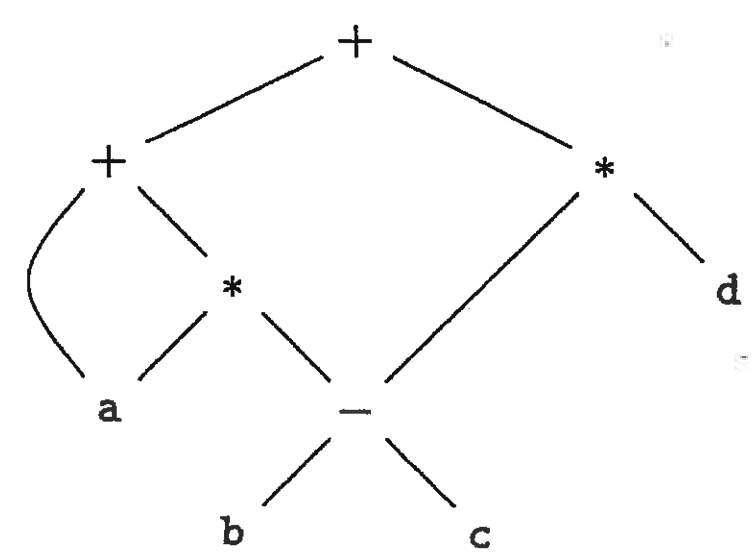


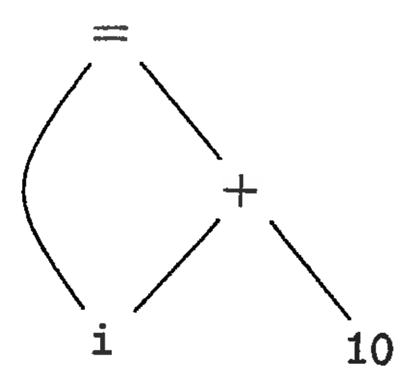
Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

1) $p_1 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}a})$
2) $p_2 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}a}) = p_1$
3) $p_3 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}b})$
4) $p_4 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}c})$
5) $p_5 = \mathit{Node}('-', p_3, p_4)$
6) $p_6 = \mathit{Node}('*', p_1, p_5)$
7) $p_7 = \mathit{Node}('+', p_1, p_6)$
8) $p_8 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}b}) = p_3$
9) $p_9 = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}c}) = p_4$
10) $p_{10} = \mathit{Node}('-', p_3, p_4) = p_5$
11) $p_{11} = \mathit{Leaf}(\mathbf{id}, \mathit{entry\text{-}d})$
12) $p_{12} = \mathit{Node}('*', p_5, p_{11})$
13) $p_{13} = \mathit{Node}('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

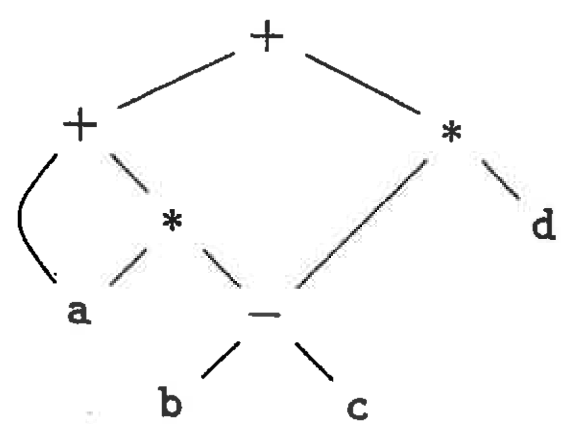


(a) DAG

1	id	→ to entry for i	
2	num	10	
3	+	1	2
4	=	1	3
5	...		

