语义分析与中间代码生成

第 6 章　语义分析与中间代码生成

6.1　语法制导翻译

6.2　符号表

6.3　类型检查

6.4　中间语言

6.5　中间代码生成

- 源程序流基本组成单位

固定单词(关键字、运算符、界限符…):

表示语句性质，反映语句结构;

用户定义的单词—标识符:

表示程序中各种实体;

如，变量名; 常量名; 标号名; 过程名; 函数名; 文件名，数组名 …

一般变量：name , type , addr , level …

数组：name , type , addr , dim , level , size …

语句标号：name , addr , 定义否 …

过程、函数：name , addr , 形参 …

反映了标识符的语义特征属性，是翻译的依据。

记录于符号表中（namelist）

整个编译过程中动态地采集、记录、变更、引用

符号表

存放源程序中有关标识符的属性信息的数据结构。

符号表结构

名字域	属性信息域

符号表作用

- 收集标识符属性信息；
- 语义检查依据；
- 代码生成时地址分配依据。

例如，设有C程序片段

```
        :
  int  i , a[4] ;
{  …
   i=a[2] ;
   …
}
```

词法语法语义分析

i, a ⇒ 符号表

i.type ⇒符号表

a.type ⇒符号表

a.维数 ⇒ 符号表

a.每维大小 ⇒ 符号表

……

类型检查;

数组越界检查;

回填addr; …

```
          <prog> ::= #createGlobalTable# <classDecl>*<progBody>
     <classDecl> ::= class id #createClassEntryAndTable# {<varDecl>*<funcDef>*};
      <progBody> ::= program #createProgramTable# <funcBody>;<funcDef>*
      <funcHead> ::= <type>id(<fParams>) #createFunctionEntryAndTable#
       <funcDef> ::= <funcHead> <funcBody>;
      <funcBody> ::= {<varDecl>*<statement>*}
       <varDecl> ::= <type>id<arraySize>*; #createVariableEntry#
     <statement> ::= <assignStat>;
                   | if(<expr>)then<statBlock>else<statBlock>;
                   | for(<type>id<assignOp><expr>;<relExpr>;<assignStat>)<statBlock>;
                   | get(<variable>);
                   | put(<expr>);
                   | return(<expr>);
    <assignStat> ::= <variable><assignOp><expr>
     <statBlock> ::= {<statement>*} | <statement> | ε
          <expr> ::= <arithExpr> | <relExpr>
       <relExpr> ::= <arithExpr><relOp><arithExpr>
     <arithExpr> ::= <arithExpr><addOp><term> | <term>
          <sign> ::= + | -
          <term> ::= <term><multOp><factor> | <factor>
        <factor> ::= <variable>
                   | <idnest>*id(<aParams>)
                   | num
                   | (<arithExpr>)
                   | not<factor>
                   | <sign><factor>
      <variable> ::= <idnest>*id<indice>*
        <idnest> ::= id<indice>*.
        <indice> ::= [<arithExpr>]
     <arraySize> ::= [ int ]
          <type> ::= int | float | id
       <fParams> ::= <type>id<arraySize>* #createParameterEntry# <fParamsTail>* | ε
       <aParams> ::= <expr><aParamsTail>* | ε
   <fParamsTail> ::= ,<type>id<arraySize>* #createParameterEntry#
   <aParamsTail> ::= ,<expr>
```

Symbol table: Global

name	kind	type	link
f1	function	float : int[2][2], float	•
f2	function	int : nil	•
MyClass1	class		•
MyClass2	class		•
program	function		•

Symbol table: f1

name	kind	type	link
fp1	parameter	int[2][2]	
fp2	parameter	float	
fv1	variable	MyClass1[3]	
fv2	variable	int	

Symbol table: f2

name	kind	type	link

Symbol table: MyClass1

name	kind	type	link
mc1v1	variable	int[2][4]	
mc1v2	variable	float	
mc1v3	variable	MyClass2[3]	
mc1f1	function	int : int, MyClass2[3]	•
f2	function	int : MyClass1[3]	•

Symbol table: MyClass1:mc1f1

name	kind	type	link
p1	parameter	int	
p2	parameter	MyClass2[3]	
fv1	variable	MyClass2[3]	

Symbol table: MyClass1:f2

name	kind	type	link
f2p1	parameter	MyClass1[3]	
mc1v1	variable	int	

Symbol table: MyClass2

name	kind	type	link
mc1v1	variable	int[2][4]	
fp1	variable	float	
m2	variable	MyClass2[3]	

Symbol table: program

name	kind	type	link
m1	variable	int	
m2	variable	float[3][2]	
m3	variable	MyClass2[2]	

```
class MyClass1 {
  int mc1v1[2][4];
  float mc1v2;
  MyClass2 mc1v3[3];
  int mc1f1(int p1, MyClass2 p2[3]) {
    MyClass2 fv1[3];
    …
  }
  int f2(MyClass1 f2p1[3]) {
    int mc1v1;
    …
  }
}

class MyClass2 {
  int mc1v1[2][4];
  float fp1;
  MyClass2 m2[3];
  …
}

program {
  int m1;
  float[3][2] m2;
  MyClass2[2] m3;
  ...
}
float f1(int fp1[2][2], float fp2) {
  MyClass1[3] fv1;
  int fv2;
  …
}

int f2() {
...
}
```

根据作用域分别建立符号表

常见标识符类的主要属性与作用

1. 标识符名。作用：查重(考虑作用域和可视性前提)
2. 类型。作用：存储空间分配；可施加运算的检查等
3. 存储类别。作用：提供语义处理、检查、存储分配的依据。
4. 作用域。作用：可视性。

第 6 章　语义分析与中间代码生成

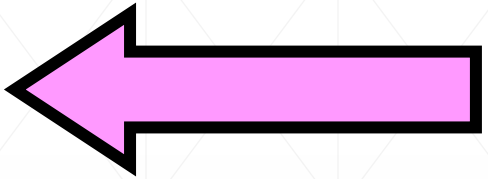
6.1　语法制导翻译

6.2　符号表

6.3　类型检查

6.4　中间语言

6.5　中间代码生成



类型的作用：

关联运行时刻的行为

确定占用的存储空间

类型检查要素

类型描述(结构)

—类型表达式

类型处理

—类型说明语句中的类型处理

—表达式中的类型处理

类型表达式

1.基本类型，int，char，float，…

2. 构造类型

数组

记录

函数

指针

类型处理

1.类型说明语句中的类型处理

将类型表达式信息填入符号表

2.表达式或语句中的类型处理

表达式：

基本类型：获取类型

构造类型：检查，没错误获取类型

语句：

检查是否匹配

不匹配是否可以类型转换

可以转换的实施转换

后序说明语句的处理中讲

表达式中的类型处理

1.简单表达式

E→字符常数 E.type=char

E→整型常数 E.type=int

E→i E.type=lookup(i.entry)

E→E[E] if(E_2.type==int& E_1.type==array(s,t)
then E.type=t,
else type_error

获取类型

检查，没有错误获取类型

语句中的类型处理

2.一般语句

产生式	语义动作
$S \rightarrow i=E$	if(*lookup*(i.entry)==E.type) then S.type=void, else type_error
$S \rightarrow$ *if E then S*	if(*E*.type==boolean) then S.type=void, else type_error

第 6 章　语义分析与中间代码生成

6.1　语法制导翻译

6.2　符号表

6.3　类型检查

6.4　中间语言

6.5　中间代码生成

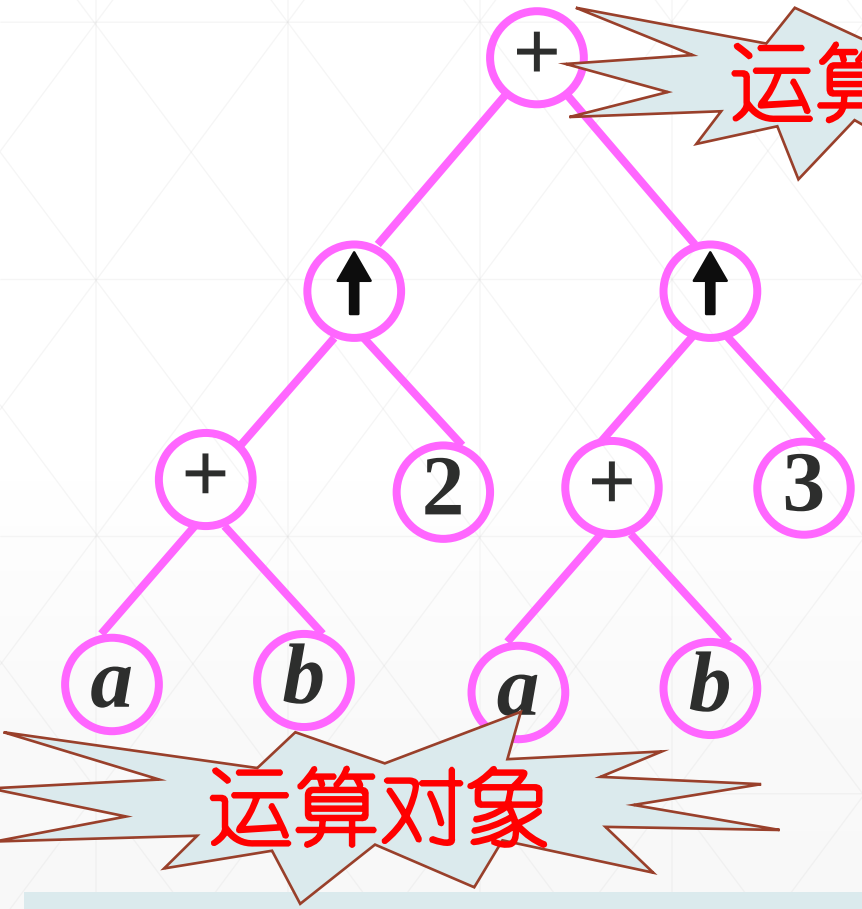
中间语言(中间代码)是语义程序的输出。

中间语言的设计与应用考虑的要素：

- 从源语言到目标语言的翻译跨度
- 目标机的指令集特点。

中间语言
- 树(AST)
- 逆波兰式
- N元式(三元式、间接三元式四元式)

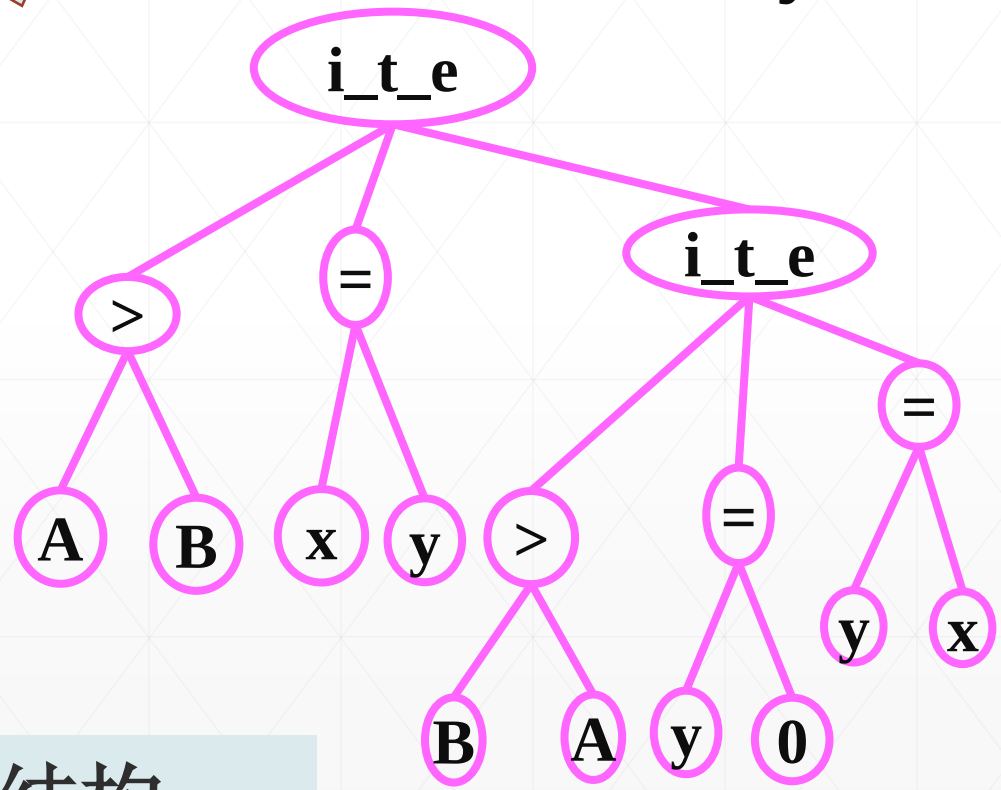
$(a+b)^2 + (a+b)^3$



if A>B then x=y

else if B>A then y=0

else y:=x



所有的结构都看成运算结构，
由运算符和运算对象构成

二. 逆波兰式(后缀式)

