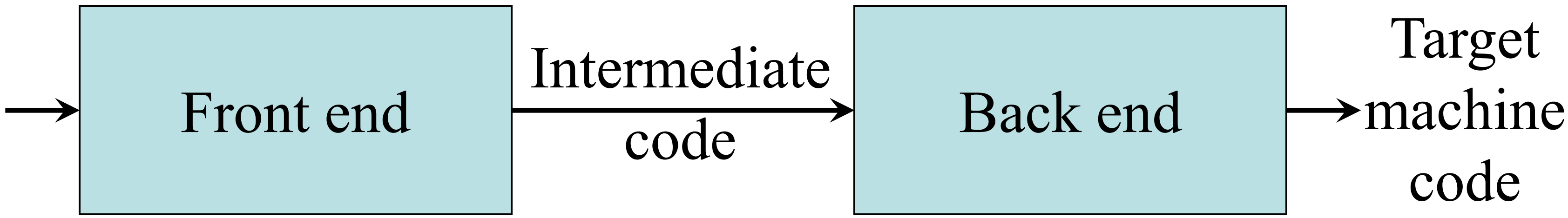
CODE GENERATION

UNIT IV

Intermediate Code Generation

Intermediate Code Generation

- Facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end



- Enables machine-independent code optimization

Intermediate Representations

- *Graphical representations* (e.g. AST)
- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (e.g. *triples* and *quads*)

  $x := y \text{ op } z$
- *Two-address code*:

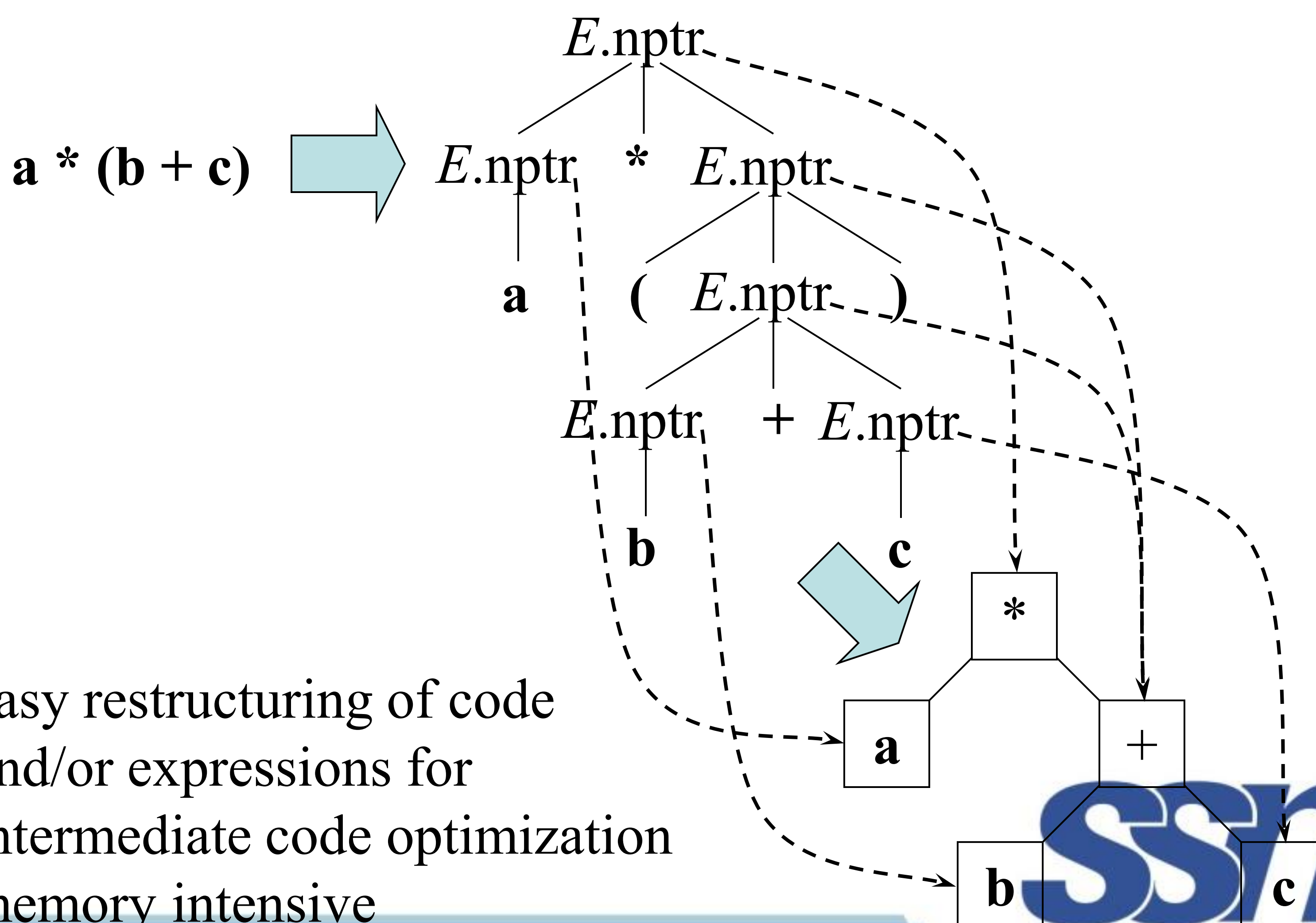
  $x := \text{op } y$

  which is the same as $x := x \text{ op } y$

Syntax-Directed Translation of Abstract Syntax Trees

Production	Semantic Rule
$S \to \textbf{id} := E$	S.nptr := *mknode*(':=' , *mkleaf*(**id**, **id**.entry), E.nptr)
$E \to E_1 + E_2$	E.nptr := *mknode*('+' , E_1.nptr, E_2.nptr)
$E \to E_1 * E_2$	E.nptr := *mknode*('*' , E_1.nptr, E_2.nptr)
$E \to \textbf{-}\ E_1$	E.nptr := *mknode*('uminus' , E_1.nptr)
$E \to \textbf{(}\ E_1\ \textbf{)}$	E.nptr := E_1.nptr
$E \to \textbf{id}$	E.nptr := *mkleaf*(**id**, **id**.entry)

Abstract Syntax Trees

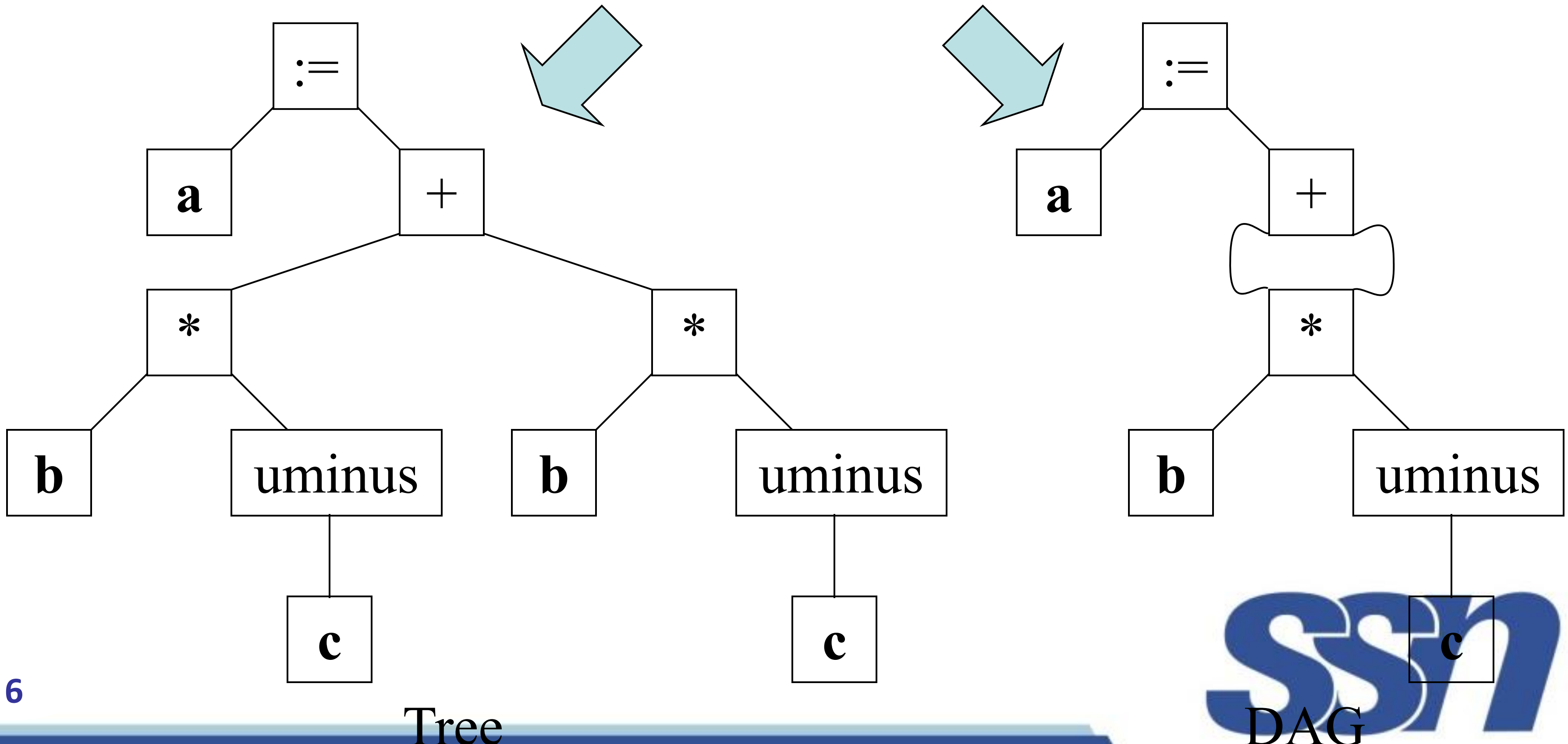


Pro: easy restructuring of code and/or expressions for intermediate code optimization