(b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array



(a) DAG

```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

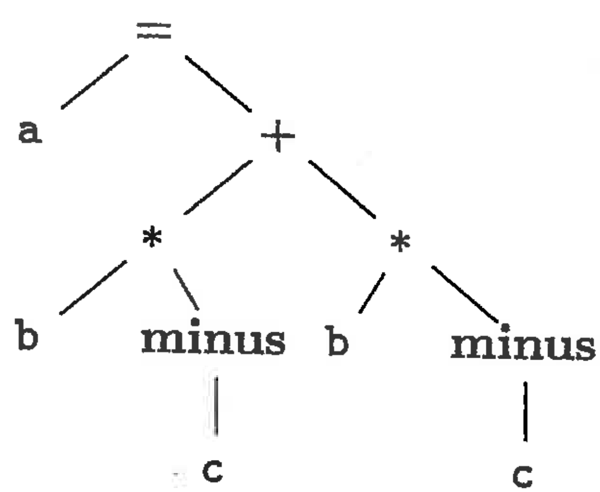
(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

For $a = b * -c + b * -c$



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

(b) Triples

Figure 6.11: Representations of $\cancel{a + a * (b - c) + (b - c) * d}$
$a = b * -c + b * -c$

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Figure 6.12: Indirect triples representation of three-address code

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

$$
\begin{aligned}
p_1 &= a + b\\
q_1 &= p_1 - c\\
p_2 &= q_1 * d\\
p_3 &= e - p_2\\
q_2 &= p_3 + q_1
\end{aligned}
$$

(b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

$$
\begin{array}{lll}
P \rightarrow & & \{\ \mathit{offset} = 0;\ \} \\
& D & \\
D \rightarrow & T\ \textbf{id}\ ; & \{\ \mathit{top.put}(\textbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\
& & \quad \mathit{offset} = \mathit{offset} + T.\mathit{width};\ \} \\
& D_1 & \\