波兰逻辑学家卢卡西维奇发明

形式：

$$e_1 e_2 \dots e_n \theta \quad (n \geq 1)$$

运算对象 运算符

为了使中间代码更接近机器语言，在逆波兰式和多元式中，我们把结构化的程序块都转换为跳转指令goto。

例：

$a \times b \Rightarrow ab\times$

$-a \Rightarrow a@$ (@表示单目－)

$a+b \times(c+d) \times(e+f) \Rightarrow a\ b\ c\ d\ +\times e\ f\ +\times+$

GOTO label $\Rightarrow$ *label J*

J：单目操作符，产生一个到运算对象的转移。

IF e GOTO label $\Rightarrow$ *e label* J_T

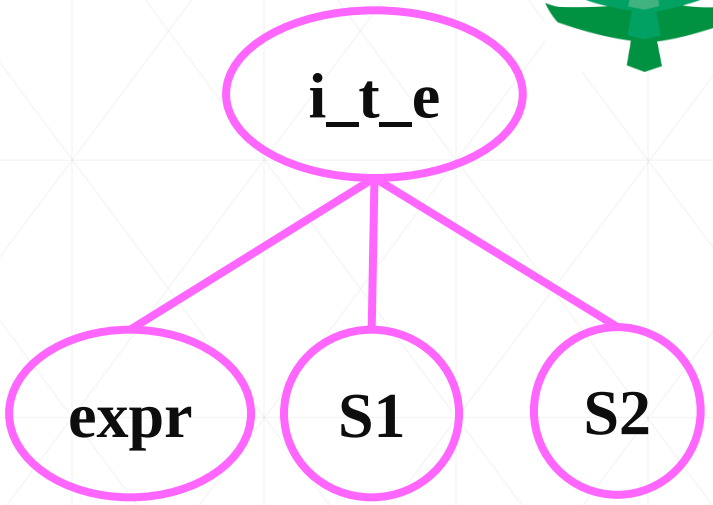
J_T：二目操作符，若第一运算对象(布尔式)的值为真，则产生一个到第二运算对象(标号)的转移。

J_F：二目操作符，若第一运算对象(布尔式)的值为假，则产生一个到第二运算对象(标号)的转移。

Label1

IF<expr>then<s1>else<s2>

Label2



编号

(1)<expr> (4) J_F

(2)<s1>

(3)(5) J

(4)<s2>

(5)

例：设有如下C程序片段

```
{
      int  k;
      i=j=0;
h:    k=100;
      if (k>i+j)
          { k--;  i++; }
      else
       ▼k=i*2-j*2;
    ▼goto h;
 }
```

标号处理

逆波兰式代码：

(1) i j 0 = =

(2) k 100 =

(3) k i j + > (7) J_F

(4) k k 1 – =

(5) i i 1 + =

(6) (8) J

(7) k i 2*j2*– =

(8) (2) J

编号

三. N元式 N个域的记录结构

（D_1，D_2，D_3，…，D_n）

OP域 操作对象域

- 常见N元式

三元式、间接三元式 ⟺ 双地址机指令形式

四元式 ⟺ 三地址机指令形式

三元式形式：

$$(OP , ARG_1 , ARG_2)$$

- 操作符
- 第一操作数
- 第二操作数

操作数可以是三元式的编号， 代表三元式的计算结果

间接三元式形式：

间接码表 + 三元式表

例如，语句 X=A+B*C 的三元式表示为

NO.	OP	ARG1	ARG2
(1)	*	B	C
(2)	+	A	(1)
(3)	=	X	(2)

if X>Y then Z = X else Z = Y+1

的三元式可以表示为

NO.	OP	ARG1	ARG2
(1)	>	X	Y
(2)	J_F	(1)	(5)
(3)	=	Z	X
(4)	J		(7)
(5)	+	Y	1
(6)	=	Z	(5)
(7)			

$(a+b)^2 + (a+b)^3$

NO	OP	ARG1	ARG2
①	+	a	b
②	↑	①	2
③	+	a	b
④	↑	③	3
⑤	+	②	④

其中：“↑” 表示幂运算。

$(a+b)^2 + (a+b)^3$ 的间接三元式

间接码表

控制三元式代码执行顺序

①
②
①
③
④

三元式表

①	+， a， b
②	↑， ①， 2
③	↑， ①， 3
④	+， ②， ③

四元式形式定义：

(OP , ARG_1 , ARG_2 , Result)

$(a+b)^2 + (a+b)^3$的四元式：

(0)	+ ,	a,	b,	T_1
(1)	↑ ,	T_1,	2,	T_2
(2)	+ ,	a,	b,	T_3
(3)	↑ ,	T_3,	3,	T_4
(4)	+,	T_2,	T_4 ,	T_5

** 注：T_i是临时变量。

if X>Y then Z = X else Z = Y+1

的四元式可以表示为

NO.	OP	ARG1	ARG2	Result
(1)	>	X	Y	t1
(2)	J_F	t1		(5)
(3)	=	X		Z
(4)	J			(7)
(5)	+	Y	1	t2
(6)	=	t2		Z
(7)				

跟语法分析的结果最相似的表示形式为树；

逆波兰式适于栈式存储的计算机（堆栈机）；

四元式便于优化；

间接三元式优化时的时空效率类似于四元式。

语法分析树 → AST

去除非终结符号及无语义的终结符号

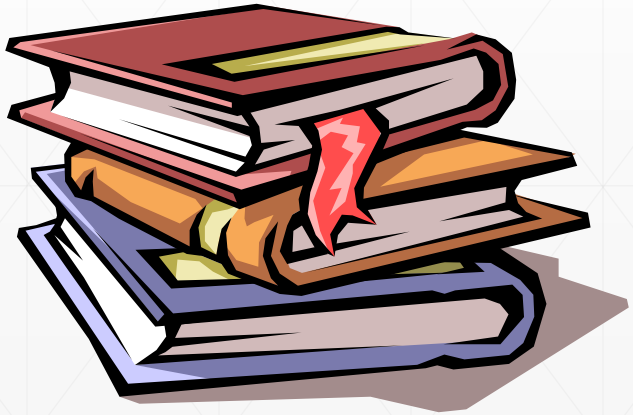
线性化：后序遍历

逆波兰式

“单步”表示

三元式

增加结果量 → 四元式



第 6 章　语义分析与中间代码生成

6.1　语法制导翻译

6.2　符号表

6.3　类型检查

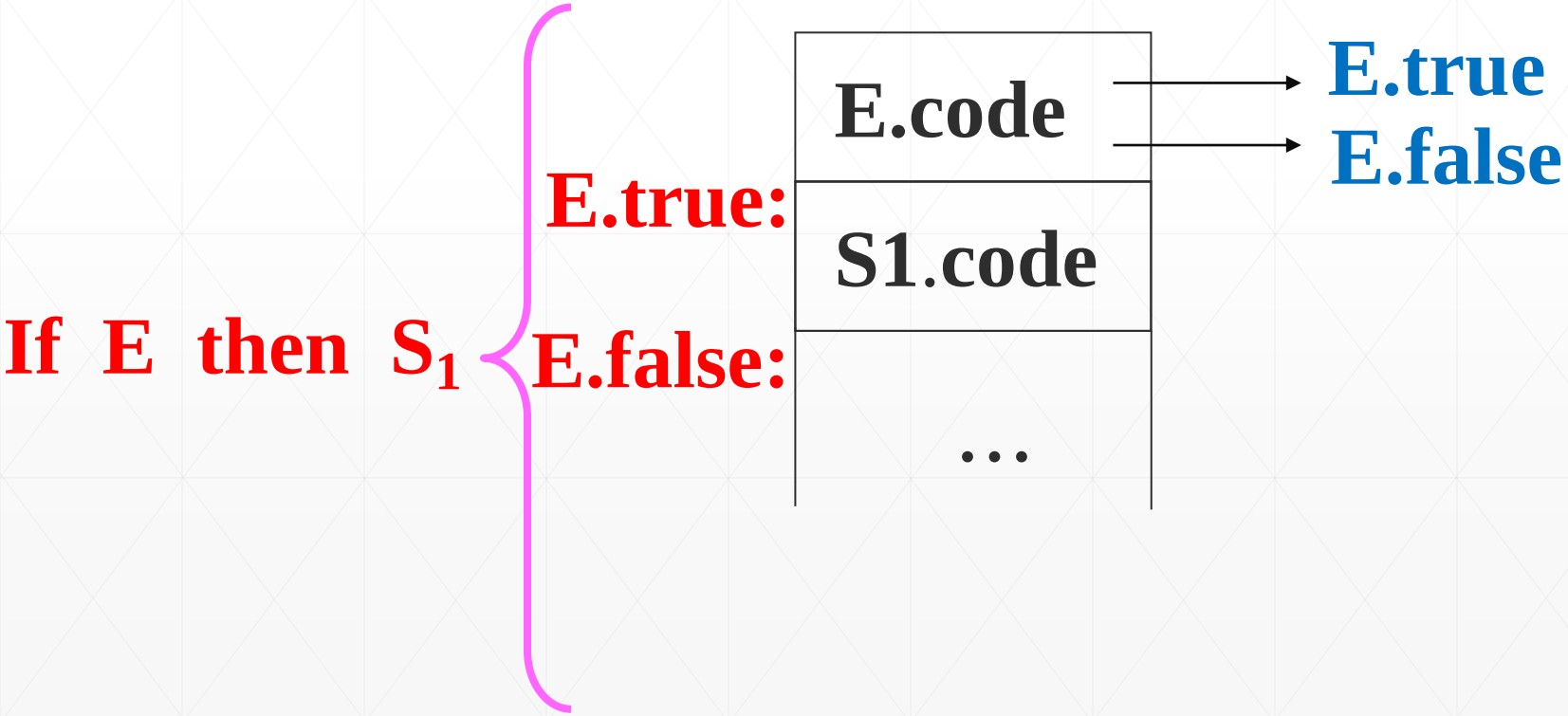
6.4　中间语言

6.5　中间代码生成

语句翻译设计要点

1. 根据语义确定语句的目标结构；

源语句 ⇒ 中间及目标代码的布局



语句翻译设计要点

1. 根据语义确定语句的目标结构；

2. 确定中间代码；

3. 根据目标结构和语义规则，构造合适的SDT或属性翻译文法；

4. 涉及的实现技术

6.6 语句翻译与中间代码生成

6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译

6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译

- 说明类语句

语言中定义性信息，一般不产生目标代码，其作用是辅助完成编译。

例如， 常量说明，变量说明，类型说明，
对象说明，标号说明 ……

- 说明类语句的处理

相关说明的属性信息填入符号表，提供语义检查和存储分配的依据。

6.6.1 说明类语句的翻译

一. 常量定义语句的翻译

二. 简单说明类语句

三. 复合类型说明语句

一. 常量定义语句的翻译

常量定义语句的文法

<C_D> → CONST <C_L>;

<C_L> → <C_L>;<CD>|<CD>

<CD> → id=num

常量说明语句的语义处理：

1.等号右边的常量是第一次出现，则将其填入常量表且回送常量表序号，

2.将等号左边的标识符在符号表中登记新记录，记录信息包括：常量标志，类型，常量表序号

一. 常量定义语句的翻译

常量说明语句的SDT

L属性

左递归

<C_D> → CONST {<C_L>.att=cons}
<C_L>;

<C_L> → {$<C\_L>_1$.att=<C_L>.att}
<C_L>; {<CD>.att=<C_L>.att;}
<CD>

<C_L> → {<CD>.att=<C_L>.att;}
<CD>

<CD> → id=num
{look_con_table(num.lexval);
enter(id, <CD>.att,num.type,num.addr); }

一. 常量定义语句的翻译

常量说明语句的SDT

S属性

<C_D> → CONST id=num;

{look_con_table(num.lexval);

enter(id, cons,num.type,num.addr);

<C_D>.att=constant； }

<C_D> →<C_D> id=num;

{look_con_table(num.lexval);

enter(id, $<C\_D_1>$.att,num.type,num.addr);

<C_D>.att= $<C\_D_1>$.att}

举例说明：

形式：CONST 标识符＝常量；…

CONST pi=3.1416; true=1;

CONST pi=3.1415926;

namelist

name	kind	type	addr	
pi	CONS	R		
true	CONS	B		
…				

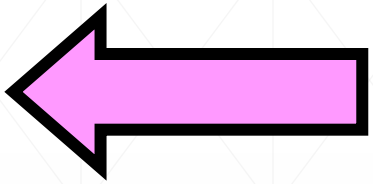
conslist

value
3.1416
1
3.1415926

6.6.1 说明类语句的翻译

一. 常量定义语句的翻译

二. 简单说明类语句



三. 复合类型说明语句

二. 简单说明类语句

- 简单说明语句的SDT

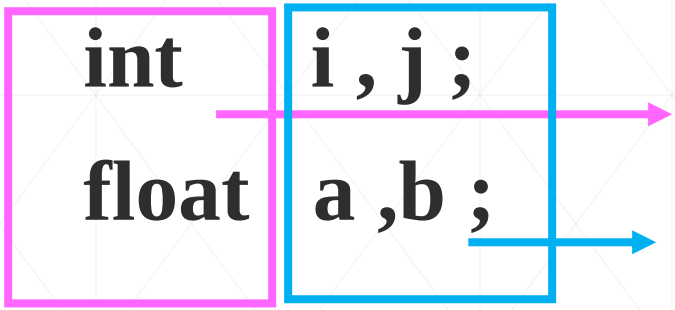
D→int id {enter(id,v,int,offset);D.att=int;D.width=4;
offset=offset+D.width;}

