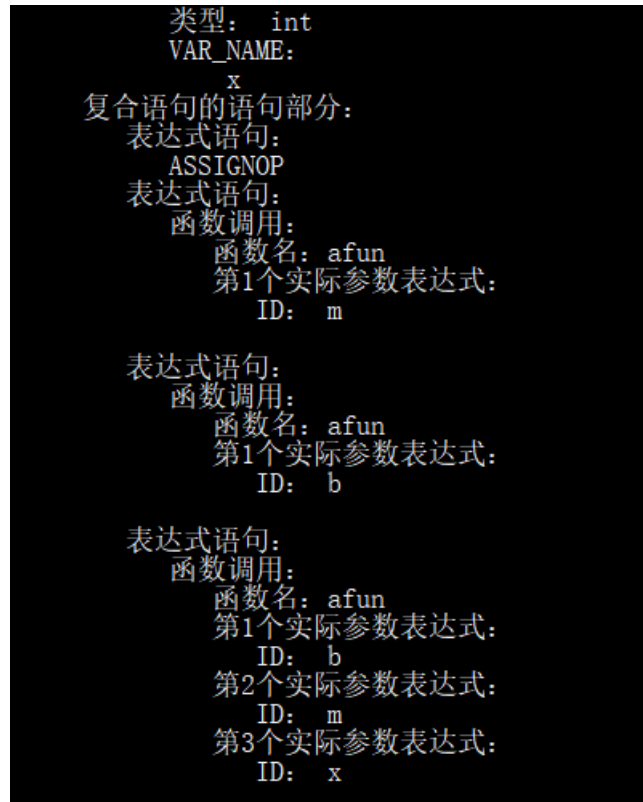


图 3-7 语法树显示

3. 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

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

需要自己实验 break 的位置检查。

这里需要实现 break 的位置检查，必须出现在循环语句中。这里的实现方法是在树节点里加一个属性 hasBreak，默认为 0。如果检测到了 break，那么这个节点的 hasBreak 属性就为 1，如果检测到了 while，则将这个节点的 hasBreak 属性设置为 1。同时，对于一个树节点，如果它的三棵子树中有一个节点的 hasBreak 为 1，那么这个节点的 hasBreak 属性就为 1（一种特殊情况是，当此节点为 while 时，while 节点的 hasBreak 为 0）。这样就只需要看树根节点的 hasBreak 属性值为 0 还是 1 了，如果为 1，说明 break 出现的位置非法，否则 break 出现的位置合法。

(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。

唯一性检查需要用到 L 这个属性，这个 L 表明了标识符所在的块，每进入一个块，L 加 1；退出，L 减 1。

详细的唯一性检查工作在 fillSymbolTable 中已经解决了，通过函数的返回值能够知道是否重复定义了。如果返回-1，那么就说明重复定义。遍历完同一层（L 相同）的符号后，如果名字重复，直接返回-1。在这里还需要考虑全局变量的情况，全局变量 L 设置为 0，如果 L（L 不等于 0）层里的元素与全局变量同名，这种情况写也不属于重复定义。

（3）类型检查

首先就是形参与实参类型是否一致，这个根据符号表取出形参实参，得到他们的类型然后再比较即可。

然后就是检查赋值语句，在这里没有进行强制转换，因此直接比较两者的类型，根据结果来判断是否匹配。

最后就是函数返回值类型的检查，这个根据符号表得到函数返回值类型和函数应该返回的类型，比较一下就可以知道是否匹配了。

4. 错误判断

根据静态语义分析，可以检查出很多错误了。在 **3.2 编译程序报错功能和第二章中**已经说明了错误类型，展示了示例。在此就不再赘述。

3.5 中间代码生成功能（实验三）

由之前生成的语法树，再对其进行一次前序遍历，就可以获得中间代码了。采用三地址代码 TAC 作为中间语言，中间语言代码的定义如表 3-2 所示。

表 3-2 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
LABEL x	定义标号 x	LABEL			X
FUNCTION f:	定义函数 f	FUNCTION			F
x := y	赋值操作	ASSIGN	X		X
x := y + z	加法操作	PLUS	Y	Z	X
x := y - z	减法操作	MINUS	Y	Z	X
x := y * z	乘法操作	STAR	Y	Z	X
x := y / z	除法操作	DIV	Y	Z	X
GOTO x	无条件转移	GOTO			X
IF x [relop] y GOTO z	条件转移	[relop]	X	Y	Z
RETURN x	返回语句	RETURN			X

ARG x	传实参 x	ARG			X
x:=CALL f	调用函数	CALL	F		X
PARAM x	函数形参	PARAM			X
READ x	读入	READ			X
WRITE x	打印	WRITE			X

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

- (1) 运算符表示需要完成的运算。
- (2) 操作数与运算结果，类型使用 kind。
- (3) 用双向循环链表表示 TAC 代码序列。

为了完成中间代码的生成，对于 AST 中的结点，需要考虑设置以下属性，在遍历过程中，根据翻译模式给出的计算方法完成属性的计算。

.place 记录该结点操作数在符号表中的位置序号，这里包括变量在符号表中的位置，以及每次完成了计算后，中间结果需要用一个临时变量保存，临时变量也需要登记到符号表中。另外由于使用复合语句，可以使作用域嵌套，不同的作用域中的变量可以同名，这是在 mini-c 中，和 C 语言一样采用就近优先的原则，但在中间语言中，没有复合语句区分层次，所以每次登记一个变量到符号表中时，会多增加一个**别名 (alias)**的表项，通过别名实现数据的唯一性。翻译时，对变量的操作替换成对别名的操作，别名命名形式为 **v+序号**。生成临时变量时，命名形式为 **temp+序号**，在填符号表时，可以在符号名称这栏填写一个空串，临时变量名直接填写到别名这栏。

.type 一个结点表示数据时，记录该数据的类型，用于表达式的计算中。该属性也可用于语句，表示语句语义分析的正确性 (OK 或 ERROR)。

.offset 记录外部变量在静态数据区中的偏移量以及局部变量和临时变量在活动记录中的偏移量。另外对函数，利用这项保存活动记录的大小。

.width 记录一个结点表示的语法单位中，定义的变量和临时单元所需要占用的字节数，方便计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小。

.code 记录中间代码序列的起始位置，如采用链表表示中间代码序列，该属性就是一个链表的头指针。

.Etrue 和.Efalse 在完成布尔表达式翻译时，表达式值为真假时要转移的程序位置 (标号的字符串形式)。

.Snext 该结点的语句序列执行完后，要转移或到的程序位置（标号的字符串形式）。

为了生成中间代码序列，定义了几个函数：

newtemp 生成一临时变量，登记到符号表中，以 **temp+序号** 的形式组成的符号串作为别名，符号名称栏用空串登记到符号表中。

newLabel 生成一个标号，标号命名形式为 **LABEL+序号**。

genIR 生成一条 TAC 的中间代码语句。一般情况下，TAC 中，涉及到 2 个运算对象和运算结果。如果是局部变量或临时变量，表示在运行时，其对应的存储单元在活动记录中，这时需要将其偏移量（offset）这个属性和数据类型同时带上，方便最后的目标代码生成。全局变量也需要带上偏移量。

genLabel 生成标号语句。

以上 3 个函数，在实验时，也可以合并在一起，如何处理，可自行确定。

merge 将多个语句序列顺序连接在一起。

定义完这些属性和函数后，就需要根据翻译模式表示的计算次序，计算规则右部各个符号对应结点的代码段，再按语句的语义，将这些代码段拼接在一起，组成规则左部非终结符对应结点的代码段。过程可参考课件 if_then_else 语句的翻译过程。

额外：

对语法树进行先根遍历，对操作数 op1, op2 的类型进行判断，并且在前面加上 # 输出。然后根据 op 操作符选择情况输出：

比如赋值语句对应的情况就如下：

```
1. case ASSIGNOP: printf(" %s := %s\n",resultstr,opnstr1);
2.                break;
```

如果是一个赋值语句，就会出现 resultstr := opnstr1。

对每一个操作符进行都进行这样的输出就可以获得中间代码了。

3.6 汇编代码生成功能（实验四）

之后就要生成目标代码了目标语言可选定 MIPS32 指令序列。

中间代码与 MIPS32 指令的关系已经在上文中给出了，在此不再赘述。

寄存器的分配

按照实验指导，采用朴素的寄存器分配方案，翻译方法如下表。

表 3-4 朴素寄存器分配的翻译

中间代码	MIPS32 指令
$x := \#k$	li \$t3,k sw \$t3, x 的偏移量(\$sp)
$x := y$	lw \$t1, y的偏移量(\$sp) move \$t3,\$t1 sw \$t3, x的偏移量(\$sp)
$x := y + z$	lw \$t1, y的偏移量(\$sp) lw \$t2, z的偏移量(\$sp) add \$t3,\$t1,\$t2 sw \$t3, x的偏移量(\$sp)
$x := y - z$	lw \$t1, y的偏移量(\$sp) lw \$t2, z的偏移量(\$sp) sub \$t3,\$t1,\$t2 sw \$t3, x的偏移量(\$sp)
$x := y * z$	lw \$t1, y的偏移量(\$sp) lw \$t2, z的偏移量(\$sp) mul \$t3,\$t1,\$t2 sw \$t3, x的偏移量(\$sp)
$x := y / z$	lw \$t1, y的偏移量(\$sp) lw \$t2, z的偏移量(\$sp) div \$t1,\$t2 mflo \$t3 sw \$t3, x 的偏移量(\$sp)
RETURN x	move \$v0, x 的偏移量(\$sp) jr \$ra
IF $x==y$ GOTO z	lw \$t1, x的偏移量(\$sp) lw \$t2, y的偏移量(\$sp) beq \$t1,\$t2,z
IF $x!=y$ GOTO z	lw \$t1, x的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) bne \$t1,\$t2,z
IF $x>y$ GOTO z	lw \$t1, x的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) bgt \$t1,\$t2,z

IF x>=y GOTO z	lw \$t1, x的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) bge \$t1,\$t2,z
IF x<y GOTO z	lw \$t1, x的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) ble \$t1,\$t2,z
IF x<=y GOTO z	lw \$t1, x的偏移量(\$sp) lw \$t2, y 的偏移量(\$sp) blt \$t1,\$t2,z
X:=CALL f	

对于 **call** 命令需要进行以下操作：

- (1) 首先根据保存在函数调用指令中的 **offset**，找到符号表中的函数定义点，获取函数的参数个数 **i**，这样就可得到在 **X:=CALL f** 之前的 **i** 个 **ARG** 形式的中间代码，获得 **i** 个实参值所存放的单元，取出后送到形式参数的单元中。再活动记录的空间。
- (2) 根据符号表记录的活动记录大小，开辟活动记录空间和保存返回地址。
- (3) 使用 **jal f** 转到函数 **f** 处
- (4) 释放活动记录空间和恢复返回地址。
- (5) 使用 **sw \$v0, x 的偏移量(\$sp)** 获取返回值送到 **X** 的存储单元中。

其中有几个比较值得注意的细节：

- a) 首先需要自己先定义 **read** 和 **write** 两个函数，这样可以保证程序能够读取输入，根据手册，使用 **syscall** 指令调用系统中断，读取输入。然后与中间代码类似，根据 **op** 来选择对应的输出，（翻译的代码在上表已经给出）。
 - b) **call** 需要特别注意，对于 **read** 和 **write** 函数需要特别处理。需要保留返回地址，函数处理完后需要回复返回地址。**write** 还需要传入参数。
 - c) 对于 **main** 函数也要特别注意，**main** 可能需要用到全局变量，调用其他函数因此需要 **\$sp** 减去 **main** 偏移量。
- 做完这些工作，就可以生成目标代码了。