D \rightarrow & \epsilon &
\end{array}
$$

Figure 6.17: Computing the relative addresses of declared names

$$
\begin{array}{lll}
T \rightarrow & B & \{\ t = B.\mathit{type};\ w = B.\mathit{width};\ \} \\
& C & \{ T.type = C.type;\ T.width = C.width; \} \\
B \rightarrow & \textbf{int} & \{\ B.\mathit{type} = \mathit{integer};\ B.\mathit{width} = 4;\ \} \\
B \rightarrow & \textbf{float} & \{\ B.\mathit{type} = \mathit{float};\ B.\mathit{width} = 8;\ \} \\
C \rightarrow & \epsilon & \{\ C.\mathit{type} = t;\ C.\mathit{width} = w;\ \} \\
C \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ C.type = \mathit{array}(\textbf{num}.\mathit{value},\ C_1.\mathit{type}); \\
& & \quad C.\mathit{width} = \textbf{num}.\mathit{value} \times C_1.\mathit{width};\ \}
\end{array}
$$

Figure 6.15: Computing types and their widths

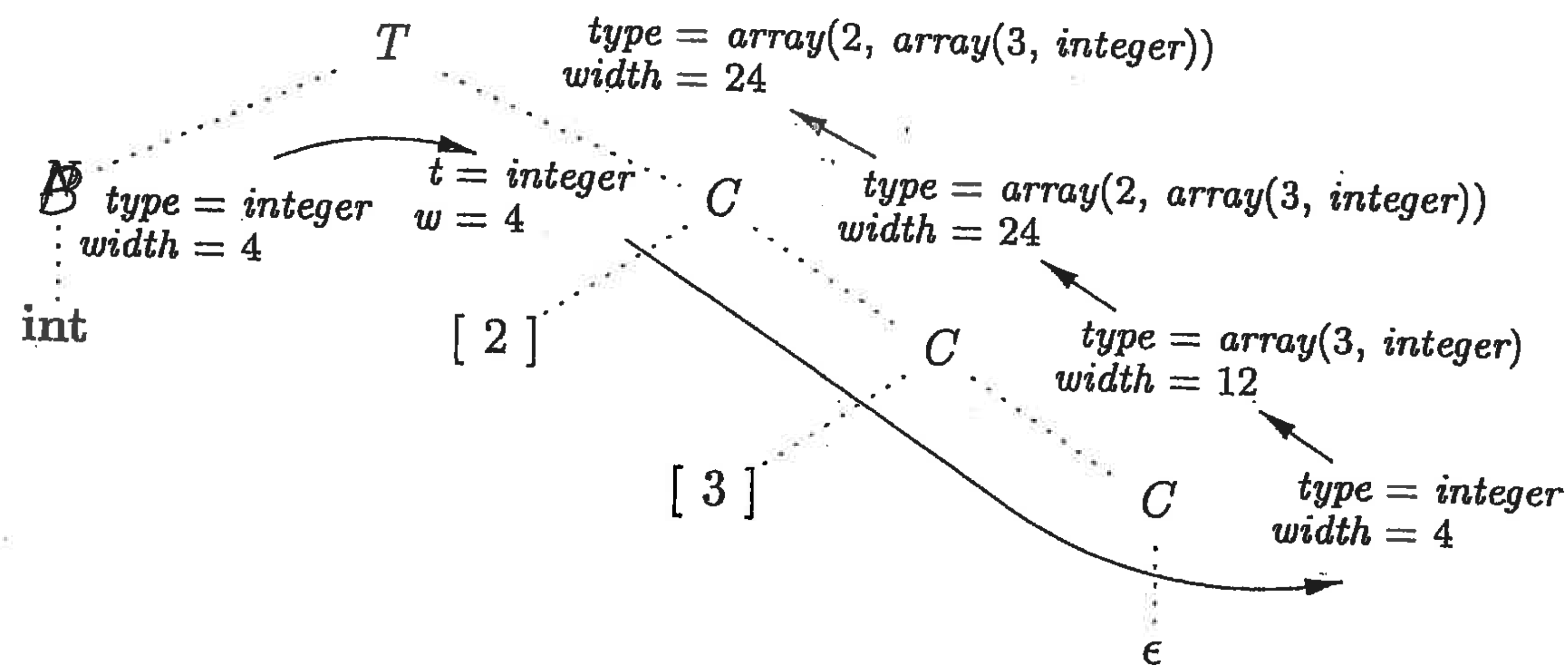


Figure 6.16: Syntax-directed translation of array types