D→float id {enter(id,v,float,offset);D.att=float;
D.width=8;offset=offset+D.width; }

D→D，id； {enter(id,v,D_1.att,offset);
offset=offset+D_1.width;D.att=D_1.att;D.width=D_1.width;}

处理变量说明前设置offset变量为0

举例说明：



namelist

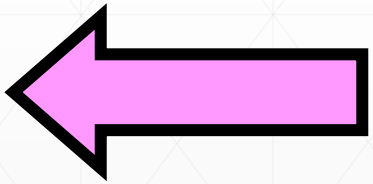
name	kind	type	offset	
i	V	I	0	
j	V	I	4	…
a	V	R	8	
b	V	R	16	
…				

6.6.1 说明类语句的翻译

一. 常量定义语句的翻译

二. 简单说明类语句

三. 复合类型说明语句



三. 复合类型说明语句

$$T \rightarrow \text{struct } L\{D\}\ V$$

$$L \rightarrow id \mid \varepsilon$$

$$D \rightarrow D;F \mid F$$

$$F \rightarrow \text{type } V;$$

$$V \rightarrow V, id \mid \varepsilon$$

其中：

L： 结构类型名； D： 结构成员； V： 变量表；

F： 结构成员项； id: 标识符；

例如，

```
struct  date
    { int year, month, day;}  today, yesterday;
```

- 语义处理涉及

(1) 结构成员与该结构相关；

(2) 结构的存储：一个结构的所有成员项连续存放(简单方式)；

(3) 结构的引用是结构成员的引用，不能整体引用；（成员信息须单独记录）

namelist

name	kind	…	addr
k1	struct		
…			

例如，

```
struct k1{
        int  a;
        float  b;
        int c[10];
        char  d;
      }
```

结构成员分表

name	kind	type	OFFSET
a	V	I	0
b	V	R	4
c	array	I	12
d	V	C	52
…			

6.6 语句翻译与中间代码生成

6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译

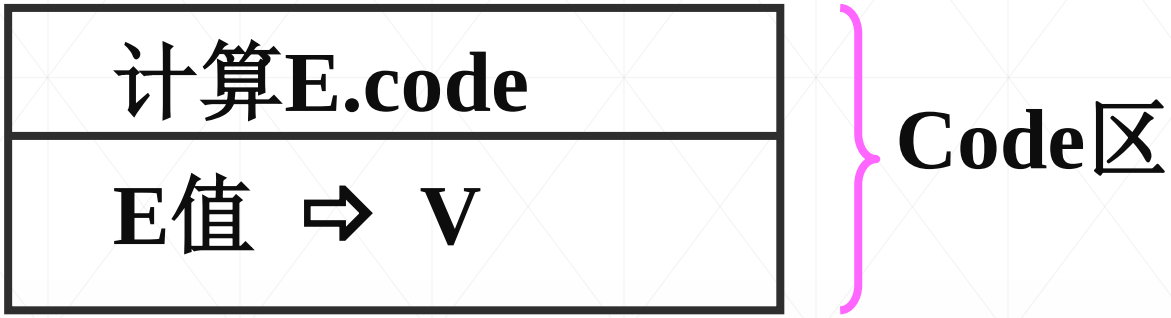
6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译

- 赋值语句形式定义

$$A \rightarrow V = E$$

- 赋值语句目标结构



- 赋值语句语义处理

(1) 表达式处理(产生表达式的中间代码);

(2) “=”的处理: “=”左右部类型相容性检查和转换;

赋值语句的四元式翻译SDT

A →i=E

{ GEN(=，E.PLACE， _，i}

其中:

GEN(OP，ARG1，ARG2，RESULT): 函数。

把四元式(OP，ARG1，ARG2，RESULT)

填入四元式表。

E. PLACE：表示存放E值的变量。

- 算术表达式形式定义

$$E \rightarrow E\ op\ E \mid op\ E \mid id$$

其中：

OP：为运算符；

E, id：运算对象。

算术表达式语句的四元式翻译SDT

(1) $E \rightarrow E\ OP\ E$

{E. PLACE=NEWTEMP;

GEN(OP, E_1. PLACE, E_2. PLACE, E.PLACE)}

(2) $E \rightarrow OP\ E$

{E. PLACE=NEWTEMP;

GEN(OP, E_1. PLACE，_，E.PLACE)}

(3) $E \rightarrow id$

{E. PLACE=id}

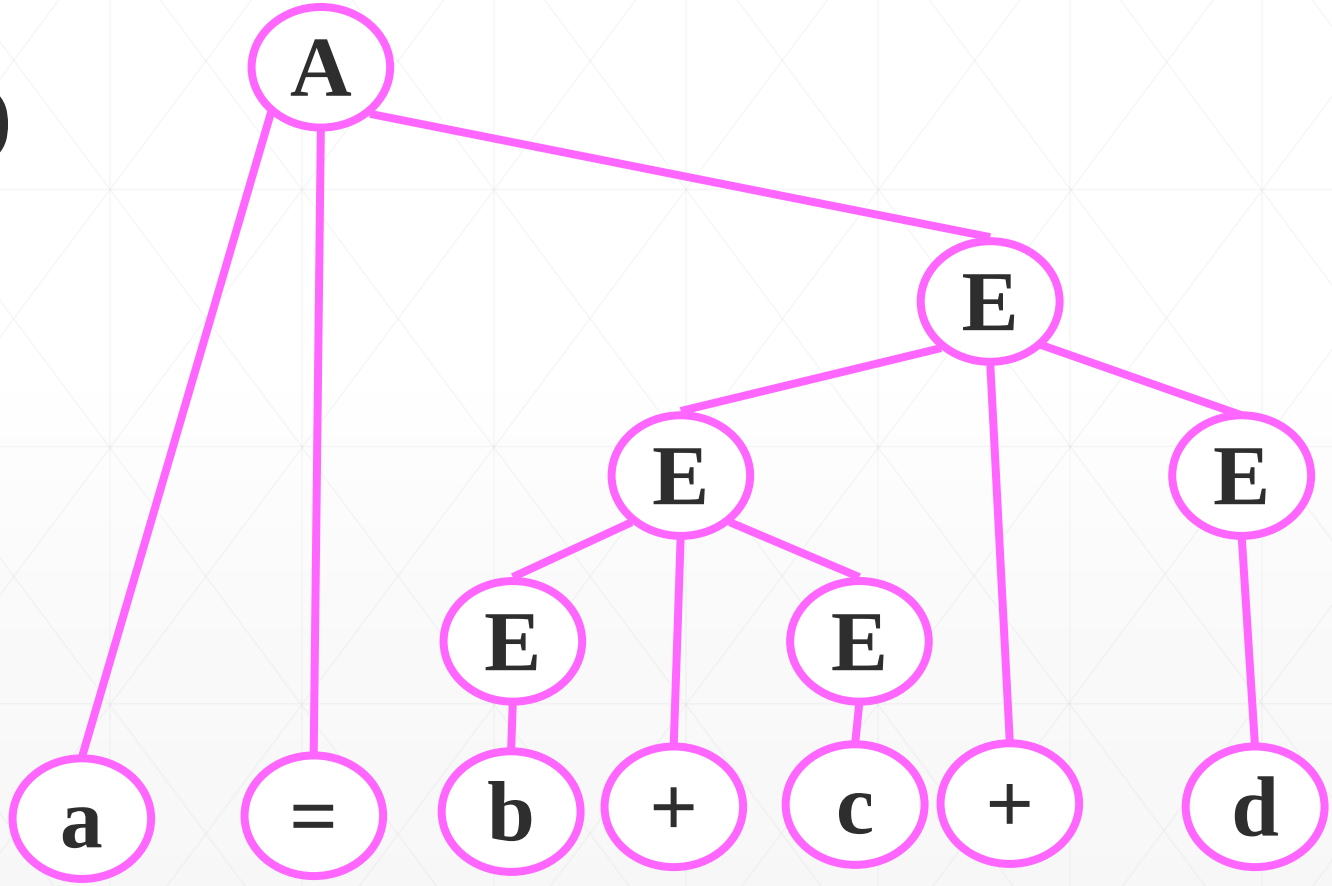
赋值 语句与表达式翻译举例：

写出C语言语句 a=b+c+d翻译后的四元式代码：

(+ b c t1)

(+ t1 d t2)

(= t2 -- a)



- 逻辑表达式形式定义

B→B||B|B&&B|！B| E rel E |true|false

其中：

||：代表或运算

&&：代表并且运算

！：代表非运算

rel: 代表关系运算比如<=、<、==、>=、>、≠等

条件语句中逻辑表达式语句的四元式翻译描述

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

- 避免生成冗余的goto指令的处理

$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code \ \|\| \ gen(\text{'if'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \text{'goto'} \ B.true) \ \|\| \ gen(\text{'goto'} \ B.false)$

语义修改为

引入特殊标号值fall

$$test = E_1.addr \ \mathbf{rel}.op \ E_2.addr$$

$$\begin{aligned} s = \ & \mathbf{if} \ B.true \neq fall \ \mathbf{and} \ B.false \neq fall \ \mathbf{then} \\ & \quad gen(\text{'if'} \ test \ \text{'goto'} \ B.true) \ \|\| \ gen(\text{'goto'} \ B.false) \\ & \mathbf{else \ if} \ B.true \neq fall \ \mathbf{then} \ gen(\text{'if'} \ test \ \text{'goto'} \ B.true) \\ & \mathbf{else \ if} \ B.false \neq fall \ \mathbf{then} \ gen(\text{'ifFalse'} \ test \ \text{'goto'} \ B.false) \\ & \mathbf{else} \ '' \end{aligned}$$

$$B.code = E_1.code \ \|\| \ E_2.code \ \|\| \ s$$

避免生成冗余的goto指令的处理

$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

语义修改为

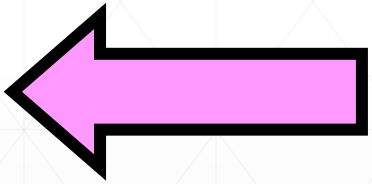
$$
\begin{aligned}
B_1.true &= \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \textbf{if } B.true \neq fall \textbf{ then } B_1.code \mid\mid B_2.code \\
&\quad \textbf{else } B_1.code \mid\mid B_2.code \mid\mid label(B_1.true)
\end{aligned}
$$