4 系统测试与评价

4.1 测试用例

1. 实验一二测试用例如下：

```
1. int a,b,c;
2. int a,b,c;// 1 v redefine
3.
4. int afun(float a, float a)// 2 para redefine
5. {
6.
7. }
8.
9. int afun(float a, int x)//3 afun redefine
10. {
11.     float b;
12.     return b;//4 return value error
13. }
14.
15. int main()
16. {
17.     int m=1;
18.     float b;
19.     int x;
20.     y = 1;//5 y undefined
21.     afun(m);//6 afun para type unmatched
22.     afun(b);//7 afun para few
23.     afun(b,m,x);//8 afun para many
24.     x();//9 x is not a func
25.     5++;//10 ++ must be left-value
26.     5 += m;//11 += must be left-value
27.     b = m;//12 type unmatched
28.     5 = b;//13 assign must be left-value
29.     ab(); //14 func undefined
```

```

30.     break;//15 cannot break here
31.     while(m){
32.         //break;
33.     }
34.     return 1;
35. }

```

2. 实验三四测试用例如下：

（同时测试了斐波那契和阶乘函数，最后决定把阶乘放上来，为了与实验指导区分）

```

1. int fibo(int a)
2. {
3.     if (a==1) return 1;
4.     return a*fibo(a-1);
5. }
6. int main()
7. {
8.     int m,n,i;
9.     m = read();
10.    i = 1;
11.    write(m);
12.    while(i<=m)
13.    {
14.        n = fibo(i);
15.        write(n);
16.        i=i+1;
17.    }
18.    return 1;
19. }

```

4.2 正确性测试

在这里正确性的验证就用实验三四的测试用例，以最后能在 SPIM 上正确运行作为通过。

图 4-1,4-2 分别是符号表与中间代码。

图 4-3 是在 SPIM 上运行的结果,可以看出阶乘的计算结果正确。(5! = 120)
至此,程序的正确性测试通过。

VarId	Alias	Level	Type	Label	Offset
read		0	int	F	4
x		1	int	P	12
write		0	int	F	4
fibonacci	v1	0	int	F	0
a	v2	1	int	P	12
temp1		1	int	T	16
temp2		1	int	T	16
temp3		1	int	T	16
temp4		1	int	T	20
temp5		1	int	T	24
temp6		1	int	T	28
temp7		1	int	T	24
temp8		1	int	T	24
temp9		1	int	T	24
temp10		2	int	T	24
temp11		2	int	T	24
temp12		2	int	T	24
temp13		2	int	T	28
temp14		1	int	T	24

图 4-1 符号表

```

FUNCTION fibonacci :
    PARAM v2
    temp1 := #1
    IF v2 == temp1 GOTO label3
    GOTO label2
LABEL label3 :
    temp2 := #1
    RETURN temp2
LABEL label2 :
    temp3 := #1
    temp4 := v2 - temp3
    ARG temp4
    temp5 := CALL fibonacci
    temp6 := v2 * temp5
    RETURN temp6
LABEL label1 :

FUNCTION main :
    temp7 := CALL read
    v4 := temp7
    temp8 := #1
    v6 := temp8
    ARG v4
    temp9 := CALL write
LABEL label10 :
    IF v6 <= v4 GOTO label9
    GOTO label8
LABEL label9 :
    ARG v6
    temp10 := CALL fibonacci
    v5 := temp10
    ARG v5
    temp11 := CALL write
    temp12 := #1
    temp13 := v6 + temp12
    v6 := temp13
    GOTO label10
LABEL label8 :
    temp14 := #1
    RETURN temp14
LABEL label4 :

```

图 4-2 中间代码

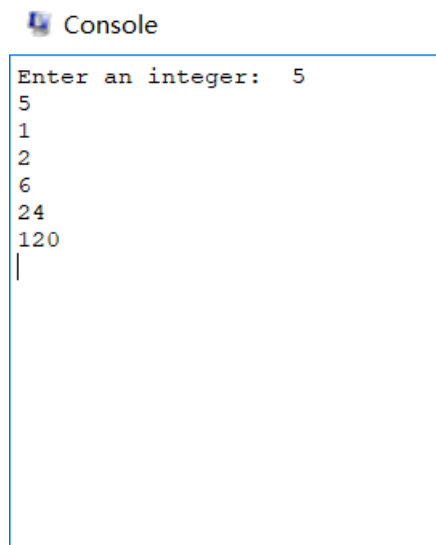


图 4-3 SPIM 测试图

4.3 报错功能测试

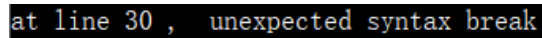
程序的报错功能的测试就要用实验一二的测试用例了, 15 种错误已经在测试用例的注释中给出了, 因此下面就直接给出结果。如下图。

```

at line 2 , a Variable redefined
at line 2 , b Variable redefined
at line 2 , c Variable redefined
at line 4 , a Paramter name cannot be the same
VarId Alias Level Type Label Offset
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
afun v7 0 int F 0
a v8 1 float P 12
at line 9 , afun function name redefined
at line 12 , Return value type unmatched
VarId Alias Level Type Label Offset
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
afun v7 0 int F 28
a v8 1 float P 12
b v11 1 float V 12
at line 20 , y Variable undefined
at line 20 , Two sides' type unmatched
at line 21 , Paramter unmatched
at line 22 , Paramter is not enough
at line 23 , Paramter is superabundant
at line 24 , x is not a function name
at line 25 , expression cannot self plus
at line 26 , Left must be left-value(+=)
at line 27 , Two sides' type unmatched
at line 28 , Left must be left-value(=)
at line 29 , ab function name undefined
VarId Alias Level Type Label Offset

```

图 4-4 程序报错测试图



```
at line 30 , unexpected syntax break
```

图 4-5 程序报错测试图

4.4 系统的优点

1. 能够在提供的代码基础上加入额外的一些功能，比如 `char` 类型、算术运算、比较运算、逻辑运算、自增自减运算和复合赋值运算，控制语句有 `if` 和 `while` 语句。
2. 能够检查出 15 种错误，包括类型检查，名字作用域分析以及控制流检查（主要是判断 `break` 是否出错）。
3. 采用朴素的寄存器分配方案，能够实现递归，能够解决一般的程序编译问题。

4.5 系统的缺点

1. 还有些基本的错误不能判断出来，可能会导致最后生成的代码出错。
2. 符号表采用顺序表管理，效率较低，没有采用 `Hash` 表。
3. 寄存器分配方法采用实验指导上提供的方法，虽然比较容易实现但是效率很低。

5 实验小结或体会

本次实验在实验指导的基础上，自己添加了部分规则完成的 miniC 编译器。借助 flex 和 bison 工具，能够有效提高工作效率。

其实从零开始写一个编译器是十分困难的，但好在实验指导提供了实验的大部分框架，因此只需要理解实验框架的思想，在其基础上进行扩展，就能很好地完成本次课程实验。

高级语言的定义已经给出，首先需要完成的就是词法分析与语法分析。通过词法分析得到种类码和 yylval 传入语法分析，如果正确机会得到抽象语法树。词法分析主要是能够识别单词，这里是通过正则表达式实现的。之后就要进行语法分析了，这部分需要处理移进规约和规约规约冲突，解决就要设定优先级和结合性，这个表已经在设计中给出，然后就需要对表达式生成对应的树节点。这些都完成之后，如果正确就会生成抽象语法树。

重要的一环就是语义分析，首先需要使用符号表管理符号。符号表的实现方式很多，指导中采用顺序表来完成，是一种简单有效的方法，但是效率很低，如果追求高效的话可以使用哈希表。在这个环节还要求程序能够检查错误，比如控制流检查，唯一性检查，名字作用与分析，类型检查等等。控制流检查只实现了 break 的检查，方法也在设计中阐述了，有一点不满意的地方就是多个 break 出现的情况，如果需要实现多个 break 同时报错的话，需要一个数组结构来保存出错 break 的位置，并且还需要判断是否出错，这个逻辑有点复杂，因此就只是实现了能不能检查 break 这个错误，如果有可能，还希望能够改进。参数重命名其实就是利用了 L 这个参数，在符号表中检查同一层的变量名，如果已经存在，则说明出现重命名了。中间代码生成比较简单，就是对不同的情况生成不同的中间代码，只需要 switch 就行，类似地生成目标代码也是进行一样的操作，只不过一开始不知道 call 的流程，实现起来会比较麻烦，后来按照指导教程上的步骤，一步步实现就能实现这些功能。如果要使用 write 和 read 函数，需要在一开始就给出这两个函数的定义，并且在符号表中加入他们。寄存器的分配方案就直接采用推荐的朴素的寄存器分配方案，直接给出也比较省精力。

完成本次课程实验，回头来看还有而很多可以优化的地方，但是通过这次实验知道了是先编译器需要完成哪些基本的并且很重要的工作，重要的是整个流程。在有限的时间内，能够对这个流程熟悉，掌握这个流程，对课程理论知识也会有一个更深的了解，如果以后从事相关方面的工作研究，也能更得心应手一些。

参考文献

- [1] John Levine 著 陆军 译. 《Flex与Bison》.东南大学出版社
- [2] 许畅等编著. 《编译原理实践与指导教程》.机械工业出版社
- [3] 王生原等编著. 《编译原理（第3版）》.清华大学出版社
- [4] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005
- [5] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005
- [6] 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
- [7] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005
- [8] 曹计昌等. C 语言程序设计. 北京：科学出版社，2008
- [9] 《编译原理实验教程》. 编译原理课程组

附件：源代码

```
1. %{
2. #include "parser.tab.h"
3. #include "string.h"
4. #include "def.h"
5. int yycolumn=1;
6. #define YY_USER_ACTION      yylloc.first_line=yylloc.last_line=yylineno; \
7.     yylloc.first_column=yycolumn;   yylloc.last_column=yycolumn+yylength-
    1; yycolumn+=yylength;
8. typedef union {
9.     int type_int;
10.    char type_char;
11.    int type_float;
12.    char type_id[32];
13.    struct node *ptr;
14. } YYLVAL;
15. #define YYSTYPE YYLVAL
16.
17. %}
18. %option yylineno
19.
20. id    [A-Za-z][A-Za-z0-9]*
21. int    [0-9]+
22. float  ([0-9]*\.[0-9]+)|([0-9]+\.)
23. char   '[A-Za-z]'
24.
25. %%
26. {int}      {yylval.type_int=atoi(yytext); return INT;}
27. {float}    {yylval.type_float=atof(yytext); return FLOAT;}
28. {char}     {yylval.type_char=atof(yytext); return CHAR;}
29.
30. "int"      {strcpy(yylval.type_id, yytext);return TYPE;}
31. "float"    {strcpy(yylval.type_id, yytext);return TYPE;}
32. "char"     {strcpy(yylval.type_id, yytext);return TYPE;}
33.
34. "return"   {return RETURN;}
35. "break"    {return BREAK;}
36. "if"       {return IF;}
37. "else"     {return ELSE;}
38. "while"    {return WHILE;}
39.
```

```

40. {id}          {strcpy(yyval.type_id, yytext); return ID; /*由于关键字的形式也符合表示符的规则，所以把关键字的处理全部放在标识符的前面，优先识别*/}
41. ";"          {return SEMI;}
42. ","          {return COMMA;}
43. ">"|"<"| ">="|"<="|"=="|"!=" {strcpy(yyval.type_id, yytext);return RELOP;}

44. "="          {return ASSIGNOP;}
45. "+="         {return PLUSASS;}
46. "-="         {return MINUSASS;}
47. "++"         {return SELF_PLUS;}
48. "+"          {return PLUS;}
49. "-"          {return MINUS;}
50. "*"          {return STAR;}
51. "//"         {return COMMENT;}
52. "/"          {return DIV;}
53. "&&"         {return AND;}
54. "||"         {return OR;}
55. "!"          {return NOT;}
56. "("          {return LP;}
57. ")"          {return RP;}
58. "{"          {return LC;}
59. "}"          {return RC;}
60. "["          {return LB;}
61. "]"          {return RB;}
62. "/*".*[\n]  {  }
63. "/*"(.|[\n])*"/" {  }
64.
65. [\n]         {yycolumn=1;}
66. [ \r\t]      {}
67. .           {printf("Error type A :Mysterious character \"%s\"\n\t at Line %d\n",yytext,yylineno);}
68. %%
69.
70. /* 和 bison 联用时，不需要这部分
71. void main()
72. {
73. yylex();
74. return 0;
75. }
76.
77. */
78. int yywrap()
79. {
80. return 1;

