CSE 304
Compiler Design
Intermediate Code Generation II

TONY MIONE

Topics

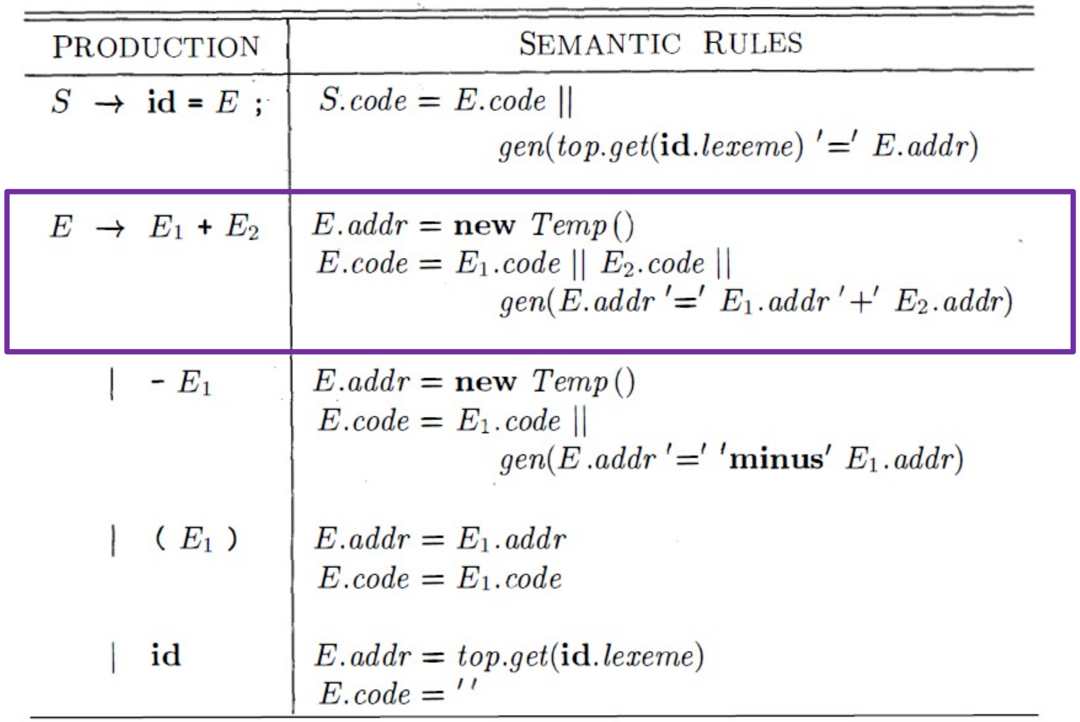
- Intermediate Code
  - Translating Expressions
  - Translating Array Elements and Array References
  - Control Flow
  - Boolean Expressions and Short circuiting
  - Avoiding Redundant gotos
  - Backpatching

Translating Expressions

- Expression evaluation can be coded by adding 2 attributes to non-terminals comprising expression:
  - .addr – Address of result
  - .code – Code to generate result

- Add primitive operations to help create intermediate code:
  - **gen()** – This generates an instruction (which is added to .code)
  - **newTemp()** – This creates a new temporary register for results
  - Concatenation operator - **||** - This is used to append .code attributes and other text for code generation

Translating Expressions



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

$E \rightarrow E_1 + E_2$

1. Generate a new Temporary
2. Append code for E_1 (generated for some subexpression)
3. Append code for E_2
4. Generate an add instruction
   a. result field is E.addr
   b. operands are E_1.addr (the temp created for E_1) and E_2.addr (temp created for E_2)

Incremental Translation

- The .code attributes in the previous translation scheme can get very long
  - It is possible to generate the code ‘on the fly’
  - Instead of having gen() write the instruction into a code attribute, write it…
    - directly to memory that is holding generated code or…
    - to a file

$$
\begin{array}{llll}
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,();\\
 & & & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}
\end{array}
$$

No code attributes needed since the instructions to compute the values in E_1.addr and E_2.addr have already been generated.

Translating Array References

- Array elements are stored consecutively in
  - Row-major order
    - Elements across a row are stored in order, then the order moves to the next row
    - Rightmost subscript changes quickest (like a car odometer)
  - Column-major order
    - Consecutive elements move down a column then continue at the top of the next column
    - Left most subscript changes the quickest
- Elements in most languages are numbered 0->n-1 where n is dimension size

Translating Array References

- To generate the location (l-value) of an element in a 1 dimensional array:
  - If *base* is the location of element 0
  - *w* is the width of an element in bytes
  - *i* is the element index
  - *Eff addr = base + i * w*
- Example: A is an array of integers (4 bytes each) and starts at memory location 0x800000. The location of element with index 5 is:
- 0x800000 + 5 * 4 = 0x800014

Translating Array References (2 or more dimensions)

- Given:
  - If *base* is the location of element 0
  - w_r is the width of a row in bytes
  - w_e is the width of an element in bytes
  - *i, j are the indices for row/element*
  - *Eff addr of A[i][j] = base +* $i * w_r + j * w_e$

- Example: A is a 3x5 array of integers at memory location 0x800000:
  - A[1][2] is at 0x800000 + 1 * 20 + 2 * 4 = 0x80001C