6.6 语句翻译与中间代码生成

6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译



6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译

- **控制流语句:** 改变程序执行顺序，引起程序执行发生跳转（向前或向后）。

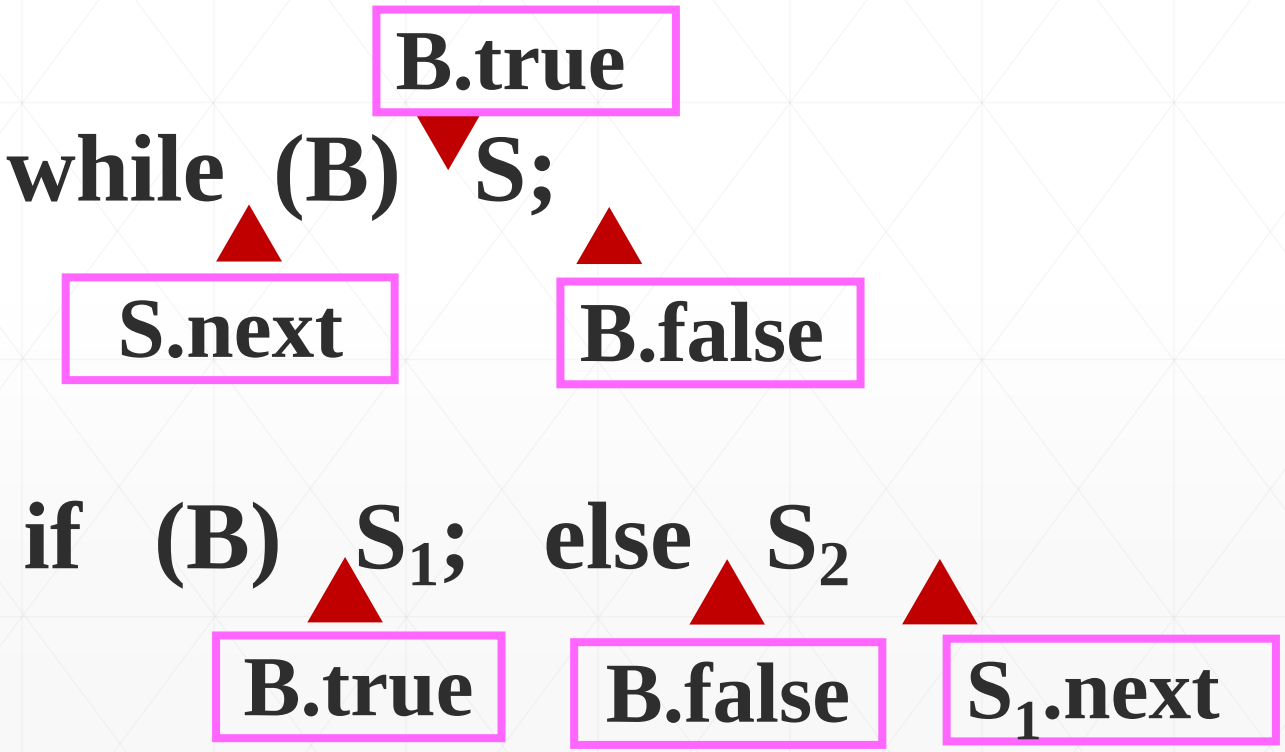
➢ 程序设计语言中出现频繁的语句；

➢ 为可执行语句，要产生相应的目标代码；

➢ 控制流程的变换，依靠代码中的**跳转指令**与对应跳转的**语句标号**。

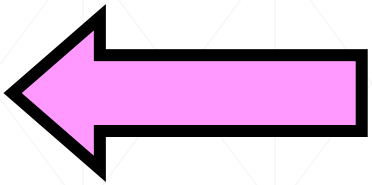
跳转目标标记

语句标号
- 显式：位于源语句之前；（如，L1： S;）
- 隐式：含于源语句之中但未标识



6.6.3 控制流类语句翻译

一.语句标号与拉链返填技术



二.布尔表达式的回填

三.条件语句的翻译

语句标号处理

控制流类语句处理面对的公共问题和实现技术

语句标号处理

- 引用时知道标号位置
  - 控制流跳转代码一次处理完成
- 引用时不知道标号位置(隐含标号)
  - 控制流跳转代码一次处理不行

语句标号

定义性出现 （如，L1: S ;）

使用性出现 （如，goto L1 ;）

语句标号表（LT）

标号名	定义否(flag)	addr

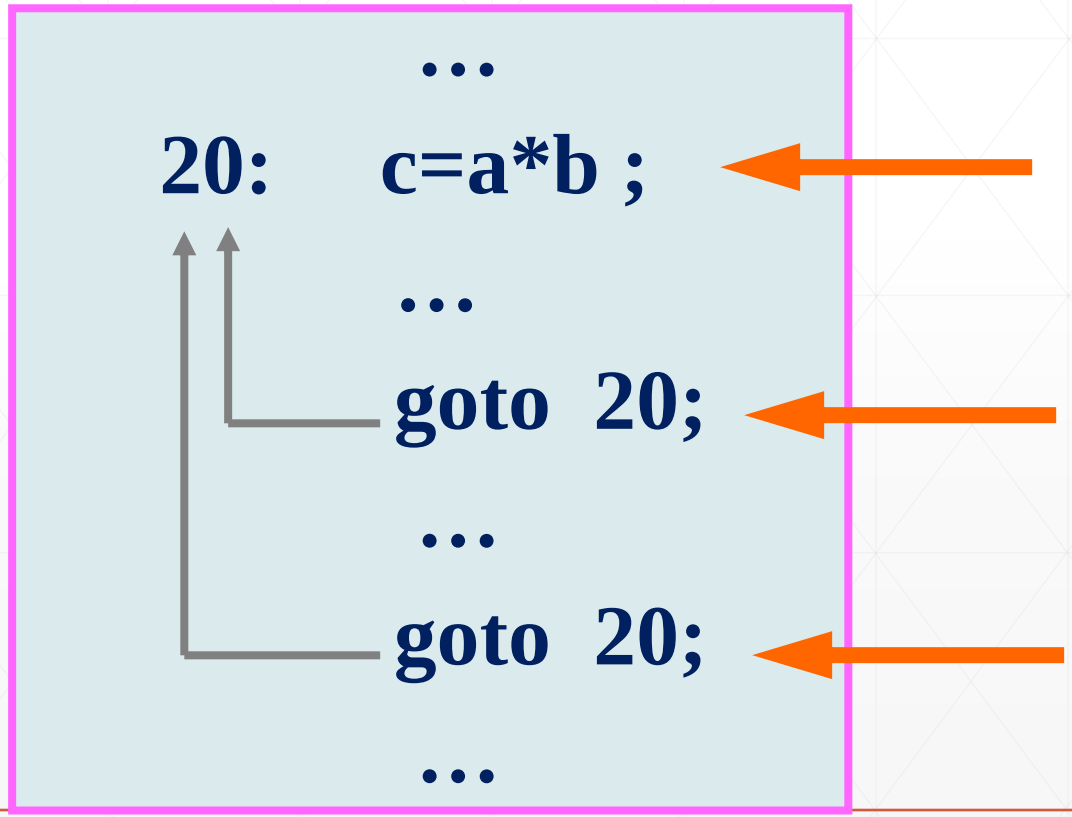
1表示定义性出现
0表示使用性出现

标号所标识的源语句的中间代码序列的第一条代码的存放地址或编号

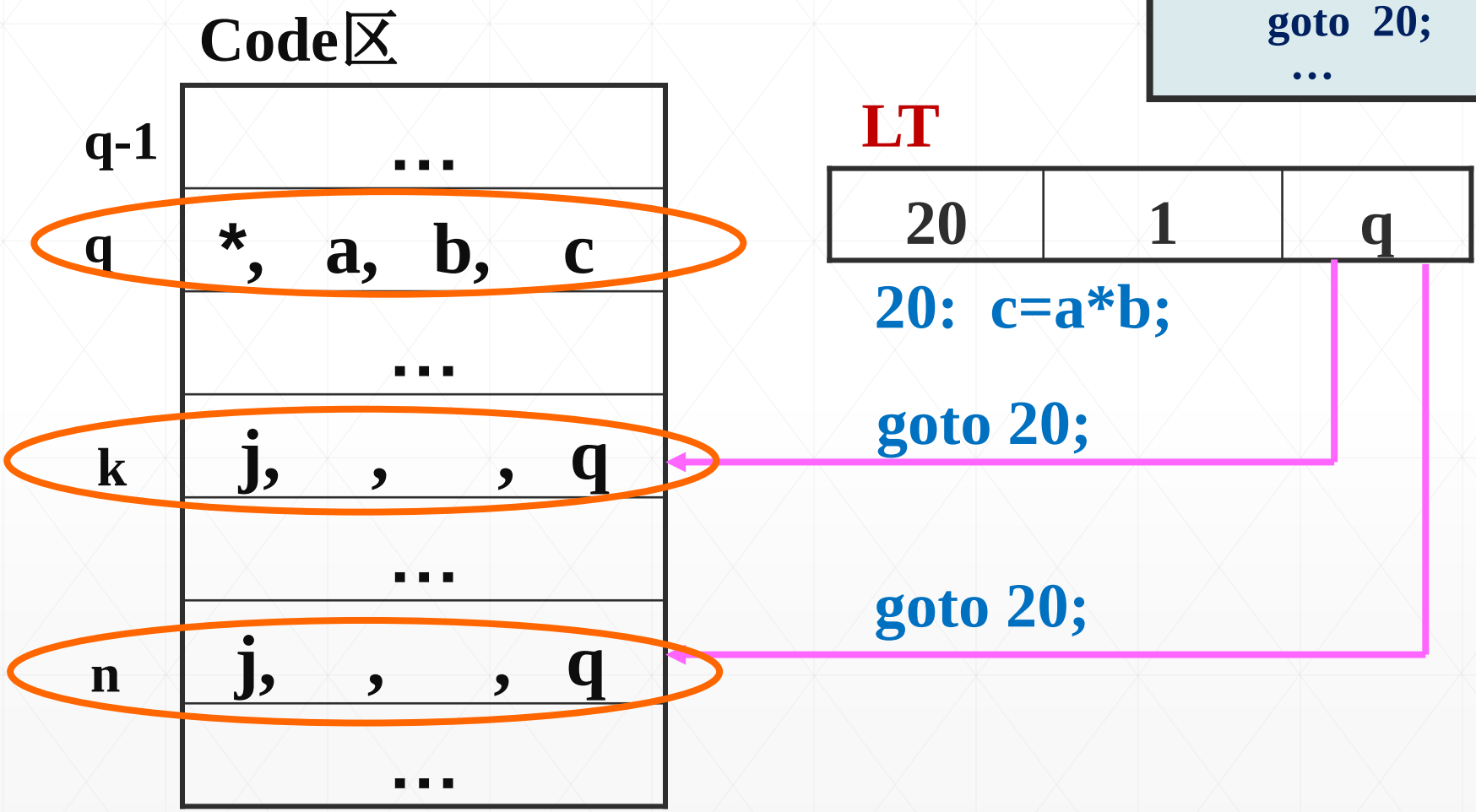
- 对多遍扫描的编译器

视为一种情况(引用时知道标号的位置) 处理。

例如，



```
20:  c=a*b ;
     …
     goto  20;
     …
     goto  20;
     …
```

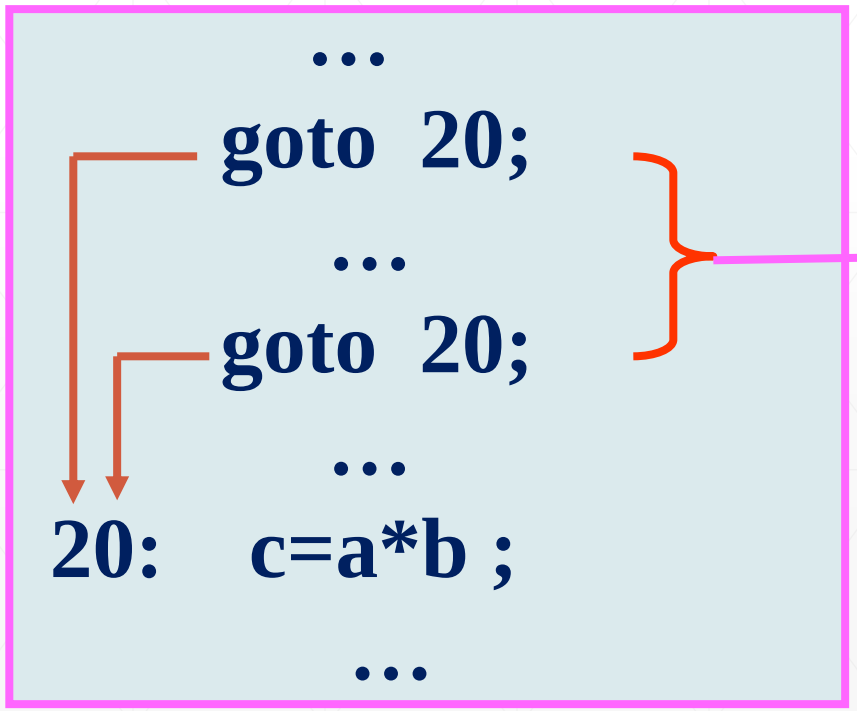


拉链-返填技术

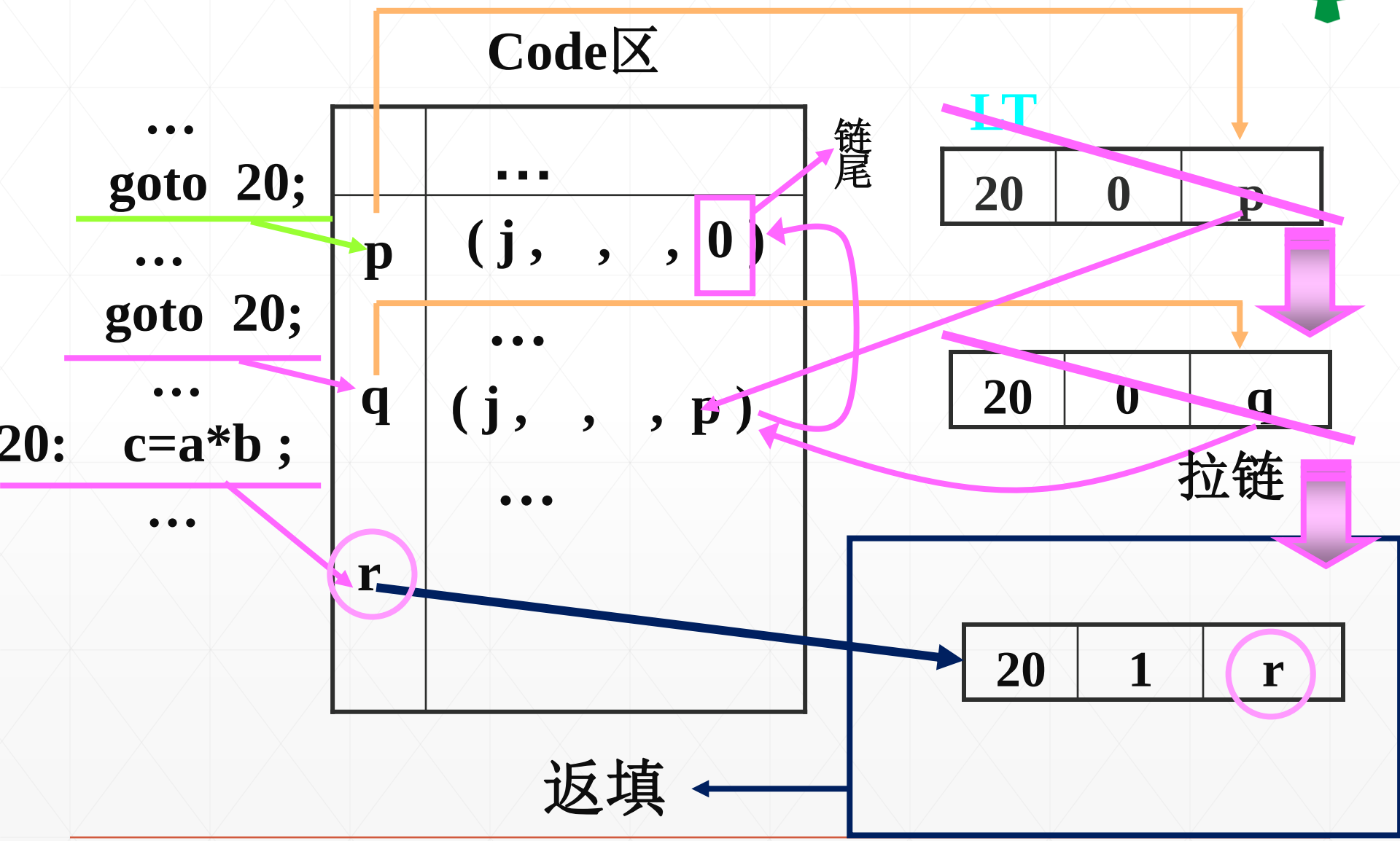
处理标号引用时不知道标号的位置的情况

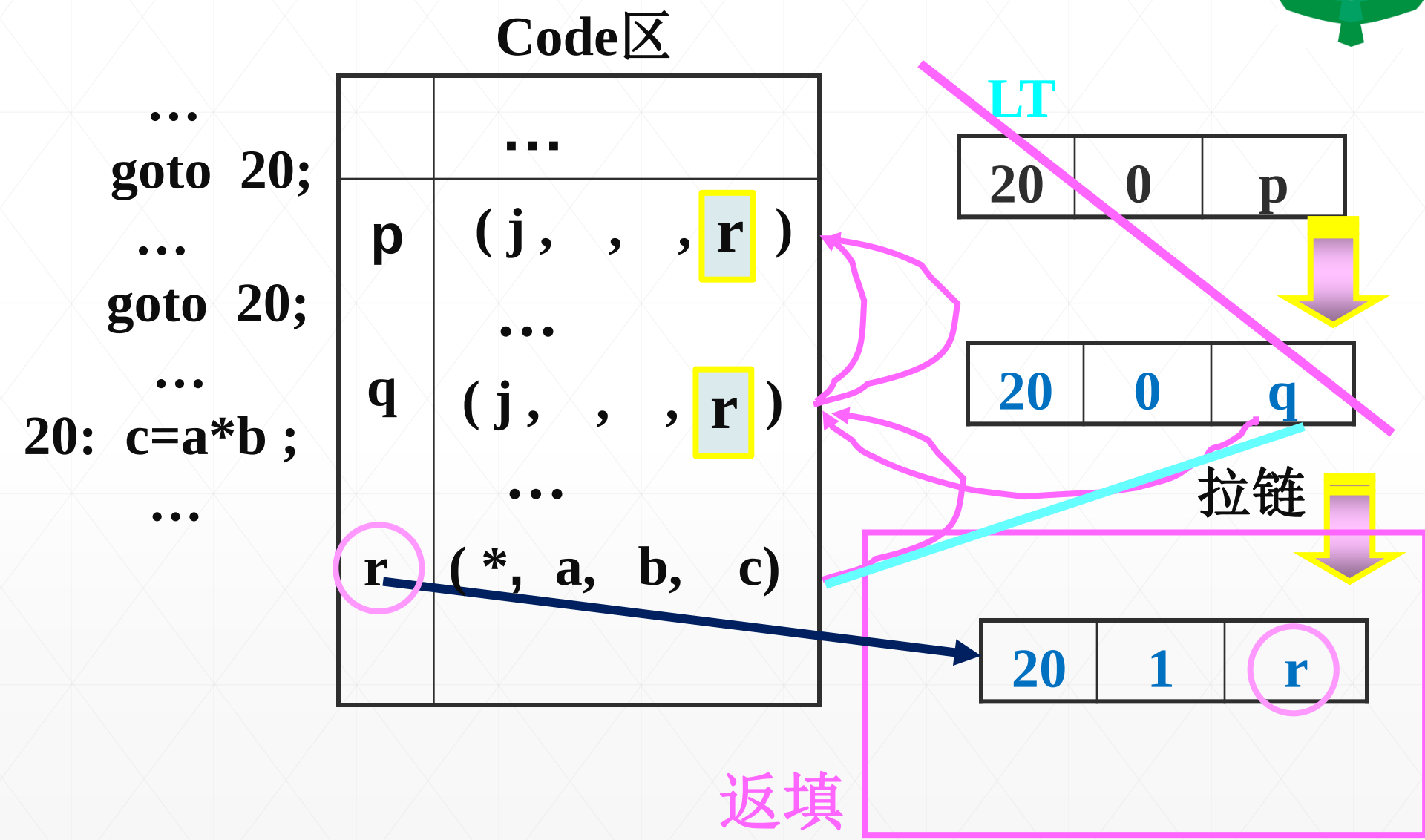
适用于一遍扫描的编译器

例如，



产生这2条语句的代码时，还未扫描到20号语句，即没有生成该语句的代码，因此转向的目标地址未定。


Code区
…
goto 20;
…
goto 20;
…
20: c=a*b ;
…
p
q
r
…
(j, , , 0)
…
(j, , , p)
…
链尾
LT
20 0 p
20 0 q
拉链
20 1 r
返填


Code区
…
goto 20;
…
goto 20;
…
20: c=a*b ;
…
…
p (j, , , r)
…
q (j, , , r)
…
r (*, a, b, c)
LT
20 0 p
20 0 q
拉链
20 1 r
返填

注意：

1. 链尾标志在代码区的四元式中，链头在标号表中，链在与同一语句标号相关的跳转代码中；

2. 拉链次序：

尾 头

p → q

3. 返填次序：

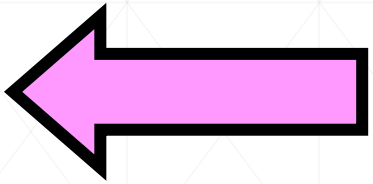
头 尾

q → p（在代码区跳转指令中(addr=0)）

6.6.3 控制流类语句翻译

一.语句标号与拉链返填技术

二.布尔表达式的回填



三.条件语句的翻译

- 条件语句中逻辑表达式语句的拉链返填

B→E

{B.true=makelist(nextinstr);
B.false=makelist(nextinstr+1);
GEN(JT E.PLACE - 0);
GEN(JF E.PLACE - 0);}