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \;\|\|$ $\quad gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \;\|\|\; E_2.code \;\|\|$ $\quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\quad \|\ -E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \;\|\|$ $\quad gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\quad \|\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad \|\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Figure 6.19: Three-address code for expressions

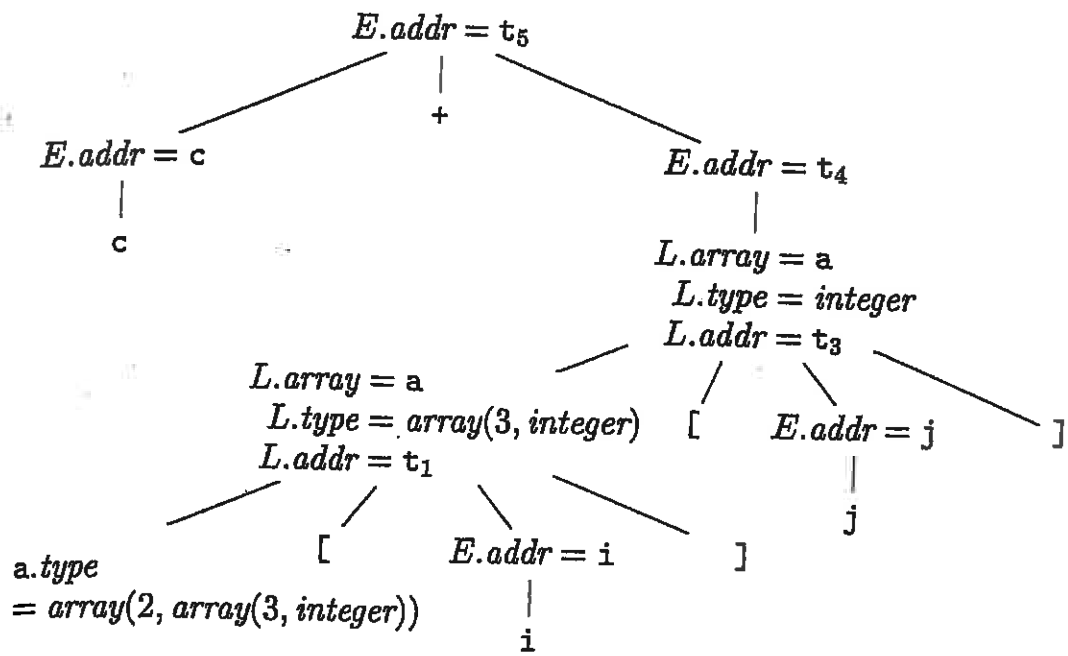


Figure 6.23: Annotated parse tree for c+a[i][j]

$$
\begin{aligned}
t_1 &= i * 12 \\
t_2 &= j * 4 \\
t_3 &= t_1 + t_2 \\
t_4 &= a\ [\ t_3\ ] \\
t_5 &= c + t_4
\end{aligned}
$$

Figure 6.24: Three-address code for expression c+a[i][j]

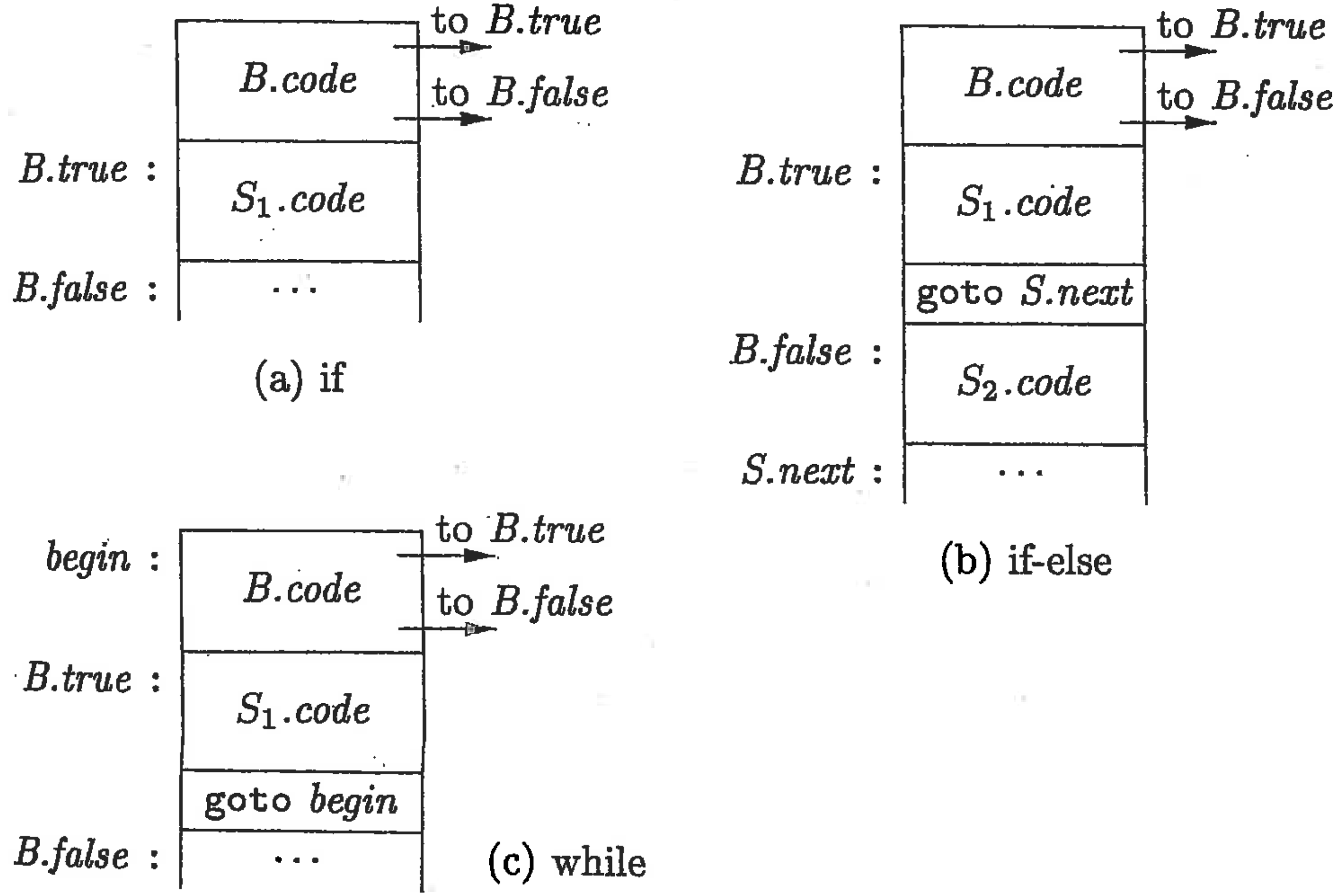


Figure 6.35: Code for if-, if-else-, and while-statements

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen(\texttt{'goto'}\ S.next)$ $\parallel label(B.false) \parallel S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen(\texttt{'goto'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

Figure 6.36: Syntax-directed definition for flow-of-control statements.

