10.1 源语言及其定义

10.1.1 语法定义

语言具有典型的嵌套块结构特征，且程序由无参数、无返回值的子程序序列组成，其典型特征用下面的上下文无关文法进行定义：

program→*fundecls*
fundecls→*function fundecls*|*function*
function→*id* ( ) *block*
block→{*decls stmts*}
decls→ *type vardefs*; *decls*|ε
type→ **int**|**char**|**bool**
vardefs→*vardef*|*vardef*, *vardefs*
vardef→*id*
stmts→*stmt stmts*|ε
stmt→*assignstmt*|*ifstmt*|*readstmt*|*writestmt*|*callstmt*|*returnstmt*|*block*
assignstmt→*variable*=*bexpr*;
variable→*id*
callstmt→*id* ( );
returnstmt→**return**;
readstmt→**read**(*varlist*);
varlist→*variable*|*variable*, *varlist*
writestmt→**write**(*exprlist*);
exprlist→*expr*, *exprlist*|*expr*
ifstmt→**if** *bexpr* **then** *stmt*|**if** *bexpr* **then** *stmt* **else** *stmt*
bexpr→*expr*|*expr* < *expr*|*expr* > *expr*|*expr* ==*expr*
expr→*expr* + *expr*|*expr* * *expr*|(*expr*)|*variable*|*num*|*letter*|**true**|**false**
id→*alphabet idtail*
idtail→*alphabet idtail*|*digit idtail*|*ε*
alphabet→ a|.....|z|A|.....|Z
digit→ 0|1|.....|9
num→ 1 *numtail* |2 *numtail*|.....|9 *numtail*
numtail→*digit numtail*|ε
letter→‘*character*’

为了方便理解，本节及后续使用的文法中斜体的字符串表示非终结符号，即 *program*、*fundecls*、*function*、*decls*、*vardefs*、*vardef*、*stmts*、*stmt*、*assignstmt*、*variable*、*varlist*、*ifstmt*、*readstmt*、*writestmt*、*returnstmt*、*bexpr*、*expr*、*exprlist*、*term*、*factor*、*id*、*idtail*、*letter*、*digit*、*num*、*numtail*、*character*、*alphabet* 是非终结符号。其次，*character* 是 ASCII 值从 32 到 126 之间的字符，*digit* 是十个基数，*alphabet* 是大小写的 26 个英文字母。此外，‘,’、‘;’、‘>’、‘<’、‘=’、‘*’、‘+’、‘(’、‘)’、‘{’、‘}’、“’和‘”是一个字符的单词，同时也充当界符的功能。

非终结符 id 和 num 表示终结符号组成的两类序列。id 是字母开头后面是字母或者数字

的标识符，num 是数字开头后面仍然是数字的无符号整数。其次，*letter* 是字符常量。此外，int、char、if、then、else、true、false、bool、return，read、write 是由终结符组成的一些特殊单词，在程序中将作为关键字表示特殊的含义。对于语法分析来说，标识符、无符号整数、字符常量、关键字、运算符和界符可以认为是终结符，这个工作由词法分析完成。

一个程序如图 10-1 所示。该程序由一个称为“main”的子程序构成，并且该子程序由两个嵌套的 block 进行定义，外层的 block 定义了两个整型变量，里层的 block 定义两个整型变量。

```
main ()
{   int a1,b;
    read(a1,b);
    {   int x,y;
        if a1<b+2 then x=5+6*3; else x=6;
        write(x,5+4);
    }
    return;
}
```

图 10-1　一个示例程序

10.1.2　语义约束

首先，程序由一系列的无参数、无返回值的子程序组成。无返回值的子程序也称为过程。每个过程由过程名进行标识，return 表示过程结束返回调用者；最后一个过程名为 main，表示主程序的入口。

其次，标识符包括变量名和过程名。所有的标识符需要先声明再使用，变量所声明的类型可以是 int、char 和 bool 类型；因为 block 可以嵌套，在同一个 block 内禁止同名变量的重复声明，但是可以在不同的 block 中进行重复定义。block 中符号的引用满足最近匹配原则，也就是引用标识符 *x* 时，从内向外依次检查 block 中的第一个对 *x* 的声明。如果没有找到，那么 *x* 就是没有声明过的标识符。在图 10-1 所示的 SIMPLE 程序中，语句“if a1<b+2 then x=5+6*3; else x=6;”对 a1 和 b 的访问都是外层 block 声明的标识符。

第三，语言定义了一种关系表达式 *bexpr*，其运算对象可以是 bool 常量 true 或者 false，也可以是代数表达式经过关系运算符“>”、“<”和“==”计算的结果；关系表达式的结果是 bool 型。代数运算可以使用乘法运算符“*”和加法运算符“+”；赋值语句使用“=”运算符。

第四，字符类型的数据仅能进行比较、赋值、读入或者输出。

和大多数程序设计语言一样，乘法“*”和加法“+”满足左结合，且乘法“*”优先于加法“+”。然而，按照上一小节定义的文法，乘法“*”和加法“+”运算在表达式中并不一定满足优先级别，也不一定满足左结合。因此，可以将 *expr* 定义改写为如下的产生式：

expr→*expr* + *term* | *term*

term→*term* * *factor* | *factor*

factor→(*expr*) | *variable* | *num* | *letter* | **true** | **false**

此外，关系运算、代数运算和赋值，需要进行类型检查，只有类型匹配才可以进行运算。

10.2 词法分析的实现

10.2.1 词法记号

单词分为如下几类（称为词法记号）：

(1) 运算符：‘>’、‘<’、‘=’、‘*’、‘+’、‘(’、‘)’；

(2) 界符：‘,’、‘;’、‘{’、‘}’；

(3) 标识符：字母开头后面是字母或者数字的序列；

(4) 关键字：int、char、if、then、else、true、false、bool、write、read、return；

(5) 无符号整数：数字开头后面仍然是数字的序列；

(6) 字符常量：ASCAII 码值在 32 和 126 之间的字符。

采用枚举型定义词法记号：

```
enum tokenkind
{
  errtoken,    endfile,    id,         num,        letter,
  addtoken,    multoken,   ltctoken,   gtctoken,   eqctoken,
  comma,       semicolon,  lparen,     rparen,     lbrace,
  rbrace,      becomes,    iftoken,    elsetoken,  thentoken,
  chartoken,   booltoken,  inttoken,   falsetoken, truetoken,
  rettoken,    writetoken, readtoken,
};
```