```

```

81. }

1. %error-verbose
2. %locations
3. %{
4. #include "stdio.h"
5. #include "math.h"
6. #include "string.h"
7. #include "def.h"
8. extern int yylineno;
9. extern char *yytext;
10. extern FILE *yyin;
11. void yyerror(const char* fmt, ...);
12. void display(struct node *,int);
13. %}
14.
15. %union {
16.     int     type_int;
17.     char     type_char;
18.     char     type_id[32];
19.     float    type_float;
20.     struct node *ptr;
21. };
22.
23. // %type 定义非终结符的语义值类型
24. %type <ptr> program ExtDefList ExtDef Specifier ExtDeclList FuncDec CompSt
    VarList VarDec ParamDec Stmt StmtList DefList Def DeclList Dec Exp Args
25.
26. // %token 定义终结符的语义值类型
27. %token <type_int> INT //指定 INT 的语义值是 type_int, 有词法分析得到
    的数值
28. %token <type_char> CHAR //指定 ID 的语义值是 type_id, 有词法分析得到的
    标识符字符串
29. %token <type_id> ID RELOP TYPE //指定 ID,RELOP 的语义值是 type_id, 有词法分析得
    到的标识符字符串
30. %token <type_float> FLOAT //指定 ID 的语义值是 type_id, 有词法分析得到的
    标识符字符串
31.
32.
33. %token LP RP LC RC LB RB SEMI COMMA //用 bison 对该文件编译时, 带参数-d, 生成
    的 exp.tab.h 中给这些单词进行编码, 可在 lex.l 中包含 parser.tab.h 使用这些单词种类
    码

```



```

34. %token PLUS MINUS STAR DIV COMMENT ASSIGNOP AND OR NOT SELF_PLUS SELF_MINUS
    IF ELSE WHILE FOR RETURN BREAK
35.
36. %left ASSIGNOP PLUSASS MINUSASS
37. %left OR
38. %left AND
39. %left RELOP
40. %left PLUS MINUS SELF_PLUS SELF_MINUS
41. %left STAR DIV
42. %left COMMENT
43. %right UMINUS NOT
44.
45. %nonassoc LOWER_THEN_ELSE
46. %nonassoc ELSE
47.
48. %%
49.
50. program: ExtDefList    { display($1,0); semantic_Analysis0($1);}    /*显示语
    法树,语义分析*/
51.          ;
52. ExtDefList: {$$=NULL;}
53.          | ExtDef ExtDefList {$$=mknode(EXT_DEF_LIST,$1,$2,NULL,yylineno);}
    //每一个 EXTDEFLIST 的结点, 其第 1 棵子树对应一个外部变量声明或函数
54.          ;
55. ExtDef:   Specifier ExtDeclList SEMI    {$$=mknode(EXT_VAR_DEF,$1,$2,NULL,yyli
    neno);}    //该结点对应一个外部变量声明
56.          | Specifier FuncDec CompSt    {$$=mknode(FUNC_DEF,$1,$2,$3,yylineno)
    ;}    //该结点对应一个函数定义
57.          | error SEMI    {$$=NULL; }
58.          ;
59. Specifier: TYPE    {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type
    _id,$1);$ $->type=!strcmp($1,"int"?INT:FLOAT);}
60.          ;
61. ExtDeclList: VarDec    {$$=$1;}    /*每一个 EXT_DECLIST 的结点, 其第一棵子
    树对应一个变量名(ID 类型的结点), 第二棵子树对应剩下的外部变量名*/
62.          | VarDec COMMA ExtDeclList {$$=mknode(EXT_DEC_LIST,$1,$3,NULL,yyli
    neno);}
63.          ;
64. VarDec:   ID    {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_
    id,$1);}    //ID 结点, 标识符字符串存放结点的 type_id
65.          | ID LB INT RB    {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->ty
    pe_id,$1);}    //数组 ID 结点, 标识符字符串存放结点的 type_id.here
66.          | ID LB RB    {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_i
    d,$1);}    //数组 ID 结点, 标识符字符串存放结点的 type_id.here

```

```

67.          ;
68. FuncDec: ID LP VarList RP    {$$=mknode(FUNC_DEC,$3,NULL,NULL,yylineno);strcpy
        y($$->type_id,$1);} //函数名存放在$$->type_id
69.          | ID LP RP    {$$=mknode(FUNC_DEC,NULL,NULL,NULL,yylineno);strcpy($$-
        >type_id,$1);} //函数名存放在$$->type_id
70.
71.          ;
72. VarList: ParamDec    {$$=mknode(PARAM_LIST,$1,NULL,NULL,yylineno);}
73.          | ParamDec COMMA VarList    {$$=mknode(PARAM_LIST,$1,$3,NULL,yylineno
        )};}
74.          ;
75. ParamDec: Specifier VarDec          {$$=mknode(PARAM_DEC,$1,$2,NULL,yylineno)
        ;}
76.          ;
77.
78. CompSt: LC DefList StmList RC    {$$=mknode(COMP_STM,$2,$3,NULL,yylineno);}
79.          ;
80. StmList: {$$=NULL; }
81.          | Stmt StmList    {$$=mknode(STM_LIST,$1,$2,NULL,yylineno);}
82.          ;
83. Stmt:    Exp SEMI    {$$=mknode(EXP_STMT,$1,NULL,NULL,yylineno);}
84.          | CompSt    {$$=$1;} //复合语句结点直接最为语句结点，不再生成新的结
        点
85.          | RETURN Exp SEMI    {$$=mknode(RETURN,$2,NULL,NULL,yylineno);}
86.          | BREAK SEMI    {$$=mknode(BREAK,NULL,NULL,NULL,yylineno);$->hasBrea
        k=1;}
87.          | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE    {$$=mknode(IF_THEN,$3,$5,N
        NULL,yylineno);}
88.          | IF LP Exp RP Stmt ELSE Stmt    {$$=mknode(IF_THEN_ELSE,$3,$5,$7,yylin
        eno);}
89.          | WHILE LP Exp RP Stmt {$$=mknode(WHILE,$3,$5,NULL,yylineno);$->hasBr
        eak=0;}
90.          ;
91.
92. DefList: {$$=NULL; }
93.          | Def DefList {$$=mknode(DEF_LIST,$1,$2,NULL,yylineno);}
94.          ;
95. Def:    Specifier Declist SEMI {$$=mknode(VAR_DEF,$1,$2,NULL,yylineno);}
96.          ;
97. Declist: Dec    {$$=mknode(DEC_LIST,$1,NULL,NULL,yylineno);}
98.          | Dec COMMA Declist    {$$=mknode(DEC_LIST,$1,$3,NULL,yylineno);}
99.          ;
100. Dec:    VarDec    {$$=$1;}

```

```

101.      | VarDec ASSIGNOP Exp  {$%=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");}
102.      ;
103. Exp:   Exp ASSIGNOP Exp  {$%=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");} // $$ 结点 type_id 空置未用，正好存放运算符
104.      | Exp PLUSASS Exp  {$%=mknode(PLUSASS,$1,$3,NULL,yylineno);strcpy($$->type_id,"PLUSASS");} // +=
105.      | Exp MINUSASS Exp  {$%=mknode(MINUSASS,$1,$3,NULL,yylineno);strcpy($$->type_id,"MINUSASS");} // -=
106.      | Exp AND Exp      {$%=mknode(AND,$1,$3,NULL,yylineno);strcpy($$->type_id,"AND");}
107.      | Exp OR Exp       {$%=mknode(OR,$1,$3,NULL,yylineno);strcpy($$->type_id,"OR");}
108.      | Exp RELOP Exp  {$%=mknode(RELOP,$1,$3,NULL,yylineno);strcpy($$->type_id,$2);} // 词法分析关系运算符自身值保存在$2 中
109.      | Exp PLUS Exp  {$%=mknode(PLUS,$1,$3,NULL,yylineno);strcpy($$->type_id,"PLUS");}
110.      | Exp MINUS Exp  {$%=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type_id,"MINUS");}
111.      | Exp STAR Exp  {$%=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type_id,"STAR");}
112.      | Exp DIV Exp  {$%=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type_id,"DIV");}
113.      | LP Exp RP      {$%=$2;}
114.      | MINUS Exp %prec UMINUS  {$%=mknode(UMINUS,$2,NULL,NULL,yylineno);strcpy($$->type_id,"UMINUS");}
115.      | NOT Exp        {$%=mknode(NOT,$2,NULL,NULL,yylineno);strcpy($$->type_id,"NOT");}
116.      | Exp SELF_PLUS  {$%=mknode(SELF_PLUS,$1,NULL,NULL,yylineno);strcpy($$->type_id,"SELF_PLUS");} // 自增
117.      | Exp SELF_MINUS  {$%=mknode(SELF_MINUS,$1,NULL,NULL,yylineno);strcpy($$->type_id,"SELF_MINUS");} // 自减
118.      | ID LP Args RP  {$%=mknode(FUNC_CALL,$3,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
119.      | ID LP RP      {$%=mknode(FUNC_CALL,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
120.      | ID            {$%=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
121.      | ID LB INT RB  {$%=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
122.      | INT           {$%=mknode(INT,NULL,NULL,NULL,yylineno);$$->type_int=$1;$$->type=INT;}
123.      | FLOAT         {$%=mknode(FLOAT,NULL,NULL,NULL,yylineno);$$->type_float=$1;$$->type=FLOAT;}

```

```

124.      | CHAR      {$$=mknode(CHAR,NULL,NULL,NULL,yylineno);$$->type_cha
      r=$1;$$->type=CHAR;}
125.      ;
126. Args:    Exp COMMA Args    {$$=mknode(ARGS,$1,$3,NULL,yylineno);}
127.      | Exp          {$$=mknode(ARGS,$1,NULL,NULL,yylineno);}
128.      ;
129.
130. %%
131.
132. int main(int argc, char *argv[]){
133.     yyin=fopen(argv[1],"r");
134.     if (!yyin) return 0;
135.     yylineno=1;
136.     yyparse();
137.     return 0;
138. }
139.
140. #include<stdarg.h>
141. void yyerror(const char* fmt, ...)
142. {
143.     va_list ap;
144.     va_start(ap, fmt);
145.     fprintf(stderr, "Grammar Error at Line %d Column %d: ", yylloc.first_li
ne,yylloc.first_column);
146.     vfprintf(stderr, fmt, ap);
147.     fprintf(stderr, ".\n");
148. }

1. #include "def.h"
2. struct node * mknode(int kind,struct node *first,struct node *second, struct
node *third,int pos ) {
3.     int i=0;
4.     struct node *T=(struct node *)malloc(sizeof(struct node));
5.     T->kind=kind;
6.     T->ptr[0]=first;
7.     T->ptr[1]=second;
8.     T->ptr[2]=third;
9.     T->pos=pos;
10.
11.     T->hasBreak = 0 ;
12.
13.     while(i<3){
14.         if(T->ptr[i]){
15.             if(T->ptr[i]->hasBreak)
16.                 {

