南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 4:
Intermediate-Code Generation

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Intermediate Representation
- Type and Declarations
- Type Checking
- Translation of Expressions
- Control Flow

Translating Expressions

- An expression* with more than one operator: $a + b * c$
  - Translate into multiple instructions with at most one operator per instruction

$$a + b * c \quad \Rightarrow \quad \begin{aligned} t_1 &= b * c \\ t_2 &= a + t_1 \end{aligned}$$

* Expressions may involve array accesses. Such cases will be discussed later.

SDD for Expression Translation – Attributes

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

S.code and *E.code* denote three-address code

E.addr denotes the address that will hold the value of *E*

top denotes the current symbol table; *get* returns the address of **id** (a variable)

gen generates three-address instructions

All attributes are synthesized. This S-attributed SDD can be implemented during bottom-up parsing.

SDD for Expression Translation – Rules

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

Temporary name generated by compiler

Check the symbol-table entry for **id** and save its address in *E.addr*

SDD for Expression Translation – Rules

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

- Generate instructions when seeing operations.
- Then concatenate instructions.

SDD for Expression Translation – Problem

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

Code attributes can be very long strings (as the expressions can be arbitrarily complex)

Redundant parts (due to value passings and concatenations) waste memory!

Incremental Translation Scheme

- In the SDT below, *gen* not only generates a three-address instruction, but also appends it to the sequence of instructions generated so far
  - In comparison, in the previous SDD, the *code* attribute can be long strings after concatenations

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

Why can this incremental approach guarantee the correct order of instructions?

Incremental Translation Scheme

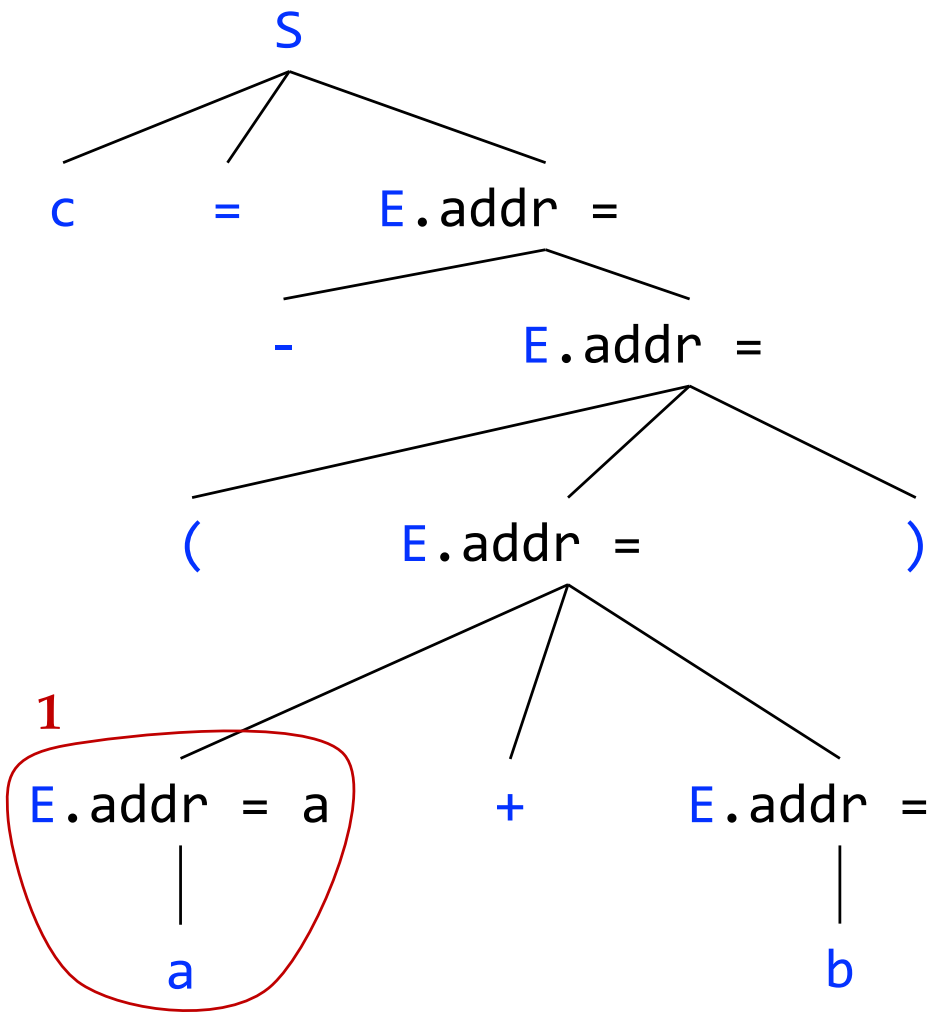
- In the SDT below, *gen* not only generates a three-address instruction, but also appends it to the sequence of instructions generated so far
  - In comparison, in the previous SDD, the *code* attribute can be long strings after concatenations

$S \rightarrow$	**id** = E ;	{ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$; }
$E \rightarrow$	E_1 + E_2	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
|	− E_1	{ $E.addr = \textbf{new}\ Temp()$; $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$; }
|	(E_1)	{ $E.addr = E_1.addr$; }
|	**id**	{ $E.addr = top.get(\textbf{id}.lexeme)$; }

This postfix SDT can be implemented in bottom-up parsing* where subexpressions are always handled first (e.g., the code of E_1 and E_2 is generated before E)

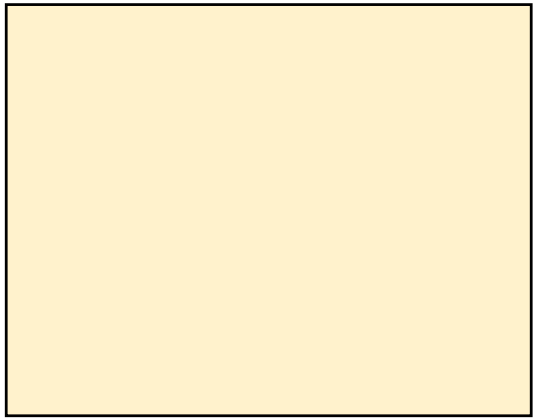
* Semantic actions are executed upon reduction.