其中，每个元素对应一个词法记号。例如 id 表示标识符，num 表示无符号整数，letter 表示字符常量，iftoken 表示关键字 if 等等。因为 id、num 和 letter 不仅仅表示一个单词，本质上分别表示一类单词，所以词法分析输出这些记号时还需要输出单词的“值”。

除了 id、num 和 letter，其它的词法记号仅表示一个单词，所以词法分析仅需要输出这些词法记号。其次，在语言基本符号之外增加了 errtoken 和 endfile 两个词法记号，errtoken 表示错误的字符串，endfile 表示文件结束标志 EOF。

10.2.2 单词的定义

单词的定义：

```
struct tokenstruct
{
    enum tokenkind kind;
    union
    {
        char *idname;
        int numval;
        char ch;
    } val;
};
struct tokenstruct token
```

采用联合体定义token的val，可以更有效利用存储空间。

如果是标识符，除了将token的kind值记为id，还需要记录该标识符本身的字符串，并将其存储在idname中。如果单词判断为无符号整数，那么将token的kind值记为num，同时将该整数对应的数值存储在numval中；如果单词判断为字符常量，将token的kind值记为letter，同时将该单词的字符存储在ch中。关键字、界符和运算符仅需要kind信息。

10.2.3 单词的识别

词法分析设计为一个函数：

struct tokenstruct gettoken()

函数 gettoken 由语法分析调用，从输入的文本中读入每个字符，剔除空格、制表符、换行，并返回一个 token。

10.3 语法分析的实现

10.3.1 文法分析和变换

乘法“*”和加法“+”需要满足左结合，且乘法“*”优先于加法“+”，所以代数表达式的 *expr* 定义如下：

expr→*expr* + *term*｜*term*

term→*term* * *factor*｜*factor*

显然，*expr* 定义包括了典型的左递归。进行等价变换，从而得到下面的产生式：

expr→*term tr*

tr→+ *term tr*｜ε

term→*factor tp*

tp→* *factor tp*｜ε

其次，包括了典型的左公因子：

vardefs→*vardef*｜*vardef* , *vardefs*

ifstmt→**if** *bexpr* **then** *stmt* ｜**if** *bexpr* **then** *stmt* **else** *stmt*

对这两个定义进行等价变换可以得到如下的产生式：

vardefs→ *vardef vardefstail*

vardefstail→, *vardef vardefstail* ｜ε

ifstmt→ **if** *bexpr* **then** *stmt iftail*

iftail→ **else** *stmt*｜ε

if 语句的二义性采用最简单的最近匹配原则来解决冲突。

10.3.2 递归下降分析的实现

递归下降分析程序的实现将一个非终结符号的识别过程编写成一个子程序，并且子程序根据非终结符号的候选式编写成分支代码。例如，对于非终结符号 *expr*、*tr* 和 *term*，实现它们语法分析的伪码如图 10-3 所示。

```
void expr() {
```

```
        term(); tr();
}
void tr(){
        if (token== addtoken) {
                gettoken(); term();tr();
        }
}
void term(){
        factor();tp();
}
```

图 10-3　expr、tr 和 term 语法分析伪代码

在实现 *expr* 语法分析的伪代码中，可以看到 tr()通过递归调用实现。因为递归可以用循环代替，从而减少子程序的数量，所以 *expr* 语法分析的伪码可以采用图 10-4 所示的伪代码实现。

```
void expr() {
        term();
        while (token== addtoken){
                gettoken(); term();
        }
}
```

图 10-4　消除 tr 递归的 expr 语法分析伪代码

和图 10-4 所示的分析过程相一致的产生式可以用下面的 BNF 来表示：

exp>::=*term* {‘+’ *term* }

在 BNF 中，引入新的元符号“{”和“}”，表示花括号内的语法成分可以重复。在花括号不加上下界时表示可重复 0 到任意次数，花括号有上下界时表示重复次数受上下界限制。此外，还引入元符号“[”、“]”，表示方括号内的成分为任选项。虽然在 BNF 中，习惯使用尖括号“<”和“>”将非终结符括起来，但是为了和 10.1 节的文法保持形式一致，仍然采用斜体字符串表示非终结符号。

将词法分析得到的 token 看作是终结符号，那么上下文无关文法可以改写为下面的 BNF：

program::= **id** ‘(’ ‘)’ *block* { **id** ‘(’ ‘)’ *block* }
block::=‘{’ *decls stmts* ‘}’
decls::={ *type vardefs* ‘;’}
type::=**int**｜**char**｜**bool**
vardefs::= *vardef*{‘,’ *vardef*}
vardef::=**id**
stmts::={ *stmt*}
stmt::= *assignstmt*｜*callstm*｜*returnstmt*｜*ifstmt*
　　｜*readstmt*｜*writestmt*｜*block*
assignstmt::= *variable* ‘=’ *bexpr*‘;’
variable::=**id**
callstm::=**id** ‘(’‘)’ ‘;’
returnstmt::=**return**‘;’
readstmt::=**read** ‘(’ *variable*{‘,’ *variable* }‘)’ ‘;’
writestmt::=**write** ‘(’ *expr* {‘,’ *expr*}‘)’ ‘;’

ifstmt::=**if** *bexpr***then** *stmt* [**else** *stmt*]

bexpr::=*expr*|*expr* '<' *expr*|*expr* '>' *expr*| *expr* '==' *expr*

e*xpr*::=*term* {'+' *term*}

term::= *factor* {'*' *factor*}

factor::='('*expr*')'|*variable*|**num** |**letter**|**true**|**false**

其中，''中的字符代表字符本身。

其次，应该注意一点，*stmts* 不能直接使用下面的 BNF 定义：

stmts::={ *assignstmt*|*callstm*|*returnstmt*|*ifstmt*|*readstmt*|*writestmt*|*block*}