```

```

17.         T->hasBreak=1;
18.         break;
19.     }
20. }
21.     i++;
22. }
23.
24. return T;
25.
26. }
27.
28. void display(struct node *T,int indent) { //对抽象语法树的先根遍历
29.     int i=1;
30.     struct node *T0;
31.     if (T)
32.     {
33.         switch (T->kind) {
34.             case EXT_DEF_LIST: display(T->ptr[0],indent); //显示该外部定义列表中的
                第一个
35.                             display(T->ptr[1],indent); //显示该外部定义列表中的
                其它外部定义
36.                             break;
37.             case EXT_VAR_DEF: printf("%*c 外部变量定义: \n",indent,' ');
38.                             display(T->ptr[0],indent+3); //显示外部变量类
                型
39.                             printf("%*c 变量名: \n",indent+3,' ');
40.                             display(T->ptr[1],indent+6); //显示变量列表
41.                             break;
42.             case TYPE:      printf("%*c 类型:  %s\n",indent,' ',T->type_id);
43.                             break;
44.             case EXT_DEC_LIST: display(T->ptr[0],indent); //依次显示外部变量
                名,
45.                             display(T->ptr[1],indent); //后续还有相同的, 仅显
                示语法树此处理代码可以和类似代码合并
46.                             break;
47.             case FUNC_DEF:   printf("%*c 函数定义: \n",indent,' ');
48.                             display(T->ptr[0],indent+3); //显示函数返回类型
49.                             display(T->ptr[1],indent+3); //显示函数名和参数
50.                             display(T->ptr[2],indent+3); //显示函数体
51.                             break;
52.             case FUNC_DEC:   printf("%*c 函数名:  %s\n",indent,' ',T->type_id);
53.                             if (T->ptr[0]) {
54.                                 printf("%*c 函数形参: \n",indent,' ');

```

```

55.                display(T->ptr[0],indent+3); //显示函数参数
           列表
56.                }
57.                else printf("%*c 无参函数\n",indent+3, ' ');
58.                break;
59.    case PARAM_LIST:    display(T->ptr[0],indent); //依次显示全部参数类型
           和名称,
60.                display(T->ptr[1],indent);
61.                break;
62.    case PARAM_DEC:    printf("%*c 类型: %s, 参数名: %s\n", indent, ' ', \
63.                T->ptr[0]->type=="INT?"int": "float",T->ptr[1]
           ->type_id);
64.                break;
65.    case EXP_STMT:    printf("%*c 表达式语句: \n",indent, ' ');
66.                display(T->ptr[0],indent+3);
67.                break;
68.    case RETURN:    printf("%*c 返回语句: \n",indent, ' ');
69.                display(T->ptr[0],indent+3);
70.                break;
71.    case COMP_STMT:    printf("%*c 复合语句: \n",indent, ' ');
72.                printf("%*c 复合语句的变量定义: \n",indent+3, ' ');
73.                display(T->ptr[0],indent+6); //显示定义部分
74.                printf("%*c 复合语句的语句部分: \n",indent+3, ' ');
75.                display(T->ptr[1],indent+6); //显示语句部分
76.                break;
77.    case STM_LIST:    display(T->ptr[0],indent); //显示第一条语句
78.                display(T->ptr[1],indent); //显示剩下语句
79.                break;
80.    case WHILE:    printf("%*c 循环语句: \n",indent, ' ');
81.                printf("%*c 循环条件: \n",indent+3, ' ');
82.                display(T->ptr[0],indent+6); //显示循环条件
83.                printf("%*c 循环体: \n",indent+3, ' ');
84.                display(T->ptr[1],indent+6); //显示循环体
85.                break;
86.    case IF_THEN:    printf("%*c 条件语句(IF_THEN): \n",indent, ' ');
87.                printf("%*c 条件: \n",indent+3, ' ');
88.                display(T->ptr[0],indent+6); //显示条件
89.                printf("%*cIF 子句: \n",indent+3, ' ');
90.                display(T->ptr[1],indent+6); //显示 if 子句
91.                break;
92.    case IF_THEN_ELSE:    printf("%*c 条件语句(IF_THEN_ELSE): \n",indent, ' ');
93.                printf("%*c 条件: \n",indent+3, ' ');
94.                display(T->ptr[0],indent+6); //显示条件
95.                printf("%*cIF 子句: \n",indent+3, ' ');

```

```

96.                display(T->ptr[1],indent+6);        //显示 if 子句
97.                printf("%*cELSE 子句: \n",indent+3,' ');
98.                display(T->ptr[2],indent+6);        //显示 else 子句
99.                break;
100.    case DEF_LIST:    display(T->ptr[0],indent);    //显示该局部变量定义列
    表中的第一个
101.                display(T->ptr[1],indent);    //显示其它局部变量定
    义
102.                break;
103.    case VAR_DEF:    printf("%*cLOCAL VAR_NAME: \n",indent,' ');
104.                display(T->ptr[0],indent+3);    //显示变量类型
105.                display(T->ptr[1],indent+3);    //显示该定义的全部变量
    名
106.                break;
107.    case DEC_LIST:    printf("%*cVAR_NAME: \n",indent,' ');
108.                T0=T;
109.                while (T0) {
110.                    if (T0->ptr[0]->kind==ID)
111.                        printf("%*c %s\n",indent+3,' ',T0->ptr[0]->
    type_id);
112.                    else if (T0->ptr[0]->kind==ASSIGNOP)
113.                        {
114.                            printf("%*c %s ASSIGNOP\n ",indent+3,' ',T0
    ->ptr[0]->ptr[0]->type_id);
115.                            //显示初始化表达式
116.                            display(T0->ptr[0]->ptr[1],indent+strlen(T0
    ->ptr[0]->ptr[0]->type_id)+4);
117.                        }
118.                    T0=T0->ptr[1];
119.                }
120.                break;
121.    case ID:    printf("%*cID: %s\n",indent,' ',T->type_id);
122.                break;
123.    case INT:    printf("%*cINT: %d\n",indent,' ',T->type_int);
124.                break;
125.    case FLOAT:    printf("%*cFLAOT: %f\n",indent,' ',T->type_float);
126.                break;
127.    case CHAR:    printf("%*cCHAR: %c\n",indent,' ',T->type_char);
128.                break;
129.    case ASSIGNOP:    printf("%*cASSIGNOP\n",indent,' ');
130.                break;
131.    case SELF_PLUS:    printf("%*cSELF_PLUS\n",indent,' ');
132.                break;

```

```

133.     case PLUSASS:      printf("%*cPLUSASS\n",indent,' ');
134.                         break;
135.     case MINUSASS:
136.
137.         // 下面几个选项共用
138.     case AND:
139.     case OR:
140.     case RELOP:
141.     case PLUS:
142.     case MINUS:
143.     case STAR:
144.     case DIV:
145.         printf("%*c%s\n",indent,' ',T->type_id);
146.         display(T->ptr[0],indent+3);
147.         display(T->ptr[1],indent+3);
148.         break;
149.     case NOT:
150.     case UMINUS:    printf("%*c%s\n",indent,' ',T->type_id);
151.                     display(T->ptr[0],indent+3);
152.                     break;
153.     case FUNC_CALL: printf("%*c 函数调用: \n",indent,' ');
154.                     printf("%*c 函数名: %s\n",indent+3,' ',T->type_id);
155.                     display(T->ptr[0],indent+3);
156.                     break;
157.     case ARGS:      i=1;
158.                     while (T) { //ARGS 表示实际参数表达式序列结点，其第一棵子
                        树为其一个实际参数表达式，第二棵子树为剩下的。
159.                         struct node *T0=T->ptr[0];
160.                         printf("%*c 第%d 个实际参数表达式:
                            \n",indent,' ',i++);
161.                         display(T0,indent+3);
162.                         T=T->ptr[1];
163.                     }
164.                     printf("\n");
165.                     break;
166.         }
167.     }
168. }

1. #include "def.h"
2.
3. extern int yylineno;
4.
5. int breakPos;//指明位置
6.

```



```

7.  char *strcat0(char *s1, char *s2) {
8.      static char result[10];
9.      strcpy(result, s1);
10.     strcat(result, s2);
11.     return result;
12. }
13.
14. char *newAlias() {
15.     static int no=1;
16.     char s[10];
17.     itoa(no++, s, 10);
18.     return strcat0("v", s);
19. }
20.
21. char *newLabel() {
22.     static int no=1;
23.     char s[10];
24.     itoa(no++, s, 10);
25.     return strcat0("label", s);
26. }
27.
28. char *newTemp(){
29.     static int no=1;
30.     char s[10];
31.     itoa(no++, s, 10);
32.     return strcat0("temp", s);
33. }
34.
35. //生成 TAC
36. struct codenode *genIR(int op, struct opn opn1, struct opn opn2, struct opn result){
37.     struct codenode *h=(struct codenode *)malloc(sizeof(struct codenode));
38.     h->op=op;
39.     h->opn1=opn1;
40.     h->opn2=opn2;
41.     h->result=result;
42.     h->next=h->prior=h;
43.     return h;
44. }
45.
46. //生成 Label
47. struct codenode *genLabel(char *label){
48.     struct codenode *h=(struct codenode *)malloc(sizeof(struct codenode));
49.     h->op=LABEL;

```

```

50.     strcpy(h->result.id,label);
51.     h->next=h->prior=h;
52.     return h;
53. }
54.
55. //生成 GOTO
56. struct codenode *genGoto(char *label){
57.     struct codenode *h=(struct codenode *)malloc(sizeof(struct codenode));
58.     h->op=GOTO;
59.     strcpy(h->result.id,label);
60.     h->next=h->prior=h;
61.     return h;
62. }
63.
64. //首尾相连
65. struct codenode *merge(int num,...){
66.     struct codenode *h1,*h2,*p,*t1,*t2;
67.     va_list ap;
68.     va_start(ap,num);
69.     h1=va_arg(ap,struct codenode *);
70.     while (--num>0) {
71.         h2=va_arg(ap,struct codenode *);
72.         if (h1==NULL) h1=h2;
73.         else if (h2){
74.             t1=h1->prior;
75.             t2=h2->prior;
76.             t1->next=h2;
77.             t2->next=h1;
78.             h1->prior=t2;
79.             h2->prior=t1;
80.         }
81.     }
82.     va_end(ap);
83.     return h1;
84. }
85.
86. //输出中间代码
87. void prnIR(struct codenode *head){
88.     char opnstr1[32],opnstr2[32],resultstr[32];
89.     struct codenode *h=head;
90.     do {
91.         if (h->opn1.kind==INT)
92.             sprintf(opnstr1,"%d",h->opn1.const_int);
93.         if (h->opn1.kind==FLOAT)