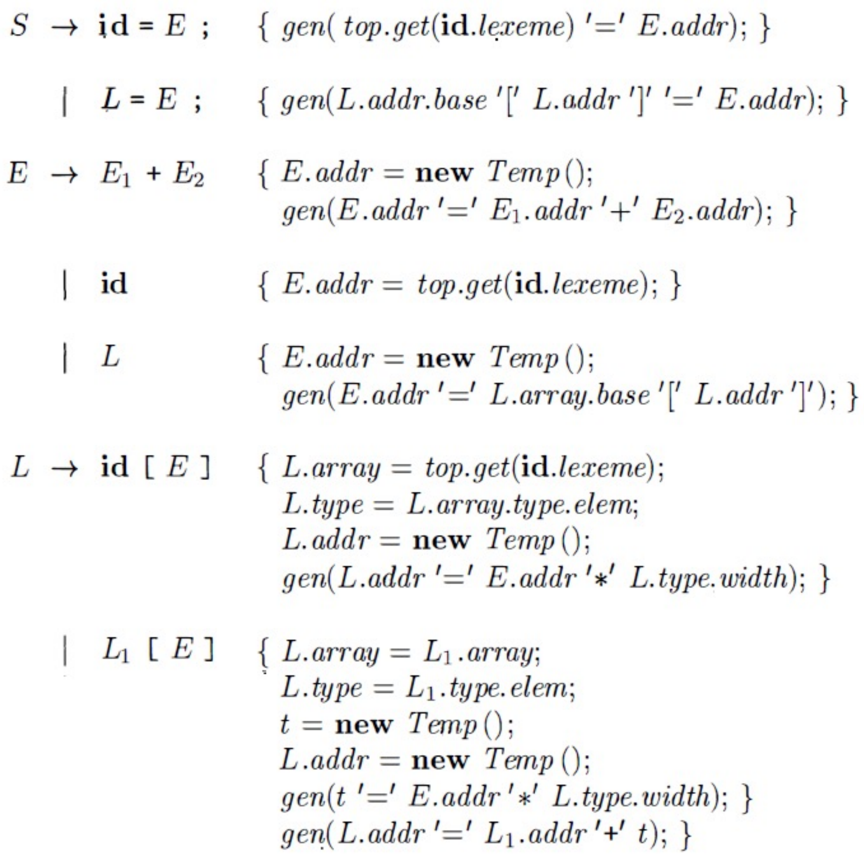
Translating Array References (2 or more dimensions)

- General formula for k-dimensional array:
  - *Eff addr of* $A[i_1][i_2]...[i_k] = base + i_l \; x \; w_l + i_2 * w_2 + . . . + i_k * w_k$

- Also, for k dimensions, can use element counts per row/column rather than width.
  - n_i is the number of elements in the row or plane
  - w is the width of the base element
  - base is the base address of the array

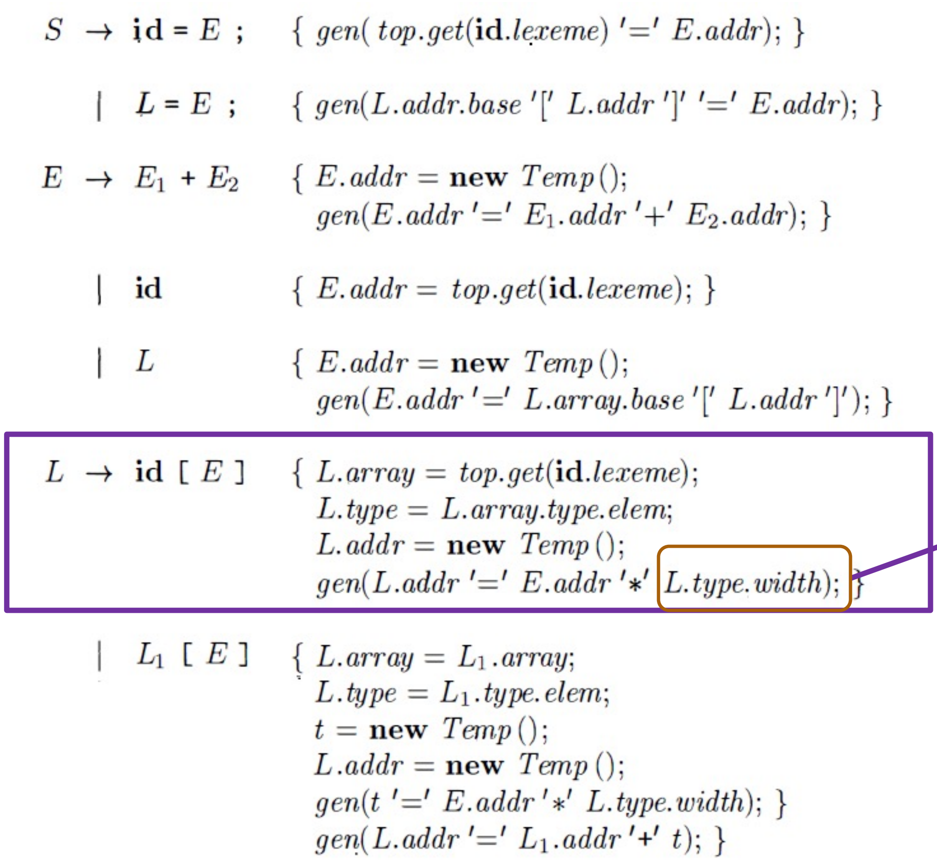
- *Eff addr of* $A[i_1][i_2]...[i_k] = base + ((...(i_1 * n_2 + i_2) * n_3 + i_3)...)*n_k + i_k) * w$

Semantic Actions for Array References



$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' *E.addr*); }
$\mid$	L = E ;	{ *gen*(*L.addr.base* '[' *L.addr* ']' '=' *E.addr*); }
$E \rightarrow$	E_1 + E_2	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* '=' E_1*.addr* '+' E_2*.addr*); }
$\mid$	**id**	{ *E.addr* = *top.get*(**id**.*lexeme*); }
$\mid$	L	{ *E.addr* = **new** *Temp*(); *gen*(*E.addr* '=' *L.array.base* '[' *L.addr* ']'); }
$L \rightarrow$	**id** [E]	{ *L.array* = *top.get*(**id**.*lexeme*); *L.type* = *L.array.type.elem*; *L.addr* = **new** *Temp*(); *gen*(*L.addr* '=' *E.addr* '*' *L.type.width*); }
$\mid$	L_1 [E]	{ *L.array* = L_1*.array*; *L.type* = L_1*.type.elem*; *t* = **new** *Temp*(); *L.addr* = **new** *Temp*(); *gen*(*t* '=' *E.addr* '*' *L.type.width*); } *gen*(*L.addr* '=' L_1*.addr* '+' *t*); }