建链

B→E rel E

{B.PLACE=NEWTEMP;
GEN(rel E_1.PLACE E_2.PLACE B.PLACE);
B.true=makelist(nextinstr);
B.false=makelist(nextinstr+1);
GEN(JT B.PLACE - 0);
GEN(JF B.PLACE - 0);}

- 条件语句中逻辑表达式语句的拉链返填

$B \rightarrow !B$ {B.true=B_1.false;
B.false=B_1.true;}

$B \rightarrow B||MB$
{backpatch(B_1.false,M.instr);
B.true=merge(B_1.true,B_2.true);
B.false=B_2.false;}

返填

拉链

$M \rightarrow \varepsilon$
{M.instr=nextinstr;}

- 条件语句中逻辑表达式语句的拉链返填

$B \rightarrow B \&\& MB$

{backpatch(B_1.true,M.instr);
B.true=B_2.true;
B.false=merge(B_1.false,B_2.false);}

返填

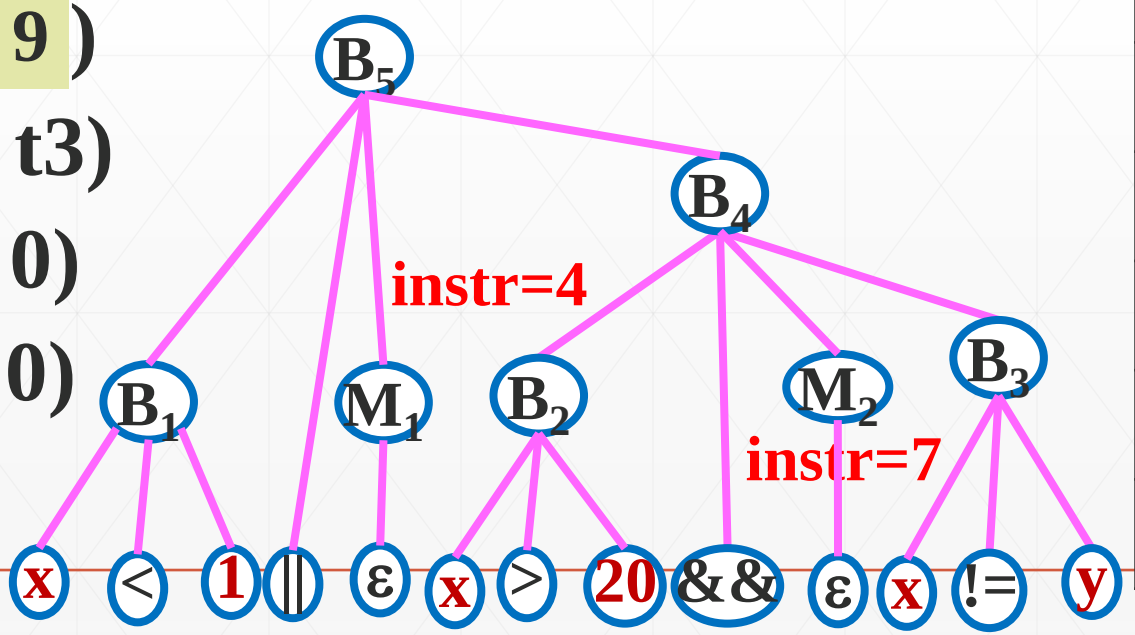
拉链

例：布尔表达 …译

1.(< x 1 t1)
2.(J_T t1 - 8)
3.(J_F t1 - 4)
4.(> x 20 t2)
5.(J_T t2 - 7)
6.(J_F t2 - 9)
7.(!= x y t3)
8.(J_T t3 - 0)
9.(J_F t3 - 0)

$B \rightarrow B\&\&MB$
{backpatch(B_1.true,M.instr);
B.true=B_2.true;
B.false=merge(B_1.false,B_2.false);}

$B \rightarrow B||MB$
{backpatch(B_1.false,M.instr);
B.true=merge(B_1.true,B_2.true);
B.false=B_2.false;}



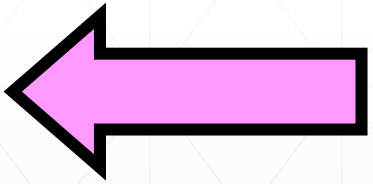
N	D	A
~~B_1T~~	~~0~~	~~2~~
B_1F	1	4
B_2T	1	7
~~B_2F~~	~~0~~	~~6~~
~~B_3T~~	~~0~~	~~8~~
~~B_3F~~	~~0~~	~~9~~
~~B_4T~~	~~0~~	~~8~~
~~B_4F~~	~~0~~	~~6~~
B_5T	0	2
B_5F	0	6

6.6.3 控制流类语句翻译

一.语句标号与拉链返填技术

二.布尔表达式的回填

三.条件语句的翻译



三. 条件语句的翻译

条件语句文法:

S → if (B) then S ① |
if (B) S else S ② |
while (B) S ③

其中:

B: 布尔表达式;

S: 语句;

S→if(B)MS
{backpatch(B.true, M.instr);
S.next=merge(B.false,S_1.next);}

S→{L}
{S.next=L.next;}

S→A;
{S.next=null}

L→LMS
{backpatch(L_1.next, M.instr);
L.next=S.next;}

L→S
{L.next=S.next;}

M→ε
{M.instr=nextinstr;}

S→if(B)MSNelseMS

{backpatch(B.true,M_1.instr);
backpatch(B.false,M_2.instr);
S.next=merge(S_1.next,N.next,S_2.next);}