Cons: memory intensive

Abstract Syntax Trees versus DAGs

a := b * -c + b * -c



Postfix Notation

a := b * -c + b * -c

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

Bytecode (for example)

```
iload 2        // push b
iload 3        // push c
ineg           // uminus
imul           // *
iload 2        // push b
iload 3        // push c
ineg           // uminus
imul           // *
iadd           // +
istore 1       // store a
```

Pro: easy to generate

Cons: stack operations are more difficult to optimize

Three-Address Code

a := b * -c + b * -c

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

Linearized representation of a syntax tree

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

Linearized representation of a syntax DAG

Three-Address Statements

- Assignment statements: *x* **:=** *y op z*, *x* **:=** *op y*
- Indexed assignments: *x* **:=** *y*[*i*], *x*[*i*] **:=** *y*
- Pointer assignments: *x* **:= &***y*, *x* **:= ****y*, ******x* **:=** *y*
- Copy statements: *x* **:=** *y*
- Unconditional jumps: **goto** *lab*
- Conditional jumps: **if** *x relop y* **goto** *lab*
- Function calls: **param** *x…* **call** *p, n*
  return *y*

Syntax-Directed Translation into Three-Address Code

Productions	Synthesized attributes:	
$S \rightarrow$ **id** **:=** E	S.code	three-address code for S
$\quad$ | **while** E **do** S	S.begin	label to start of S or nil
$E \rightarrow E + E$	S.after	label to end of S or nil
$\quad$ | E * E	E.code	three-address code for E
$\quad$ | **-** E	E.place	a name holding the value of E
$\quad$ | **(** E **)**		
$\quad$ | **id**		
$\quad$ | **num**		

$$gen(E.\mathrm{place}\ \text{':='}\ E_1.\mathrm{place}\ \text{'+'}\ E_2.\mathrm{place})$$

Code generation

```
t3 := t1 + t2
```

Syntax-Directed Translation into Three-Address Code

Productions	Semantic rules
$S \rightarrow$ **id** $:=$ E	S.code := E.code || *gen*(**id**.place ':=' E.place); S.begin := S.after := nil
$S \rightarrow$ **while** E **do** S_1	*(see next slide)*
$E \rightarrow E_1 + E_2$	E.place := *newtemp*(); E.code := E_1.code || E_2.code || *gen*(E.place ':=' E_1.place '+' E_2.place)
$E \rightarrow E_1 * E_2$	E.place := *newtemp*(); E.code := E_1.code || E_2.code || *gen*(E.place ':=' E_1.place '*' E_2.place)
$E \rightarrow$ **-** E_1	E.place := *newtemp*(); E.code := E_1.code || *gen*(E.place ':=' 'uminus' E_1.place)
$E \rightarrow$ **(** E_1 **)**	E.place := E_1.place E.code := E_1.code
$E \rightarrow$ **id**	E.place := **id**.name E.code := ''
$E \rightarrow$ **num**	E.place := *newtemp*(); E.code := *gen*(E.place ':=' **num**.value)

Syntax-Directed Translation into Three-Address Code

Production

$S \rightarrow$ **while** E **do** S_1

Semantic rule

S.begin := *newlabel*()
S.after := *newlabel*()
S.code := *gen*(S.begin ':') ||
E.code ||
gen('if' E.place '=' '0' 'goto' S.after) ||
S_1.code ||
gen('goto' S.begin) ||
gen(S.after ':')

S.begin:	E.code
	`if` E.place `= 0` `goto` S.after
	S.code
	`goto` S.begin
S.after:	…

Example

i := 2 * n + k
while i do
i := i - k

```
     t1  := 2
     t2  := t1 * n
     t3  := t2 + k
     i   := t3
L1:  if i = 0 goto L2
     t4  := i - k
     i   := t4
     goto L1
L2:
```

Implementation of Three-Address Statements: Quads

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Quads (quadruples)

Pro: easy to rearrange code for global optimization
Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Triples