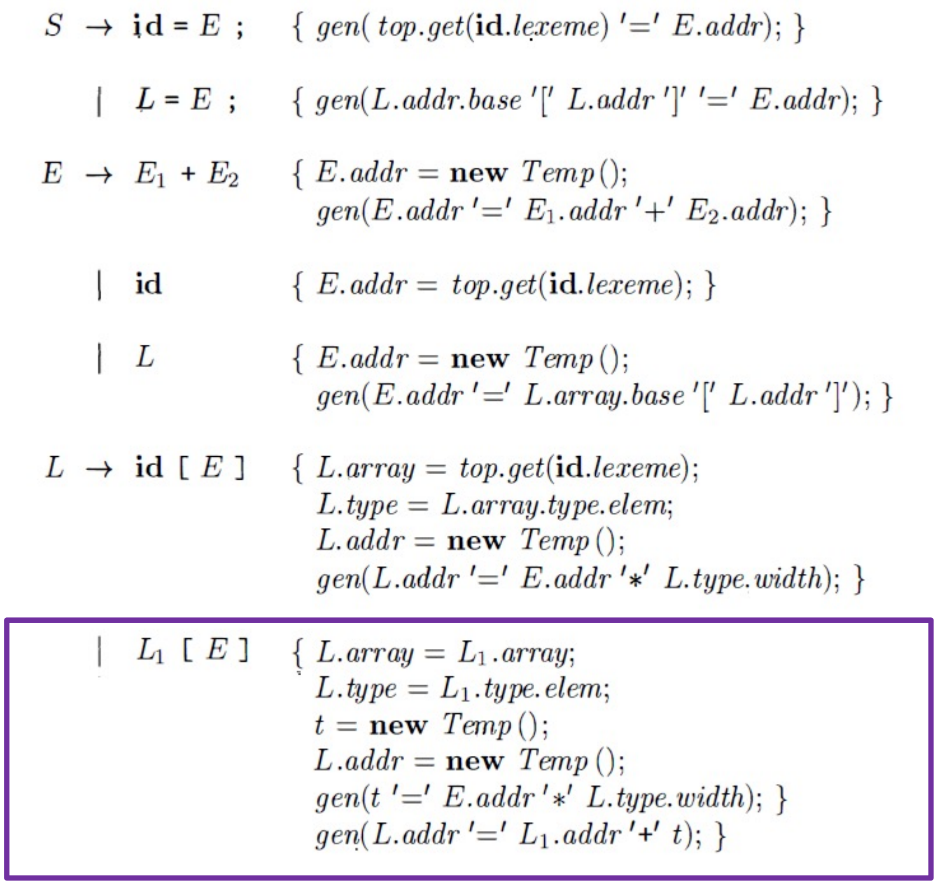
Semantic Actions for Array References



$$
\begin{array}{llll}
S & \rightarrow & \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ \text{'='}\ E.addr);\ \} \\
 & | & L = E\ ; & \{\ gen(L.addr.base\ \text{'['}\ L.addr\ \text{']'}\ \text{'='}\ E.addr);\ \} \\
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ \text{'='}\ E_1.addr\ \text{'+'}\ E_2.addr);\ \} \\
 & | & \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \} \\
 & | & L & \{\ E.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ \text{'='}\ L.array.base\ \text{'['}\ L.addr\ \text{']'});\ \} \\
L & \rightarrow & \mathbf{id}\ [\ E\ ] & \{\ L.array = top.get(\mathbf{id}.lexeme); \\
 & & & \quad L.type = L.array.type.elem; \\
 & & & \quad L.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(L.addr\ \text{'='}\ E.addr\ \text{'*'}\ \boxed{L.type.width);}\ \} \\
 & | & L_1\ [\ E\ ] & \{\ L.array = L_1.array; \\
 & & & \quad L.type = L_1.type.elem; \\
 & & & \quad t = \mathbf{new}\ Temp\,(); \\
 & & & \quad L.addr = \mathbf{new}\ Temp\,(); \\
 & & & \quad gen(t\ \text{'='}\ E.addr\ \text{'*'}\ L.type.width);\ \} \\
 & & & \quad gen(L.addr\ \text{'='}\ L_1.addr\ \text{'+'}\ t);\ \}
\end{array}
$$

1. Get symbol info into L_1.array
2. Get type into L_1.type.elem
3. L.addr is a new temporary
4. Compute address into L.addr
   (note: L.type.width is size of whole row, plane, etc)

Semantic Actions for Array References



$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
|	L = E ;	{ *gen*(L.*addr.base* '[' L.*addr* ']' '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); }
|	L	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' L.*array.base* '[' L.*addr* ']'); }
$L \rightarrow$	**id** [E]	{ L.*array* = *top.get*(**id**.*lexeme*); L.*type* = L.*array.type.elem*; L.*addr* = **new** *Temp* (); *gen*(L.*addr* '=' E.*addr* '*' L.*type.width*); }
|	L_1 [E]	{ L.*array* = L_1.*array*; L.*type* = L_1.*type.elem*; t = **new** *Temp* (); L.*addr* = **new** *Temp* (); *gen*(t '=' E.*addr* '*' L.*type.width*); } *gen*(L.*addr* '=' L_1.*addr* '+' t); }

1. Copy L_1.array to L.array
2. Get type of subarray (from L_1.type.elem)
3. Generate a new temp (t)
4. Generate a new temp (L.addr)
5. Create code to calculate t and add it to L.addr (the offset from the base)

Control Flow

- Boolean expressions can be used
  - To compute a logical value (true/false)
  - To alter control flow
- Here we are concerned with the latter
- Consider the following grammar:

B-> B||B | B&&B | !B | (B) | E rel E | true | false

- rel is one of the relational operators (<, <=, ==, !=, >, >=)
- || is logical OR, && is logical AND, ! Is NOT