Example: Translating c = - (a + b)



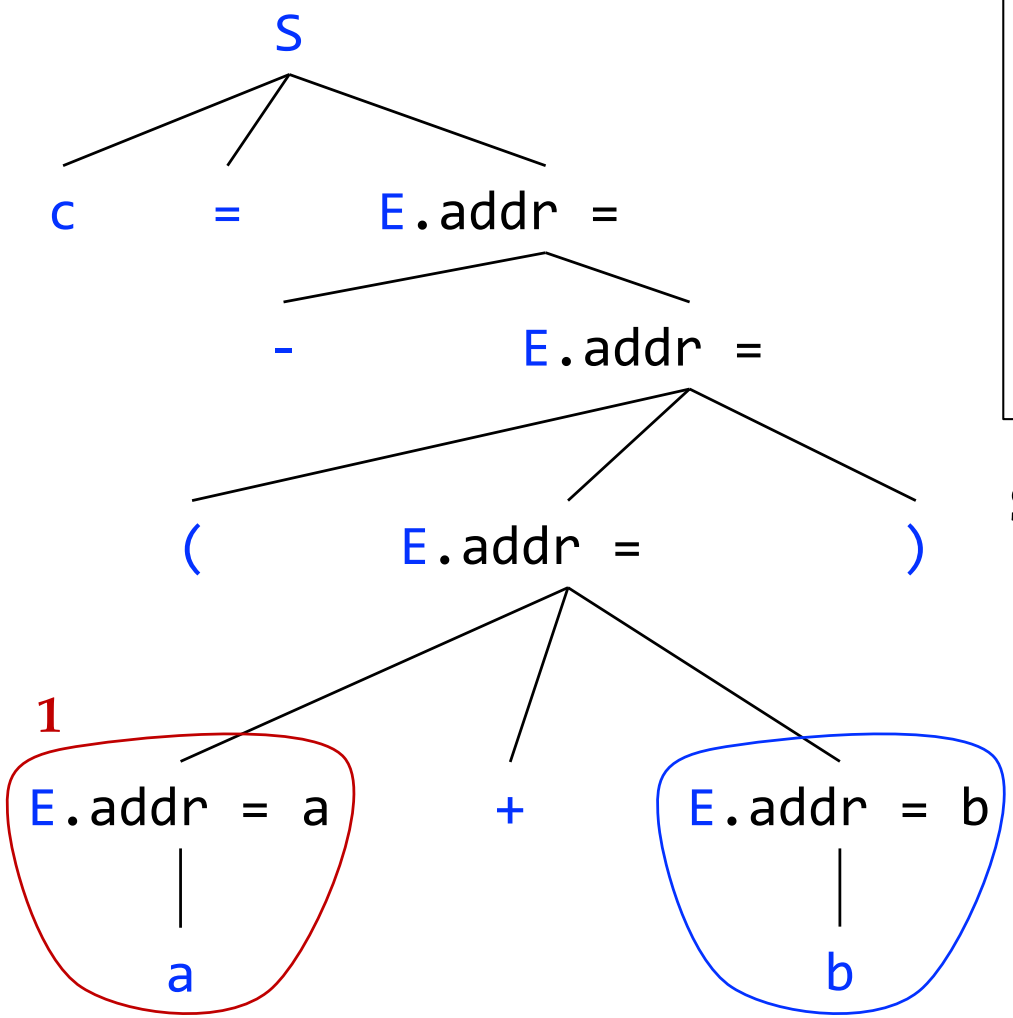
$S \rightarrow$	**id** = E ;	{ $gen($ $top.get($**id**$.lexeme)$ $'='$ $E.addr)$; }
$E \rightarrow$	E_1 + E_2	{ $E.addr =$ **new** $Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
|	- E_1	{ $E.addr =$ **new** $Temp()$; $gen(E.addr\ '='\ '$**minus**$'\ E_1.addr)$; }
|	(E_1)	{ $E.addr = E_1.addr$; }
|	**id**	{ $E.addr =$ $top.get($**id**$.lexeme)$; } **1**

Stack: c = - (a Reduce a to E



Generated code

Example: Translating c = – (a + b)



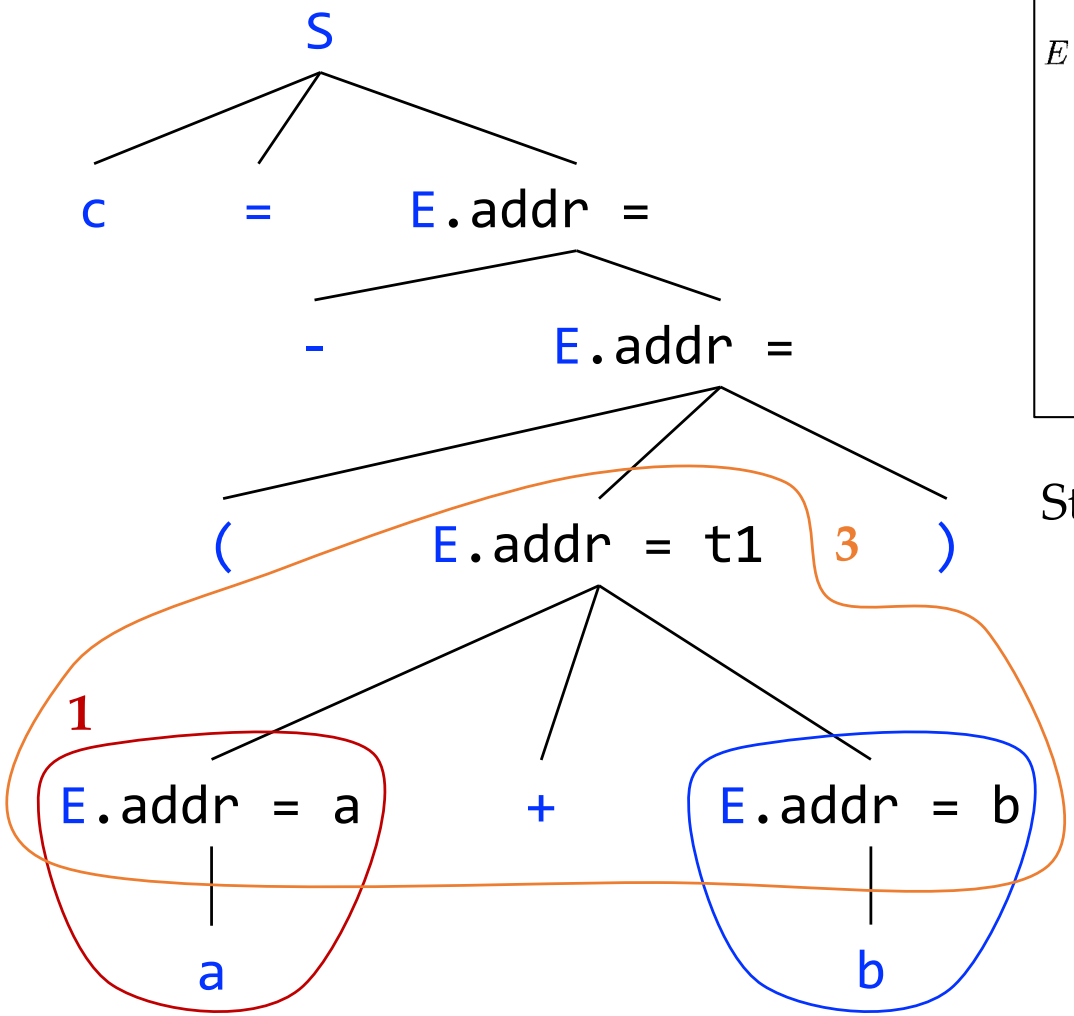
$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	– E_1	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
|	(E_1)	{ E.*addr* = E_1.*addr*; }
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); } **1 2**

Stack: c = - (E + b Reduce b to E

2

Generated code

Example: Translating c = - (a + b)



$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp* (); 3 *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	- E_1	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
|	(E_1)	{ E.*addr* = E_1.*addr*; }
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); } 1 2