因为 *ifstmt* 中真分支和假分支的语句不可以使用 *stmts* 定义，只能使用 *stmt* 定义。

10.4 符号表的实现

10.4.1 符号表的设计

为了实现标识符在不同的 block 中可以进行重复定义，在符号表中设置一个 level 属性来记录符号的作用域深度，以区分不同作用域中的符号。

因为要求标识符先声明再使用、禁止同名标识符在同一个 block 中重复声明，同时还需要进行类型的检查等，所以符号表需要存储的属性还包括每个符号的名字、类型。其次，每个变量都需要考虑它在运行时存储空间上的分配，所以每个变量还设置相对地址属性。此外，因为有些标识符表示变量，有些标识符表示过程名，而两者的用法不完全相同，所以还应该在符号表中进行区分。

符号表定义：

```
enum idform
{   var,
    proc
};
enum datatype
{   nul,
    inttype,
    chartype,
    booltype
};
struct tablestruct
{   char name[n];
    enum idform form;
    enum datatype type;
    int level;
    int address;
};
struct tablestruct table[tmax]
```

根据上面的数据结构，符号表中每个符号项的定义包括符号的名字 name、符号的种类 form、数据类型 datatype、所在块的深度 level 以及地址 address。其中，对于变量，address

表示相对地址，对于过程，address 表示子程序的入口标号。

因为函数同时具备变量和过程的一些特征，为了方便扩展函数，符号表的定义中区分了符号的种类 form 和数据类型 datatype；同时，datatype 中的 nul 值用于 proc 种类的标识符。

对于图 10-5 所示的示例程序，假设每个整型变量占 2 个存储空间，字符型和布尔型变量都占 1 个存储空间，而且最外层的 block 中的第一个变量的相对地址为 0，则当翻译程序运行到 B3 时，符号表中存储的符号如表 10-1 所示。

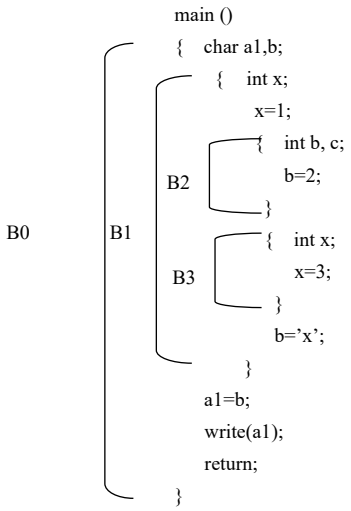

图 10-5　一个展示作用域的 SIMPLE 程序

在表 10-1 所示的符号表中，标识符 b 和 x 都重复定义了两次，但是因为它们所在的 block 不同，所以都是合法的声明。其次，当翻译程序运行到 B3 时，B2 块已经翻译结束，B2 块中的声明的变量 b 也不会再被后面的代码所引用，所以可以从符号表中删除。此外，“main”的 address 是子程序的入口标号是 L1，因为在中间代码生成阶段将会在子程序的入口处生成一条定义 L1 的三地址代码。

表 10-1　符号表示例

name	form	type	address	level
main	procedure		1	0
a1	variable	chartype	0	1
b	variable	chartype	1	1
x	variable	inttype	2	2
x	variable	inttype	4	3

10.4.2　符号表的管理

符号表由词法分析、语法分析、语义分析创建并由语义分析和中间代码生成使用。在大多数情况下，词法分析只能向语法分析返回词法记号以及单词的附加信息。因此，在实现符号表的时候，由语法分析决定符号是否需要相应的表项以及创建表项的工作。

采用两个函数 enter 和 found，实现符号的登记和查询的工作。enter 函数将当前符号的名字 id、符号种类 f、数据类型 dt、table 中的位置 tx、所在的作用域 lev、以及相对地址 off，

登记到符号表的表尾。enter 函数的伪代码如图 10-6 所示。其中，名字 id 是全局变量，并且假设 int 类型的数据占 2 个存储空间，布尔型和字符型仅占一个字节。

```
void enter(enum idform f, enum datatype dt,int lev,int * tx,int * off) {
        (*tx)++;
        table[(*tx)].form = f;
        table[(*tx)].type = dt;
        table[(*tx)].level = lev;
        table[(*tx)].address = (*off);
        if (dt==inttype)
                (*off)=(*off)+2;
        else if (dt==booltype ||dt==chartype )
                (*off)=(*off)+1;
}
```

图 10-6　登记符号表的 enter 函数

在符号表中，通过 level 值来体现符号的局部化单元，最外层 block 设置为 0，每进入一个嵌套 block，嵌套深度 level 值加 1。因为不允许重复声明，所以每个声明的符号需要先判定是否重复声明，如果是重复声明则不执行 enter 函数，并提示相应的错误信息。根据 name 和 level 值判定是否是重复声明，如果 level 相同，name 也相同，则判定为标识符重复声明。

found 函数判定名字 idname 的标识符是否登记在符号表中，主要的伪代码如图 10-7 所示。因为符号表按照先声明先填写的方法处理声明的标识符，所以嵌套最深的 block 声明的标识符在 table 的表尾。因此，found 函数从表尾开始查找，这样就满足最近声明原则。其次，在符号表的数组中，table[0]是空出的，所以每次执行 found 函数时，先把待查询的 idname 复制到 table[0].name 中，确保 idname 没有登记过符号表时，函数 found 返回 0。

```
int found(char* idname,int tx){
        strcpy(table[0].name, idname);
        i = tx;
        while (strcmp(table[i].name, idname) != 0)
                i--;
        return i;
}
```