```

```

94.         sprintf(opnstr1, "%f", h->opn1.const_float);
95.     if (h->opn1.kind==ID)
96.         sprintf(opnstr1, "%s", h->opn1.id);
97.     if (h->opn2.kind==INT)
98.         sprintf(opnstr2, "%d", h->opn2.const_int);
99.     if (h->opn2.kind==FLOAT)
100.         sprintf(opnstr2, "%f", h->opn2.const_float);
101.     if (h->opn2.kind==ID)
102.         sprintf(opnstr2, "%s", h->opn2.id);
103.     sprintf(resultstr, "%s", h->result.id);
104.     switch (h->op) {
105.         case ASSIGNOP: printf(" %s := %s\n", resultstr, opnstr1);
106.             break;
107.         case PLUSASS: printf(" %s := %s + %s\n", resultstr, resultstr, o
pnstr2); //here 复合赋值的中间代码
108.             break;
109.         case SELF_PLUS: printf(" %s := %s + %s\n", resultstr, resultstr,
opnstr1);
110.             break;
111.         case PLUS:
112.         case MINUS:
113.         case STAR:
114.         case DIV: printf(" %s := %s %c %s\n", resultstr, opnstr1, \
115.             h->op==PLUS? '+': h->op==MINUS? '-'
': h->op==STAR? '*': '\\', opnstr2);
116.             break;
117.         case FUNCTION: printf("\nFUNCTION %s :\n", h->result.id);
118.             break;
119.         case PARAM: printf(" PARAM %s\n", h->result.id);
120.             break;
121.         case LABEL: printf("LABEL %s :\n", h->result.id);
122.             break;
123.         case GOTO: printf(" GOTO %s\n", h->result.id);
124.             break;
125.         case JLE: printf(" IF %s <= %s GOTO %s\n", opnstr1, opnstr2
, resultstr);
126.             break;
127.         case JLT: printf(" IF %s < %s GOTO %s\n", opnstr1, opnstr2,
resultstr);
128.             break;
129.         case JGE: printf(" IF %s >= %s GOTO %s\n", opnstr1, opnstr2
, resultstr);
130.             break;

```

```

131.         case JGT:      printf(" IF %s > %s GOTO %s\n",opnstr1,opnstr2,
        resultstr);
132.                                     break;
133.         case EQ:       printf(" IF %s == %s GOTO %s\n",opnstr1,opnstr2
        ,resultstr);
134.                                     break;
135.         case NEQ:      printf(" IF %s != %s GOTO %s\n",opnstr1,opnstr2
        ,resultstr);
136.                                     break;
137.         case ARG:      printf(" ARG %s\n",h->result.id);
138.                                     break;
139.         case CALL:     printf(" %s := CALL %s\n",resultstr, opnstr1);
140.                                     break;
141.         case RETURN:   if (h->result.kind)
142.                         printf(" RETURN %s\n",resultstr);
143.                         else
144.                         printf(" RETURN\n");
145.                         break;
146.     }
147.     h=h->next;
148. } while (h!=head);
149. }
150.
151. void genObCode(struct codenode *head)
152. {
153.     int i;
154.     char opnstr1[32],opnstr2[32],resultstr[32];
155.     struct codenode *h=head,*p;
156.     FILE *fp;
157.     fp=fopen("OBCode.s","w");
158.     fprintf(fp, ".data\n");
159.     fprintf(fp, "_Prompt: .asciiz \"Enter an integer:  \n\n");
160.     fprintf(fp, "_ret: .asciiz \"\\n\\n\\n");
161.     fprintf(fp, ".globl main\n");
162.     fprintf(fp, ".text\n");
163.     fprintf(fp, "read:\n");
164.     fprintf(fp, "    li $v0,4\n");
165.     fprintf(fp, "    la $a0,_Prompt\n");
166.     fprintf(fp, "    syscall\n");
167.     fprintf(fp, "    li $v0,5\n");
168.     fprintf(fp, "    syscall\n");
169.     fprintf(fp, "    jr $ra\n");
170.     fprintf(fp, "write:\n");

```

```

171.     fprintf(fp, "  li $v0,1\n");
172.     fprintf(fp, "  syscall\n");
173.     fprintf(fp, "  li $v0,4\n");
174.     fprintf(fp, "  la $a0,_ret\n");
175.     fprintf(fp, "  syscall\n");
176.     fprintf(fp, "  move $v0,$0\n");
177.     fprintf(fp, "  jr $ra\n");
178.     do {
179.         switch (h->op) {
180.             case ASSIGNOP:
181.                 if (h->opn1.kind==INT)
182.                     fprintf(fp, "  li $t3, %d\n", h->opn1.const_int
183. );
184.                 else {
185.                     fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset
186. );
187.                     fprintf(fp, "  move $t3, $t1\n");
188.                 }
189.                 fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset
190. );
191.                 break;
192.             case PLUS:
193.             case MINUS:
194.             case STAR:
195.             case DIV:
196.                 fprintf(fp, "  lw $t1, %d($sp)\n", h->opn1.offset);
197.                 fprintf(fp, "  lw $t2, %d($sp)\n", h->opn2.offset);
198.                 if (h->op==PLUS)      fprintf(fp, "  add $t3,$t1,$t
199. 2\n");
200.                 else if (h->op==MINUS) fprintf(fp, "  sub $t3,$t1,$t
201. 2\n");
202.                 else if (h->op==STAR) fprintf(fp, "  mul $t3,$
203. t1,$t2\n");
204.                 else {
205.                     fprintf(fp, "  div $t1, $t2\n");
206.                     fprintf(fp, "  mflo $t3\n");
207.                 }
208.                 fprintf(fp, "  sw $t3, %d($sp)\n", h->result.offset
209. );
210.                 break;
211.             case FUNCTION:
212.                 fprintf(fp, "\n%s:\n", h->result.id);

```

```

206.                if (!strcmp(h->result.id,"main")) //特殊处理 main
207.                fprintf(fp, "    addi $sp, $sp, -%d\n",symbolTable
    e.symbols[h->result.offset].offset);
208.                break;
209.                case PARAM:
210.                break;
211.                case LABEL: fprintf(fp, "%s:\n", h->result.id);
212.                break;
213.                case GOTO:  fprintf(fp, "    j %s\n", h->result.id);
214.                break;
215.                case JLE:
216.                case JLT:
217.                case JGE:
218.                case JGT:
219.                case EQ:
220.                case NEQ:
221.                fprintf(fp, "    lw $t1, %d($sp)\n", h->opn1.offset);
222.                fprintf(fp, "    lw $t2, %d($sp)\n", h->opn2.offset);
223.                if (h->op==JLE) fprintf(fp, "    ble $t1,$t2,%s\n", h
->result.id);
224.                else if (h->op==JLT) fprintf(fp, "    blt $t1,$t2,%s\
n", h->result.id);
225.                else if (h->op==JGE) fprintf(fp, "    bge $t1,$t2,%s\
n", h->result.id);
226.                else if (h->op==JGT) fprintf(fp, "    bgt $t1,$t2,%s\
n", h->result.id);
227.                else if (h->op==EQ)  fprintf(fp, "    beq $t1,$t2,%s\
n", h->result.id);
228.                else                fprintf(fp, "    bne $t1,$t2,%s\
n", h->result.id);
229.                break;
230.                case ARG:
231.                break;
232.                case CALL:
233.                if (!strcmp(h->opn1.id,"write")){
234.                fprintf(fp, "    lw $a0, %d($sp)\n",h->prior->res
ult.offset);
235.                fprintf(fp, "    addi $sp, $sp, -4\n");
236.                fprintf(fp, "    sw $ra,0($sp)\n"); //保留返回地
址
237.                fprintf(fp, "    jal write\n");

```

```

238.                fprintf(fp, "  lw $ra,0($sp)\n"); //恢复返回地
    址
239.                fprintf(fp, "  addi $sp, $sp, 4\n");
240.                break;
241.            }
242.            if (!strcmp(h->opn1.id,"read")){
243.                fprintf(fp, "  addi $sp, $sp, -4\n");
244.                fprintf(fp, "  sw $ra,0($sp)\n"); //保留返回地
    址
245.                fprintf(fp, "  jal read\n");
246.                fprintf(fp, "  lw $ra,0($sp)\n"); //恢复返回地
    址
247.                fprintf(fp, "  addi $sp, $sp, 4\n");
248.                fprintf(fp, "  sw $v0, %d($sp)\n",h->result.off
    set);
249.                break;
250.            }
251.            for(p=h,i=0;i<symbolTable.symbols[h->opn1.offset].p
    aramnum;i++){
252.                p=p->prior;
253.            }
254.            fprintf(fp, "  move $t0,$sp\n"); //取实参表达式的
    值
255.            fprintf(fp, "  addi $sp, $sp, -%d\n", symbolTable.s
    ymbols[h->opn1.offset].offset);
256.            fprintf(fp, "  sw $ra,0($sp)\n"); //保留返回地址
257.            i=h->opn1.offset+1; //第一个形参变量
258.            while (symbolTable.symbols[i].flag=='P')
259.            {
260.                fprintf(fp, "  lw $t1, %d($t0)\n", p->resul
    t.offset);
261.                fprintf(fp, "  move $t3,$t1\n");
262.                fprintf(fp, "  sw $t3,%d($sp)\n", symbolTa
    ble.symbols[i].offset);
263.                p=p->next; i++;
264.            }
265.            fprintf(fp, "  jal %s\n",h->opn1.id); //恢复返回地
    址
266.            fprintf(fp, "  lw $ra,0($sp)\n"); //恢复返回地址
267.            fprintf(fp, "  addi $sp,$sp,%d\n",symbolTable.symbo
    ls[h->opn1.offset].offset);
268.            fprintf(fp, "  sw $v0,%d($sp)\n", h->result.offset)
    ;
269.            break;

```

```

270.         case RETURN:
271.             fprintf(fp, " lw $v0,%d($sp)\n",h->result.offset);

272.             fprintf(fp, " jr $ra\n");
273.             break;
274.
275.     }
276.     h=h->next;
277. } while (h!=head);
278. fclose(fp);
279. }
280.
281. void semantic_error(int line,char *msg1,char *msg2){
282.     printf("at line %d , %s %s\n",line,msg1,msg2);
283. }
284. void prn_symbol(){ //显示符号表
285.     int i=0;
286.     printf("%6s %6s %6s %6s %4s %6s\n","VarId","Alias","Level","Type","Label",
"Offset");
287.     for(i=0;i<symbolTable.index;i++)
288.         printf("%6s %6s %6d %6s %4c %6d\n",symbolTable.symbols[i].name,\
289.             symbolTable.symbols[i].alias,symbolTable.symbols[i].level,\

290.             symbolTable.symbols[i].type=="int"? "int": "float",\
291.             symbolTable.symbols[i].flag,symbolTable.symbols[i].offset);

292. }
293.
294. int searchSymbolTable(char *name) {
295.     int i;
296.     for(i=symbolTable.index-1;i>=0;i--)
297.         if (!strcmp(symbolTable.symbols[i].name, name)) return i;
298.     return -1;
299. }
300.
301. int fillSymbolTable(char *name,char *alias,int level,int type,char flag,int
offset) {
302.     //首先根据 name 查符号表, 不能重复定义
303.     int i;
304.     /*符号查重, 考虑外部变量 声明前有函数定义*/
305.     for(i=symbolTable.index-
1;symbolTable.symbols[i].level==level||((level==0 && i>=0);i--) {
306.         if (level==0 && symbolTable.symbols[i].level==1) continue;
307.         if (!strcmp(symbolTable.symbols[i].name, name)) return -1;