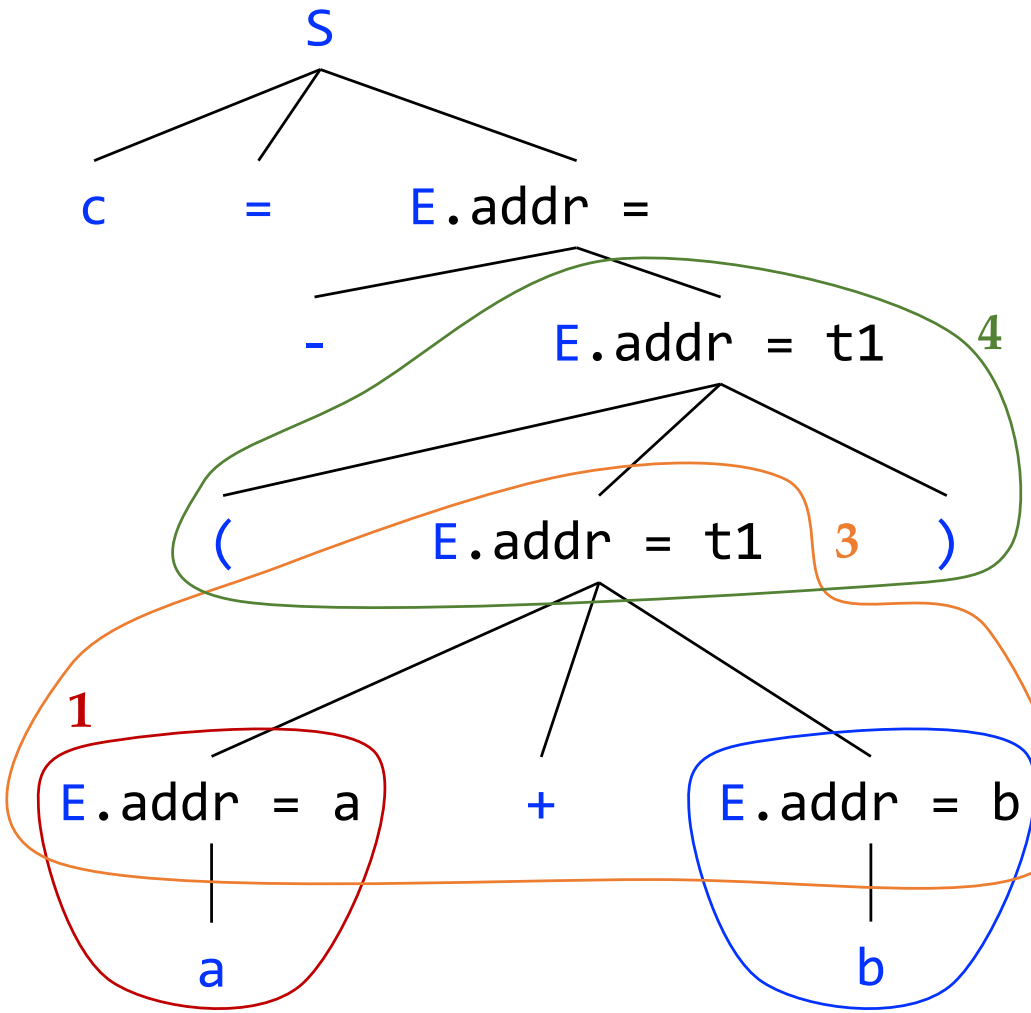
Stack: `c = - ( E + E` Reduce E+E to E

```
t1 = a + b
```

3

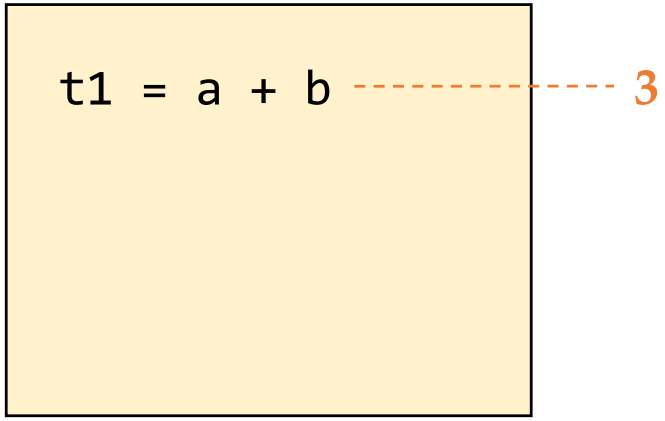
Generated code

Example: Translating c = - (a + b)



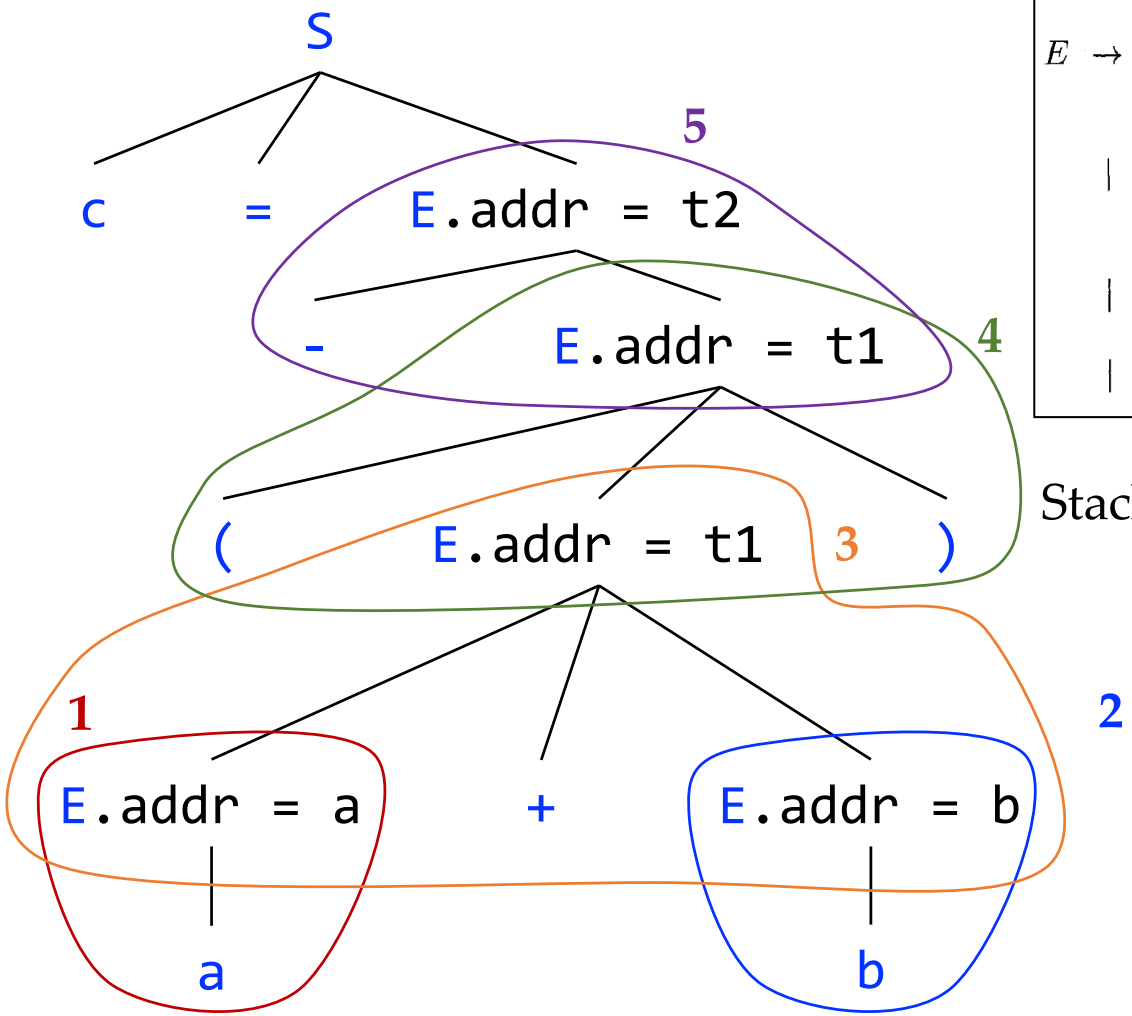
$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp* (); 3 *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	- E_1	{ E.*addr* = **new** *Temp* (); *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
|	(E_1)	{ E.*addr* = E_1.*addr*; } 4
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); } 1 2