<table>
<tr><th>PRODUCTION</th><th>SEMANTIC RULES</th></tr>
<tr><td>$B \rightarrow B_1 \;||\; B_2$</td><td>$B_1.\mathit{true} = B.\mathit{true}$
$B_1.\mathit{false} = \mathit{newlabel}()$
$B_2.\mathit{true} = B.\mathit{true}$
$B_2.\mathit{false} = B.\mathit{false}$
$B.\mathit{code} = B_1.\mathit{code} \;||\; \mathit{label}(B_1.\mathit{false}) \;||\; B_2.\mathit{code}$</td></tr>
<tr><td>$B \rightarrow B_1 \;\&\&\; B_2$</td><td>$B_1.\mathit{true} = \mathit{newlabel}()$
$B_1.\mathit{false} = B.\mathit{false}$
$B_2.\mathit{true} = B.\mathit{true}$
$B_2.\mathit{false} = B.\mathit{false}$
$B.\mathit{code} = B_1.\mathit{code} \;||\; \mathit{label}(B_1.\mathit{true}) \;||\; B_2.\mathit{code}$</td></tr>
<tr><td>$B \rightarrow \;!\, B_1$</td><td>$B_1.\mathit{true} = B.\mathit{false}$
$B_1.\mathit{false} = B.\mathit{true}$
$B.\mathit{code} = B_1.\mathit{code}$</td></tr>
<tr><td>$B \rightarrow E_1 \;\textbf{rel}\; E_2$</td><td>$B.\mathit{code} = E_1.\mathit{code} \;||\; E_2.\mathit{code}$
$\quad ||\; \mathit{gen}(\texttt{'if'}\; E_1.\mathit{addr}\; \textbf{rel}.\mathit{op}\; E_2.\mathit{addr}\; \texttt{'goto'}\; B.\mathit{true})$
$\quad ||\; \mathit{gen}(\texttt{'goto'}\; B.\mathit{false})$</td></tr>
<tr><td>$B \rightarrow \textbf{true}$</td><td>$B.\mathit{code} = \mathit{gen}(\texttt{'goto'}\; B.\mathit{true})$</td></tr>
<tr><td>$B \rightarrow \textbf{false}$</td><td>$B.\mathit{code} = \mathit{gen}(\texttt{'goto'}\; B.\mathit{false})$</td></tr>
</table>

Figure 6.37: Generating three-address code for booleans

$$B \to B_1 \mid\mid M\, B_2 \mid B_1 \,\&\&\, M\, B_2 \mid !\, B_1 \mid (\, B_1\,) \mid E_1 \textbf{ rel } E_2 \mid \textbf{true} \mid \textbf{false}$$
$$M \to \epsilon\ .$$

The translation scheme is in Fig. 6.43.