图 10-7　符号表的 found 函数

符号表由词法分析、语法分析、语义分析创建并使用。在大多数情况下，词法分析只能向语法分析返回词法记号。因此，在实现符号表的时候，由语法分析确定将符号及其属性插入到符号表中。

10.5　中间代码生成

该翻译器采用两遍扫描的组织方式，第一遍扫描的结果将源程序转换成中间代码序列。

10.5.1 中间代码的定义

采用四元式的三地址代码表示源程序翻译得到的中间代码序列，具体四元式形式如下：

(op, operand1, operand2, result)

该四元式表示的含义是 result=operand1 op operand2，表示不同的操作码 op 对运算的分量 operand1 和 operand2 进行操作。操作码 op 定义：

```
enum IRcode_name
{   ADD,      SUB,      MUL,      DIV,      EQC,      LTC,
    GTC,      ASS,      LAB,      JPC,      JUMP,     RET,
    READ,     WRITE,    ENTRY,    CALL
 };
```

运算的分量可以是标号、常量、临时变量或者地址：

```
enum addr_kind
{   labelkind,    constkind,    tempkind,    varkind };
```

为了统一表示中间代码的各操作分量和操作结果，我们统一定义 addr 结构保存它们的基本信息。addr 的结构用下面 C 语言结构体进行定义：

```
struct addrRecord
{ enum addr_kind addrkind ;
   union
   {    int value;
        char c;
        bool b;
   } constant;
   char name[al];
   enum datatype type;
};
```

在 addr 的结构中，addrkind 用来区分操作分量的具体类型，value 用来存储常量值、标号和临时变量的编号或变量的偏移地址；c 用来存储字符类型的常量值；b 用来存储布尔类型的值；name 为了直观地输出中间代码而使用；datatype 用来存储符号的数据类型。

使用下面的结构体定义每一条中间代码：

```
struct IRCodeR
{     IRcode_name IRcodename;
      addrRecord   *addr1;
      addrRecord   *addr2;
      addrRecord   *addr3;
} ;
```

表 10-2　中间代码及含义

种类	中间代码	含义
关系运算	(EQC, addr1, addr2, addr3)	If addr1=addr2 then addr3=1 else addr3=0
	(LTC, addr1, addr2, addr3)	If addr1<addr2 then addr3=1 else addr3=0
	(GTC, addr1, addr2, addr3)	If addr1>addr2 then addr3=1 else addr3=0

代数运算	(ADD, addr1, addr2, addr3)	Addr3= addr1+addr2
	(MUL, addr1, addr2, addr3)	Addr3= addr1*addr2
语句	(READ, -, -, addr3)	输入语句
	(WRITE, -, -, addr3)	输出语句
	(LAB, -, -, addr3)	标号语句
	(ASS, addr1, -, addr3)	Addr3= addr1
	(JUMP, -, -, addr3)	无条件调转到 addr3
	(JPC, addr1, -, addr3)	若 addr1=0 则跳转到 addr3，否则顺序执行
	(ENTRY, addr1, addr2, addr3)	过程入口。其中 addr1 是过程产生临时变量所占的空间大小；addr2 是过程声明变量所占的空间大小；addr3 是过程入口标号
	(CALL, -, -, addr3)	调用入口地址为 addr1 的过程
	(RET, -, -, -)	返回调用过程

一个程序最后翻译得到的四元式序列存储到一个一维的全局结构体数组中：

struct IRCodeR IRcode[IRmax]

10.5.2 生成中间代码的构造

1.声明语句

声明仅包括变量和过程的声明，变量声明部分主要是结合程序的语义进行符号信息的登记和管理。除了过程体的代码，过程声明主要完成活动记录的空间计算并生成相应的指令，基本思想和过程如下：

(1) 编译程序开始，生成一条跳转指令确保程序永远从主程序开始执行。非终结符 *program* 的中间代码生成方法可以表示为下面的语法制导定义：

program→*fundecls*

{ *mainlab*=addrLabel(NewLabel());
program.code=genIR(JUMP,NULL, NULL, *mainlab*)
|| *fundecls.code*}

其中，*mainlab* 是主程序的入口标号，函数 addrLabel 创建一个标号类型的四元式分量，函数 NewLabel 创建一个新的标号；*code* 属性表示生成的中间代码序列；*program* 翻译的中间代码由 JUMP 指令和 *fundecls* 代码顺序组成。

(2) 子程序声明中，当分析到子程序名 id 的时候，生成子程序的入口指令。子程序声明的中间代码生成方法可以表示为下面的语法制导定义：

fundecls→*function fundecls*$_1$

{ *fundecls.code*= *function.code* || *fundecls*$_1$*.code*}

fundecls→*function*

{ *fundecls.code*= *function.code* }

function→**id** ‘(‘‘)’ *block*

{ *tempsize*=addrconst();
varsize= addrconst();
lab=addrLabel(NewLabel());
enter(proc,nul, lev, &tx, &off);

function.code= genIR(ENTRY, *tempsize*, *varsize*, *lab*)
|| *block.code*}

其中，函数 addrconst 是创建一个常量类型的四元式分量；*tempsize* 记录子程序生成的临时变量的空间大小，*varsize* 记录子程序声明的变量空间大小，*lab* 是子程序的入口标号。如果 id 为“main”，则 *lab* 就是 *mainlab*；enter 将过程 id 及其相关信息登记到符号表中。其次，*varsize* 和 *tempsize* 属性需要子程序的 block 分析结束进行回填。

(3) 子程序 *block* 除了对符号表中变量的初始偏移地址、符号表的指针、过程的临时变量以及空间初始化，主要是根据 *decls* 定义完成变量声明处理。*block* 的中间代码主要由 *stmts* 决定：

block→‘{’ *decls stmts*‘}’
{ *block.code*= *stmts.code*}

其中，*decls* 是变量声明，涉及下面的产生式：

decls→ *type vardefs*; *decls* | ε
type→ **int** | **char** | **bool**
vardefs→*vardef* | *vardef*, *vardefs*
vardef→*id*