Stack: `c = - ( E )` Reduce (E) to E



Generated code

Example: Translating c = – (a + b)



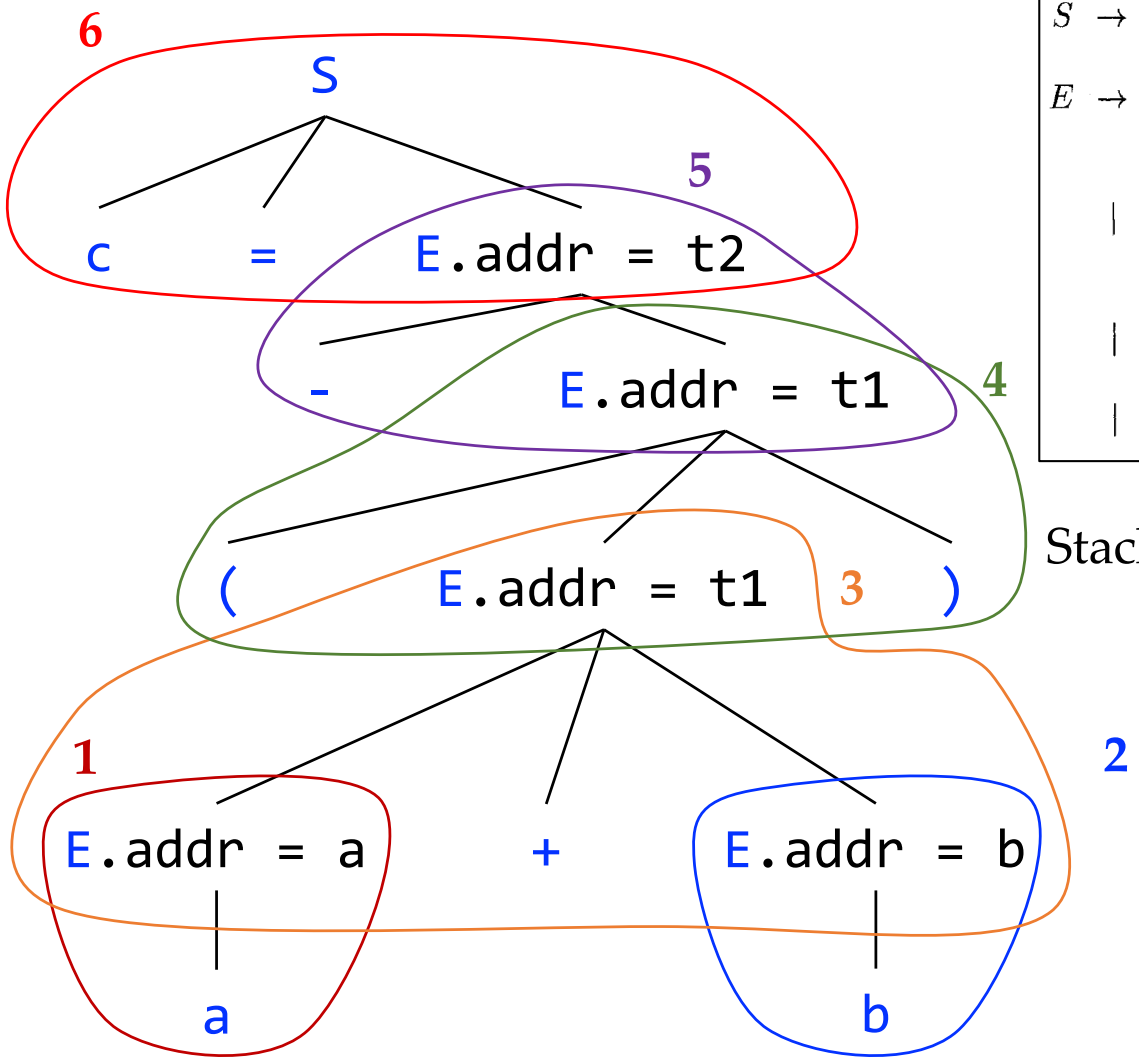
$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp* (); 3 *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	– E_1	{ E.*addr* = **new** *Temp* (); 5 *gen*(E.*addr* '=' 'minus' E_1.*addr*); }
|	(E_1)	{ E.*addr* = E_1.*addr*; } 4
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); } 1 2

Stack: c = - E Reduce -E to E

```
t1 = a + b      ---------- 3
t2 = - t1       ---------- 5
```

Generated code

Example: Translating c = – (a + b)



$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); } 6
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp* (); 3 *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	– E_1	{ E.*addr* = **new** *Temp* (); 5 *gen*(E.*addr* '=' 'minus' E_1.*addr*); }
|	(E_1)	{ E.*addr* = E_1.*addr*; } 4
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); } 1 2

