CS323 Lab 12

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Translation of Expressions (with array accesses)
- Control Flow
- Backpatching (self-study materials)
- Symbol Table Management
- Scope Checking

Dealing with Arrays (Lab)

- An expression involve array accesses: $c + a[i][j]$
- An array reference $A[i][j]$ will expand into a sequence of three-address instructions that calculate an address for the reference

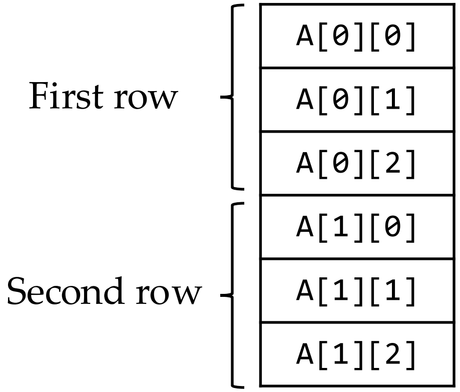
```
c + a[i][j]
```

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
```

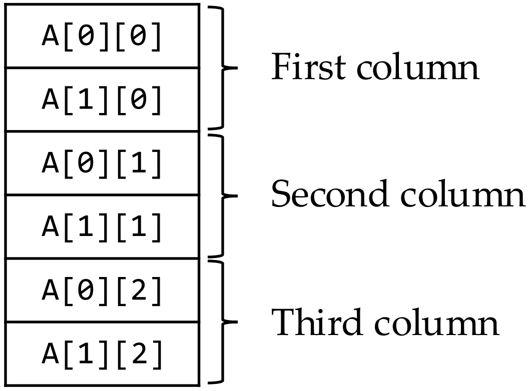
calculate address

Addressing Array Elements

- Array elements can be accessed quickly if they are stored consecutively
- For an array A with n elements, the relative address of $A[i]$ is:
  - $\boldsymbol{base + i * w}$ ($base$ is the relative address of $A[0]$, w is the width of an element)
- For a 2D array A (row-major layout), the relative address of $A[i_1][i_2]$ is:
  - $\boldsymbol{base + i_1 * w_1 + i_2 * w_2}$ (w_1 is the width of a row, w_2 is the width of an element)



Row-major (C)



Column-major (Fortran)

Addressing Array Elements

- Array elements can be accessed quickly if they are stored consecutively
- For an array A with n elements, the relative address of $A[i]$ is:
  - $\boldsymbol{base + i * w}$ ($base$ is the relative address of $A[0]$, w is the width of an element)
- For a 2D array A (row-major layout), the relative address of $A[i_1][i_2]$ is:
  - $\boldsymbol{base + i_1 * w_1 + i_2 * w_2}$ (w_1 is the width of a row, w_2 is the width of an element)
- Further generalize to k-dimensional array A (row-major layout), the relative address of $A[i_1][i_2] \ldots [i_k]$ is:
  - $\boldsymbol{base + i_1 * w_1 + i_2 * w_2 + \cdots + i_k * w_k}$ (w's can be generalized as above)

Translation of Array References

- The main problem in generating code for array references is to relate the address-calculation formula to the grammar
  - The relative address of $A[i_1][i_2] \ldots [i_k]$ is $base\ +\ i_1 * w_1 + i_2 * w_2 + \cdots + i_k * w_k$
  - Productions for generating array references: $L \rightarrow L\ [\ E\ ]\ |\ \mathbf{id}\ [\ E\ ]$

第一部分语义动作是在计算元素所在行之前所有行的元素所占的内存空间的总和，第二部分语义动作是在计算元素所在行的前序元素所占的内存空间。

SDT for Array References (1)

$$
\begin{array}{lll}
L \rightarrow \textbf{id}\ [\ E\ ] & \{ & L.array = top.get(\textbf{id}.lexeme); \\
 & & L.type = L.array.type.elem; \\
 & & L.addr = \textbf{new}\ Temp\,(); \\
 & & gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
\mid\ L_1\ [\ E\ ] & \{ & L.array = L_1.array; \\
 & & L.type = L_1.type.elem; \\
 & & t = \textbf{new}\ Temp\,(); \\
 & & L.addr = \textbf{new}\ Temp\,(); \\
 & & gen(t\ '='\ E.addr\ '*'\ L.type.width); \\
 & & gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

Compute offset

A is a 2*3 array of integers
Translate A[i][j]

L.type is the type of A's element: array(3, int)

A[i]
Reduce using prod. #1
L

L[j]
Reduce using prod. #2
L

L.type is the type of A[i]'s element:

int

L.array: a pointer to the symbol-table entry for the array name

L.array.base: the base address of the array

L.addr: a temporary for computing the offset for the array reference

L.type: the type of the subarray generated by *L*

t.elem: for any array type *t*, *t.elem* gives the element type

SDT for Array References (2)

- The semantic actions of L-productions compute offsets
- The address of an array element is $base + offset$

$$
\begin{array}{lll}
E & \rightarrow E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,();\\
& & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}\\
& \mid\ \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}\\
& \mid\ L & \{\ E.addr = \mathbf{new}\ Temp\,();\\
& & \quad gen(\underline{E.addr\ '='\ L.array.base\ '['\ L.addr\ ']'});\ \}
\end{array}
$$

Instruction of the form $x = a[i]$

Array references can be part of an expression

SDT for Array References (3)

$$
\begin{array}{lll}
S & \rightarrow \ \mathbf{id} = E \ ; & \{ \ gen(\ top.get(\mathbf{id}.lexeme) \ '=' \ E.addr); \ \} \\
& | \quad L = E \ ; & \{ \ gen(L.addr.base \ '[' \ L.addr \ ']' \ '=' \ E.addr); \ \}
\end{array}
$$

Instruction of form $a[i] = x$

Array references can appear at the LHS of an assignment statement

$E \rightarrow E_1 + E_2$ { $E.addr$ = **new** $Temp\,()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; } 6

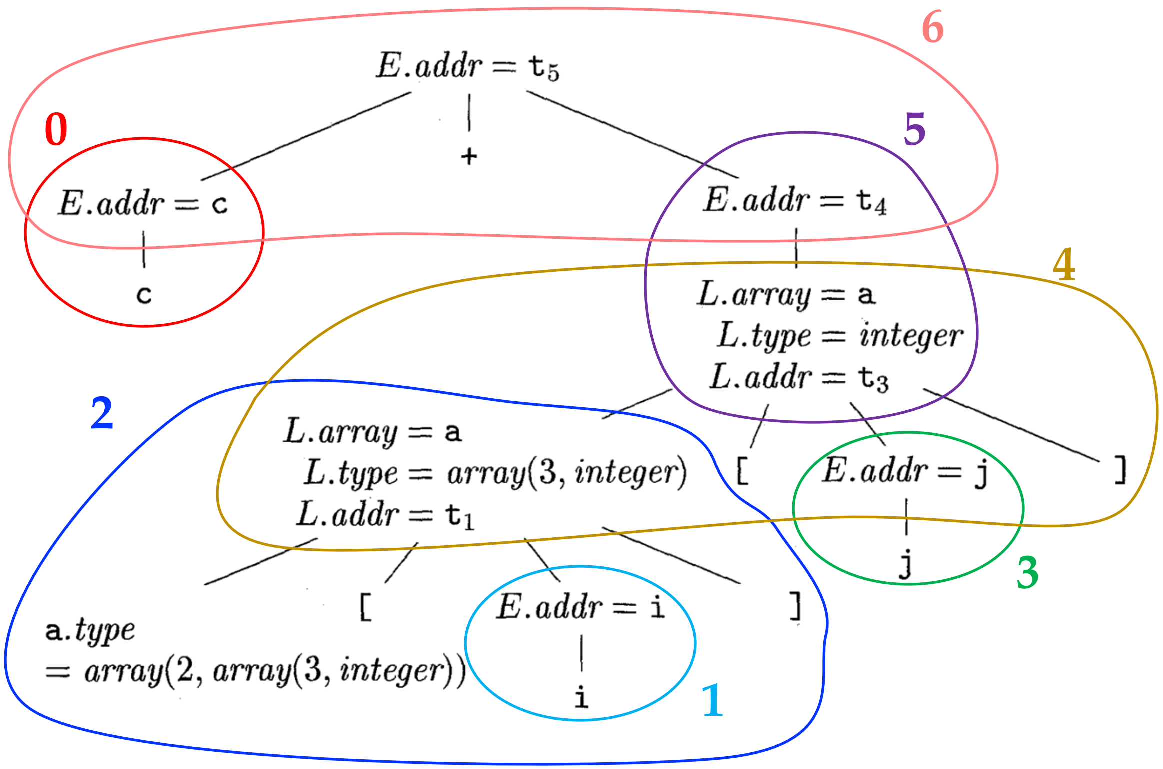
| **id** { $E.addr = top.get(\mathbf{id}.lexeme)$; } 0 1 3

| L { $E.addr$ = **new** $Temp\,()$; $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; } 5

$L \rightarrow$ **id** [E] { $L.array = top.get(\mathbf{id}.lexeme)$; $L.type = L.array.type.elem$; $L.addr$ = **new** $Temp\,()$; $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; } 2

| L_1 [E] { $L.array = L_1.array$; $L.type = L_1.type.elem$; t = **new** $Temp\,()$; $L.addr$ = **new** $Temp\,()$; $gen(t\ '='\ E.addr\ '*'\ L.type.width)$; $gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; } 4

Translating `c + a[i][j]`



Generated code:

```
t1 = i * 12     2
t2 = j * 4      4
t3 = t1 + t2
t4 = a[t3]      5
t5 = c + t4     6
```

Outline

- Translation of Expressions (with array accesses)
- Control Flow
- Backpatching (self-study materials)
- Symbol Table Management
- Scope Checking

Control Flow

- Boolean expressions are often used to alter the flow of control or compute logical values

- Grammar: $B \rightarrow B \parallel B \mid B \mathbin{\&\&} B \mid !\,B \mid (B) \mid E\ \textbf{rel}\ E \mid \textbf{true} \mid \textbf{false}$

- Given the expression $B_1 \parallel B_2$, if B_1 is true, then the expression is true without having to evaluate B_2 *.

If B_2 has side effect (e.g., changing the value of a global variable), then the effect may not occur

Short-Circuit Code Example

- In *short-circuit* code, the boolean operators &&, ||, ! translate into jumps. The operators do not appear in the code.

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

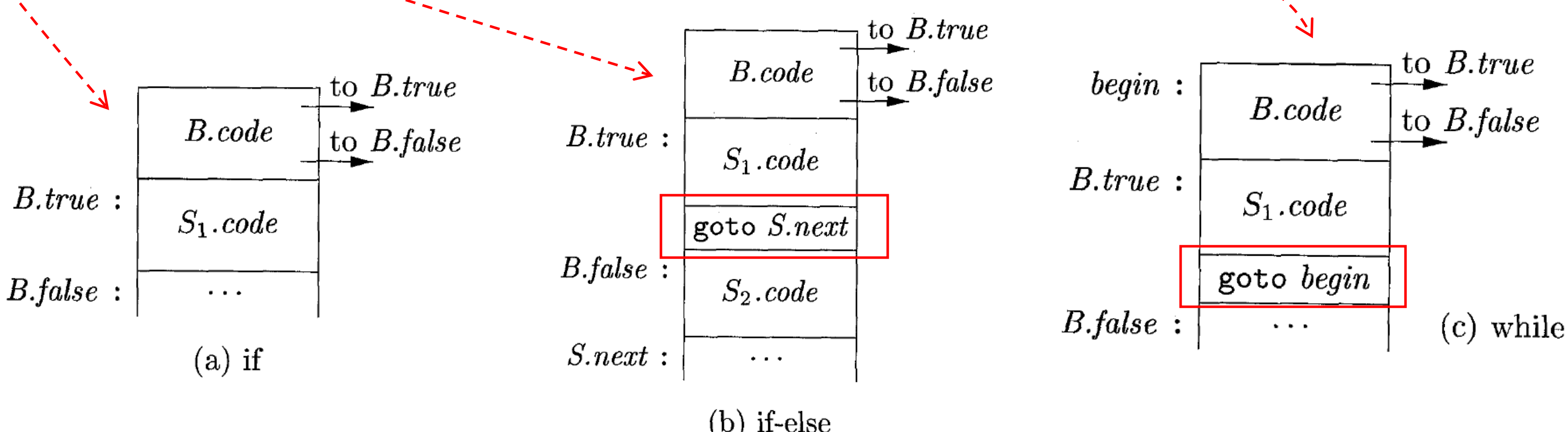
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Flow-of-Control Statements

- Grammar:
  - $S \rightarrow \textbf{if} \; (\; B \;) \; S_1$
  - $S \rightarrow \textbf{if} \; (\; B \;) \; S_1 \; \textbf{else} \; S_2$
  - $S \rightarrow \textbf{while} \; (\; B \;) \; S_1$

Inherited attributes:

- $B.true$: the label to which control flows if B is true
- $B.false$: the label to which control flows if B is false
- $S.next$: the label for the instruction immediately after the code for S



$S.next$ is not needed

$S.next$ is not needed

SDD for Flow-of-Control Statements (1)

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

Illustrated by previous figures

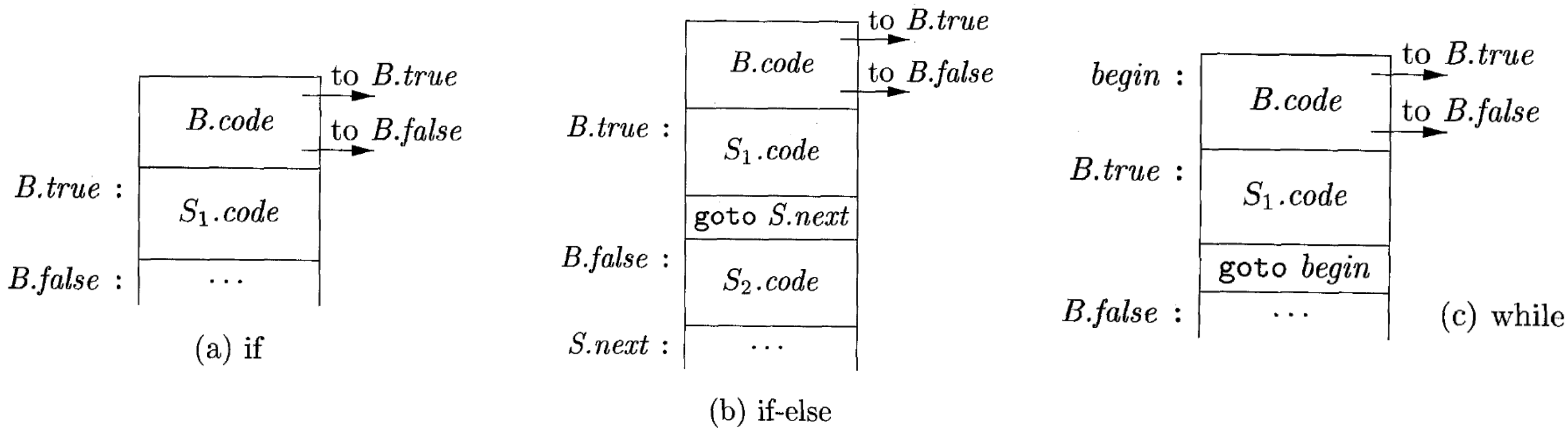
SDD for Flow-of-Control Statements (2)

Illustrated by previous figure

Production	Semantic Rules
$S \rightarrow \textbf{while} (B) S_1$	$\boxed{\begin{array}{l} begin = newlabel() \\ B.true = newlabel() \\ B.false = S.next \\ S_1.next = begin \\ S.code = label(begin) \parallel B.code \\ \quad \parallel label(B.true) \parallel S_1.code \\ \quad \parallel gen('\texttt{goto}'\ begin) \end{array}}$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

Translating Boolean Expressions in Flow-of-Control Statements

- A boolean expression B is translated into <u>three-address instructions that evaluate B using conditional and unconditional jumps</u> to one of two labels: $B.true$ and $B.false$
  - $B.true$ and $B.false$ are two inherited attributes. Their value depends on the context of B (e.g., if statement, if-$else$ statement, $while$ statement)



Generating Three-Address Code for Booleans (1)

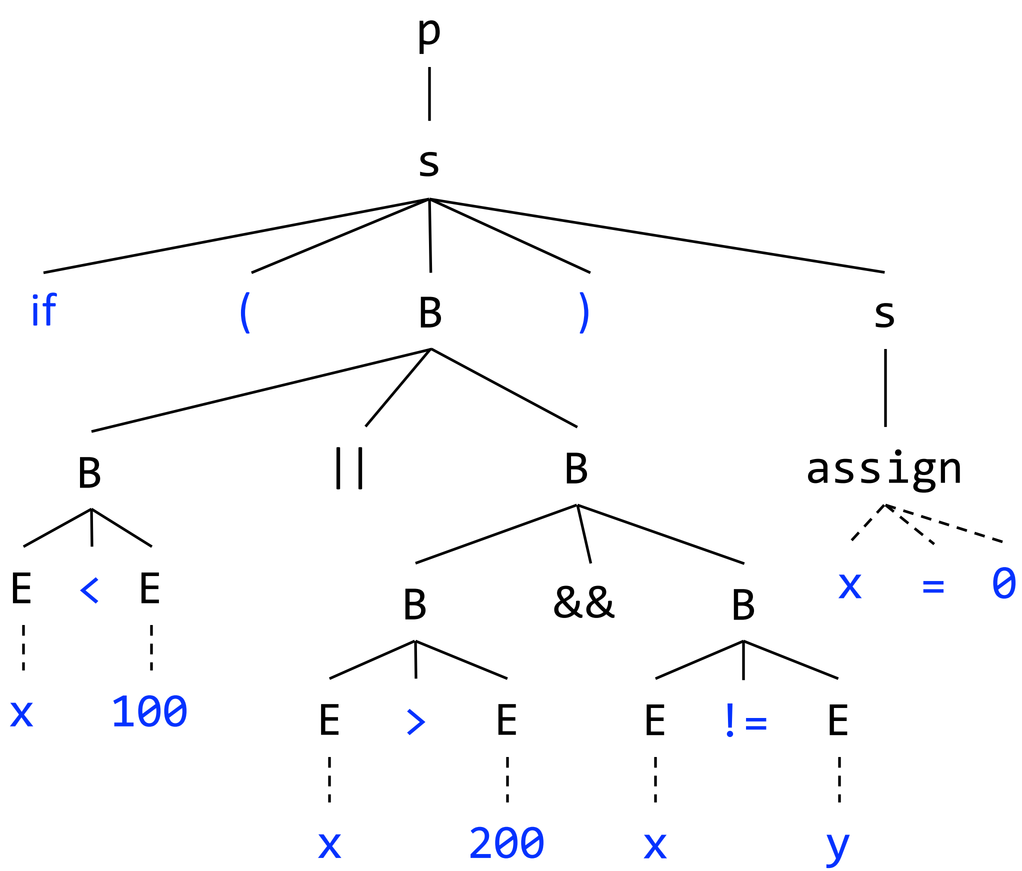
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow \textbf{false}$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

Generating Three-Address Code for Booleans (2)

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ // short-circuiting $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $\boxed{B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code}$
$B \rightarrow B_1 \; \&\& \; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ // short-circuiting $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \; ! \; B_1$	$B_1.true = B.false$ // targets reversed $B_1.false = B.true$ $\boxed{B.code = B_1.code}$

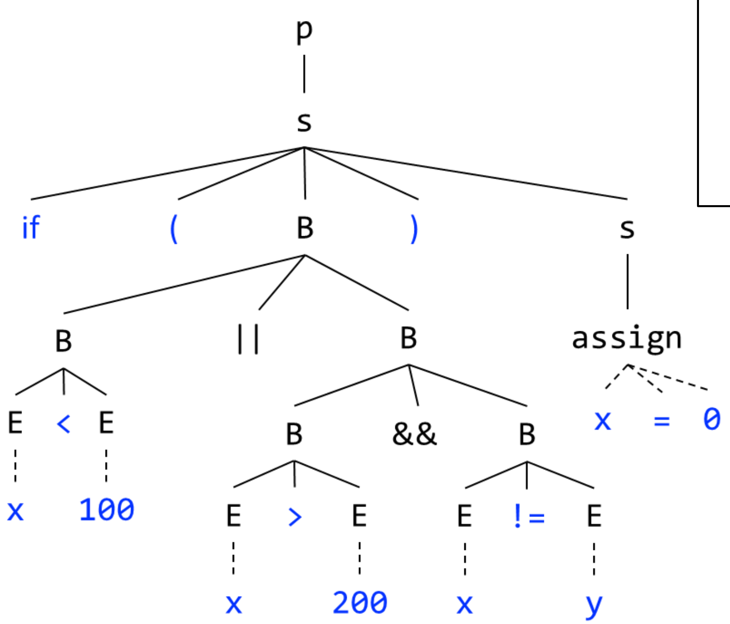
Example

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



Dashed lines mean that the reduction may consist of multiple steps

Example



PRODUCTION	SEMANTIC RULES
$P \to S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \to \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \to \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

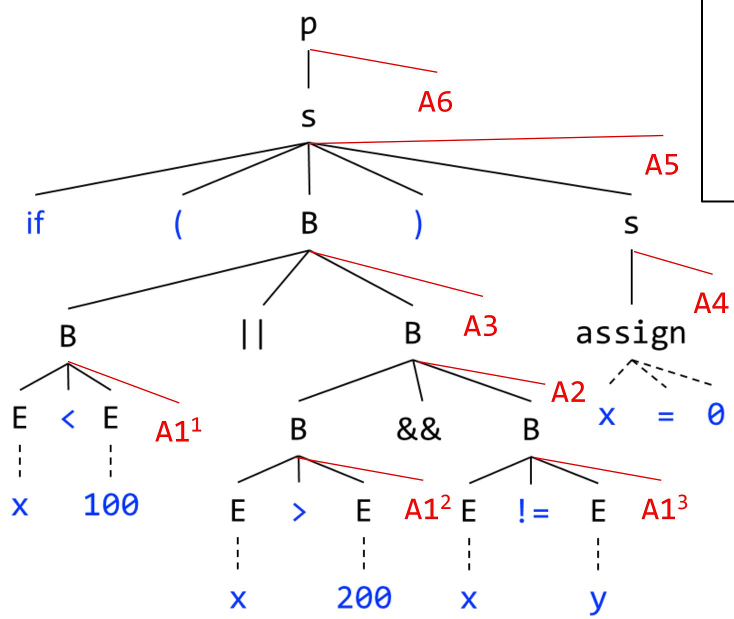
$B \to B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \to B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code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

$B \to E_1\ \textbf{rel}\ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$

This SDD is L-attributed, not S-attributed. The grammar is not LL. There is no way to implement the SDD directly during parsing.

Traversing the parse tree to evaluate the attributes helps generate the intermediate code

Example



Virtual nodes are in red color

Application order of actions (preorder traversal of the tree):

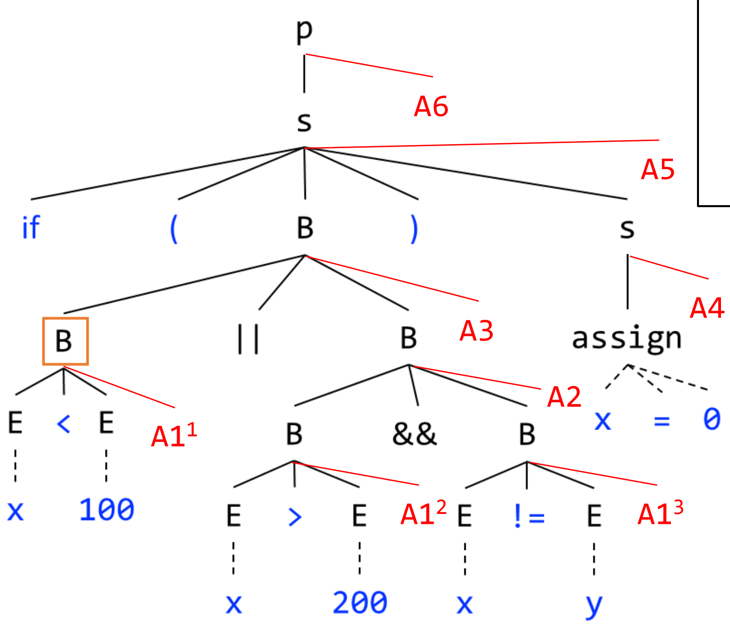
A1¹ A1² A1³ A2 A3 A4 A5 A6

PRODUCTION	SEMANTIC RULES	
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$	A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$	A4
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$	A5

$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$	A2

$B \rightarrow E_1\ \textbf{rel}\ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$	A1

Example



A1¹ A1² A1³ A2 A3 A4 A5 A6

Generated code:

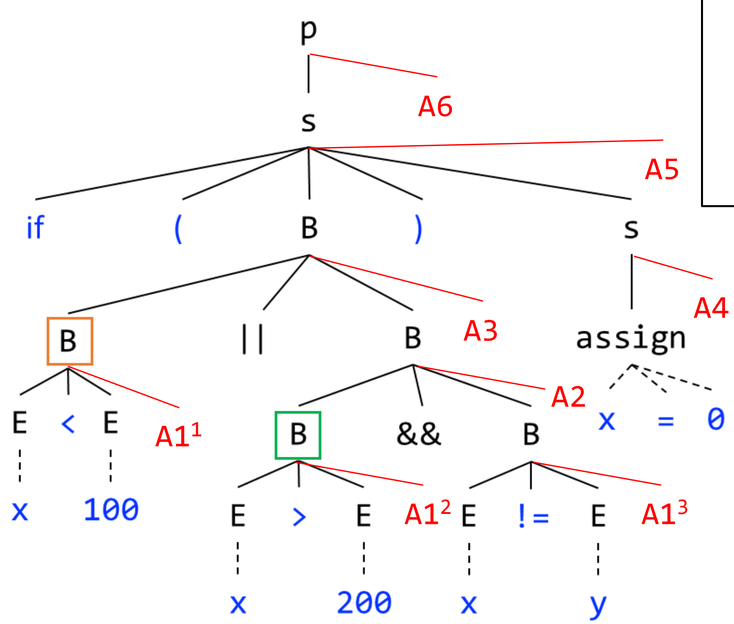
```
if x < 100 goto B.true
goto B.false
```

Production	Semantic Rules	
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$	A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$	A4
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$	A5

$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$	A3
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$	A2

$B \rightarrow E_1\ \textbf{rel}\ E_2$	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\|\; gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\|\|\; gen(\text{'goto'}\ B.false)$	A1

Example



$A1^1$ $A1^2$ $A1^3$ A2 A3 A4 A5 A6

Generated code:

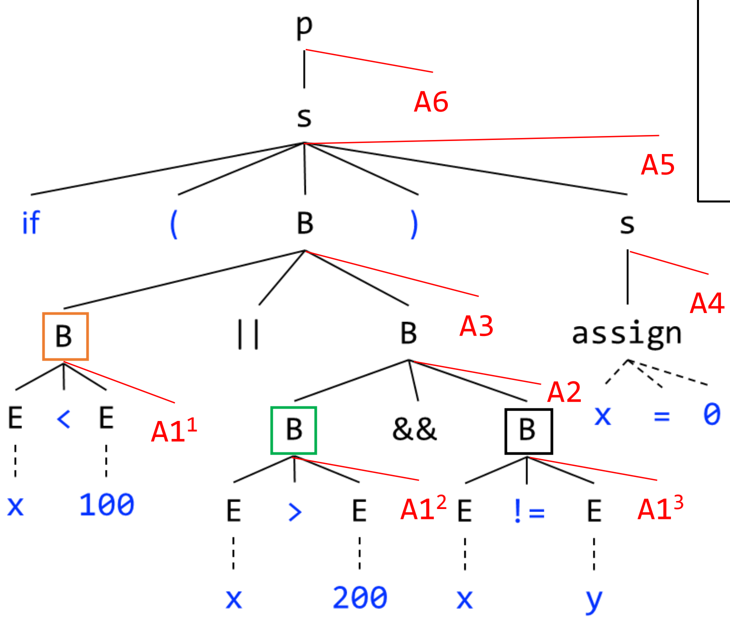
```
if x < 100 goto B.true
goto B.false
if x > 200 goto B.true
goto B.false
```

PRODUCTION	SEMANTIC RULES	
$P \to S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$	A6
$S \to \textbf{assign}$	$S.code = \textbf{assign}.code$	A4
$S \to \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$	A5

$B \to B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \to B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$	A2

$B \to E_1\ \textbf{rel}\ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$	A1

Example



A1¹ A1² **A1³** A2 A3 A4 A5 A6

Generated code:

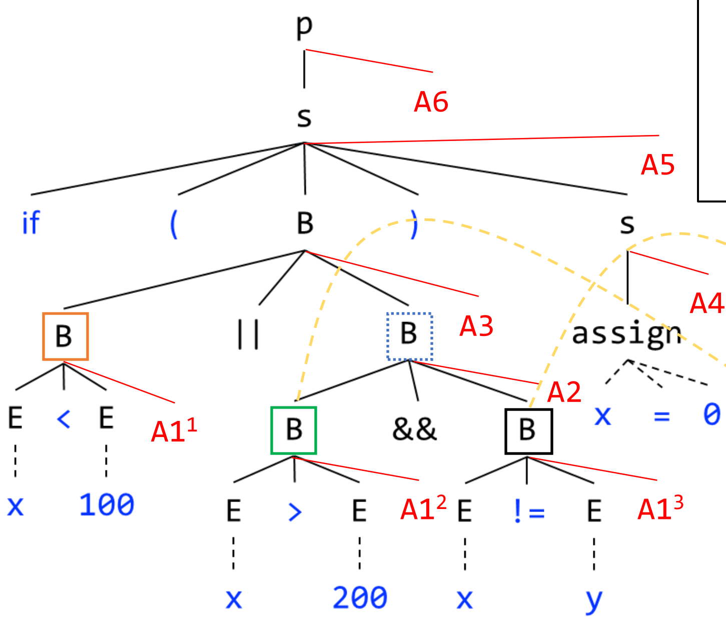
```
if x < 100 goto B.true
goto B.false
if x > 200 goto B.true
goto B.false
if x != y goto B.true
goto B.false
```

PRODUCTION	SEMANTIC RULES	
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$	A6
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$	A4
$S \rightarrow$ **if** $(B)\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$	A5

$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$	A3
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$	A2

$B \rightarrow E_1$ **rel** E_2	A1	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\| \; gen($'**if**' $E_1.addr$ **rel**$.op$ $E_2.addr$ '**goto**' $B.true)$ $\|\| \; gen($'**goto**' $B.false)$

Example



A1¹ A1² A1³ **A2** A3 A4 A5 A6

PRODUCTION	SEMANTIC RULES	
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$	A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$	A4
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$	A5

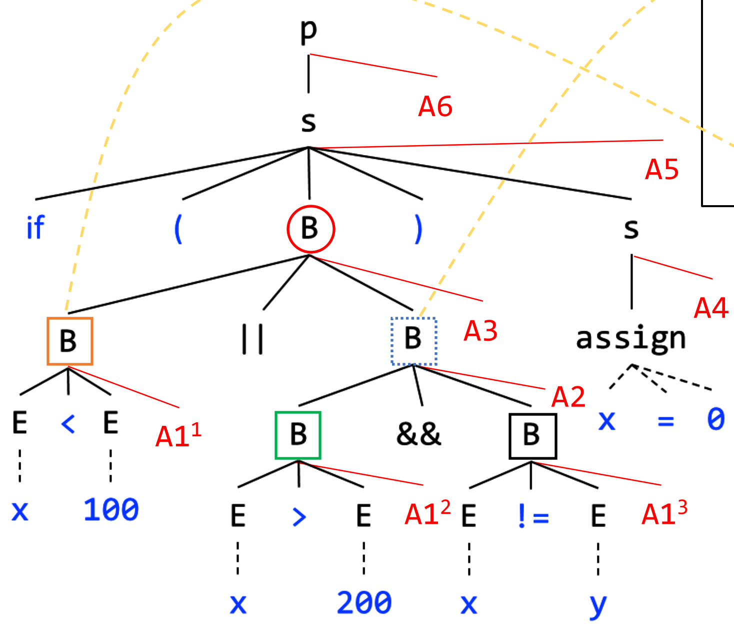
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$	A2

$B \rightarrow E_1 \textbf{ rel } E_2$	A1	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$

Generated code:

```
    if x < 100 goto B.true
    goto B.false
    if x > 200 goto B.true  = L4
    goto B.false  = B.false
L4: if x != y goto B.true = B.true
    goto B.false = B.false
```

Example



A1¹ A1² A1³ A2 **A3** A4 A5 A6

Generated code:

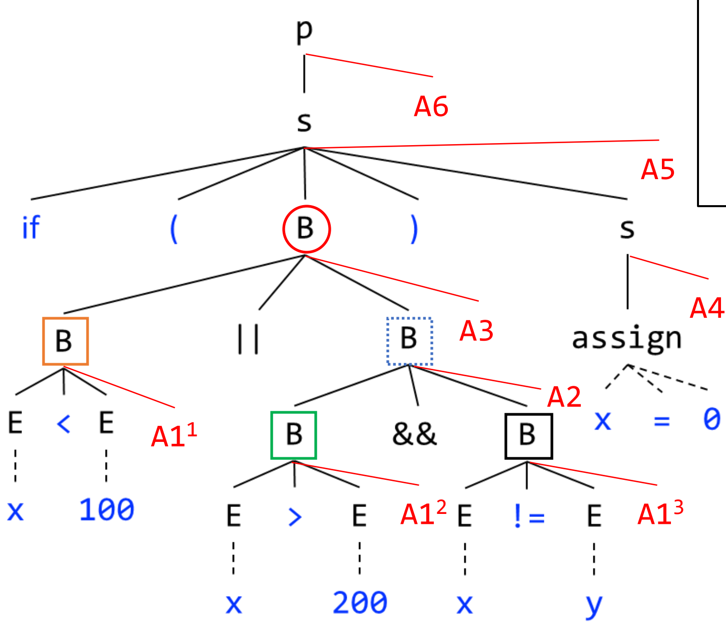
```
    if x < 100 goto B.true = B.true
    goto B.false = L3
L3: if x > 200 goto B.true = L4
    goto B.false = B.false = B.false
L4: if x != y goto B.true = B.true = B.true
    goto B.false = B.false = B.false
```

PRODUCTION	SEMANTIC RULES	
$P \to S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$	A6
$S \to \textbf{assign}$	$S.code = \textbf{assign}.code$	A4
$S \to \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$	A5

$B \to B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \to B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$	A2

$B \to E_1 \ \textbf{rel} \ E_2$	A1	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}' \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\mid\mid gen('\texttt{goto}' \ B.false)$

Example



A1¹ A1² A1³ A2 A3 **A4** A5 A6

PRODUCTION	SEMANTIC RULES	
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$	A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$	A4
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$	A5

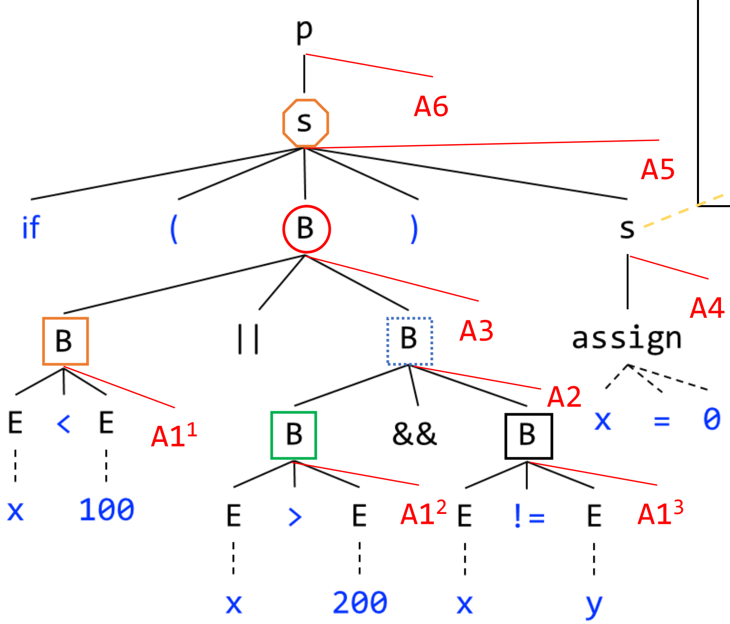
$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$	A3
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$	A2

$B \rightarrow E_1\ \textbf{rel}\ E_2$	A1	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\|\; gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\|\|\; gen('\texttt{goto}'\ B.false)$

Generated code:

```
    if x < 100 goto B.true  = B.true
    goto B.false = L3
L3: if x > 200 goto B.true  = L4
    goto B.false  = B.false = B.false
L4: if x != y goto B.true = B.true = B.true
    goto B.false = B.false = B.false
    x = 0
```

Example



A1¹ A1² A1³ A2 A3 A4 A5 A6

PRODUCTION	SEMANTIC RULES	
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$	A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$	A4
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$	A5

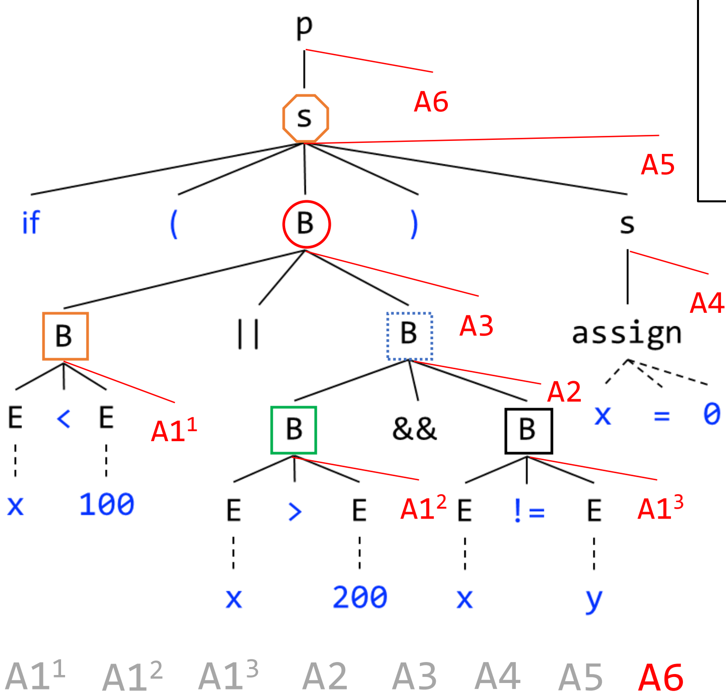
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \rightarrow B_1 \; \&\& \; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$	A2

$B \rightarrow E_1 \; \textbf{rel} \; E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}' \; E_1.addr \; \textbf{rel}.op \; E_2.addr \; '\texttt{goto}' \; B.true)$ $\mid\mid gen('\texttt{goto}' \; B.false)$	A1

Generated code:

```
    if x < 100 goto B.true = B.true = L2
    goto B.false = L3
L3: if x > 200 goto B.true = L4
    goto B.false = B.false = B.false = S.next
L4: if x != y goto B.true = B.true = B.true = L2
    goto B.false = B.false = B.false = S.next
L2: x = 0
```

Example



PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$ A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$ A4
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ A5 $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ A3 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ A2 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$

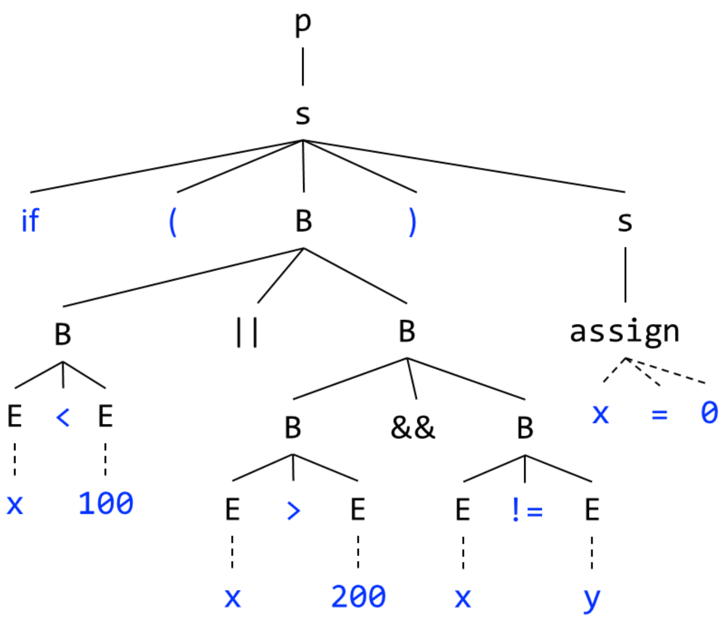
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ A1 $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$

Generated code:

```
    if x < 100 goto B.true = B.true  = L2
    goto B.false = L3
L3: if x > 200 goto B.true  = L4
    goto B.false  = B.false = B.false  = S.next = L1
L4: if x != y goto B.true = B.true = B.true = L2
    goto B.false = B.false = B.false  = S.next  = L1
L2: x = 0
L1: ...
```

Example

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



Generated code:

```
      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:
```

Outline

- Translation of Expressions (with array accesses)
- Control Flow
- Backpatching (self-study materials)
- Symbol Table Management
- Scope Checking

Backpatching (回填)

- A **key problem** when generating code for boolean expressions and flow-of-control statements is to match a jump instruction with the jump target
- **Example: if (*B*) *S***
  - According to the short-circuit translation, B's code contains a jump to the instruction following the code for S (executed when B is false)
  - However, B must be translated before S. The jump target is unknown when translating B
  - Earlier, we address the problem by passing labels as inherited attributes ($S.next$), but this requires another separate pass (traversing the parse tree) after parsing

How to address the problem in one pass?

One-Pass Code Generation Using Backpatching

- **Basic idea of backpatching (基本思想):**
  - When a jump is generated, its target is temporarily left unspecified.
  - Incomplete jumps are grouped into lists. All jumps on a list have the same target.
  - Fill in the labels for incomplete jumps when the targets become known.

- **The technique (技术细节):**
  - For a nonterminal B that represents a boolean expression, we define two synthesized attributes: $truelist$ and $falselist$
  - $truelist$: a list of jump instructions whose target is the jump target when B is true
  - $falselist$: a list of jump instructions whose target is the jump target when B is false

One-Pass Code Generation Using Backpatching

- **The technique (技术细节) Cont.:**
  - $makelist(i)$: create a new list containing only i, the index of a jump instruction, and return the pointer to the list
  - $merge(p_1, p_2)$: concatenate the lists pointed by p_1 and p_2, and return a pointer to the concatenated list
  - $backpatch(p, i)$: insert i as the target for each of the jump instructions on the list pointed by p

Backpatching for Boolean Expressions (布尔表达式的回填)

- An SDT suitable for generating code for boolean expressions during bottom-up parsing
- Grammar:
  - $B \rightarrow B_1 \parallel M B_2 \mid B_1 \, \&\& \, M B_2 \mid ! B_1 \mid (B_1) \mid E_1 \textbf{ rel } E_2 \mid \textbf{true} \mid \textbf{false}$
  - $M \rightarrow \epsilon$

Keep this question in mind: Why do we introduce M before B_2?

2

1

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) $B \rightarrow B_1 \; \&\& \; M \; B_2$ $\{$ $backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.falselist);$ $\}$

When finishing processing B1 && B2, we know the jump target for B1.truelist

3) $B \rightarrow \; ! \; B_1$ $\{$ $B.truelist = B_1.falselist;$
$B.falselist = B_1.truelist;$ $\}$

4) $B \rightarrow (\; B_1 \;)$ $\{$ $B.truelist = B_1.truelist;$
$B.falselist = B_1.falselist;$ $\}$

When finishing processing E1 rel E2, we do not know the jump targets, so generate incomplete instructions first

5) $B \rightarrow E_1 \; \textbf{rel} \; E_2$ $\{$ $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
gen('if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto _');
gen('goto _'); $\}$

6) $B \rightarrow \textbf{true}$ $\{$ $B.truelist = makelist(nextinstr);$
gen('goto _'); $\}$

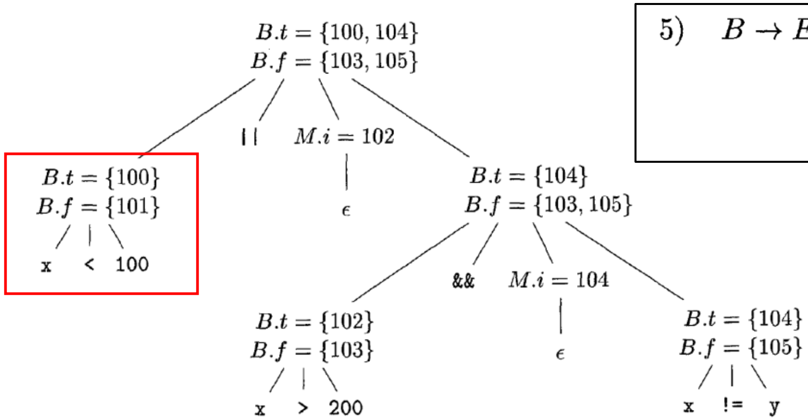
7) $B \rightarrow \textbf{false}$ $\{$ $B.falselist = makelist(nextinstr);$
gen('goto _'); $\}$

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

Tip: understand 1 and 2 at a high level first and then revisit this slide after you understand the later examples.

Example – Boolean Expressions

- The earlier SDT is a postfix SDT. The semantic actions can be performed during a bottom-up parse.
- Boolean expression: $x < 100 \parallel x > 200 \,\&\&\, x \,!= y$
- **Step 1:** reduce $x < 100$ to B by production (5)



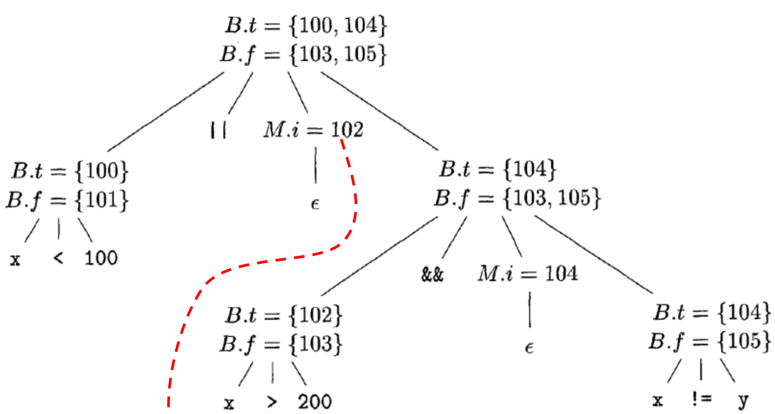
5)	$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ *gen*('if' $E_1.addr$ **rel**.*op* $E_2.addr$ 'goto _'); *gen*('goto _'); }

Generated code:

```
100:   if x < 100 goto _
101:   goto _
```

Example – Boolean Expressions

- The earlier SDT is a postfix SDT. The semantic actions can be performed during a bottom-up parse.
- Boolean expression: $x < 100 \parallel x > 200 \; \&\& \; x \, != y$
- **Step 2:** reduce ϵ to M by production (8)

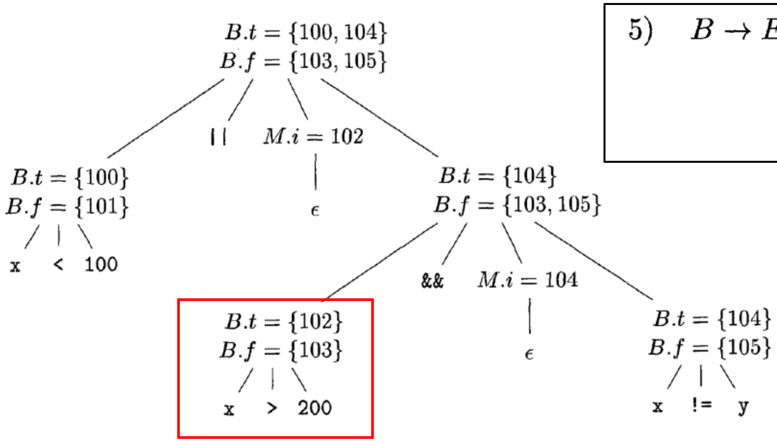


8)	$M \rightarrow \epsilon$	$\{ \; M.instr = nextinstr; \; \}$

The marker nonterminal records the value of *nextinstr*, 102

Example – Boolean Expressions

- Boolean expression: $x < 100 \,\|\, x > 200 \;\&\&\; x \mathbin{!}= y$
- **Step 3:** reduce $x > 200$ to B by production (5)



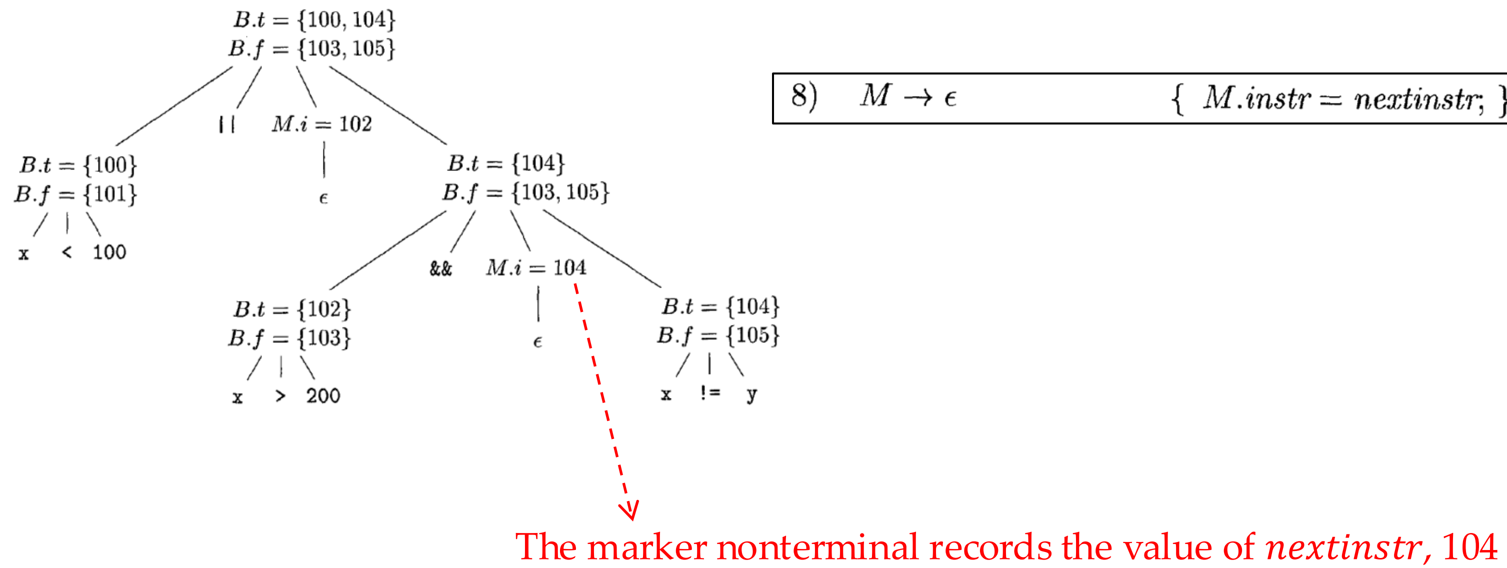
5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$gen($'`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto _`'$);$
$gen($'`goto _`'$);$ }

Generated code:

```
102:  if x > 200 goto _
103:  goto _
```

Example – Boolean Expressions

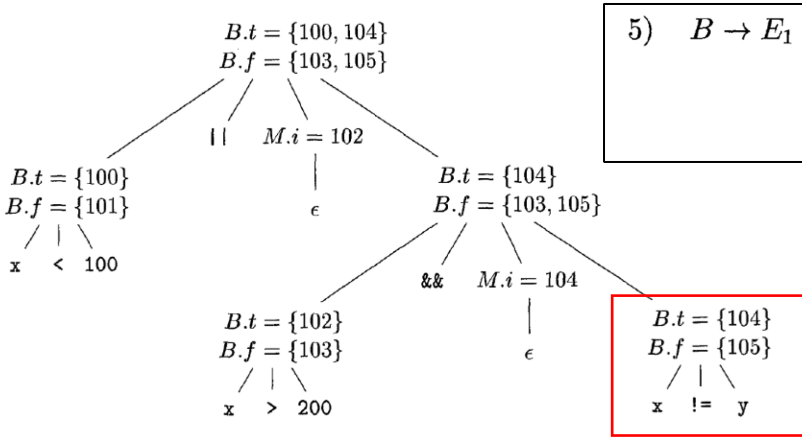
- Boolean expression: $x < 100 \parallel x > 200$ && $x\,! = y$
- **Step 4:** reduce ϵ to M by production (8)



$$\boxed{8)\quad M \rightarrow \epsilon \qquad \{\ M.instr = nextinstr;\ \}}$$

Example – Boolean Expressions

- Boolean expression: $x\ <\ 100 \parallel x > 200\ \&\&\ x\,!= y$
- **Step 5:** reduce $x! = y$ to B by production (5)



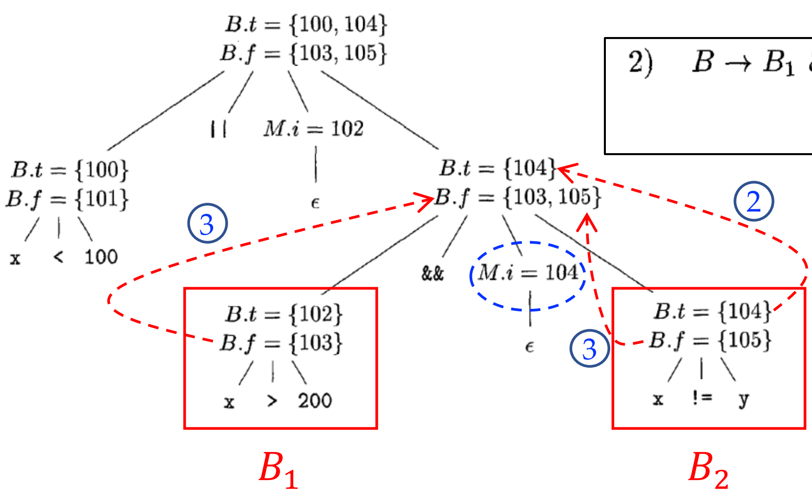
5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
gen('`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto _`');
gen('`goto _`'); }

Generated code:

```
104:  if x != y goto _
105:  goto _
```

Example – Boolean Expressions

- Boolean expression: $x < 100 \parallel x > 200 \;\&\&\; x != y$
- **Step 6:** reduce B_1 && MB_2 to B by production (2)



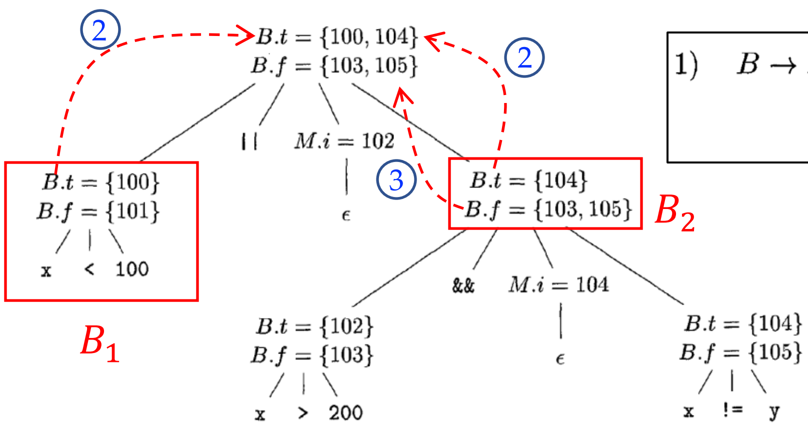
2) $B \rightarrow B_1 \;\&\&\; M\; B_2 \quad \{\; backpatch(B_1.truelist, M.instr);$ 1
$\quad B.truelist = B_2.truelist;$ 2
$\quad B.falselist = merge(B_1.falselist, B_2.falselist);$ 3

Backpatch instruction 102: 1

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

Example – Boolean Expressions

- Boolean expression: $x < 100 \parallel x > 200 \&\& x \mathrel{!=} y$
- **Step 7:** reduce $B_1 \parallel MB_2$ to B by production (1)



1)	$B \to B_1$ `||` $M\ B_2$	{ $backpatch(B_1.falselist, M.instr);$ 1 $B.truelist = merge(B_1.truelist, B_2.truelist);$ 2 $B.falselist = B_2.falselist;$ } 3

Backpatch instruction 101: 1

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

The remaining jump targets will be filled in later parsing steps

Backpatching vs. Non-Backpatching (1)

(1) Non-backpatching SDD with inherited attributes:

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

(2) Backpatching scheme:

$B \rightarrow E_1$ **rel** E_2	$\{\; B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $gen(\texttt{'if'}\; E_1.addr\; \mathbf{rel}.op\; E_2.addr\; \texttt{'goto \_'});$ $gen(\texttt{'goto \_'});\; \}$

Comparison:

- In (2), incomplete instructions (指令坯) are added to corresponding lists
- The instruction jumping to $B.true$ in (1) is added to $B.truelist$ in (2)
- The instruction jumping to $B.false$ in (1) is added to $B.falselist$ in (2)

Backpatching vs. Non-Backpatching (2)

(1) Non-backpatching SDD with inherited attributes:

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

(2) Backpatching scheme:

$B \rightarrow B_1 \mid\mid M\ B_2$	$\{$ $backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;$ $\}$

Comparison:

- The assignments to $true/false$ attributes in (1) correspond to the manipulations of $truelist/falselist$ in (2)

Outline

- Translation of Expressions (with array accesses)
- Control Flow
- Backpatching (self-study materials)
- Symbol Table Management
- Scope Checking

Symbol Table

- A *symbol table* maps an identifier (name) to its associated information
    - **identifier**: variable name, function name, user-defined type name (the name of the struct type in SPL), …
    - **information**: types, array dimension, struct members, initial values, …

Key: identifier → **Value:** the associated information

A symbol table is essentially a set of such key-value pairs

Symbol Table Operations

- **Symbol table operations during compilation**
  - **lookup**: check for variable existence, type definition, …
  - **insert**: when seeing function/variable/type declarations, …
  - **delete**: current scope finished, delete all identifiers inside (may not need this operation if only global scope is supported)

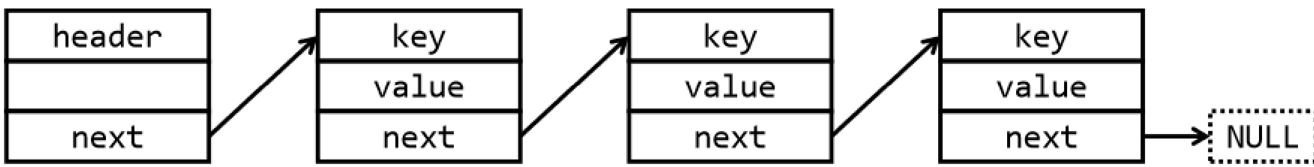
```
ExtDef -> Specifier ExtDecList        <- Handle global variables when reducing using this production
ExtDef -> Specifier SEMI              <- Handle user-defined types
ExtDef -> Specifier FunDec CompSt     <- Handle functions
Def -> Specifier DecList SEMI         <- Handle local variables
```

Symbol Table Implementation

- You are free to implement symbol table in terms of:
  - Stored information
    - Our suggestion: only store type information, including type info for variables, function return values, function parameters, and self-defined data types
  - Possible choices of abstract data types:
    - linked list, hash table, binary search tree, …

Abstract Data Types

- **Linked list**



  - **Lookup**: O(n) in worst case
  - **Insert**: O(1) at head, O(n) at tail
  - **Delete**: O(n) in worst case

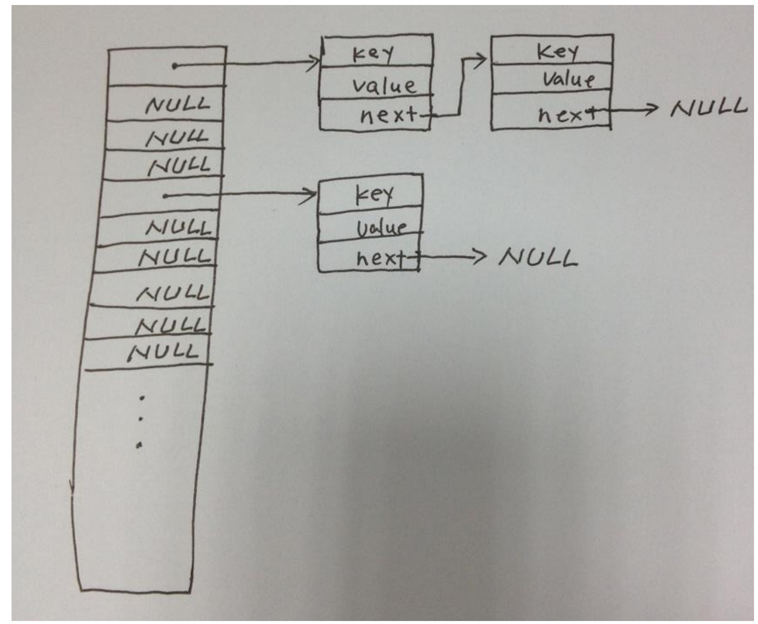
Abstract Data Types

- **Hash table**
  - Allocate a large consecutive space
  - Compress key to index (hash function)*
  - Most operations can be done in O(1)
  - Drawback: space consumption

* You may consider using https://en.wikipedia.org/wiki/PJW_hash_function

Abstract Data Types

- **Hash table conflicts**
  - When the hash functions maps multiple keys to the same index
  - **Solution #1**: Separate chaining (分离链接法)
  - **Solution #2**: Rehashing (再哈希法), which uses multiple hash functions and recomputes the hash value by an alternative hash function upon collisions



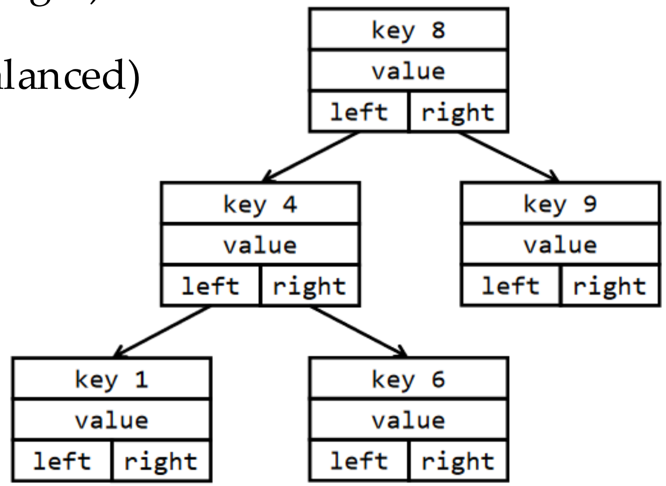
Separate chaining

Abstract Data Types

- **Binary search tree**
  - The key in each node is greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree
  - Ideally, the time complexity of operations: O(log n)
  - O(n) in worst case (when tree extremely imbalanced)
  - Balance strategies:*

    + AVL tree

    + Red-black tree



* https://www.javatpoint.com/red-black-tree-vs-avl-tree

Outline

- Translation of Expressions (with array accesses)
- Control Flow
- Backpatching (self-study materials)
- Symbol Table Management
- Scope Checking

Scope Checking

- Variables in a program are only visible within certain sections, called *scope*

- For program without scopes, we say there is only global scope (the assumption of our SPL)

- *Scope checking* refers to the process of determining if an identifier (symbol) is accessible at a program location

Scope Example