```



```

308.     }
309.     //填写符号表内容
310.     strcpy(symbolTable.symbols[symbolTable.index].name,name);
311.     strcpy(symbolTable.symbols[symbolTable.index].alias,alias);
312.     symbolTable.symbols[symbolTable.index].level=level;
313.     symbolTable.symbols[symbolTable.index].type=type;
314.     symbolTable.symbols[symbolTable.index].flag=flag;
315.     symbolTable.symbols[symbolTable.index].offset=offset;
316.     return symbolTable.index++; //符号在符号表中的位置序号
317. }
318.
319.
320. int fill_Temp(char *name,int level,int type,char flag,int offset) {
321.     strcpy(symbolTable.symbols[symbolTable.index].name,"");
322.     strcpy(symbolTable.symbols[symbolTable.index].alias,name);
323.     symbolTable.symbols[symbolTable.index].level=level;
324.     symbolTable.symbols[symbolTable.index].type=type;
325.     symbolTable.symbols[symbolTable.index].flag=flag;
326.     symbolTable.symbols[symbolTable.index].offset=offset;
327.     return symbolTable.index++;
328. }
329.
330. int LEV=0;      //层号
331. int func_size;
332.
333. void ext_var_list(struct node *T){ //处理变量列表
334.     int rtn,num=1;
335.     switch (T->kind){
336.         case EXT_DEC_LIST:
337.             T->ptr[0]->type=T->type;
338.             T->ptr[0]->offset=T->offset;
339.             T->ptr[1]->type=T->type;
340.             T->ptr[1]->offset=T->offset+T->width;
341.             T->ptr[1]->width=T->width;
342.             ext_var_list(T->ptr[0]);
343.             ext_var_list(T->ptr[1]);
344.             T->num=T->ptr[1]->num+1;
345.             break;
346.         case ID:
347.             rtn=fillSymbolTable(T->type_id,newAlias(),LEV,T->type,'V',T->of
fset); //最后一个
348.             if (rtn==-1)
349.                 semantic_error(T->pos,T->type_id, "Variable redefined");
350.             else T->place=rtn;

```

```

351.         T->num=1;
352.         break;
353.     }
354. }
355.
356. int match_param(int i, struct node *T, int pos){
357.     int j, num=symbolTable.symbols[i].paramnum; //
358.     int type1, type2;
359.     if (num==0 && T==NULL) return 1;
360.     for (j=1; j<=num; j++) {
361.         if (!T){
362.             semantic_error(pos, "", "Paramter is not enough");
363.             return 0;
364.         }
365.         type1=symbolTable.symbols[i+j].type; //形参类型
366.         type2=T->ptr[0]->type;
367.         if (type1!=type2){
368.             semantic_error(T->pos, "", "Paramter unmatched");
369.             return 0;
370.         }
371.         T=T->ptr[1];
372.     }
373.     if (T){
374.         semantic_error(T->pos, "", "Paramter is superabundant");
375.         return 0;
376.     }
377.     return 1;
378. }
379.
380. void boolExp(struct node *T){ // 参考文献[2]p84 的思想
381.     struct opn opn1, opn2, result;
382.     int op;
383.     int rtn;
384.     if (T)
385.     {
386.         switch (T->kind) {
387.             case INT: if (T->type_int!=0)
388.                 T->code=genGoto(T->Etrue);
389.                 else T->code=genGoto(T->Efalse);
390.                 T->width=0;
391.                 break;
392.             case FLOAT: if (T->type_float!=0.0)
393.                 T->code=genGoto(T->Etrue);
394.                 else T->code=genGoto(T->Efalse);

```

```

395.             T->width=0;
396.             break;
397.         case ID: //查符号表, 获得符号表中的位置
398.             rtn=searchSymbolTable(T->type_id);
399.             if (rtn==-1)
400.                 semantic_error(T->pos,T->type_id, "Variable undefined");
401.             if (symbolTable.symbols[rtn].flag=='F')
402.                 semantic_error(T->pos,T->type_id, "is function name");
403.             else {
404.                 opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[rtn].alias);
405.                 opn1.offset=symbolTable.symbols[rtn].offset;
406.                 opn2.kind=INT; opn2.const_int=0;
407.                 result.kind=ID; strcpy(result.id,T->Etrue);
408.                 T->code=genIR(NEQ,opn1,opn2,result);
409.                 T->code=merge(2,T->code,genGoto(T->Efalse));
410.             }
411.             T->width=0;
412.             break;
413.         case RELOP: //处理关系运算 exp , 2 个操作数都按基本 exp 处理
414.             T->ptr[0]->offset=T->ptr[1]->offset=T->offset;
415.             Exp(T->ptr[0]);
416.             T->width=T->ptr[0]->width;
417.             Exp(T->ptr[1]);
418.             if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;
419.             opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);
420.             opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;
421.             opn2.kind=ID; strcpy(opn2.id,symbolTable.symbols[T->ptr[1]->place].alias);
422.             opn2.offset=symbolTable.symbols[T->ptr[1]->place].offset;
423.             result.kind=ID; strcpy(result.id,T->Etrue);
424.             if (strcmp(T->type_id,"<")==0)
425.                 op=JLT;
426.             else if (strcmp(T->type_id,"<=")==0)
427.                 op=JLE;
428.             else if (strcmp(T->type_id,">")==0)
429.                 op=JGT;
430.             else if (strcmp(T->type_id,">=")==0)

```

```

431.                op=JGE;
432.                else if (strcmp(T->type_id,"==")==0)
433.                    op=EQ;
434.                else if (strcmp(T->type_id,"!=")==0)
435.                    op=NEQ;
436.                T->code=genIR(op,opn1,opn2,result);
437.                T->code=merge(4,T->ptr[0]->code,T->ptr[1]->code,T->code
,genGoto(T->Efalse));
438.                break;
439.            case AND:
440.            case OR:
441.                if (T->kind==AND) {
442.                    strcpy(T->ptr[0]->Etrue,newLabel());
443.                    strcpy(T->ptr[0]->Efalse,T->Efalse);
444.                }
445.                else {
446.                    strcpy(T->ptr[0]->Etrue,T->Etrue);
447.                    strcpy(T->ptr[0]->Efalse,newLabel());
448.                }
449.                strcpy(T->ptr[1]->Etrue,T->Etrue);
450.                strcpy(T->ptr[1]->Efalse,T->Efalse);
451.                T->ptr[0]->offset=T->ptr[1]->offset=T->offset;
452.                boolExp(T->ptr[0]);
453.                T->width=T->ptr[0]->width;
454.                boolExp(T->ptr[1]);
455.                if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->widt
h;
456.                if (T->kind==AND)
457.                    T->code=merge(3,T->ptr[0]->code,genLabel(T->ptr[0]-
>Etrue),T->ptr[1]->code);
458.                else
459.                    T->code=merge(3,T->ptr[0]->code,genLabel(T->ptr[0]-
>Efalse),T->ptr[1]->code);
460.                break;
461.            case NOT:  strcpy(T->ptr[0]->Etrue,T->Efalse);
462.                    strcpy(T->ptr[0]->Efalse,T->Etrue);
463.                    boolExp(T->ptr[0]);
464.                    T->code=T->ptr[0]->code;
465.                    break;
466.        }
467.    }
468. }
469.
470.

```

```

471. void Exp(struct node *T)
472. { //处理基本 exp , 参考文献[2]p82 的思想
473.     int rtn,num,width;
474.     struct node *T0,*Ttemp;
475.     struct opn opn1,opn2,result;
476.     if (T)
477.     {
478.         switch (T->kind) {
479.             case ID: //查符号表, 获得符号表中的位置
480.                 rtn=searchSymbolTable(T->type_id);
481.                 if (rtn==-1)
482.                     semantic_error(T->pos,T->type_id, "Variable undefined");
483.                 if (symbolTable.symbols[rtn].flag=='F')
484.                     semantic_error(T->pos,T->type_id, "is function name");
485.                 else {
486.                     T->place=rtn;
487.                     T->code=NULL; //标识符不需要生成 TAC
488.                     T->type=symbolTable.symbols[rtn].type;
489.                     T->offset=symbolTable.symbols[rtn].offset;
490.                     T->width=0; //未再使用新单元
491.                 }
492.                 break;
493.             case INT: T->place=fill_Temp(newTemp(),LEV,T->type,'T',T->offset); //
//为整常量生成 一个临时变量
494.                 T->type=INT;
495.                 opn1.kind=INT;opn1.const_int=T->type_int;
496.                 result.kind=ID; strcpy(result.id,symbolTable.symbols[T->place].alias);
497.                 result.offset=symbolTable.symbols[T->place].offset;
498.                 T->code=genIR(ASSIGNOP,opn1,opn2,result);
499.                 T->width=4;
500.                 break;
501.             case FLOAT: T->place=fill_Temp(newTemp(),LEV,T->type,'T',T->offset);
//为浮点常量生成 一个临时变量
502.                 T->type=FLOAT;
503.                 opn1.kind=FLOAT; opn1.const_float=T->type_float;
504.                 result.kind=ID; strcpy(result.id,symbolTable.symbols[T->place].alias);
505.                 result.offset=symbolTable.symbols[T->place].offset;
506.                 T->code=genIR(ASSIGNOP,opn1,opn2,result);
507.                 T->width=4;
508.                 break;

```

```

509.     case ASSIGNOP:
510.         if (T->ptr[0]->kind!=ID){
511.             semantic_error(T->pos,"", "Left must be left-
value(=)");
512.         }
513.         else {
514.             Exp(T->ptr[0]);
515.             T->ptr[1]->offset=T->offset;
516.             Exp(T->ptr[1]);
517.             T->type=T->ptr[0]->type;
518.             rtn=searchSymbolTable(T->ptr[0]->type_id);
519.             if ( symbolTable.symbols[rtn].type != T->ptr[1]->type){
//不匹配
520.                 semantic_error(T->pos,"", "Two sides' type unmatched"
);
521.             }
522.             T->width=T->ptr[1]->width;
523.             T->code=merge(2,T->ptr[0]->code,T->ptr[1]->code);
524.             opn1.kind=ID;   strcpy(opn1.id,symbolTable.symbols[T->p
tr[1]->place].alias);
525.             opn1.offset=symbolTable.symbols[T->ptr[1]->place].offse
t;
526.             result.kind=ID; strcpy(result.id,symbolTable.symbols[T-
>ptr[0]->place].alias);
527.             result.offset=symbolTable.symbols[T->ptr[0]->place].off
set;
528.             T->code=merge(2,T->code,genIR(ASSIGNOP,opn1,opn2,result
));
529.         }
530.         break;
531.     case PLUSASS:
532.         if (T->ptr[0]->kind!=ID){
533.             semantic_error(T->pos,"", "Left must be left-
value(+=)");
534.         }
535.         else {
536.             T->ptr[0]->offset = T->offset;
537.             Exp(T->ptr[0]);
538.             T->ptr[1]->offset = T->offset + T->ptr[0]->width;
539.             Exp(T->ptr[1]);
540.             T->type = T->ptr[0]->type;
541.             rtn = fill_Temp(newTemp(), LEV, T->type, 'T', T->offset
+ T->ptr[1]->width);