变量声明部分主要是结合程序的语义进行符号信息的登记和管理，不需要生成中间代码。

2.语句序列

语句序列的中间代码分别由具体的语句决定：

stmts→*stmt stmts*$_1$
{ *stmts.code*= *stmt.code*||*stmts*$_1$*.code*}
stmts→ε
{ *stmts.code*={}}
stmt→*assignstmt*
{ *stmt.code*= *assignstmt.code* }
stmt→*callstmt*
{ *stmt.code*=*callstmt.code*}
stmt→*returnstmt*
{ *stmt.code*= *returnstmt.code* }
stmt→*ifstmt*
{ *stmt.code*= *ifstmt.code* }
stmt→*readstmt*
{ *stmt.code*= *readstmt.code* }
stmt→*writestmt*
{ *stmt.code*= *writestmt.code* }
stmt→*block*
{ *stmt.code*= *block.code* }

在语句序列中，处理难点是 *stmt* 中 *block* 定义。因为 block 可以顺序定义，也可以嵌套定义。如果两个 block 结构 B1 和 B2 是顺序定义，那么分析 B2 之前，需要将 B2 的变量初始偏移地址、符号表的指针恢复为与 B1 分析前一致；其次，需要比较 B1 和 B2 声明的变量空间大小，并使用大者作为分析 B2 之后的空间偏移。这个值除了用于分配嵌套在 B2 中 block 声明的每一个变量，也最终用于计算子程序的 *varsize*。

3. 表达式

关系表达式是运算对象为常量、变量或者代数表达式的表达式。对于“<”的关系运算，生成中间代码的翻译模式如下：

bexpr→*expr*$_1$ < *expr*$_2$

{ *bexpr.addr*= NewTemp();
bexpr.code=*expr*$_1$.*code* || *expr*$_2$.*code*
||genIR(LTC,*expr*$_1$.*addr*, *expr*$_2$.*addr*, *bexpr.addr*)}

其中，NewTemp()函数生成一个临时变量类型的四元式分量，这里为了简洁省略了NewTemp()的参数。其次，*addr* 属性表示四元式的运算分量；*expr.addr* 和 *expr.code* 是 *expr* 的综合属性，在翻译关系运算之前可以获得。

对定义“==”和“>”关系的产生式，处理方法和“<”产生式类似，仅仅只是最后一条指令不同：

bexpr→*expr*$_1$ > *expr*$_2$

{ *bexpr.addr*=NewTemp();
bexpr.code== *expr*$_1$.*code* || *expr*$_2$.*code*
||genIR(GTC,*expr*$_1$.*addr*, *expr*$_2$.*addr*, *bexpr.addr*)}

bexpr→*expr*$_1$==*expr*$_2$

{ *bexpr.addr*= NewTemp();
bexpr.code= *expr*$_1$.*code* || *expr*$_2$.*code*
||genIR(EQC,*expr*$_1$.*addr*, *expr*$_2$.*addr*, *bexpr.addr*)}

类似地，对代数运算表达式，生成中间代码的翻译模式如下：

expr→*expr*$_1$ + *term*

{ *expr.addr*= NewTemp();
expr.code=genIR(ADD,*expr*$_1$.*addr*, *term.addr*, *expr.addr*)}

expr→*term*

{ *expr.addr*= *term.addr* ;*expr.code*=*term.code* }

expr→(*expr*$_1$)

{*expr.addr*=*expr*$_1$.*addr*; *expr.code*= *expr*$_1$.*code* }

term→*term*$_1$ **factor*

{ *term.addr* =NewTemp();
term.code= *term*$_1$.*code*|| *factor*.code
||genIR(MUL,*term*$_1$.*addr*, *factor.addr*, *term.addr*)}

term→*factor*

{ *term.addr* =*factor.addr*; *term.code*=*factor.code*}

factor→(*expr*)

{ *factor.addr* = *expr.addr*; *factor.code*= *expr.code*}

factor→*variable*

{ *factor.addr* = *variable.addr*; *factor.code*={} }

factor→*num*

{ *factor.addr* =*addrconst*(); *factor.code*={}}

factor→*letter*

{ *factor.addr* =*addrconst*(); *factor.code*={}}

factor→**true**

{*factor.addr* =*addrconst*(); *factor.code*={}}

factor→**false**

{*factor.addr* =*addrconst*(); *factor.code*={}}

其中，*addrconst*()是创建一个存储常量的四元式分量，对应不同类型的常量，将常量值存储到四元式分量中。

4. 赋值语句

对于赋值语句，生成式中间代码的翻译模式如下：

assignstmt→*variable*=*bexpr*;

{ *assignstmt*.code=genIR(ASS,*bexpr.addr*, NULL, *variable.addr*)}

variable→**id**

{ *i*=found(*id.name*,**tx*);

if (i!=0) *variable. addr*=addrvar()}

其中，函数 addrvar 是创建一个变量类型的四元式分量，为了简洁省略了 addrvar 的参数。如果采用自顶向下分析方法实现翻译程序，删除表达式文法的左递归还需要等价地变换产生式的语义。表达式左递归删除对语义的影响和变换，在 5.4.1 一节给出详细的变换方法以及结果，此处不再赘述。

5. if 语句

if 语句生成中间代码的翻译模式可以定义如下：

stmt→ **if** *bexpr* **then** $stmt_1$

{ *bexpr_false* =addrLabel(NewLabel());

stmt.code=*bexpr.code*

|| (JPC, *bexpr.addr*, NULL, *bexpr_false*)

|| $stmt_1$.*code*

|| (LAB, *bexpr_false*, NULL, NULL)}

stmt→ **if** *bexpr* **then** $stmt_1$ **else** $stmt_2$

{ *bexpr_false*=addrLabel(NewLabel());

stmt_next =addrLabel(NewLabel());

stmt.code=*bexpr.code*

|| (JPC, *bexpr.addr*, NULL, *bexpr_false*)

|| $stmt_1$.*code*

|| (JUMP, NULL,NULL, *stmt_next*)

|| (LAB, *bexpr_false*, NULL, NULL)

|| $stmt_2$.*code*

|| (LAB, *stmt_next*, NULL, NULL) }

其中，*bexpr_false* 和 *stmt_next* 是标号，分别表示 if 语句中布尔表达式为假或者 if 语句结束跳转的目标。结合中间代码的设计，采用指令 LAB 完成创建标号的工作。