```
int test_2_o01(){
    int a, b, c;
    a = a + b;
    if(b > 0){
        int a = c * 7;
        b = b - a;
    }
    return a + c*b;
}
```

Which a does it refer to?

Which a does it refer to?

Scope Checking

- What if there is only one symbol table?

```
int test_2_o01(){
    int a=0, b=1, c=2;
    a = a + b;
    if(b > 0){
        int a = c * 7;
        b = b - a;
    }
    return a + c*b;
}
```

id	type	Declaration location
a	int	Line 2
b	int	Line 2
c	int	Line 2

Scope Checking

- What if there is only one symbol table?

```
int test_2_o01(){
    int a=0, b=1, c=2;
    a = a + b;
    if(b > 0){
        int a = c * 7;
        b = b - a;
    }
    return a + c*b;
}
```

Shall we update it to line 5?

id	type	Declaration location
a	int	Line 2
b	int	Line 2
c	int	Line 2

Scope Checking

- What if there is only one symbol table?

```
int test_2_o01(){
    int a=0, b=1, c=2;
    a = a + b;
    if(b > 0){
        int a = c * 7;
        b = b - a;
    }
    return a + c*b;
}
```

id	type	Declaration location
a	int	Line 5
b	int	Line 2
c	int	Line 2

If we update to line 5 earlier, then at line 7, the compiler would consider *a* to be defined at line 5, which is not correct…

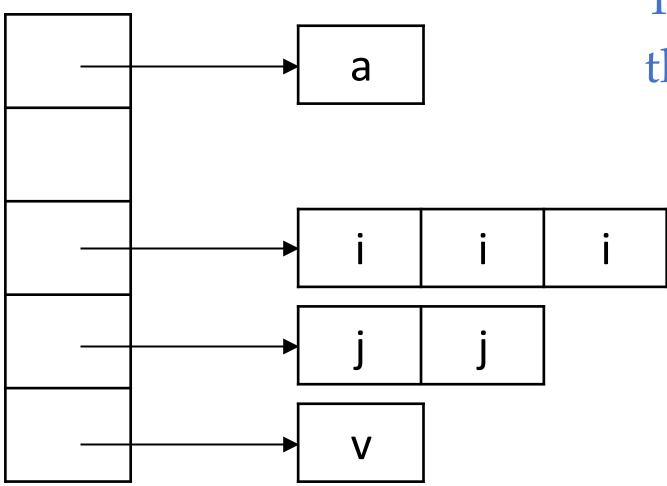
Implementing Scope Checking

- Two common strategies
  - Single table (also known as “imperative style”)
  - Multiple tables (also known as “functional style”)

- Both need to delete symbols when leaving a scope

Single Table Strategy

- The naïve implementation:
  - Use a hash table to implement the symbol table
  - Use separated chaining to address conflicts
  - Insert duplicate keys at the head of the corresponding list



The innermost definition always appears at the head of list (easy to find ☺)

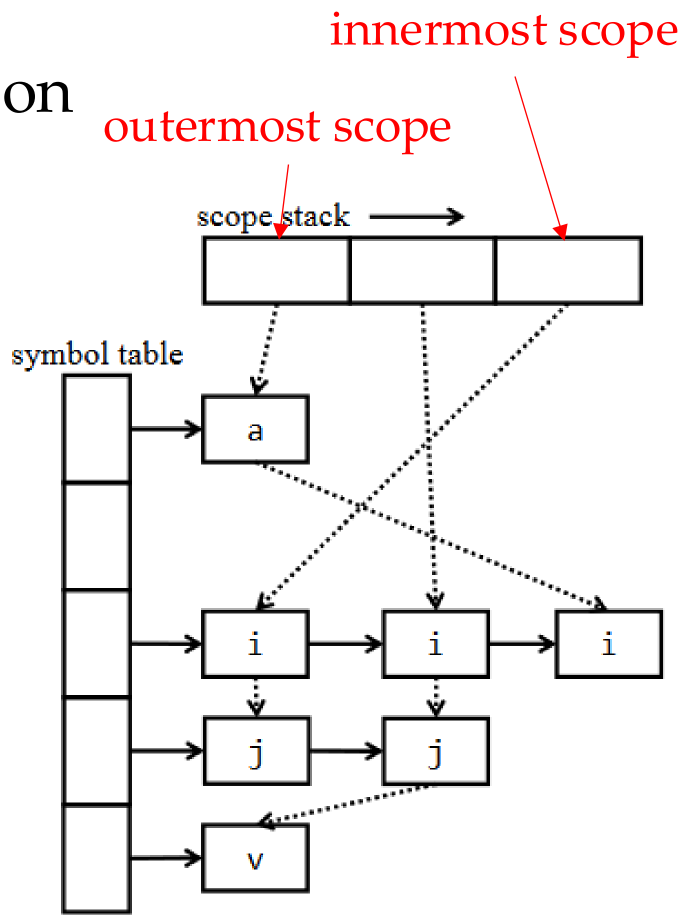
Disadvantage:

When the current scope is closed, we need to remove symbols, which is not easy:

- Need to traverse all linked lists to check if the symbol at the head is available only for the current scope (i.e., defined there)

Single Table Strategy

- The orthogonal list implementation



Advantages:

- Still easy to find the innermost definition (at the head of each list)
- When closing the current scope, removing symbols is easy:
  - Tracing through the list corresponding to the stack top can efficiently locate the "to be removed" symbols

Multiple Tables Strategy

- Maintaining scope stack, each element is a symbol table
- Push new table when entering a new scope
- Pop the topmost table when leaving a scope

Disadvantage: When analyzing the scope for a variable, one may need to search all the way down the stack (from the symbol table at the top of the stack to the symbol table at the bottom of the stack)

Project Milestone Check #2

- During the lab session on week #13 (Dec. 8)
- Expected progress
  - Lexical analysis (100% done)
  - Parsing (100% done)
  - Semantic analysis (simple rules such as those related to basic type checking 100% done)
  - Intermediate code generation (in progress)