```

```

542.            T->place = fill_Temp(newTemp(), LEV, T->type, 'T', T->offset + T->ptr[1]->width + 4);
543.            T->width = T->ptr[1]->width + 4 * 2;
544.            T->code = merge(2, T->ptr[0]->code, T->ptr[1]->code);
545.            opn1.kind=ID; strcpy(opn1.id, symbolTable.symbols[T->ptr[0]->place].alias);
546.            opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;
547.            opn2.kind=ID; strcpy(opn2.id, symbolTable.symbols[T->ptr[1]->place].alias);
548.            opn2.offset=symbolTable.symbols[T->ptr[1]->place].offset;
549.            result.kind=ID; strcpy(result.id, symbolTable.symbols[rtn].alias);
550.            result.offset=symbolTable.symbols[rtn].offset;
551.            T->code = merge(2, T->code, genIR(PLUS, opn1, opn2, result));
552.            strcpy(opn1.id, symbolTable.symbols[rtn].alias);
553.            opn1.offset = symbolTable.symbols[rtn].offset;
554.            strcpy(result.id, symbolTable.symbols[T->ptr[0]->place].alias);
555.            result.offset = symbolTable.symbols[T->ptr[0]->place].offset;
556.            T->code = merge(2, T->code, genIR(ASSIGNOP, opn1, opn2, result));
557.        }
558.        break;
559.    case SELF_PLUS:
560.        if( T->ptr[1] ){//若有两个子树，则为非法
561.            semantic_error(T->pos, "", "expression cannot self plus");
562.        }
563.        if(T->type != INT ){//只能整型自增
564.            semantic_error(T->pos, "", "expression cannot self plus");
565.        }
566.        T->ptr[0]->offset = T->offset;
567.        Exp(T->ptr[0]);
568.
569.        T->type = T->ptr[0]->type;
570.        rtn = fill_Temp(newTemp(), LEV, T->type, 'T', T->offset );
571.        T->place = fill_Temp(newTemp(), LEV, T->type, 'T', T->offset + 4);
572.        T->width = T->ptr[0]->width + 4 * 2;
573.        opn1.kind=ID; strcpy(opn1.id, symbolTable.symbols[T->ptr[0]->place].alias);

```

```

574.         opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;
575.         T->place=fill_Temp(newTemp(),LEV,T->type,'T',T->offset);
576.         T->type=INT;
577.         opn2.kind=INT;opn2.const_int=1;
578.         T->width=4;
579.         result.kind=ID; strcpy(result.id, symbolTable.symbols[rtn].alias);
580.         result.offset=symbolTable.symbols[rtn].offset;
581.         T->code = genIR(PLUS, opn1, opn2, result);
582.         strcpy(opn1.id, symbolTable.symbols[rtn].alias);
583.         opn1.offset = symbolTable.symbols[rtn].offset;
584.         strcpy(result.id, symbolTable.symbols[T->ptr[0]->place].alias);

585.         result.offset = symbolTable.symbols[T->ptr[0]->place].offset;
586.         T->code = merge(2, T->code, genIR(ASSIGNOP, opn1, opn2, result)
    );
587.         break;
588.
589.     case AND:
590.     case OR:
591.     case RELOP:
592.         T->type=INT;
593.         T->ptr[0]->offset=T->ptr[1]->offset=T->offset;
594.         Exp(T->ptr[0]);
595.         Exp(T->ptr[1]);
596.         break;
597.
598.     case PLUS:
599.     case MINUS:
600.     case STAR:
601.     case DIV:    T->ptr[0]->offset=T->offset;
602.                 Exp(T->ptr[0]);
603.                 T->ptr[1]->offset=T->offset+T->ptr[0]->width;
604.                 Exp(T->ptr[1]);
605.                 if (T->ptr[0]->type==FLOAT || T->ptr[1]->type==FLOAT)
606.                     T->type=FLOAT,T->width=T->ptr[0]->width+T->ptr[1]->width+4;
607.                 else T->type=INT,T->width=T->ptr[0]->width+T->ptr[1]->width+2;
608.                 T->place=fill_Temp(newTemp(),LEV,T->type,'T',T->offset+T->ptr[0]->
width+T->ptr[1]->width);//
609.                 opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->
place].alias);

```



```

610.         opn1.type=T->ptr[0]->type;opn1.offset=symbolTable.symbols[T
->ptr[0]->place].offset;
611.         opn2.kind=ID; strcpy(opn2.id,symbolTable.symbols[T->ptr[1]-
->place].alias);
612.         opn2.type=T->ptr[1]->type;opn2.offset=symbolTable.symbols[T
->ptr[1]->place].offset;
613.         result.kind=ID; strcpy(result.id,symbolTable.symbols[T->pla
ce].alias);
614.         result.type=T->type;result.offset=symbolTable.symbols[T->pl
ace].offset;
615.         T->code=merge(3,T->ptr[0]->code,T->ptr[1]->code,genIR(T->ki
nd,opn1,opn2,result));
616.         T->width=T->ptr[0]->width+T->ptr[1]->width+(T->type==INT?4:
8);
617.         break;
618.     case NOT:
619.         break;
620.     case UMINUS:
621.         break;
622.     case FUNC_CALL: //根据 T->type_id 查出函数的定义, 如果语言中增加了实验教材的
read, write 需要单独处理一下
623.         rtn=searchSymbolTable(T->type_id);
624.         if (rtn!=-1){
625.             semantic_error(T->pos,T->type_id, "function name undefi
ned");
626.             break;
627.         }
628.         if (symbolTable.symbols[rtn].flag!='F'){
629.             semantic_error(T->pos,T->type_id, "is not a function na
me");
630.             break;
631.         }
632.         T->type=symbolTable.symbols[rtn].type;
633.         width=T->type==INT?4:8;
634.         if (T->ptr[0]){
635.             T->ptr[0]->offset=T->offset;
636.             Exp(T->ptr[0]);
637.             T->width=T->ptr[0]->width+width;
638.             T->code=T->ptr[0]->code;
639.         }
640.         else {T->width=width; T->code=NULL;}
641.         match_param(rtn,T->ptr[0], T->pos);
642.         T0=T->ptr[0];
643.         while (T0) {

```

```

644.         result.kind=ID;  strcpy(result.id,symbolTable.symbols[T
        0->ptr[0]->place].alias);
645.         result.offset=symbolTable.symbols[T0->ptr[0]->place].of
        fset;
646.         T->code=merge(2,T->code,genIR(ARG,opn1,opn2,result));
647.         T0=T0->ptr[1];
648.     }
649.     T->place=fill_Temp(newTemp(),LEV,T->type,'T',T->offset+T->w
        idth-width);
650.     opn1.kind=ID;    strcpy(opn1.id,T->type_id);
651.     opn1.offset=rtn;
652.     result.kind=ID;  strcpy(result.id,symbolTable.symbols[T->p
        lace].alias);
653.     result.offset=symbolTable.symbols[T->place].offset;
654.     T->code=merge(2,T->code,genIR(CALL,opn1,opn2,result));
655.     break;
656. case ARGS:
657.     T->ptr[0]->offset=T->offset;
658.     Exp(T->ptr[0]);
659.     T->width=T->ptr[0]->width;
660.     T->code=T->ptr[0]->code;
661.     if (T->ptr[1]) {
662.         T->ptr[1]->offset=T->offset+T->ptr[0]->width;
663.         Exp(T->ptr[1]);
664.         T->width+=T->ptr[1]->width;
665.         T->code=merge(2,T->code,T->ptr[1]->code);
666.     }
667.     break;
668. }
669. }
670. }
671.
672. void semantic_Analysis(struct node *T)
673. { //对抽象语法树的先根遍历
674.     int rtn,num,width;
675.     struct node *T0;
676.     struct opn opn1,opn2,result;
677.     if (T)
678.     {
679.         switch (T->kind) {
680.         case EXT_DEF_LIST:
681.             if (!T->ptr[0]) break;
682.             T->ptr[0]->offset=T->offset;
683.             semantic_Analysis(T->ptr[0]);    //访问外部定义列表中的第一个

```

```

684.         T->code=T->ptr[0]->code;
685.         if (T->ptr[1]){
686.             T->ptr[1]->offset=T->ptr[0]->offset+T->ptr[0]->width;
687.             semantic_Analysis(T->ptr[1]); //访问该外部定义列表中的其它外部
        定义
688.             T->code=merge(2,T->code,T->ptr[1]->code);
689.         }
690.         break;
691.     case EXT_VAR_DEF: //处理外部说明
692.         T->type=T->ptr[1]->type=!strcmp(T->ptr[0]->type_id,"int"?INT:F
        LOAT;
693.         T->ptr[1]->offset=T->offset; //这个外部变量 的偏移量向下传
        递
694.         T->ptr[1]->width=T->type==INT?4:8; //将一个变量 的宽度向下传递
695.         ext_var_list(T->ptr[1]); //处理外部变量 说明中的标识符
        序列
696.         T->width=(T->type==INT?4:8)* T->ptr[1]->num; //计算这个外部变
        量 说明的宽度
697.         T->code=NULL; //这里假定外部变量 不支持初始化
698.         break;
699.     case FUNC_DEF: //填写函数定义信息到符号表
700.         T->ptr[1]->type=!strcmp(T->ptr[0]->type_id,"int"?INT:FLOAT;//
        获取函数
701.         T->width=0; //函数的宽度设置为 0, 不会对外部变量 的地址分配产生
        影响
702.         T->offset=DX; //设置局部变量 在活动记录中的偏移量初值
703.         semantic_Analysis(T->ptr[1]);
704.         T->offset+=T->ptr[1]->width;
705.         T->ptr[2]->offset=T->offset;
706.         strcpy(T->ptr[2]->Snext,newLabel());
707.         semantic_Analysis(T->ptr[2]); //处理函数体
708.         //计算活动记录大小,这里 offset 属性存放的是活动记录大小, 不是偏移
709.         symbolTable.symbols[T->ptr[1]->place].offset=T->offset+T->ptr[2
        ]->width;
710.         T->code=merge(3,T->ptr[1]->code,T->ptr[2]->code,genLabel(T->ptr
        [2]->Snext));
711.         break;
712.     case FUNC_DEC:
713.         rtn=fillSymbolTable(T->type_id,newAlias(),LEV,T->type,'F',0);//
        函数不在数据区中分配单元, 偏移量为 0
714.         if (rtn==-1){
715.             semantic_error(T->pos,T->type_id, "function name redefined"
        );
716.         break;

```

```

717.         }
718.         else T->place=rtn;
719.         result.kind=ID;   strcpy(result.id,T->type_id);
720.         result.offset=rtn;
721.         T->code=genIR(FUNCTION,opn1,opn2,result);
722.         T->offset=DX;    //设置形参在活动记录中的偏移量初值
723.         if (T->ptr[0]) { //判断
724.             T->ptr[0]->offset=T->offset;
725.             semantic_Analysis(T->ptr[0]); //处理
726.             T->width=T->ptr[0]->width;
727.             symbolTable.symbols[rtn].paramnum=T->ptr[0]->num;
728.             T->code=merge(2,T->code,T->ptr[0]->code);
729.         }
730.         else symbolTable.symbols[rtn].paramnum=0,T->width=0;
731.         break;
732.     case PARAM_LIST:
733.         T->ptr[0]->offset=T->offset;
734.         semantic_Analysis(T->ptr[0]);
735.         if (T->ptr[1]){
736.             T->ptr[1]->offset=T->offset+T->ptr[0]->width;
737.             semantic_Analysis(T->ptr[1]);
738.             T->num=T->ptr[0]->num+T->ptr[1]->num;
739.             T->width=T->ptr[0]->width+T->ptr[1]->width;
740.             T->code=merge(2,T->ptr[0]->code,T->ptr[1]->code);
741.         }
742.         else {
743.             T->num=T->ptr[0]->num;
744.             T->width=T->ptr[0]->width;
745.             T->code=T->ptr[0]->code;
746.         }
747.         break;
748.     case PARAM_DEC:
749.         rtn=fillSymbolTable(T->ptr[1]->type_id,newAlias(),1,T->ptr[0]->
type, 'P',T->offset);
750.         if (rtn==-1)
751.             semantic_error(T->ptr[1]->pos,T->ptr[1]->type_id, "Paramter
name cannot be the same");
752.         else T->ptr[1]->place=rtn;
753.         T->num=1;
754.         T->width=T->ptr[0]->type==INT?4:8;
755.         result.kind=ID;   strcpy(result.id, symbolTable.symbols[rtn].al
ias);
756.         result.offset=T->offset;
757.         T->code=genIR(PARAM,opn1,opn2,result);