6. read 语句

readstmt→**read**(*varlist*);

{ *readstmt.code*= *varlist.code*}

varlist→*variable*

{ *varlist.code*=genIR(READ,NULL,NULL, *variable.addr*)}

varlist→*variable*, *varlist*$_1$

{ *varlist.code*=genIR(READ,NULL,NULL, *variable.addr*)
|| *varlist*$_1$.*code* }

variable→**id**

{ *i*=found(*idname*,**tx*);
if (i!=0) *variable. addr*=addrvar() else error();}

7. write 语句

writestmt→**write**(*exprlist*);

{ *writestmt.code*= *exprlist.code*}

exprlist→*expr*

{ *exprlist.code*=genIR(WRITE,NULL,NULL, *expr.addr*)}

exprlist→*expr*, *exprlist*$_1$

{ *exprlist.code*=genIR(WRITE,NULL,NULL, *expr.addr*)
|| *exprlist*$_1$.*code* }

8. call 语句

callstmt→**id** ( );

{ *i*=found(*id.name*);
if (i!=0)
{*lab*=addrLabel(*table*[*i*].*address*);
callstmt.code=genIR(CALL,NULL,NULL,*lab*);
else error();}
}

其中，lab 引用子程序 id 定义的入口标号，如果 id 没有声明过则报错。

9. return 语句

returnstmt→**return**;

{ *returnstmt.code*=genIR(RET,NULL,NULL,NULL)}

10.5.3 中间代码生成和优化

采用增量翻译的方式将生成的中间代码增量地存储到一个全局空间 IRcode 中。子程序 genIR 实现生成一条中间代码并将其写到 IRcode，写入的位置通过中间代码指针 NextIR 指示，每生成一条中间代码，NextIR 计算器增加 1。NextIR 是一个全局变量，但是仅仅 genIR 有权修改 NextI。

当一个程序翻译结束，得到的中间代码存储在 IRcode 数组中，通过子程序 PrintIR 可以直观的查看源程序翻译的结果。

例如，图 10-1 所示的程序经过翻译和常量合并得到的中间代码序列如表 10-3 所示。

表 10-3 (图 10-1)的中间代码和常量合并之后的优化代码

中间代码	常量合并之后的优化代码
[0] (JUMP,-,-,L1) [1] (ENTRY,9,8,L1)	[0] (JUMP,-,-,L1) [1] (ENTRY,9,8,L1)

[2] (READ,-,-,a1)	[2] (READ,-,-,a1)
[3] (READ,-,-,b)	[3] (READ,-,-,b)
[4] (ADD,b,2,T1)	[4] (ADD,b,2,T1)
[5] (LTC,a1,T1,T2)	[5] (LTC,a1,T1,T2)
[6] (JPC,T2,-,L2)	[6] (JPC,T2,-,L2)
[7] (MUL,6,3,T3)	[7] (ASS,18,-,T3)
[8] (ADD,5,T3,T4)	[8] (ADD,5,T3,T4)
[9] (ASS,T4,-,x)	[9] (ASS,T4,-,x)
[10] (JUMP,-,-,L3)	[10] (JUMP,-,-,L3)
[11] (LAB,-,-,L2)	[11] (LAB,-,-,L2)
[12] (ASS,6,-,x)	[12] (ASS,6,-,x)
[13] (LAB,-,-,L3)	[13] (LAB,-,-,L3)
[14] (WRITE,-,-,x)	[14] (WRITE,-,-,x)
[15] (ADD,5,4,T5)	[15] (ASS,9,-,T5)
[16] (WRITE,-,-,T5)	[16] (WRITE,-,-,T5)
[17] (RET,-,-,-)	[17] (RET,-,-,-)

在表 10-3 所示的中间代码中，为了提高可读性，使用 L1、L2、L3 分别对应标号 1，2、3。同理，使用 T1、T2、T3、T4、T5 分别对应临时变量编号 1、2、3、4、和 5。

10.6 目标代码生成

10.6.1 虚拟目标机

1. 寄存器和存储器

虚拟目标机包括 9 个寄存器，其中 6 个是通用寄存器 ax、bx、cx、dx、top 和 bp。通用寄存器可以作为普通的数据寄存器使用，其中 bp 和 top 是指针寄存器，主要用于存储器操作数的寻址，且 bp 记录基地址，top 记录栈顶。3 个专用寄存器是 pc、flag 和 ip，其中 flag 是标志寄存器，用于指示比较指令的状态；pc 是程序计数器，用于存放指令的地址；ip 是指令寄存器，用来保存当前正在执行的一条指令。

为了使翻译过程简单并且结构清晰，翻译器使用一个整型数组进行定义 6 个是通用寄存器和 2 个专用寄存器 pc 和 flag。ip 是一个结构体变量，与代码段的存储单元的结构一致。也就是说，字符型和布尔型在目标代码中都转换成整型数据进行访问。

虚拟目标机还包括两个存储器，分别用于存储指令和数据。存储指令的代码段由结构数组定义，存储数据的数据段是一个整型数组。定义它们的数据结构如下：

```
int reg [Rnum];
int dMem [DSIZE];
typedef struct {
        int iop;
        int iarg1;
        int iarg2;
        int iarg3;
    } INSTRUCTION;
INSTRUCTION iMem [ISIZE]
```