N→ε {N.next=makelist(nextinstr);
GEN(J - -0)}

S→whileM (B) MS

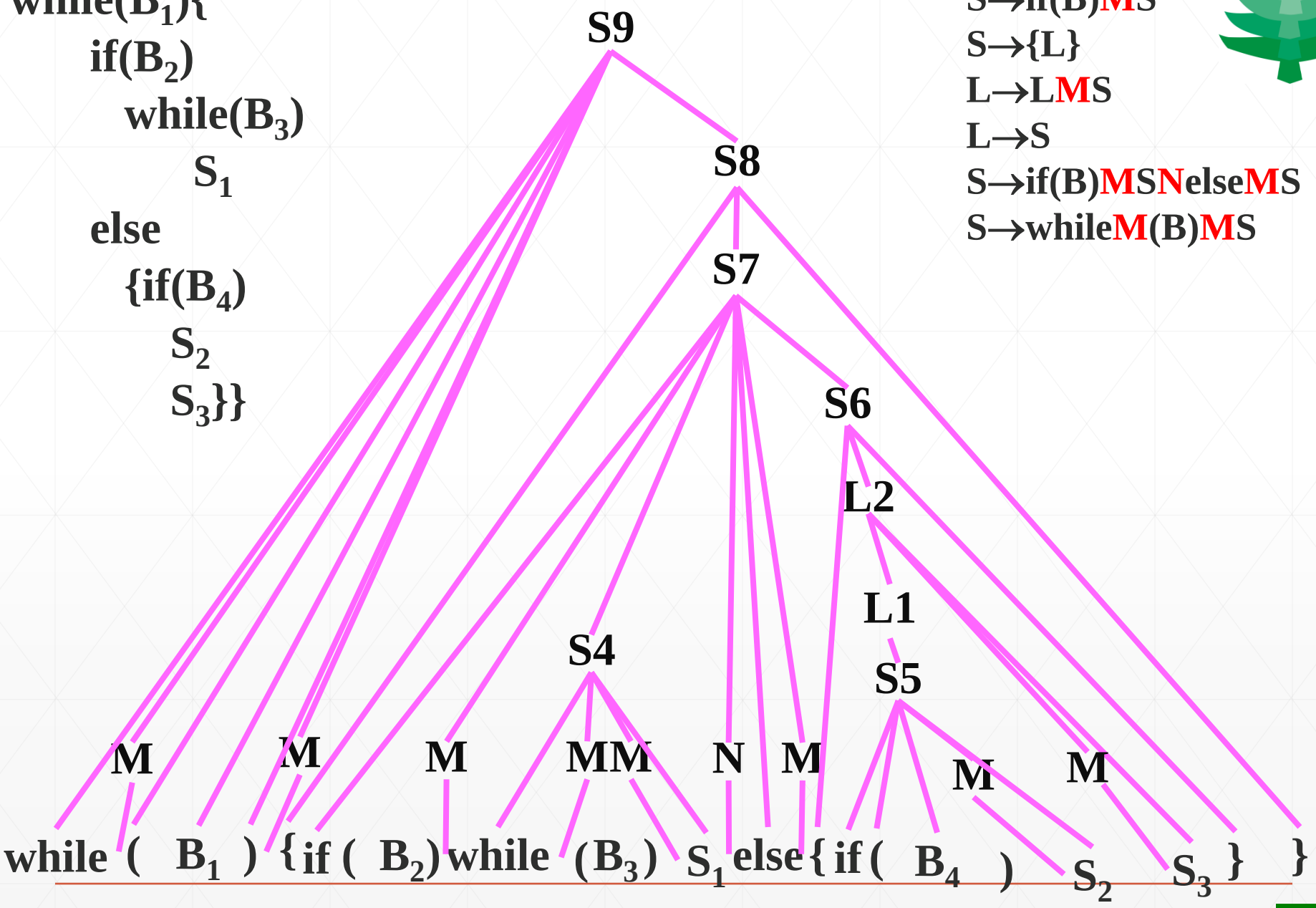
{backpatch(S_1.next,M_1.instr);
backpatch(B.true,M_2.instr);
S.next=B.false;
GEN(J - - M_1.instr)}

例：模拟一遍编译的技术将下面的类C语言代码翻译为四元式形式的中间代码,并给出最后的标号表。

```
while(B1){
    if(B2)
      while(B3)
          S1
    else
      {if(B4)
          S2
          S3}}
```

```
while(B1){
    if(B2)
      while(B3)
          S1
    else
      {if(B4)
          S2
          S3}}
```

S→if(B)MS
S→{L}
L→LMS
L→S
S→if(B)MSNelseMS
S→whileM(B)MS



N	D	A
B1T	0	2
B1F	0	3
B2T	0	5
B2F	0	6
B3T	1	10
B3F	0	9
S4N	0	9
N1N	0	12
B4T	0	14
B4F	0	15

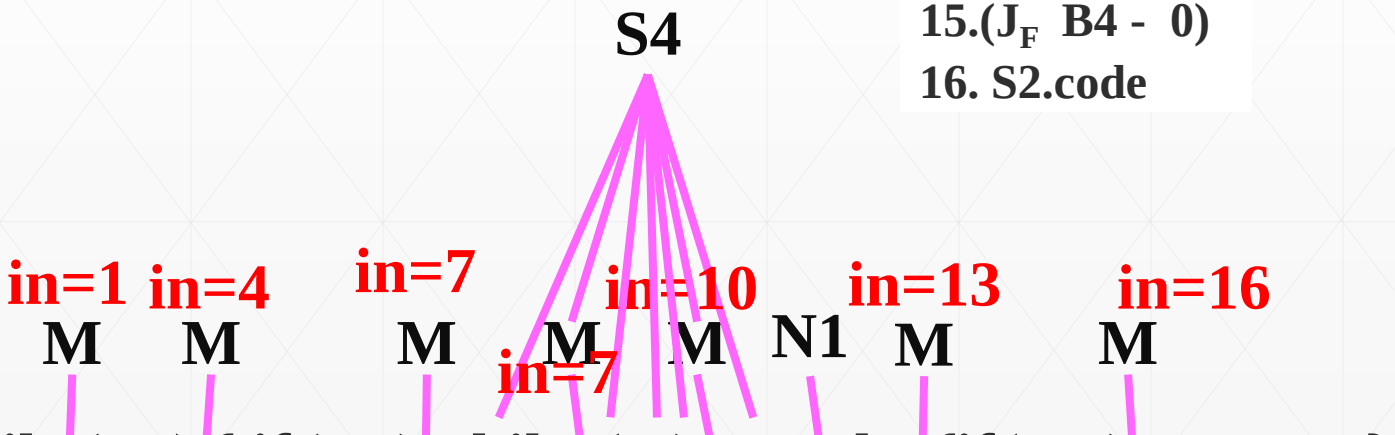
$M \rightarrow \varepsilon$
{M.instr=nextinstr;}

$N \rightarrow \varepsilon$
{N.next=makelist(nextinstr);
GEN(J - -N.next)}

$S \rightarrow$whileM (B) MS
{backpatch(S_1.next,M_1.instr);
backpatch(B.true, M_2.instr);
S.next=B.false;
GEN(J - - M_1.instr)}

1.B1.code
2.(J_T B1 - 0)
3.(J_F B1 - 0)
4.B2.code
5.(J_T B2 - 0)
6.(J_F B2 - 0)
7.B3.code
8.(J_T B3 - 10
9.(J_F B3 - 0)
10.S1.code
11.(J - - 7)
12. (J - - 0)
13.B4.code
14.(J_T B4 - 0)
15.(J_F B4 - 0)
16. S2.code

翻译

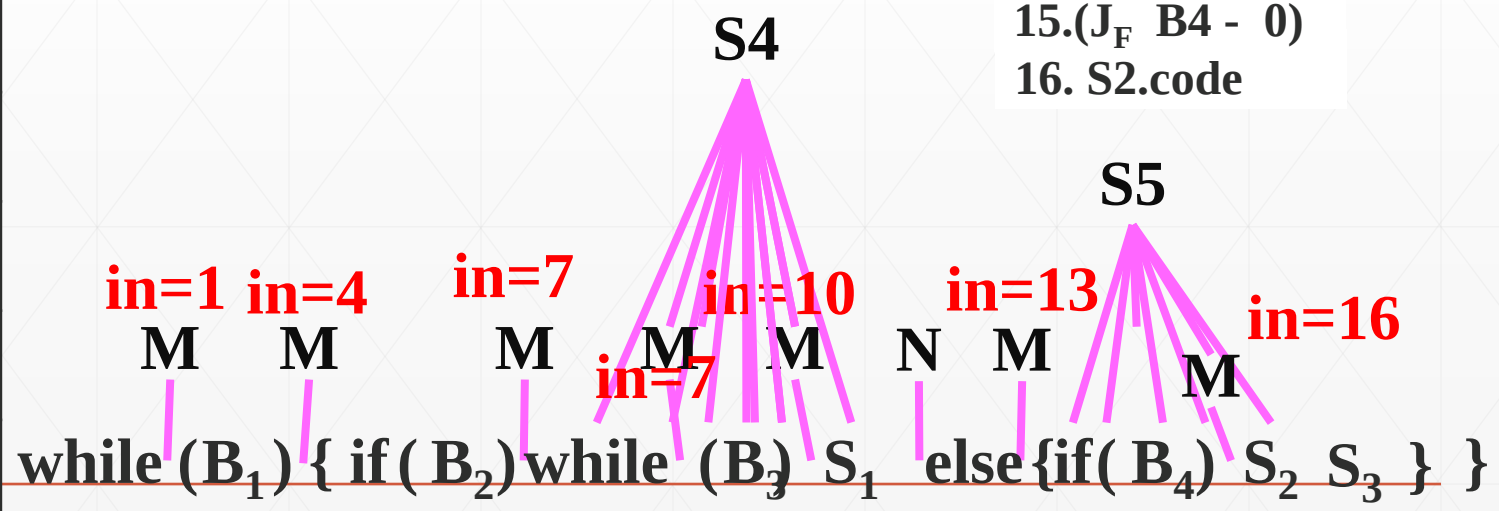


N	D	A
B1T	0	2
B1F	0	3
B2T	0	5
B2F	0	6
B3T	1	10
B3F	0	9
S4N	0	9
N1N	0	12
B4T	1	16
B4F	0	15
S5N	0	15

M→ε
{M.instr=nextinstr;

S→if(B)MS
{backpatch(B.true, M.instr);
S.next=merge(B.false,S_1.next);}

1.B1.code
2.(J_T B1 - 0)
3.(J_F B1 - 0)
4.B2.code
5.(J_T B2 - 0)
6.(J_F B2 - 0)
7.B3.code
8.(J_T B3 - 10
9.(J_F B3 - 0)
10.S1.code
11.(J - - 7)
12. (J - - 0)
13.B4.code
14.(J_T B4 - 16
15.(J_F B4 - 0)
16. S2.code



N	D	A
B1T	0	2
B1F	0	3
B2T	0	5
B2F	0	6
B3T	1	10
B3F	0	9
S4N	0	9
N1N	0	12
B4T	1	16
B4F	0	15
S5N	0	15

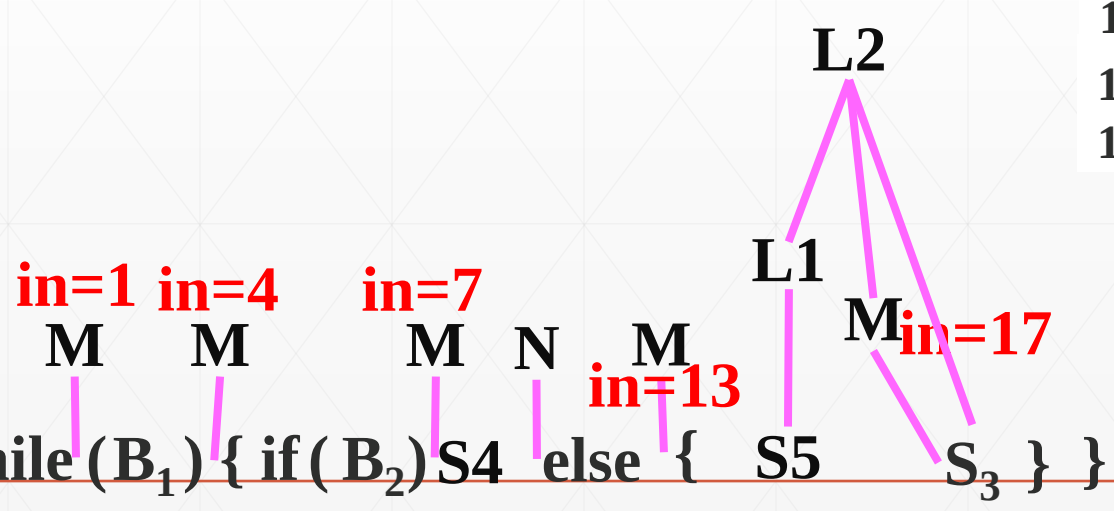
M→ε
{M.instr=nextinstr;}

L→S
{L.next=S.next;}

L→LMS
{backpatch(L_1.next, M.instr);
L.next=S.next;}

N	D	A
L1N	1	17

1.B1.code
2.(J_T B1 - 0)
3.(J_F B1 - 0)
4.B2.code
5.(J_T B2 - 0)
6.(J_F B2 - 0)
7.B3.code
8.(J_T B3 - 10
9.(J_F B3 - 0)
10.S1.code
11.(J - - 7)
12. (J - - 0)
13.B4.code
14.(J_T B4 - 16
15.(J_F B4 - 17)
16. S2.code
17.S3.code