```

```

758.         break;
759.     case COMP_STM:
760.         LEV++;
761.         //设置层号加 1, 并且保存该层局部变量 在符号表中的起始位置
762.         symbol_scope_TX.TX[symbol_scope_TX.top++]=symbolTable.index;
763.         T->width=0;
764.         T->code=NULL;
765.         if (T->ptr[0]) {
766.             T->ptr[0]->offset=T->offset;
767.             semantic_Analysis(T->ptr[0]); //处理该层的局部变量
768.             T->width+=T->ptr[0]->width;
769.             T->code=T->ptr[0]->code;
770.         }
771.         if (T->ptr[1]){
772.             T->ptr[1]->offset=T->offset+T->width;
773.             strcpy(T->ptr[1]->Snext,T->Snext); //S.next 属性向下传递
774.             semantic_Analysis(T->ptr[1]);
775.             T->width+=T->ptr[1]->width;
776.             T->code=merge(2,T->code,T->ptr[1]->code);
777.         }
778.         prn_symbol();
779.         LEV--;
780.         symbolTable.index=symbol_scope_TX.TX[--
symbol_scope_TX.top]; //删除该作用域中的符号
781.         break;
782.     case DEF_LIST:
783.         T->code=NULL;
784.         if (T->ptr[0]){
785.             T->ptr[0]->offset=T->offset;
786.             semantic_Analysis(T->ptr[0]); //处理一个局部变量 定义
787.             T->code=T->ptr[0]->code;
788.             T->width=T->ptr[0]->width;
789.         }
790.         if (T->ptr[1]) {
791.             T->ptr[1]->offset=T->offset+T->ptr[0]->width;
792.             semantic_Analysis(T->ptr[1]); //处理剩下的局部变量 定义
793.             T->code=merge(2,T->code,T->ptr[1]->code);
794.             T->width+=T->ptr[1]->width;
795.         }
796.         break;
797.     case VAR_DEF://类似于上面的外部变量 EXT_VAR_DEF
798.         T->code=NULL;
799.         T->ptr[1]->type=!strcmp(T->ptr[0]->type_id,"int"?INT:FLOAT
;

```

```

800.          T0=T->ptr[1];
801.          num=0;
802.          T0->offset=T->offset;
803.          T->width=0;
804.          width=T->ptr[1]->type==INT?4:8;  //一个变量 宽度
805.          while (T0) {  //处理所以 DEC_LISTNode
806.              num++;
807.              T0->ptr[0]->type=T0->type;  //类型属性向下传递
808.              if (T0->ptr[1]) T0->ptr[1]->type=T0->type;
809.              T0->ptr[0]->offset=T0->offset;  //类型属性向下传递
810.              if (T0->ptr[1]) T0->ptr[1]->offset=T0->offset+width;
811.              if (T0->ptr[0]->kind==ID){
812.                  rtn=fillSymbolTable(T0->ptr[0]->type_id,newAlias(),
      LEV,T0->ptr[0]->type,'V',T->offset+T->width);//此处偏移量未计算, 暂时为 0
813.                  if (rtn==-1)
814.                      semantic_error(T0->ptr[0]->pos,T0->ptr[0]->type
      _id, "Variable undefined");
815.                  else T0->ptr[0]->place=rtn;
816.                  T->width+=width;
817.              }
818.              else if (T0->ptr[0]->kind==ASSIGNOP){
819.                  rtn=fillSymbolTable(T0->ptr[0]->ptr[0]->type_id
      ,newAlias(),LEV,T0->ptr[0]->type,'V',T->offset+T->width);//此处偏移量未计
      算, 暂时为 0
820.                  if (rtn==-1)
821.                      semantic_error(T0->ptr[0]->ptr[0]->pos,T0->
      ptr[0]->ptr[0]->type_id, "Variable redefined");
822.                  else {
823.                      T0->ptr[0]->place=rtn;
824.                      T0->ptr[0]->ptr[1]->offset=T->offset+T->wid
      th+width;
825.                      Exp(T0->ptr[0]->ptr[1]);
826.                      opn1.kind=ID; strcpy(opn1.id,symbolTable.sy
      mbols[T0->ptr[0]->ptr[1]->place].alias);
827.                      opn1.offset = symbolTable.symbols[T0->ptr[0]
      ]->ptr[1]->place].offset;
828.                      result.kind=ID; strcpy(result.id,symbolTabl
      e.symbols[T0->ptr[0]->place].alias);
829.                      result.offset = symbolTable.symbols[T0->ptr
      [0]->place].offset;
830.                      T->code=merge(3,T->code,T0->ptr[0]->ptr[1]-
      >code,genIR(ASSIGNOP,opn1,opn2,result));
831.                  }
832.                  T->width+=width+T0->ptr[0]->ptr[1]->width;

```

```

833.         }
834.         T0=T0->ptr[1];
835.     }
836.     break;
837.     case STM_LIST:
838.         if (!T->ptr[0]) { T->code=NULL; T->width=0; break;}
839.         if (T->ptr[1])
840.             strcpy(T->ptr[0]->Snext,newLabel());
841.         else
842.             strcpy(T->ptr[0]->Snext,T->Snext);
843.         T->ptr[0]->offset=T->offset;
844.         semantic_Analysis(T->ptr[0]);
845.         T->code=T->ptr[0]->code;
846.         T->width=T->ptr[0]->width;
847.         if (T->ptr[1]){
848.             strcpy(T->ptr[1]->Snext,T->Snext);
849.             T->ptr[1]->offset=T->offset; //顺序结构共享单元方式
850.             semantic_Analysis(T->ptr[1]);
851.             if (T->ptr[0]->kind==RETURN || T->ptr[0]->kind==EXP_STMT
                || T->ptr[0]->kind==COMP_STMT)
852.                 T->code=merge(2,T->code,T->ptr[1]->code);
853.             else
854.                 T->code=merge(3,T->code,genLabel(T->ptr[0]->Snext),
                    T->ptr[1]->code);
855.             if (T->ptr[1]->width>T->width) T->width=T->ptr[1]->width;
                //顺序结构共享单元方式
856.         }
857.         break;
858.     case IF_THEN:
859.         strcpy(T->ptr[0]->Etrue,newLabel());
860.         strcpy(T->ptr[0]->Efalse,T->Snext);
861.         T->ptr[0]->offset=T->ptr[1]->offset=T->offset;
862.         boolExp(T->ptr[0]);
863.         T->width=T->ptr[0]->width;
864.         strcpy(T->ptr[1]->Snext,T->Snext);
865.         semantic_Analysis(T->ptr[1]); //if 子句
866.         if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;
867.         T->code=merge(3,T->ptr[0]->code, genLabel(T->ptr[0]->Etrue)
            ,T->ptr[1]->code);
868.         break;
869.     case IF_THEN_ELSE:
870.         strcpy(T->ptr[0]->Etrue,newLabel()); //设置条件语句真假转移
            位置
871.         strcpy(T->ptr[0]->Efalse,newLabel());

```

```

872.          T->ptr[0]->offset=T->ptr[1]->offset=T->ptr[2]->offset=T->of
      fset;
873.          boolExp(T->ptr[0]);
874.          T->width=T->ptr[0]->width;
875.          strcpy(T->ptr[1]->Snext,T->Snext);
876.          semantic_Analysis(T->ptr[1]);          //if 子句
877.          if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;
878.          strcpy(T->ptr[2]->Snext,T->Snext);
879.          semantic_Analysis(T->ptr[2]);          //else 子句
880.          if (T->width<T->ptr[2]->width) T->width=T->ptr[2]->width;
881.          T->code=merge(6,T->ptr[0]->code,genLabel(T->ptr[0]->Etrue),
      T->ptr[1]->code,\
882.                      genGoto(T->Snext),genLabel(T->ptr[0]->Efalse)
      ,T->ptr[2]->code);
883.          break;
884.      case WHILE: strcpy(T->ptr[0]->Etrue,newLabel());
885.                  strcpy(T->ptr[0]->Efalse,T->Snext);
886.                  T->ptr[0]->offset=T->ptr[1]->offset=T->offset;
887.                  boolExp(T->ptr[0]);          //循环条件
888.                  T->width=T->ptr[0]->width;
889.                  strcpy(T->ptr[1]->Snext,newLabel());
890.                  semantic_Analysis(T->ptr[1]);          //循环体
891.                  if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;
892.                  T->code=merge(5,genLabel(T->ptr[1]->Snext),T->ptr[0]->code,
      \
893.                      genLabel(T->ptr[0]->Etrue),T->ptr[1]->code,genGoto(T->ptr[1]
      ]->Snext));
894.          break;
895.      case EXP_STMT:
896.                  T->ptr[0]->offset=T->offset;
897.                  semantic_Analysis(T->ptr[0]);
898.                  T->code=T->ptr[0]->code;
899.                  T->width=T->ptr[0]->width;
900.          break;
901.      case BREAK:
902.                  breakPos = T->pos;
903.          break;
904.      case RETURN:if (T->ptr[0]){
905.                  T->ptr[0]->offset=T->offset;
906.                  Exp(T->ptr[0]);
907.                  num=symbolTable.index;
908.                  do num--; while (symbolTable.symbols[num].flag!='F');
909.                  if (T->ptr[0]->type!=symbolTable.symbols[num].type) {

```



```

910.                semantic_error(T->pos, "", "Return value type unmatc
    h");
911.                T->width=0;T->code=NULL;
912.                break;
913.            }
914.            T->width=T->ptr[0]->width;
915.            result.kind=ID; strcpy(result.id,symbolTable.symbols[T-
    >ptr[0]->place].alias);
916.            result.offset=symbolTable.symbols[T->ptr[0]->place].off
    set;
917.            T->code=merge(2,T->ptr[0]->code,genIR(RETURN,opn1,opn2,
    result));
918.        }
919.        else{
920.            T->width=0;
921.            result.kind=0;
922.            T->code=genIR(RETURN,opn1,opn2,result);
923.        }
924.        break;
925.    case ID:
926.    case INT:
927.    case FLOAT:
928.    case ASSIGNOP:
929.    case PLUSASS:
930.    case SELF_PLUS:
931.    case AND:
932.    case OR:
933.    case RELOP:
934.    case PLUS:
935.    case MINUS:
936.    case STAR:
937.    case DIV:
938.    case NOT:
939.    case UMINUS:
940.    case FUNC_CALL:
941.        Exp(T);          //处理基本 exp
942.        break;
943.    }
944.    }
945. }
946.
947. void semantic_Analysis0(struct node *T) {
948.     symbolTable.index=0;
949.     fillSymbolTable("read", "", 0, INT, 'F', 4);

```

```

950.    symbolTable.symbols[0].paramnum=0;
951.    fillSymbolTable("x","",1,INT,'P',12);
952.    fillSymbolTable("write","",0,INT,'F',4);
953.    symbolTable.symbols[2].paramnum=1;
954.    symbol_scope_TX.TX[0]=0;    //外部变量 在符号表中的起始序号为 0
955.    symbol_scope_TX.top=1;
956.    T->offset=0;                //外部变量 在数据区的偏移量
957.    semantic_Analysis(T);
958.    if(T->hasBreak){
959.        semantic_error(breakPos,"", "unexpected syntax break");
960.    }
961.    prnIR(T->code);
962.    genObCode(T->code);
963. }

```