2. 指令集

虚拟目标机指令一般包括四个可选的组成部分：

iop r, d, s

其中，iop 是必需的操作码，表示要执行的操作，如加减乘除等；r、d、s 在不同的指令中表示的含义不同。下面结合虚拟机提供的指令具体解释如下：

(1) opIN：有两个参数 r 和 d，如果 d=0，指令将外部整型变量读入寄存器 r；如果 d=1，指令将外部字符型变量读入寄存器 r；
(2) opOUT：有两个参数 r 和 d，如果 d=0，指令将寄存器 r 的内容按照整型输出，如果 d=1，指令将寄存器 r 的内容按照字符型输出；
(3) opADD：有三个参数，指令将寄存器 d 的值加寄存器 s 的值存入寄存器 r；
(4) opSUB：有三个参数，指令将寄存器 d 的值减寄存器 s 的值存入寄存器 r；
(5) opMUL：有三个参数，指令将寄存器 d 的值乘寄存器 s 的值存入寄存器 r；
(6) opDIV：有三个参数，指令将寄存器 d 的值除寄存器 s 的值存入寄存器 r；
(7) opLD：有三个参数，并且 r 和 s 是寄存器，d 是立即数，指令将地址为 d+reg(s)的内存单元的值存入寄存器 r；
(8) opST：有三个参数，并且 r 和 s 是寄存器，d 是立即数，指令将寄存器 r 中的值存入地址 d+reg(s)的内存单元；
(9) opLDA：有三个参数，并且 r 和 s 是寄存器，d 是立即数，指令将 d+reg(s)值存入寄存器 r；
(10) opLDC：有两个参数 r 和 d，指令将立即数 d 放入寄存器 r；
(11) opMOV：有两个参数 r 和 d，指令将寄存器 d 的值存储到寄存器 r 中；
(12) opPUSH：有一个参数 r，指令将寄存器 r 的值压入栈顶，top 指针加 1；
(13) opPOP：有一个参数 r，指令将 top 指针减 1，且栈顶值存入寄存器 r；
(14) opJNL：如果寄存器 flag 不小于 0，则 r 的值存入寄存器 pc，即下一条指令将跳转到 r；
(15) opJNG：如果寄存器 flag 不大于 0，则 r 的值存入寄存器 pc，即下一条指令将跳转到 r；
(16) opJNE：如果寄存器 flag 的值不等于 0，则 r 的值存入寄存器 pc，即下一条指令将跳转到 r；
(17) opJUMP：r 的值存入寄存器 pc，即下一条指令将无条件跳转到 r。

10.6.2 运行时刻环境

过程的活动记录设计如图 10-8 所示。其中，返回地址用于存储被调过程返回时的返回地址，控制链用于存储主调过程活动记录的基地址；活动记录的每一个单元可以通过 bp 以及与 bp 的相对偏移地址进行访问。其次，虽然临时变量的空间在编译时最终可以确定，但代码生成或优化可能会缩减过程所需的临时变量，因此，把临时变量安排在局部变量的后面，因为它长度的改变不会影响其它的数据对象。

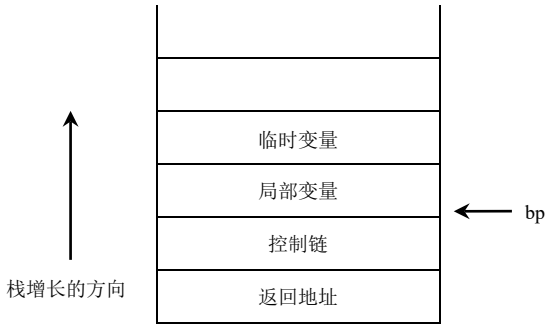

图 10-8　活动记录的设计方案

虽然语言的 block 具有嵌套定义的特征，但是这种嵌套仅局限于子程序内部的 block。为了提高空间的利用率，进入 block 时分配变量声明的空间，在退出 block 时释放占用的空间。一个子程序内声明的嵌套 block 依次分配在活动记录中，但是顺序声明的 block 不是同时活跃，所以这些 block 在数据空间上是重叠的。

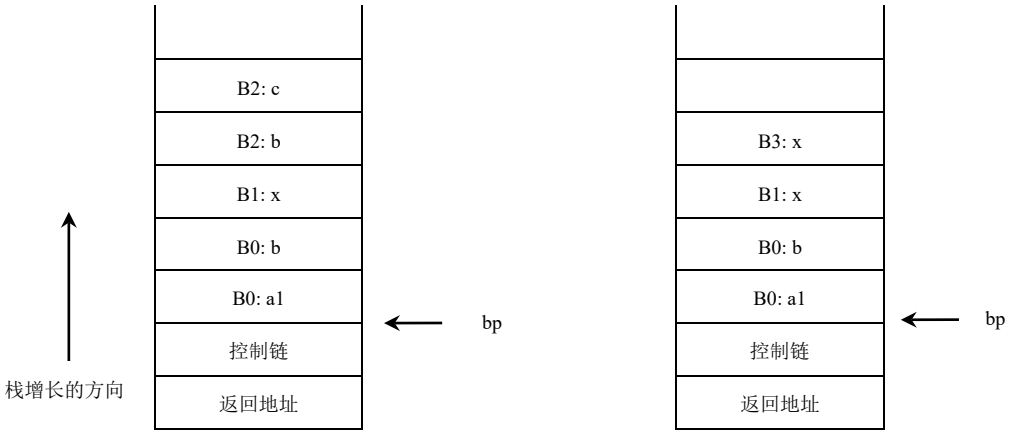

图 10-9　SIMPLE 程序(图 10-5)分别运行 B2 时刻和 B3 时刻的活动记录布局

对图 10-5 所示的程序，块结构 B2 和 B3 就是顺序声明。当程序运行到 B2 时，数据段中的空间布局仅包含 B0、B1、B2，当程序运行到 B3 时，数据段中的空间布局仅包含 B0、B1、B3，如图 10-9 所示。因为 B2 和 B3 不同时活跃，所以 B2 中的 b 和 B3 中的 x 是重叠的，也就是说 B2 中 b 和 B3 中 x 的地址是相同的。