N	D	A
B1T	0	2
B1F	0	3
B2T	1	7
B2F	1	13
B3T	1	10
B3F	0	9
S4N	0	9
N1N	0	12
B4T	1	16
B4F	0	15
S5N	0	15

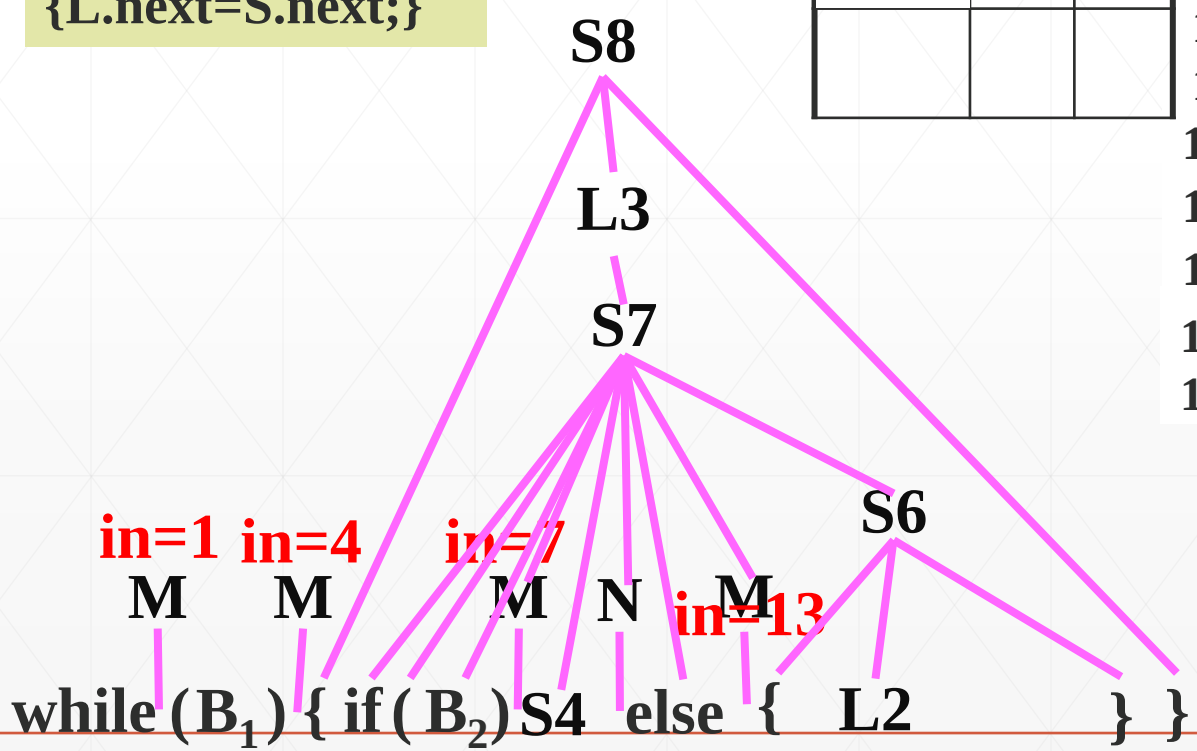
S→{L}
{S.next=L.next;}

S→if(B)MSNelseMS
{backpatch(B.true,M_1.instr);
backpatch(B.false,M_2.instr);
S.next=merge(S_1.next,N.next,S_2.next);}

L→S
{L.next=S.next;}

N	D	A
L1N	1	17
S7N	0	9
L3N	0	9
S8N	0	9

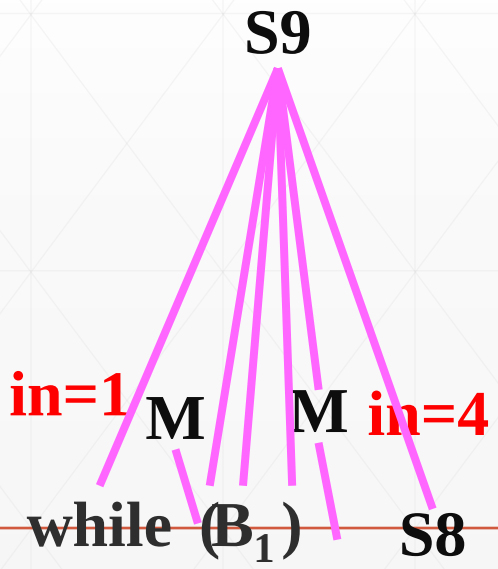
1.B1.code
2.(J_T B1 - 0)
3.(J_F B1 - 0)
4.B2.code
5.(J_T B2 - 7)
6.(J_F B2 - 13
7.B3.code
8.(J_T B3 - 10
9.(J_F B3 - 12
10.S1.code
11.(J - - 7)
12. (J - - 0)
13.B4.code
14.(J_T B4 - 16
15.(J_F B4 - 17)
16. S2.code
17.S3.code



N	D	A
B1T	1	4
B1F	0	3
B2T	1	7
B2F	1	13
B3T	1	10
B3F	0	9
S4N	0	9
N1N	0	12
B4T	1	16
B4F	0	15
S5N	0	15

S→whileM (B) MS
{backpatch(S_1.next,M_1.instr);
backpatch(B.true,M_2.instr);
S.next=B.false;
GEN(J - - M_1.instr)}

N	D	A
L1N	1	17
S7N	0	9
L3N	0	9
S8N	1	1
S9N	0	3



1.B1.code
2.(J_T B1 - 4
3.(J_F B1 - 0)
4.B2.code
5.(J_T B2 - 7)
6.(J_F B2 - 13
7.B3.code
8.(J_T B3 - 10
9.(J_F B3 - 1
10.S1.code
11.(J - - 7)
12. (J - - 1
13.B4.code
14.(J_T B4 - 16
15.(J_F B4 - 17)
16. S2.code
17.S3.code
18.(J - - 1)

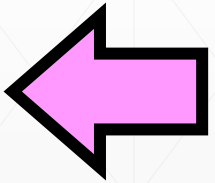
6.6 语句翻译与中间代码生成

6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译

6.6.4 数组说明与数组元素引用的翻译



6.6.5 过程、函数说明和调用的翻译

一. 数组说明

type A $[d_1]$ $[d_2]$… $[d_n]$;

元素类型

数组名

d_i的值
每维大小

n: d_i个数
数组维数

d_1*d_2*… *d_n
数组体积

type A $[d_1 .. d_1']$ $[d_2 .. d_2']$… $[d_n .. d_n']$;

数组说明的语义处理

填表，登记数组的属性信息。

type
name
dim
dim_value
vol

} 符号表 {

公共、等长信息
（符号表）

与计算数组元素地址有关、不等长信息
（信息向量表）

return

例:设有说明

int a[2][2];

float b[4];

namelist

name	kind	type	addr
a	array	I	
b	array	R	

信息向量表

a	2
	2
	2
	2*2
	C_a
	a_0
b	1
	4
	4*1
	C_b
	b_0

信息向量表

维数
第一维大小
第二维大小
…
第n维大小
体积 $=d_1*d_2*\ldots*d_n$
计算数元地址不变部分C
数组第一个元素地址a
…

若为上下界需计算:

a(2..10)→10-2+1=9

计算时间:

尽量编译时算

动态数组如何处理?

二. 数组元素引用

type A $[d_1]$ $[d_2]$… $[d_n]$; 数组说明

A $[i_1]$ $[i_2]$… $[i_n]$ 数组引用

** 不能整体引用，仅对单一数组元素引用。

- 数组元素引用的语义处理

语义检查： 类型匹配；下标越界检查；

产生代码：数组元素地址计算的中间代码。

数组元素地址计算编译时能否完成？

一般不能

∵ A $[i_1]$ $[i_2]$… $[i_n]$ 中 i_k多为表达式

如， a[i+j-1][j++] 运行时得到下标值

涉及数组元素地址计算的有关知识

数组存储方式
(线性连续)
- 按行存储
- 按列存储

例如， int a[2][2];

按行

a_0	a[0][0]
	a[0][1]
	a[1][0]
	a[1][1]

按列

a_0	a[0][0]
	a[1][0]
	a[0][1]
	a[1][1]

$a[0][1]_{add} = a_0 + 1*width$

$a[0][1]_{add} = a_0 + 2*width$

数组元素地址计算

据存储方式，通过下述规则计算数组元素位置

(顺序号) *width + a_0

- 对一维数组 int a[n]; // 默认下标下界=0

则 $a[i]_{addr} = a_0 + (i) * width$

(若数组下标下限=1，则$a[i] = a_0 + (i-1) * width$)

- 对二维数组 int a[n][m] ;

则 $a[i][j]_{addr} = a_0 +(i *m + j)*width$

若数组下标下限=1，则

$a[i][j]_{add} = a_0 +((i-1) *m + (j-1))*width$

- 对n维数组 int a[d_1] [d_2]… [d_n];

则 a[i_1] [i_2] … [i_n] add

= a_0 +(i_1 *d_2*d_3 * … *d_n

+ i_2 *d_3 *d_4*… *d_n

+ …

+ i_{n-1} *d_n + i_n) *width

含i_k,是可变部分，程序运行时方可知。

进一步考虑更一般的情况：若数组下标下限≠0

若数组下标下限=1

则 $a[i_1][i_2]\dots[i_n]_{add}$

$= a_0 + ((i_1-1)*d_2*d_3*\dots*d_n$

$+ (i_2-1)*d_3*d_4*\dots*d_n$

$+ \dots$

$+ (i_{n-1}-1)*d_n + (i_n-1))*width$

变换

V

$=a_0+(i_1d_2d_3\dots d_n + i_2d_3d_4\dots d_n+\dots+i_{n-1}d_n+i_n)*width$

$-(d_2d_3\dots d_n+d_3d_4\dots d_n+\dots+d_n+1)*width$

不变

C

$a[i_1][i_2]\ldots[i_n]$ add

V

$=a_0+(i_1d_2d_3\ldots d_n+i_2d_3d_4\ldots d_n+\ldots+i_{n-1}d_n+i_n)*width$

$-(d_2d_3\ldots d_n+d_3d_4\ldots d_n+\ldots+d_n+1)*width$

不变

C

$= a_0 - C + V$

conspart varpart

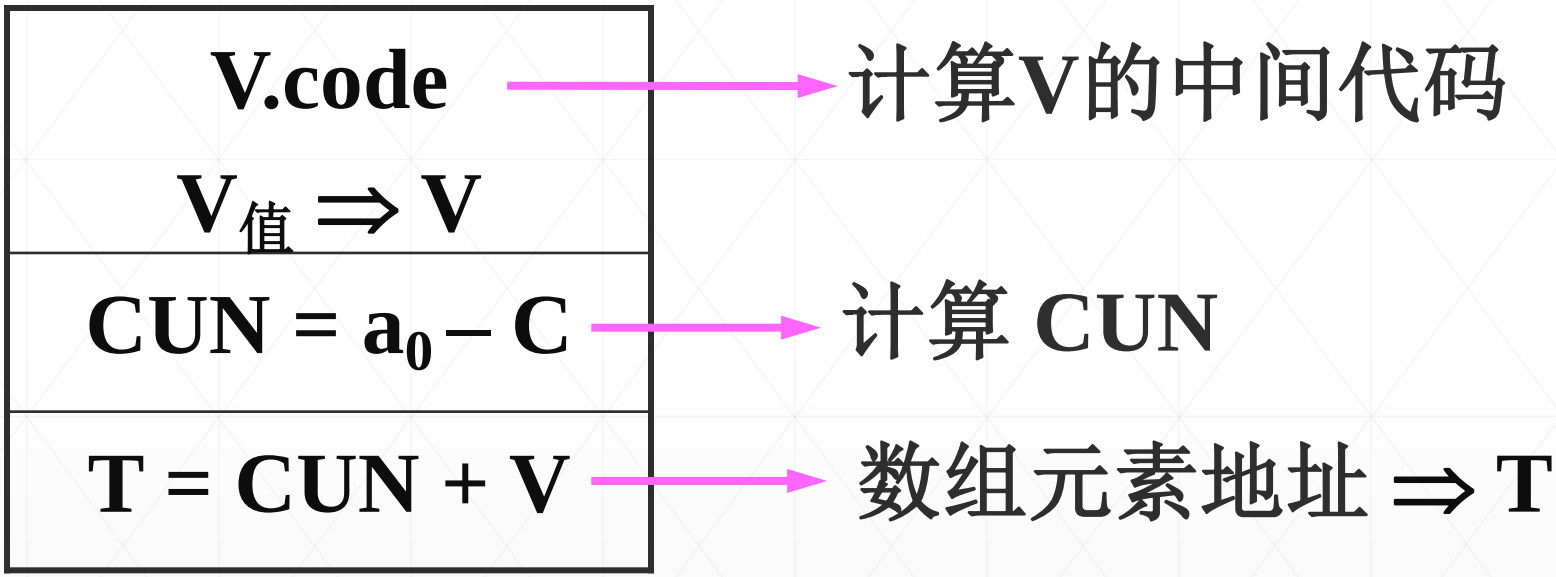
$= CUN + V$

编译时计算,填入内情向量表

据数元引用情况，产生计算的中间代码，程序运行时算出。

信息向量表

数组元素引用目标结构



例 设有程序段

```
…
int  a[10][20]
…
s= a[x][y];
…
a[x][y]=t;
```

namelist

name	kind	type	addr
a	array	I	
…			

V.code
$V_{值} \Rightarrow V$
CUN = a_0 – C
T = CUN + V

a[x][y]$_{addr}$•code

① (*, x ,20,t1)
② (+, t1,y,t2)
③ (*,4,t2 ,t3)
④ (–, a_0, 84,t4)
⑤ (=[],t4[t3],-,t5)
⑥ (=,t5,-,s)
…
([]=,t,-, t9[t8])