1)	$B \to B_1 \mid\mid M\, B_2$	$\{$ *backpatch*$(B_1.\mathit{falselist}, M.\mathit{instr})$; $B.\mathit{truelist} = \mathit{merge}(B_1.\mathit{truelist}, B_2.\mathit{truelist})$; $B.\mathit{falselist} = B_2.\mathit{falselist}$; $\}$
2)	$B \to B_1 \,\&\&\, M\, B_2$	$\{$ *backpatch*$(B_1.\mathit{truelist}, M.\mathit{instr})$; $B.\mathit{truelist} = B_2.\mathit{truelist}$; $B.\mathit{falselist} = \mathit{merge}(B_1.\mathit{falselist}, B_2.\mathit{falselist})$; $\}$
3)	$B \to\ !\, B_1$	$\{$ $B.\mathit{truelist} = B_1.\mathit{falselist}$; $B.\mathit{falselist} = B_1.\mathit{truelist}$; $\}$
4)	$B \to (\, B_1\,)$	$\{$ $B.\mathit{truelist} = B_1.\mathit{truelist}$; $B.\mathit{falselist} = B_1.\mathit{falselist}$; $\}$
5)	$B \to E_1 \textbf{ rel } E_2$	$\{$ $B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr})$; $B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr} + 1)$; *gen*('if' $E_1.\mathit{addr}$ **rel**.*op* $E_2.\mathit{addr}$ 'goto _'); *gen*('goto _'); $\}$
6)	$B \to \textbf{true}$	$\{$ $B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr})$; *gen*('goto _'); $\}$
7)	$B \to \textbf{false}$	$\{$ $B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr})$; *gen*('goto _'); $\}$
8)	$M \to \epsilon$	$\{$ $M.\mathit{instr} = \mathit{nextinstr}$; $\}$

Figure 6.43: Translation scheme for boolean expressions

1) $S \rightarrow \textbf{if} \, (\, B \,) \, M \, S_1$ $\{$ $backpatch(B.truelist,\ M.instr);$
$S.nextlist \ = \ merge(B.falselist,\ S_1.nextlist);$ $\}$

2) $S \rightarrow \ \textbf{if} \, (\, B \,) \, M_1 \, S_1 \, N \, \textbf{else} \, M_2 \, S_2$
$\{$ $backpatch(B.truelist,\ M_1.instr);$
$backpatch(B.falselist,\ M_2.instr);$
$temp \ = \ merge(S_1.nextlist,\ N.nextlist);$
$S.nextlist \ = \ merge(temp,\ S_2.nextlist);$ $\}$

3) $S \rightarrow \ \textbf{while} \, M_1 \, (\, B \,) \, M_2 \, S_1$
$\{$ $backpatch(S_1.nextlist,\ M_1.instr);$
$backpatch(B.truelist,\ M_2.instr);$
$S.nextlist \ = \ B.falselist;$
$gen($'goto' $M_1.instr);$ $\}$

4) $S \rightarrow \{ \, L \, \}$ $\{$ $S.nextlist \ = \ L.nextlist;$ $\}$

5) $S \rightarrow A \ ;$ $\{$ $S.nextlist \ = \ \textbf{null};$ $\}$

6) $M \rightarrow \epsilon$ $\{$ $M.instr \ = \ nextinstr;$ $\}$

7) $N \rightarrow \epsilon$ $\{$ $N.nextlist \ = \ makelist(nextinstr);$
$gen($'goto _'$);$ $\}$

8) $L \rightarrow L_1 \ M \ S$ $\{$ $backpatch(L_1.nextlist,\ M.instr);$
$L.nextlist \ = \ S.nextlist;$ $\}$

9) $L \rightarrow S$ $\{$ $L.nextlist \ = \ S.nextlist;$ $\}$

Figure 6.46: Translation of statements

```
          code to evaluate E into t
          goto test
L1:       code for S1
          goto next
L2:       code for S2
          goto next
          ...
Ln-1:     code for Sn-1
          goto next
Ln:       code for Sn
          goto next
test:     if t = V1 goto L1
          if t = V2 goto L2
          ...
          if t = Vn-1 goto Ln-1
          goto Ln
next:
```

Figure 6.49: Translation of a switch-statement

```
          code to evaluate E into t
          if t != V_1 goto L_1
          code for S_1
          goto next
L_1:      if t != V_2 goto L_2
          code for S_2
          goto next
L_2:
          ...
L_{n-2}:  if t != V_{n-1} goto L_{n-1}
          code for S_{n-1}
          goto next
L_{n-1}:  code for S_n
next:
```

Figure 6.50: Another translation of a switch statement