需要注意的是，为了展示不同的数据类型所占空间大小不同的具体处理，翻译器虽然为源程序中的整型变量分配两个单元的空间，但是在虚拟机的数据段中，整型数据仅使用低地址单元进行存储。

10.6.3　目标代码生成

目标代码生成在中间代码生成的基础上进行，因此目标代码生成本质上是进行第二遍扫描。其次，因为中间代码的每条四元式都由有限个操作码中的一个唯一地进行确定，并且每个操作码对应的四元式有固定的格式，所以中间代码生成的基本思想就是依次扫描每一条四元式，并且根据每一个操作码以及固定的结构将其翻译成目标代码。

下面针对一些典型的四元式，解释它们的翻译过程。例如，四元式(ADD,5, 3,T4)可以根

据下面的过程进行翻译：

(1) 取立即数 5 到寄存器 ax，生成指令 LDC ax 5；

(2) 取立即数 3 到寄存器 bx，生成指令 LDC bx 3；

(3) 生成加法指令 ADD ax ax bx

(4) 将运算结果储存到 T4 对应的空间中，生成指令 ST ax 10 bp。

过程入口的四元式(ENTRY,9,8,L1)可以根据下面的过程进行翻译：

(1) 保存主调过程的基地址，生成指令 PUSH bp；

(2) 建立当前过程的基地址，将 top 寄存器中的值储存到 bp 中，生成指令 MOV bp top；

(3) 开辟当前过程的空间，生成指令 LDA top 17 bp。

过程退出的四元式(RET,-,-,-)可以根据下面的过程进行翻译：

(1) 释放过程占用的空间，生成指令 MOV top bp；

(2) 恢复主调过程的基地址，生成指令 POP bp；

(3) 跳转到返回地址，生成指令 POP pc。

临时变量和跳转指令仍然是目标代码生成过程中面临的两个关键问题。

临时变量是中间代码生成过程中产生的变量，它们的数量和具体的源程序、中间代码形式以及中间代码生成的算法都有关系。临时变量在目标代码中可以映射为内存单元或者寄存器。如果临时变量映射到数据段的内存单元，那么编译器还需要将临时变量映射为地址。翻译器在目标代码阶段完成临时变量的地址映射。

跳转指令主要涉及到三个操作码：JUMP、JPC 和 CALL。在这三个操作码对应的四元式中，第三个操作数引用的是标号。这些标号对应的目标可能是通过操作码 LAB 和 ENTRY 进行定义。因为目标代码生成过程中，翻译 JUMP、JPC 和 CALL 的时候，它们的跳转目标可能还未进行翻译，其次翻译 LAB 和 ENTRY 的时候，可能还存跳转到这些位置跳转指令还未进行翻译，所以需要将定义性出现的标号和引用性出现的标号都记录下来，并在适当的时机进行回填相应目标指令的跳转目标。

对 LAB 和 ENTRY 定义的标号 L，标号 L 都是四元式中的第四个单元，可以采用如下基本相同的处理方法：

(1) 在标号地址表中，检查标号 L 是否存在；如果存在报错，不存在执行下一步；

(2) 将标号 L 和它对应的目标指令地址登记到标号地址表中。

对 JUMP、JPC 和 CALL 引用的标号 L，也可以采取基本相同的方法：

(1) 在标号地址表中，检查标号 L 是否存在；如果存在，则使用标号地址表中 L 的地址生成完整的跳转指令，不存在则执行下一步；

(2) 将该跳转指令的地址和标号 L 登记到跳转指令表中。

对于不满足上面的跳转指令，可以在每次碰到 LAB 和 ENTRY 查询定义的标号是否已经存在对应的跳转指令，并进行回填处理，也可以在翻译完所有的指令之后进行回填处理。图 10-1 所示的源代码经过翻译器翻译，最终可以获得目标代码。使用函数 PrintObject 显示的目标代码如图 10-10 所示。

[0] JUMP 1	[20] LDC ax 5
[1] PUSH bp	[21] LD bx 10 bp
[2] MOV bp top	[22] ADD ax ax bx
[3] LDA top 8 bp	[23] LDA top 2 top
[4] IN ax 0	[24] ST ax 12 bp
[5] ST ax 0 bp	[25] LD ax 12 bp
[6] IN ax 0	[26] ST ax 4 bp
[7] ST ax 2 bp	[27] JUMP 30

[8] LD ax 2 bp	[28] LDC ax 6
[9] LDC bx 2	[29] ST ax 4 bp
[10] ADD ax ax bx	[30] LD ax 4 bp
[11] LDA top 2 top	[31] OUT ax 0
[12] ST ax 8 bp	[32] LDC ax 9
[13] LD ax 0 bp	[33] LDA top 2 top
[14] LD bx 8 bp	[34] ST ax 14 bp
[15] SUB flag ax bx	[35] LD ax 14 bp
[16] JNL 28	[36] OUT ax 0
[17] LDC ax 18	[37] MOV top bp
[18] LDA top 2 top	[38] POP bp
[19] ST ax 10 bp	[39] POP pc

图 10-10　示例程序(图 10-1)的目标程序

10.6.4　虚拟机解释程序

虚拟机 VM 是一个取值、执行的循环，执行的基本过程如图 10-11 所示。虚拟机 VM 执行在开始执行时，先将寄存器 pc、top、sb 设置为 0，然后从 iMem[0]取出放入 ip 寄存器开始执行，除非执行的指令修改了 pc 寄存器，否则 pc 加 1 执行下一条指令。

```
void VM() {
        reg[top]= 0;
        reg[bp]= 0;
        reg[pc]= 0;
        do{
            ip=iMem[reg[pc]];
            reg[pc]=reg[pc]+1;
            switch (ip.op){
                        case opIN:……;break;
                        case opOUT:…;break;
                        case opADD:…;break;
                        ……
            }
        }while reg[pc]!=0)
    }
```

图 10-11　虚拟机 VM 基本过程

虚拟机 VM 执行过程中，循环中主要是一个 switch 语句，根据每条指令的操作码进行解释执行。