Control Flow

- Depending on language semantics, Boolean expressions may NOT need to be completely evaluated
- Most languages allow ‘short circuit’ evaluation (quit once you know the result
  - B_1 || B_2 – If B_1 is true, we know the whole expression is true so skip B_2 eval
  - B_1 && B_2 – If B_1 is false, we know expression is false so skip B_2 eval

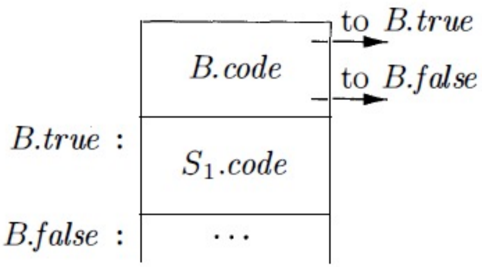
Flow of Control Statements

Here is a small grammar of Flow of Control Statements

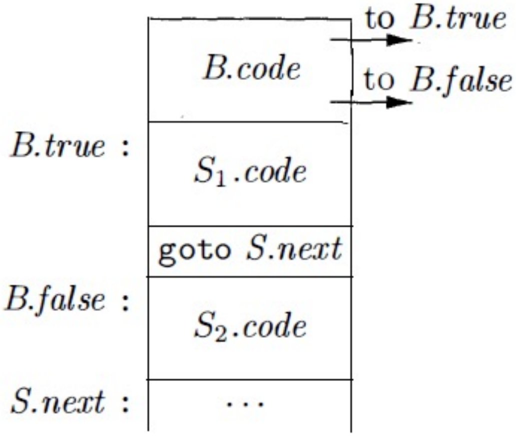
$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if}\ (\ B\ )\ S_1 \\
S & \rightarrow & \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2 \\
S & \rightarrow & \textbf{while}\ (\ B\ )\ S_1
\end{array}
$$

B represents a Boolean expression
S represents statements in the language

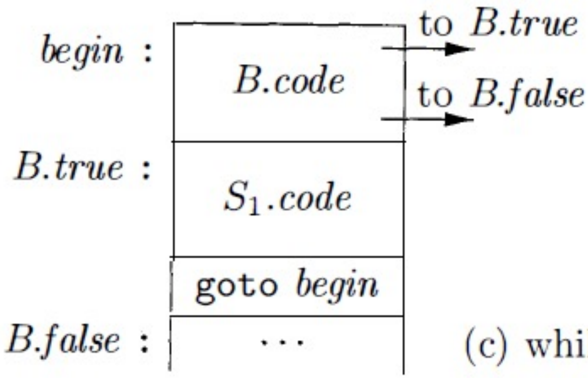
Need to create semantic actions that generate the following patterns of code:



(a) if

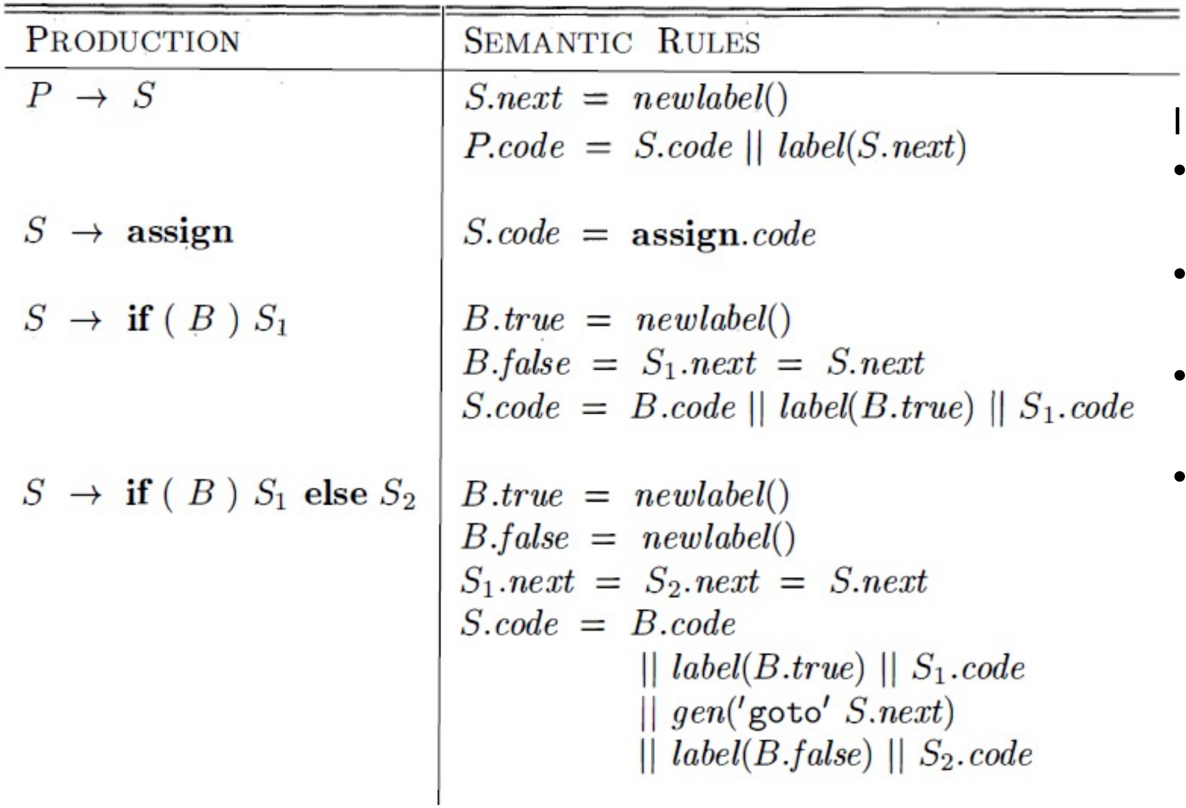


(b) if-else



(c) while

Flow of Control Statements

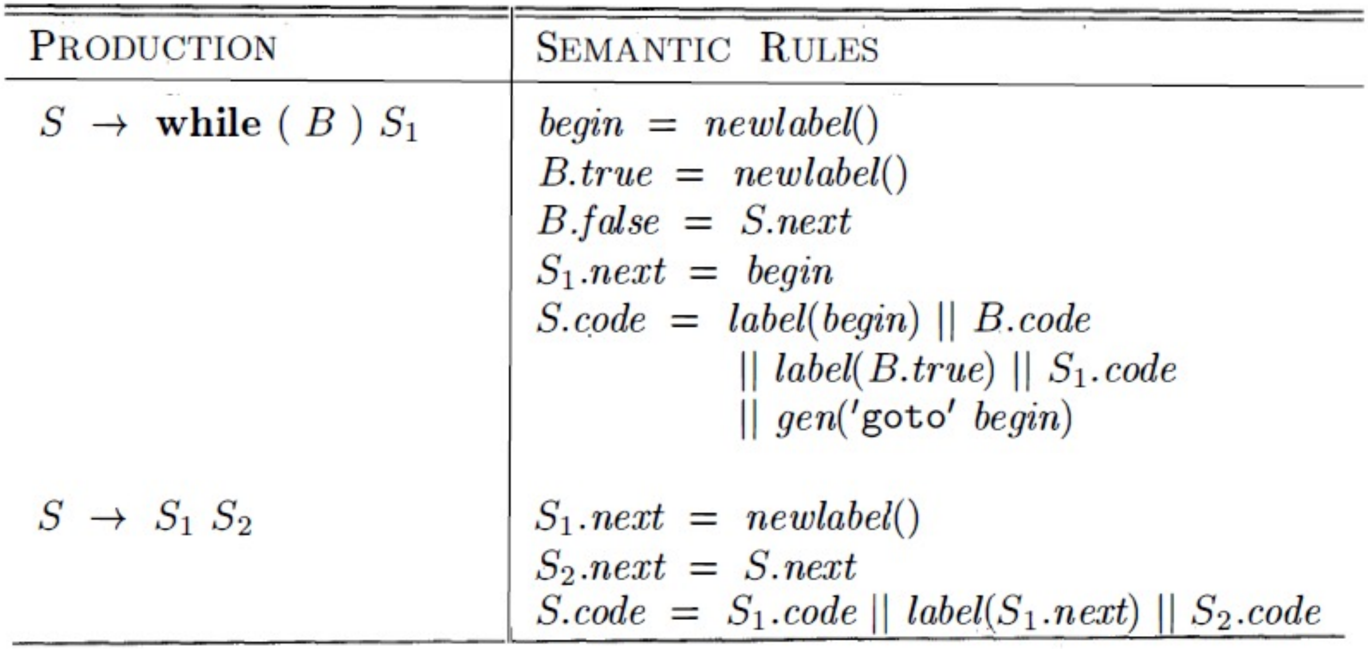


PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mathbin{\|\|} label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} \; (\; B \;) \; S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mathbin{\|\|}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

In this and following slides:

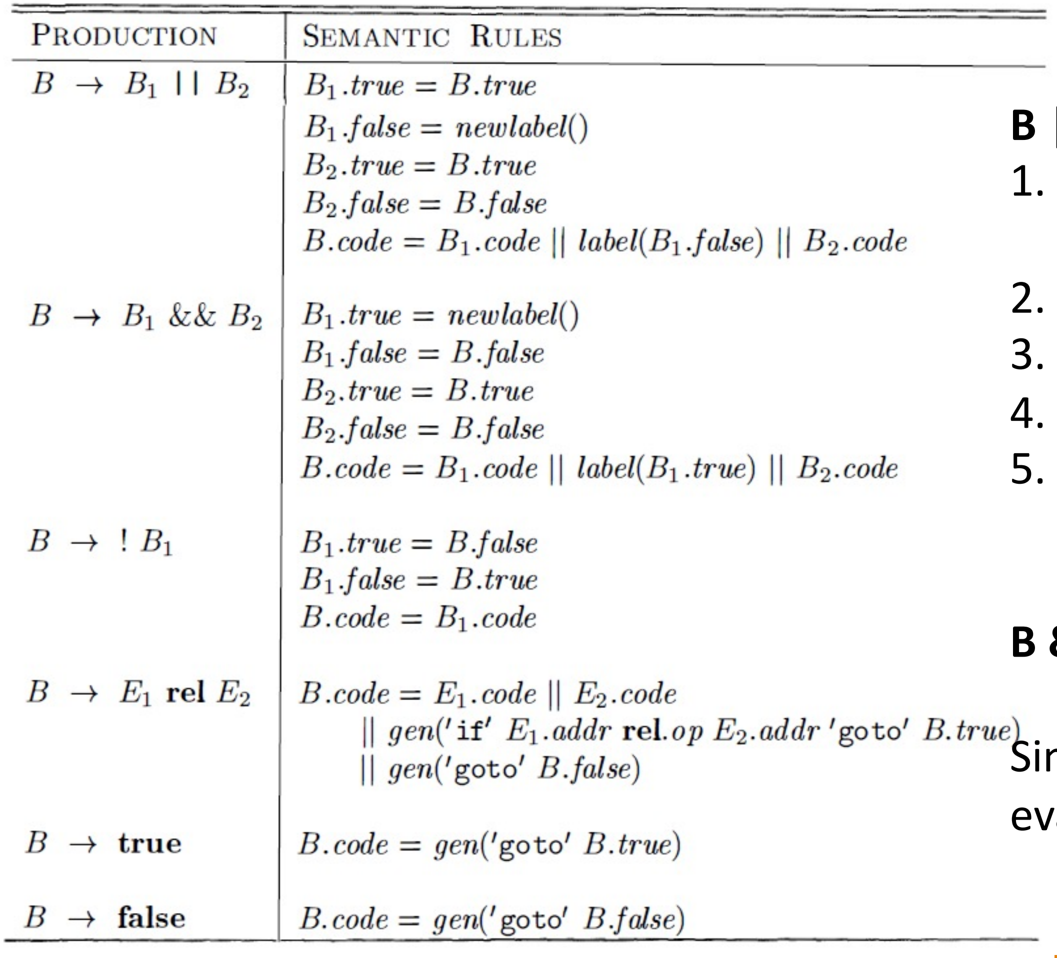
- B.true, B.false, S.next, S_1.next, etc are labels for branch transfers
- **B.true** – Branch here when B is true
- **B.false** – Branch here when B is false
- **S.next, S_1.next** – This is the target of the next statement after S, S1, etc