Pro: temporaries are implicit
Cons: difficult to rearrange code

Implementation of Three-Address Stmts: Indirect Triples

#	*Stmt*
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Triple container

Pro: temporaries are implicit & easier to rearrange code

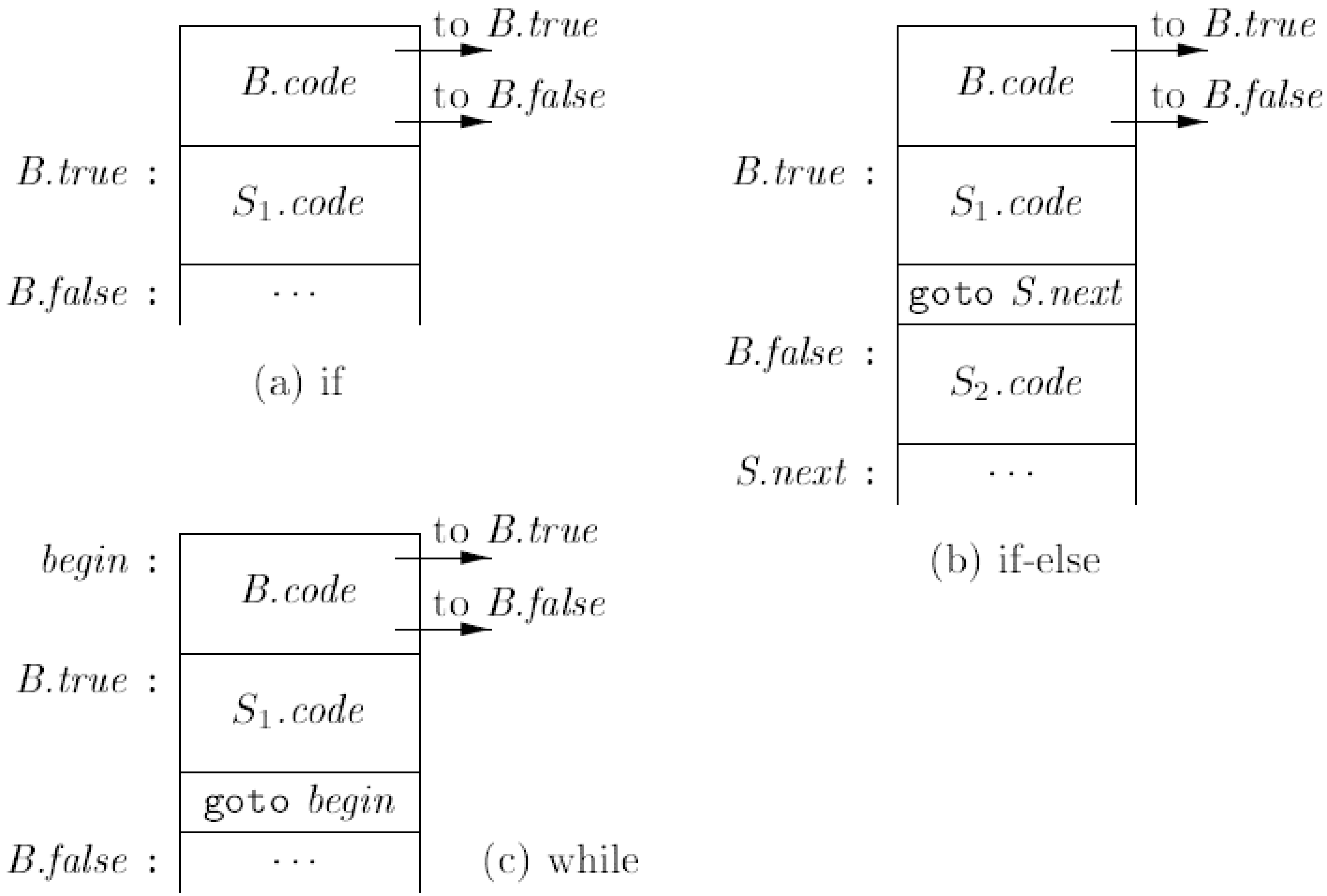
Example

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Flow-of-Control Statements

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if}\ (\ B\ )\ S_1 \\
S & \rightarrow & \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2 \\
S & \rightarrow & \textbf{while}\ (\ B\ )\ S_1
\end{array}
$$



(a) if

(b) if-else

(c) while

SDT for Boolean expressions

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow \textbf{if} (B) S_1 \textbf{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen(\texttt{'goto'}\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$

SDT for Boolean expressions

$S \rightarrow \textbf{while} \; (\; B \;) \; S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen(\texttt{'goto'}\ begin)$
$S \rightarrow S_1 \; S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

3AC for Boolean

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	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen($'if' $E_1.addr$ **rel**$.op$ $E_2.addr$ 'goto' $B.true)$ $\mid\mid gen($'goto' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'goto' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'goto' $B.false)$

Example

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

Example

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```