内情向量表
2
10
20
10*20=200
C=(20+1)*4=84
a_0
…

- 动态数组？

  type A $[d_1]$ $[d_2]$… $[d_n]$； （d_i是变量）

  (每维大小、体积等信息程序运行时确定)

- 动态数组处理

  内情向量表在程序运行时建立，编译时仅分配内情向量表所占存储空间(据维数)。在运行时， d_i确定后，计算体积、计算C、为数组分配存储空间且将这些信息登录内情向量表。

∴ 动态数组：产生此部分代码 + 静态数组元素引用

动态数组处理目标结构

取$d_i \Rightarrow$ 内情向量表code	动态数组说明
计算体积、C的code	
为数组分配存储的code	
V.code $V_{值} \Rightarrow V$	计算V的中间代码
CUN = a_0 – C	计算不变部分 $\Rightarrow$ CUN
T = CUN + V	数组元素地址 $\Rightarrow$ T

6.6 语句翻译与中间代码生成

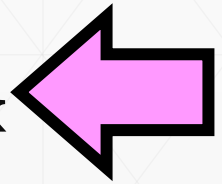
6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译

6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译



函数翻译
- 处理说明（定义）
- 处理调用

- 函数(过程)说明和函数调用

常见的语法结构, 形式随语言的不同而有所不同在功能上和需做的语义处理工作上基本类似。

比如：有的语言说明由关键字(PROCEDURE, FUCTION)引导，有些语言则直接定义。

一. 函数说明的翻译

1. 函数类型（构造类型）

返回值类型和形式参数类型

2. 填写符号表

建立针对该函数的符号表，并填入相关信息

3.形式参数存储单元管理

4. 产生执行函数体的代码注意：

(1) 返回地址保存，调用前状态信息存储

(2) 恢复调用前状态信息

例6.5 设有C函数

```
fun1( int a,b)
 {
    int   c;
    c=a+b ;
    return ( c )
 }
```

namelist

name	kind	type	len	offset	addr
fun1	函数	I			→ code
a	形参	I	4	0	
b	形参	I	4	4	
c	V	I	4	8	

二. 函数调用的翻译

G: S → id (Elist)

Elist → Elist，E | E

其中： id: 函数名；

Elist: 实参表；

E: 表达式。

■ 函数(过程) 调用的翻译

依赖于形式参数与实在参数结合的方式以及数据存储空间的分配方式。

- 函数调用的语义处理：

① 检查所调用的过程或函数是否定义；与所定义的过程或函数的类型、实参与形参的数量及类型是否一致；

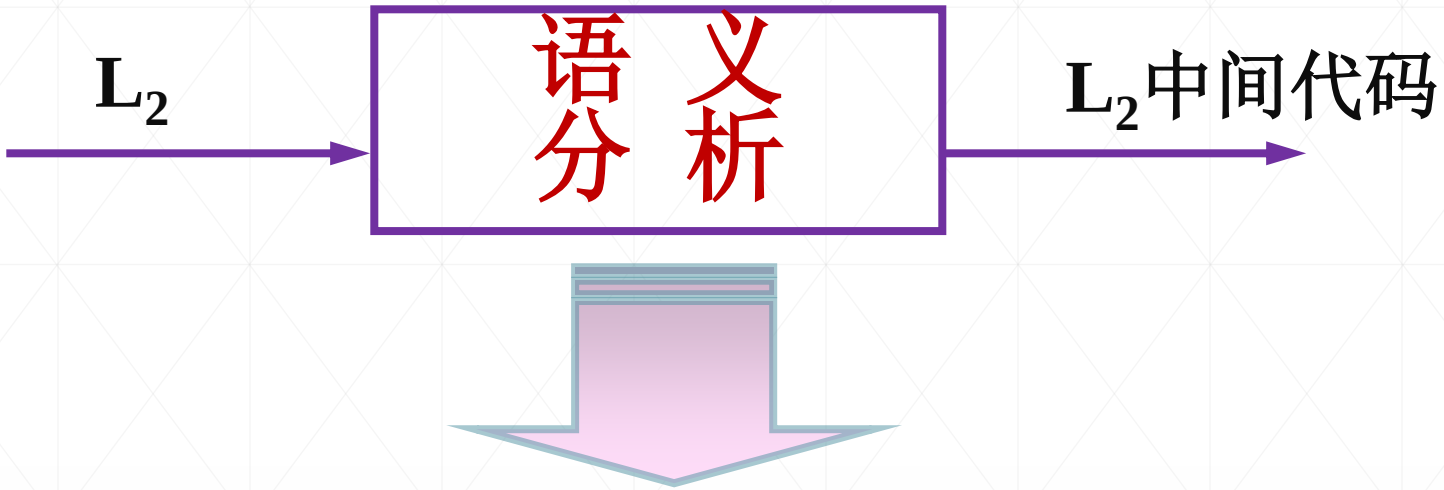
② 给被调过程或函数分配活动记录所需的存储空间；

③ 传送实参并执行代码；

例：n=f(a[i]) a为整数数组，数组下标从0开始

```
* i 4 t1
=[] a[t1]   t2
param t2
call f  1 t3
= t3        n
```

语义分析与中间代码生成总结



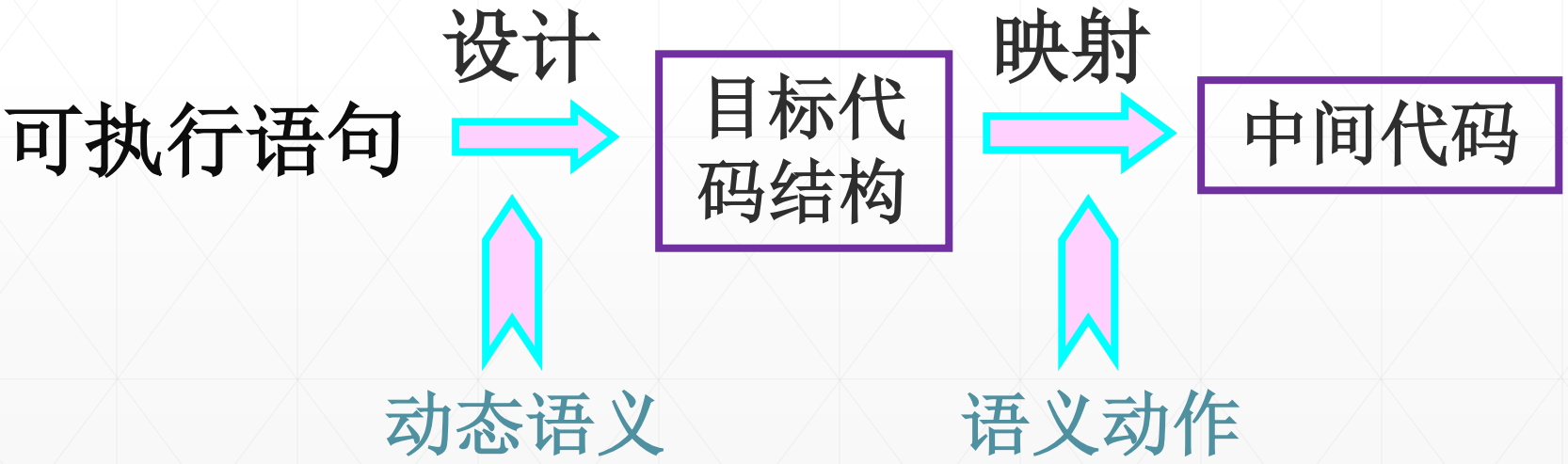
根据语义进行语义检查和代码生成

对L_2扫描的顺序性；产生的代码执行的线性性；

语义映射的准确性；

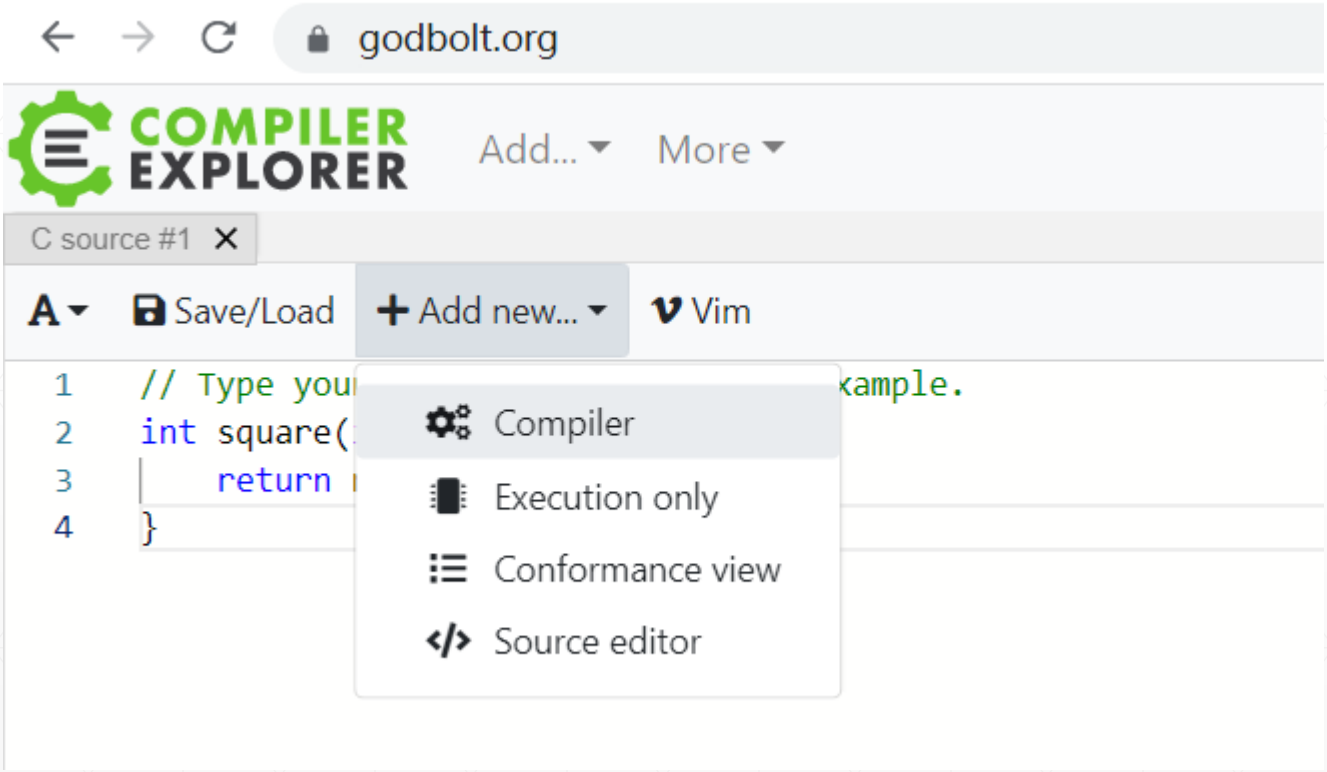
语义分析与中间代码生成

说明性语句 → 登录编译信息、为可执行语句提供语义检查和语句翻译的信息依据。



语句翻译

- 观察中间代码输出



第六章 结束