Flow of Control Statements



PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{while} \; (\; B \;) \; S_1$	$begin \; = \; newlabel()$ $B.true \; = \; newlabel()$ $B.false \; = \; S.next$ $S_1.next \; = \; begin$ $S.code \; = \; label(begin) \; \| \; B.code$ $\| \; label(B.true) \; \| \; S_1.code$ $\| \; gen(\text{'goto'} \; begin)$
$S \rightarrow S_1 \; S_2$	$S_1.next \; = \; newlabel()$ $S_2.next \; = \; S.next$ $S.code \; = \; S_1.code \; \| \; label(S_1.next) \; \| \; S_2.code$

Boolean Expressions



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

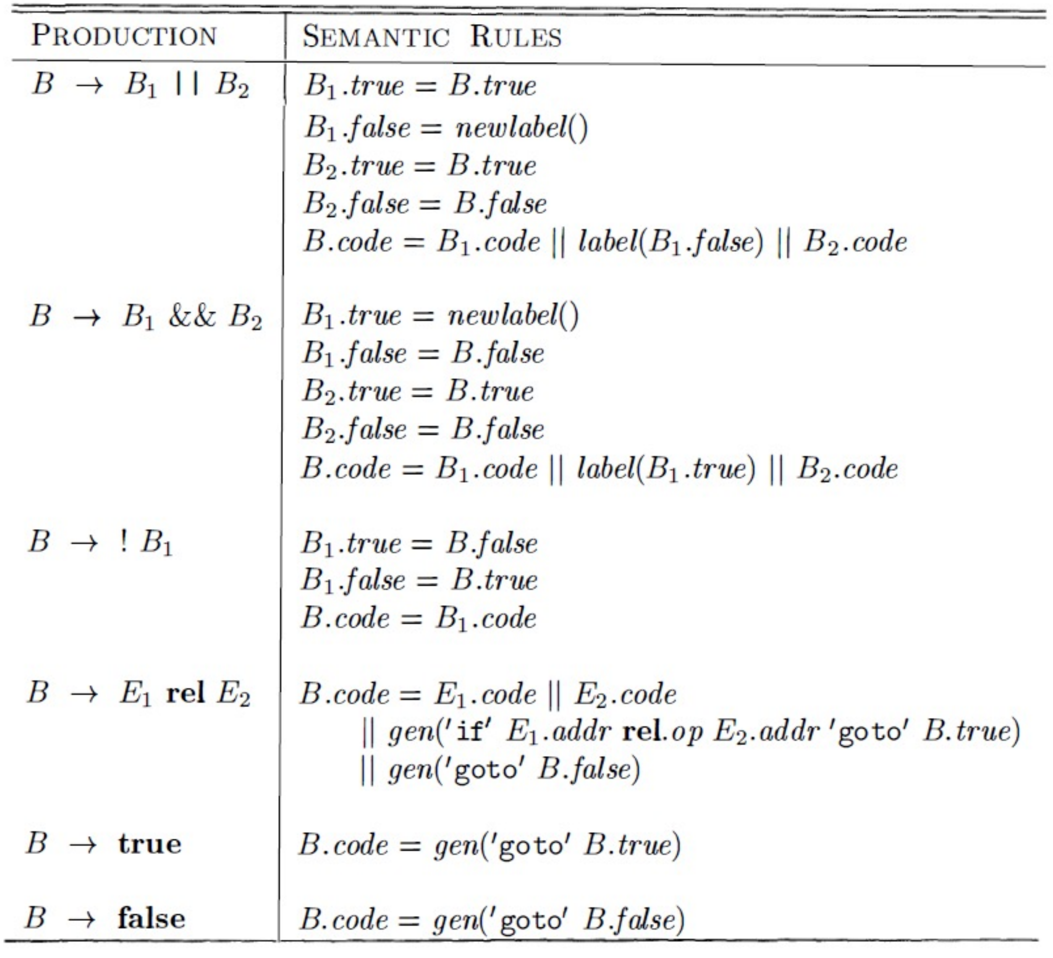
B || B

1. Target true result of B_1 to overall result (short circuit)
2. False result needs a label
3. B_2.true goes to overall true result target
4. B_2.false goes to false target of B
5. Code is B_1 eval code, the label B1.false, and the B_2 eval code

B && B

Similar to B || B but reverse false and true evaluation targets

Boolean Expressions



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

!B

Just reverse true and false labels from B_1!

Avoiding Redundant Gotos

- Can reduce gotos by clever reorganization of tests and control transfers
- Consider: S -> if (B) then S_1 => Actions are:

B.true = newlabel()
B.false = S_1.next = S.next
B.code II label(B.true) II S_l.code

Now, use a new operator ***fall*** meaning *do not generate a goto*

S-> if (B) then S_1 => actions are now:
B.true = fall
B.false = S_1.next = S.next
B.code II S_1.code

Avoiding Redundant Gotos

- Can reduce gotos by clever reorganization of tests and control transfers
- Consider: S -> if (B) then S_1 else S_2 => actions are:
  B.true = newlabel()
  B.false = newlabel()
  S_1.next = S_2.next = S.next
  S.code = B.code II label(B. true) II S_1.code II gen('goto‘ S.next) II label(B.false) II S_2.code

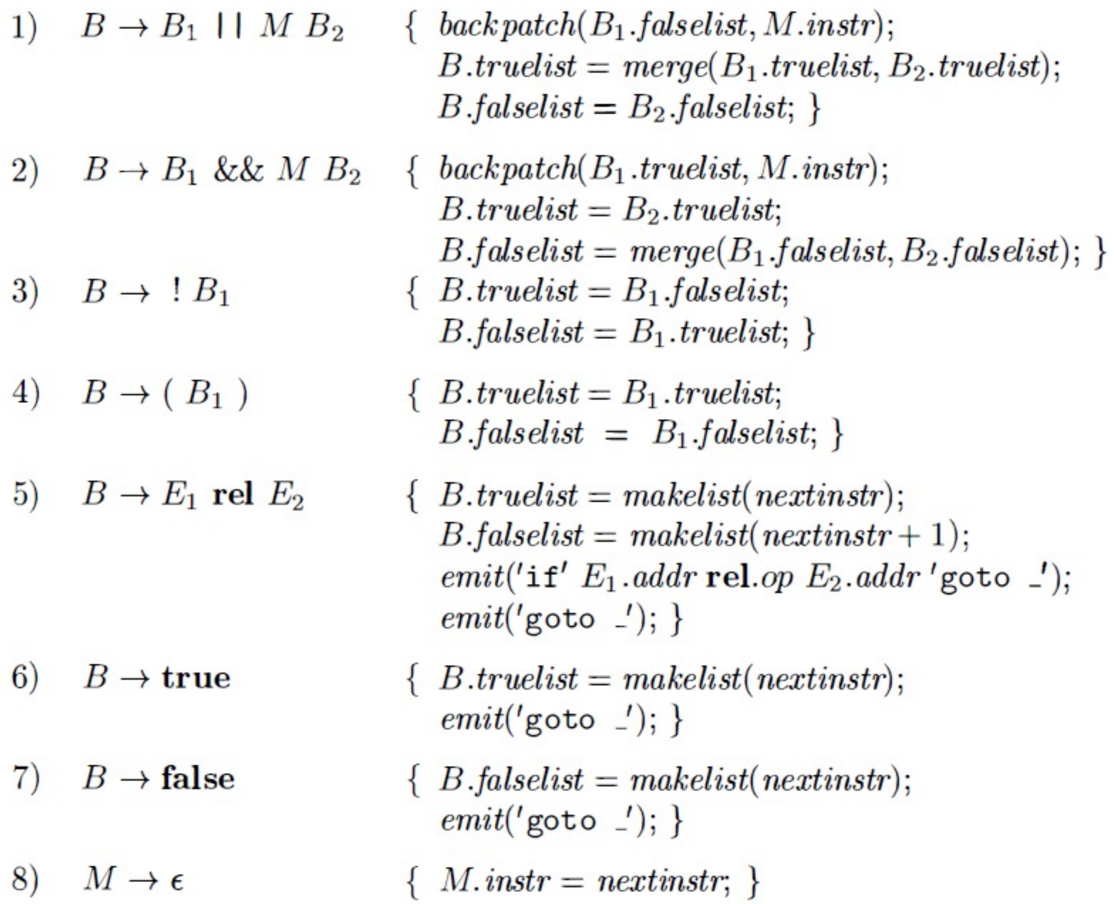
  Now, use a new operator ***fall*** meaning *do not generate a goto*

  S-> if (B) then S_1 else S_2 => actions are now:
  B.true = fall
  B.false = newlabel()
  S.next = newlabel()
  S.code = B.code II S_1.code II gen('goto‘ S.next) II label(B.false) II S_2.code

Backpatching

- Generating jump instructions on the fly may require a second pass to determine the address of labels generated.
- **Backpatching** allows 1 pass translation by keeping lists of jump targets created as synthesized attributes
- Need 3 lists:
  - **B.truelist** – instructions that need a target when B is *true*
  - **B.falselist** – instructions that need a target when B is *false*
  - **S.nextlist** – instructions that need to jump to the instruction after the code in S.
- 3 functions are used:
  - **makelist(i)** – Creates a new list containing only i, an index into the instruction list
  - **merge(p1,p2)** – merges two lists and returns the merged lists
  - **backpatch(p, i)** – Patches (fixes) the targets of all conditional and unconditional jumps in the locations found in list p to point at instruction i.

Backpatching – Boolean Expressions



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

Marker non terminal picks up location of next instruction generated (see production 8)

Backpatching done during compound expression evaluation

Lists are either reversed or copied in productions 3 and 4

makelist() is needed for productions 5, 6, and 7

Backpatching – Boolean Expressions

Consider

$$
\begin{array}{lll}
1) & B \rightarrow B_1 \;||\; M\; B_2 & \{\; \mathit{backpatch}(B_1.\mathit{falselist}, M.\mathit{instr}); \\
 & & \phantom{\{\;} B.\mathit{truelist} = \mathit{merge}(B_1.\mathit{truelist}, B_2.\mathit{truelist}); \\
 & & \phantom{\{\;} B.\mathit{falselist} = B_2.\mathit{falselist};\; \}
\end{array}
$$

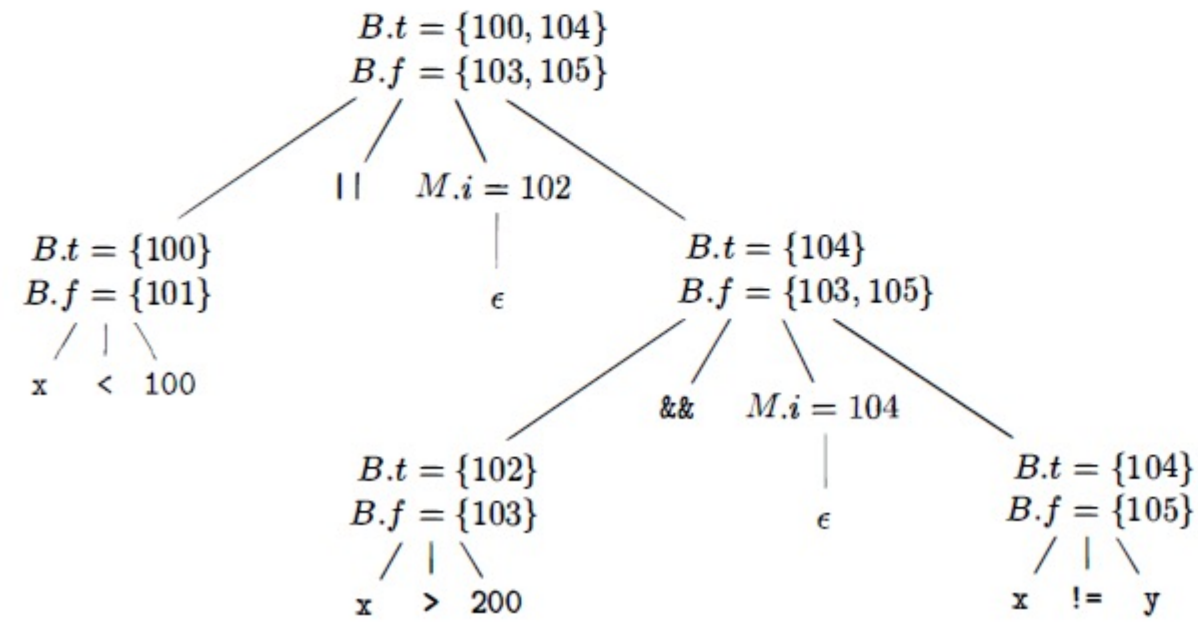
If B_1 is true, control can jump past the test in B_2

However, if it is false, it must jump to the test in B_2 in order to test the complete conditional.

- So the backpatch() operation causes all jumps on the false list to point at M (M.instr) which will be the start of the code in B_2
- Meanwhile, the truelist is set to the combination of truelists for B_1 and B_2 since both of those mean the overall expression is true.
- Finally, B’s synthesized falselist should be wherever the falselist for B_2 points.

Backpatching Example

Parse tree for: x < 100 I I x > 200 && x ! = Y



Generates:

100: if x < 100 goto –

101: goto –

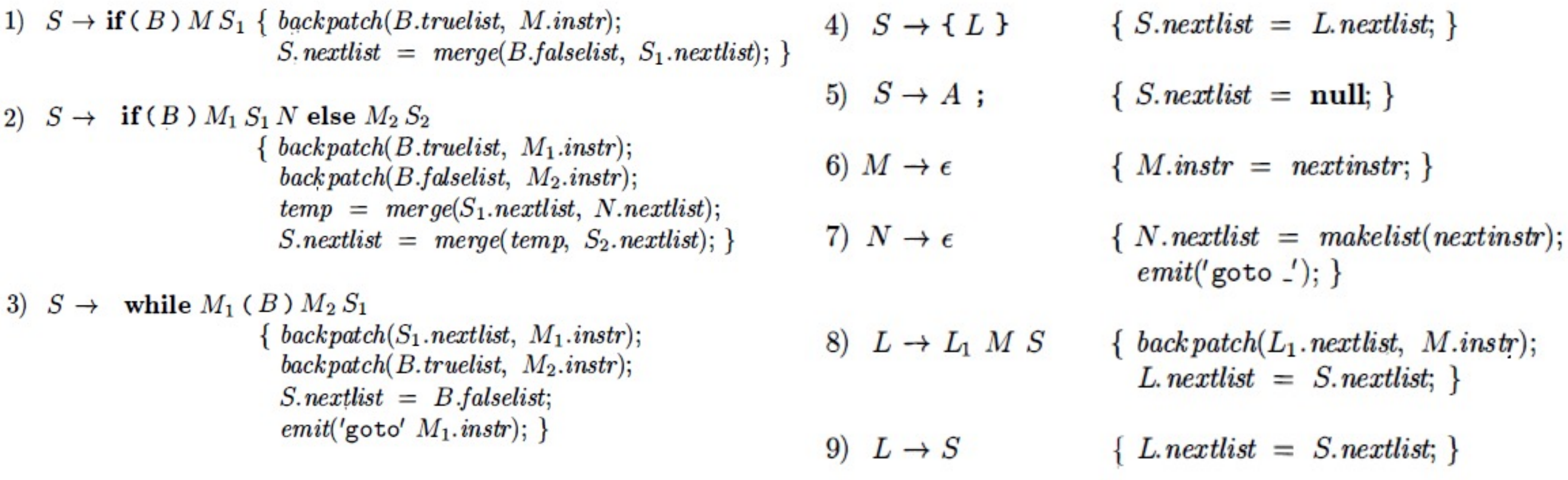
102: if x > 200 goto –

103: goto –

104: if x != y goto –

105: goto –

Backpatching Flow of Control



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) $S \rightarrow \textbf{while}\; M_1\,(\,B\,)\; M_2\; S_1$	$\{\; backpatch(S_1.nextlist,\ M_1.instr);$ $backpatch(B.truelist,\ M_2.instr);$ $S.nextlist\ =\ B.falselist;$ $emit(\text{'}\texttt{goto}\text{'}\ M_1.instr);\ \}$
4) $S \rightarrow \{\ L\ \}$	$\{\; S.nextlist\ =\ L.nextlist;\ \}$
5) $S \rightarrow A\ ;$	$\{\; S.nextlist\ =\ \textbf{null};\ \}$
6) $M \rightarrow \epsilon$	$\{\; M.instr\ =\ nextinstr;\ \}$
7) $N \rightarrow \epsilon$	$\{\; N.nextlist\ =\ makelist(nextinstr);$ $emit(\text{'}\texttt{goto \_}\text{'});\ \}$
8) $L \rightarrow L_1\ M\ S$	$\{\; backpatch(L_1.nextlist,\ M.instr);$ $L.nextlist\ =\ S.nextlist;\ \}$
9) $L \rightarrow S$	$\{\; L.nextlist\ =\ S.nextlist;\ \}$

Questions?