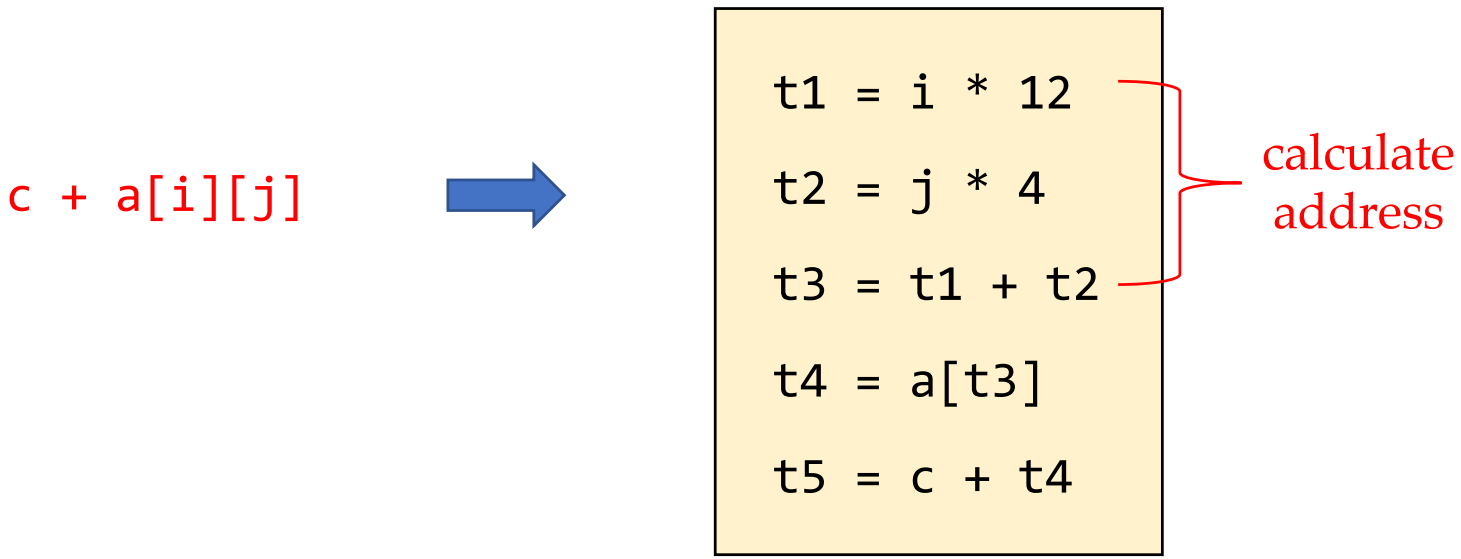
Stack: `c = E` Reduce c = E to E

```
t1 = a + b      ---------- 3
t2 = - t1       ---------- 5
c = t2          ---------- 6
```

Generated code

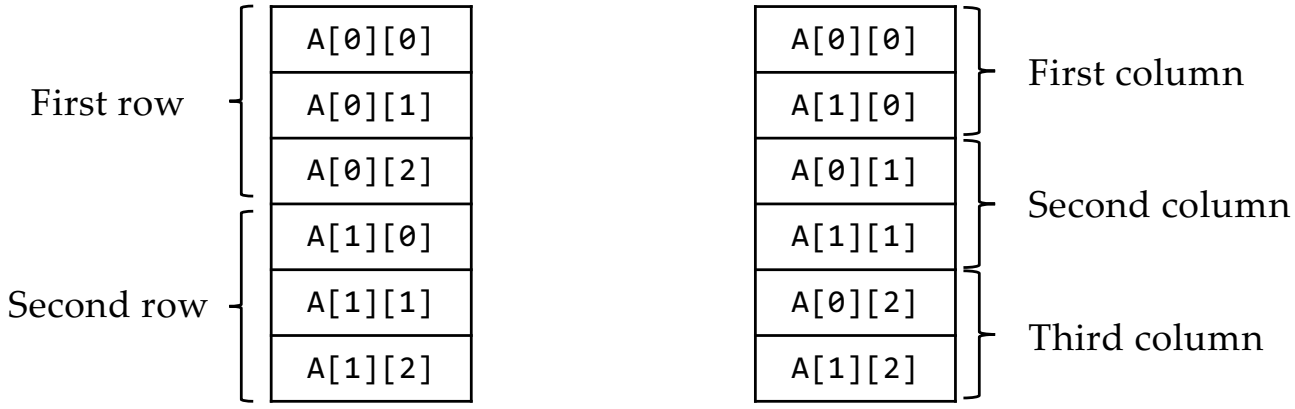
Dealing with Arrays

- An expression involving 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



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)



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)

$\boldsymbol{w_1}$ is the width of the (n-1)-dimensional subarray of A (suppose A has n dimensions)

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: $L \rightarrow L\ [\ E\ ]\ |\ \mathbf{id}\ [\ E\ ]$

SDT for Array References (1)

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

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

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 $base + offset$

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

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

Array references can be part of an expression

SDT for Array References (3)

$$S \rightarrow \mathbf{id} = E\ ; \quad \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \}$$

$$\mid\ \ L = E\ ; \quad \{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \}$$

Instruction of form $a[i] = x$

Array references can appear at the LHS of an assignment statement

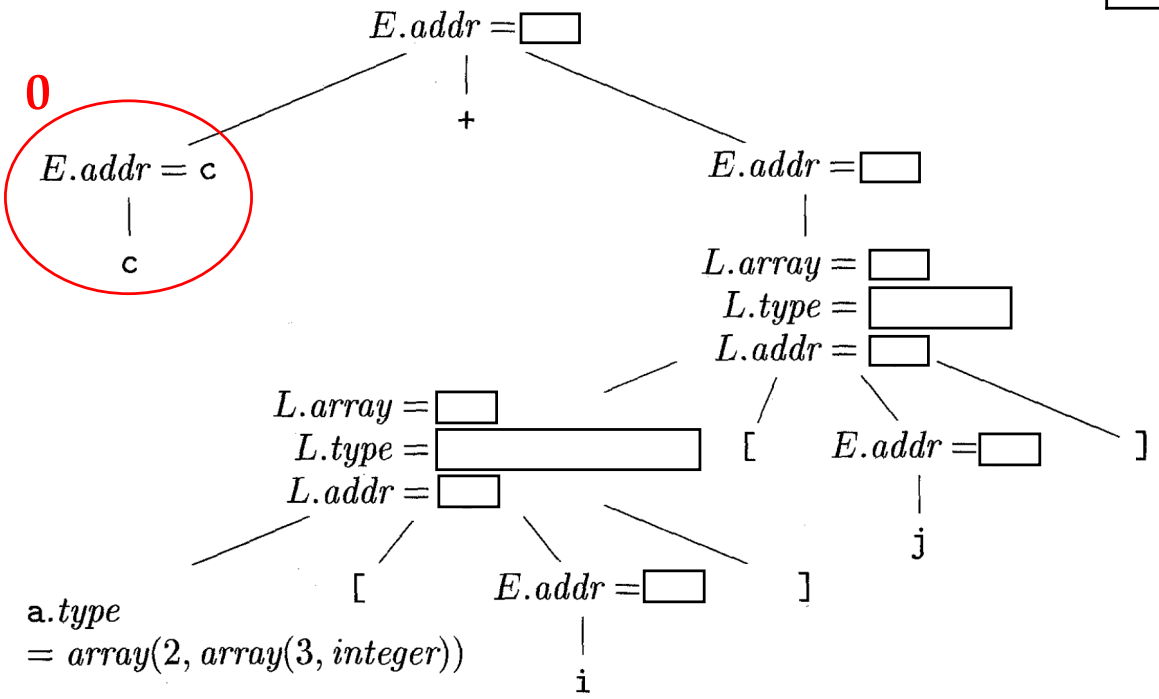
$E \rightarrow E_1 + E_2$	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
| **id**	{ $E.addr = top.get(\textbf{id}.lexeme);$ } **0**
| L	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

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

Translating c + a[i][j]

Stack: c

Reduce c to E



Generated code:

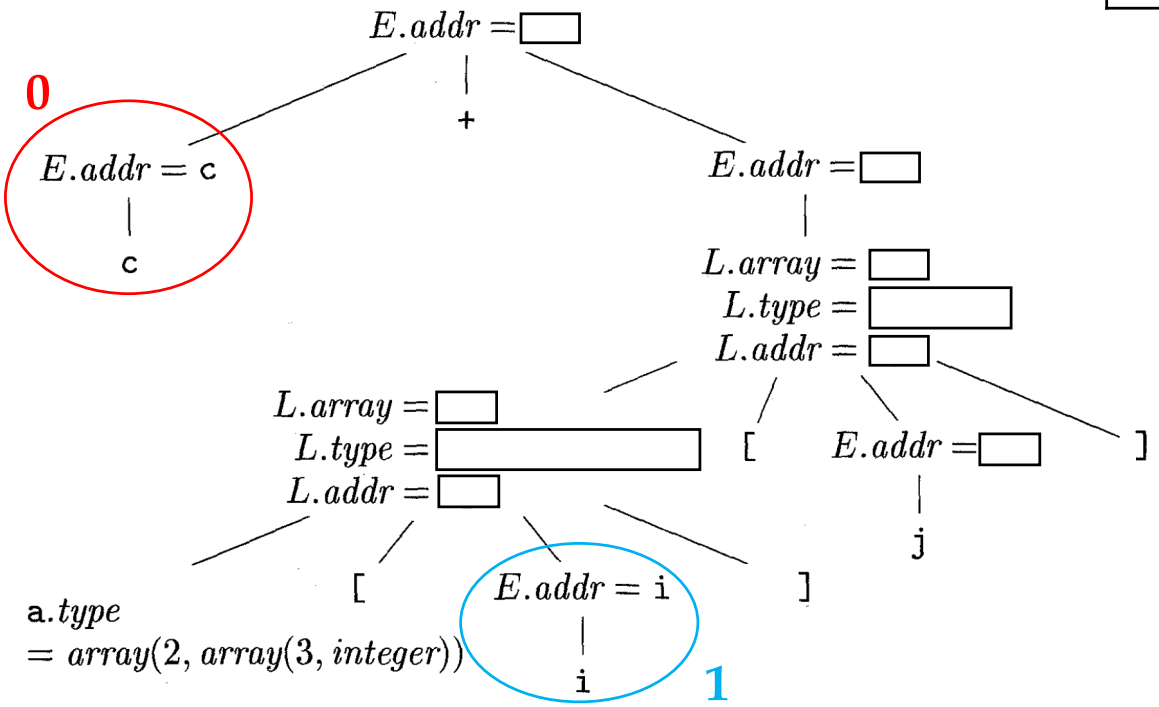
$E \rightarrow E_1 + E_2$	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
| **id**	{ $E.addr = top.get(\textbf{id}.lexeme);$ } **0** **1**
| L	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

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

Translating c + a[i][j]

Stack: E + a[i]

Reduce i to E



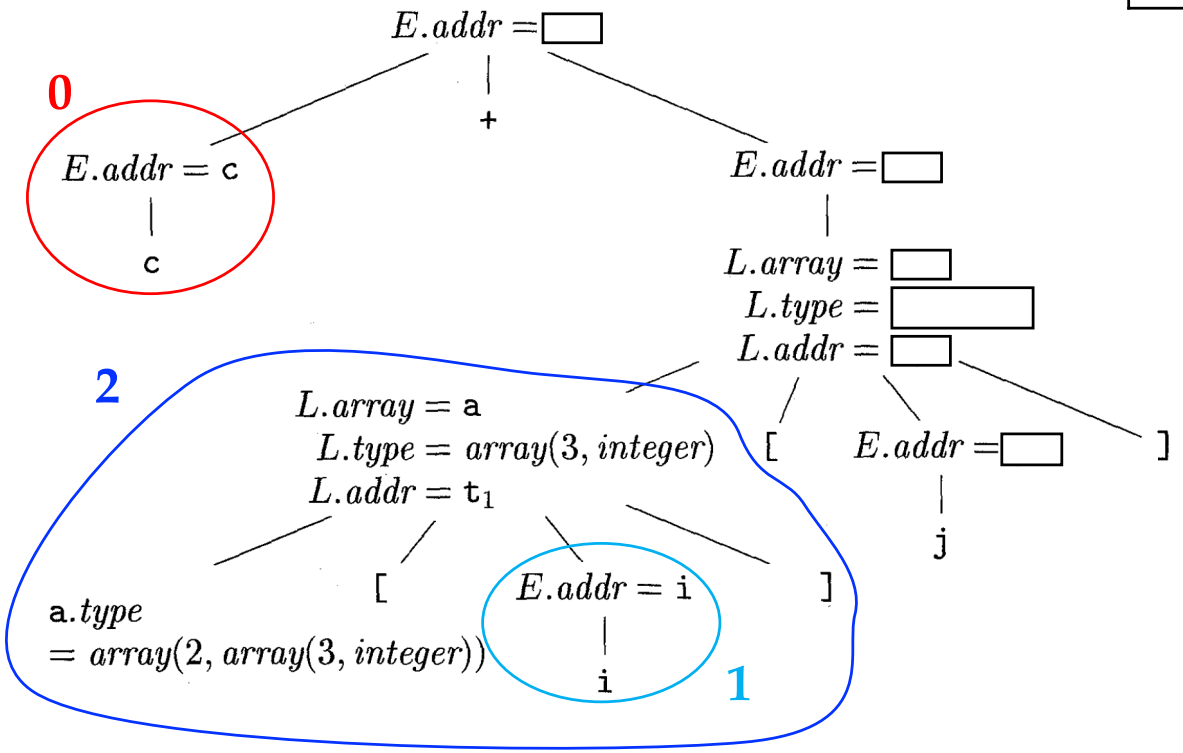
Generated code:

$E \rightarrow E_1 + E_2$	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
| **id**	{ $E.addr = top.get(\textbf{id}.lexeme);$ } **0** **1**
| L	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

$L \rightarrow$ **id** [E]	{ $L.array = top.get(\textbf{id}.lexeme);$ **2** $L.type = L.array.type.elem;$ $L.addr = \textbf{new}\ Temp();$ $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }
| L_1 [E]	{ $L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \textbf{new}\ Temp();$ $L.addr = \textbf{new}\ Temp();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Translating c + a[i][j]

Stack: E + a[E] Reduce a[E] to L



Generated code:

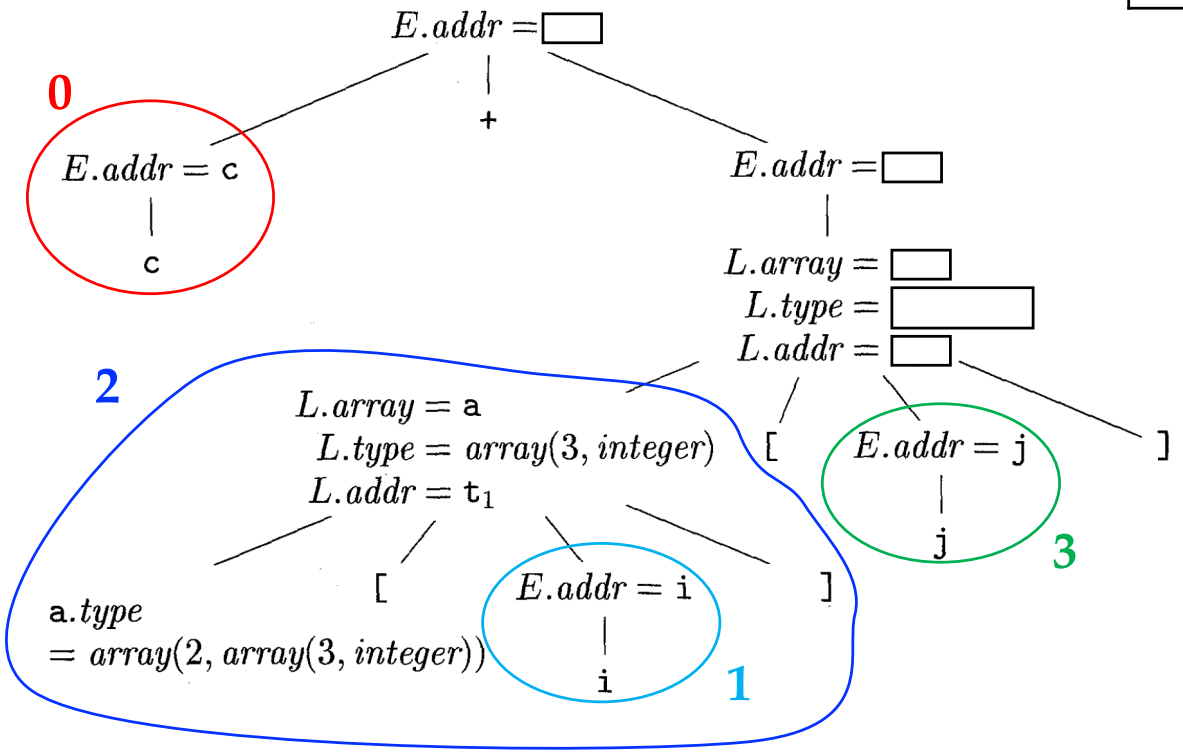
```
t1 = i * 12 --------- 2
```

$E \rightarrow E_1 + E_2$	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
| **id**	{ $E.addr = top.get(\textbf{id}.lexeme);$ } **0 1 3**
| L	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

$L \rightarrow$ **id** [E]	{ $L.array = top.get(\textbf{id}.lexeme);$ $L.type = L.array.type.elem;$ $L.addr = \textbf{new}\ Temp();$ $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ } **2**
| L_1 [E]	{ $L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \textbf{new}\ Temp();$ $L.addr = \textbf{new}\ Temp();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Translating c + a[i][j]

Stack: E + L[j] Reduce j to E



Generated code:

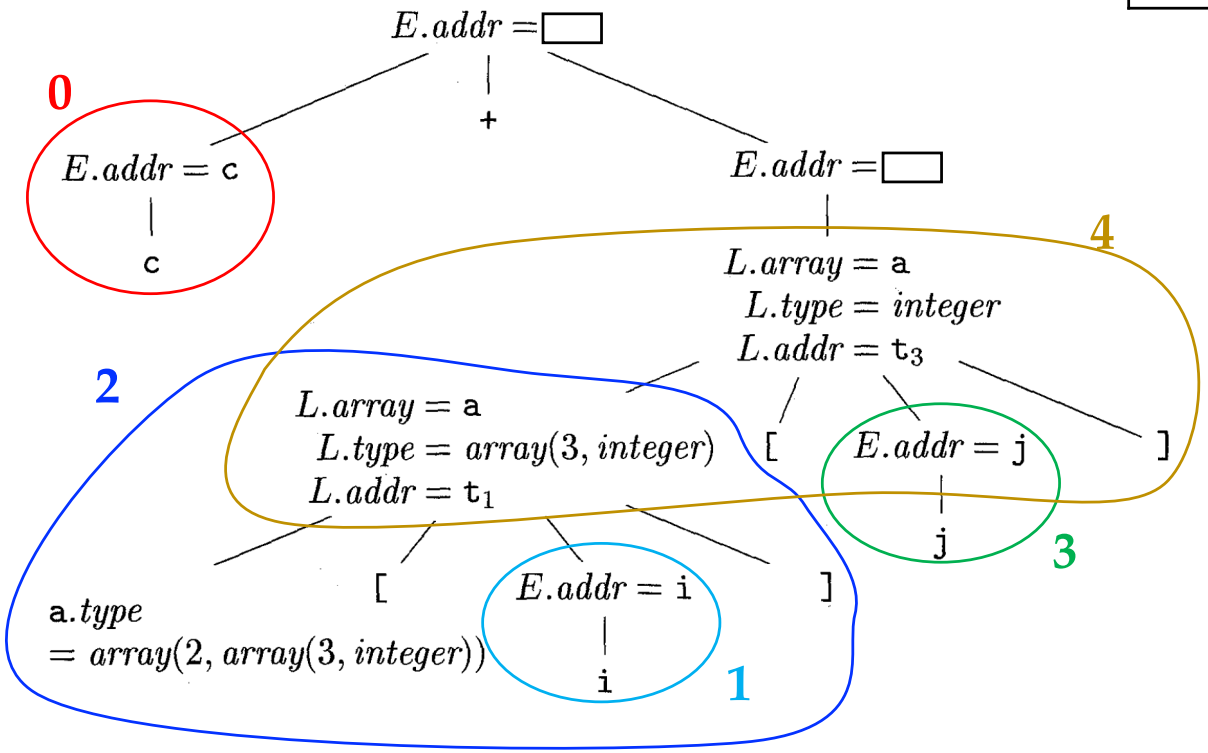
```
t1 = i * 12 --------- 2
```

$E \rightarrow E_1 + E_2$		$\{ E.addr = \textbf{new}\ Temp(); \; gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr); \}$
|	**id**	$\{ E.addr = top.get(\textbf{id}.lexeme); \}$ **0 1 3**
|	L	$\{ E.addr = \textbf{new}\ Temp(); \; gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']'); \}$

$L \rightarrow$	**id** [E]	$\{ L.array = top.get(\textbf{id}.lexeme); \; L.type = L.array.type.elem; \; L.addr = \textbf{new}\ Temp(); \; gen(L.addr\ '='\ E.addr\ '*'\ L.type.width); \}$ **2**
|	L_1 [E]	$\{ L.array = L_1.array; \; L.type = L_1.type.elem; \; t = \textbf{new}\ Temp(); \; L.addr = \textbf{new}\ Temp(); \; gen(t\ '='\ E.addr\ '*'\ L.type.width); \; gen(L.addr\ '='\ L_1.addr\ '+'\ t); \}$ **4**

Translating c + a[i][j]

Stack: E + L[E] Reduce L[E] to L



Generated code:

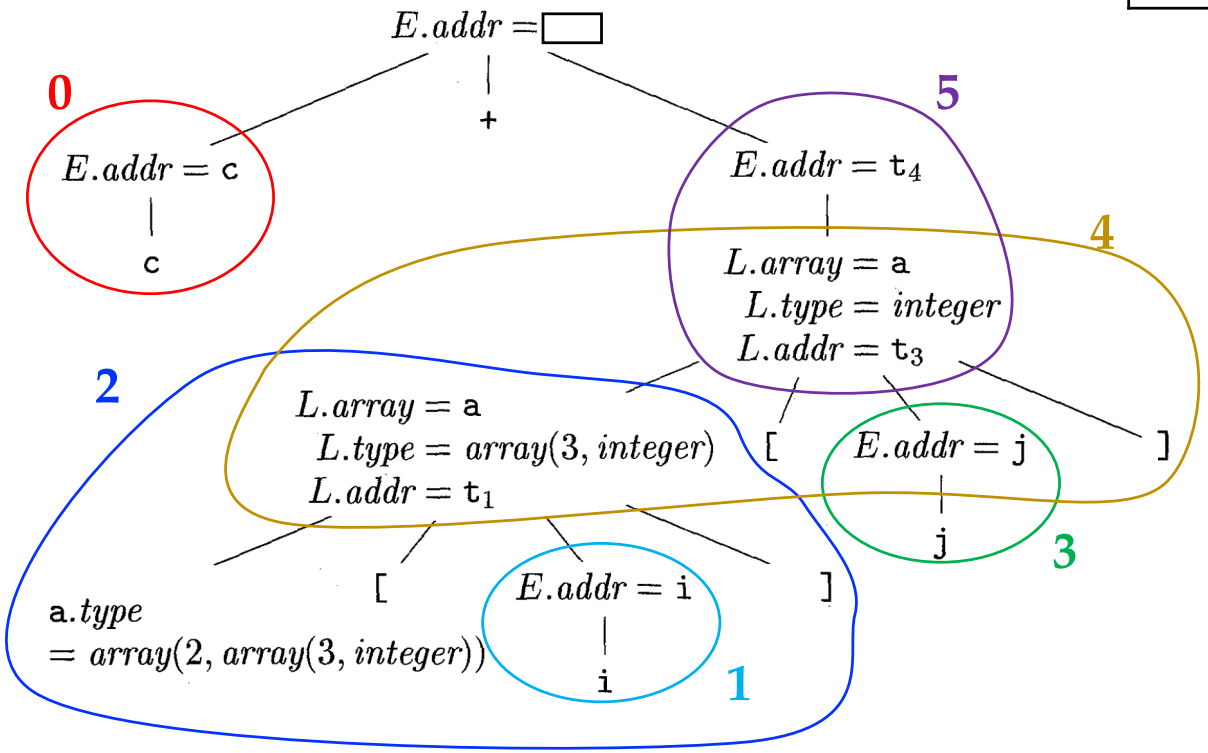
```
t1 = i * 12      ------- 2
t2 = j * 4       -.
                   4
t3 = t1 + t2     -'
```

$E \rightarrow E_1 + E_2$	{ $E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
| **id**	{ $E.addr = top.get(\textbf{id}.lexeme);$ } **0 1 3**
| L	{ $E.addr = \textbf{new}\ Temp();$ **5** $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

$L \rightarrow$ **id** [E]	{ $L.array = top.get(\textbf{id}.lexeme);$ $L.type = L.array.type.elem;$ **2** $L.addr = \textbf{new}\ Temp();$ $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }
| L_1 [E]	{ $L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \textbf{new}\ Temp();$ **4** $L.addr = \textbf{new}\ Temp();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Translating c + a[i][j]

Stack: E + L Reduce L to E



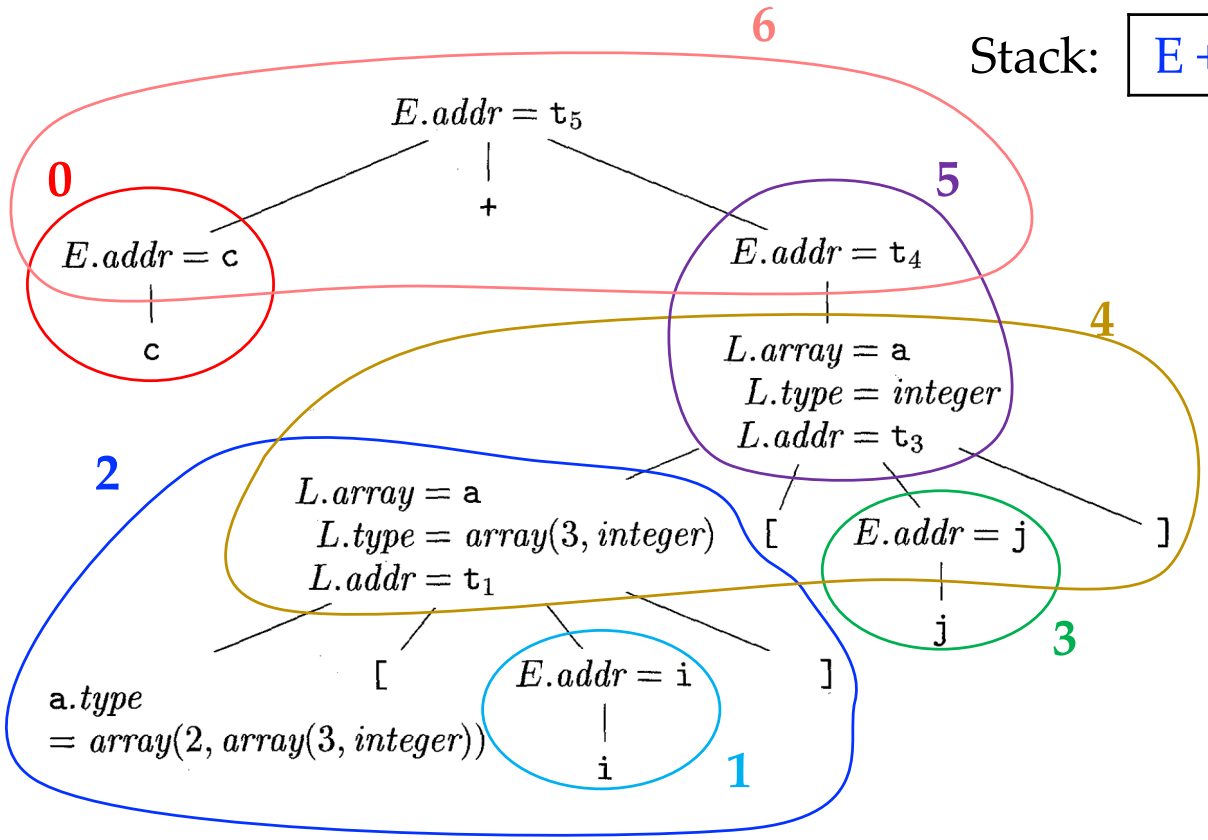
Generated code:

```
t1 = i * 12      ------- 2
t2 = j * 4       -.
                   4
t3 = t1 + t2     -'
t4 = a[t3]       ---------- 5
```

$E \rightarrow E_1 + E_2$	{ $E.addr = \textbf{new}\ Temp();$ 6 $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }
| **id**	{ $E.addr = top.get(\textbf{id}.lexeme);$ } 0 1 3
| L	{ $E.addr = \textbf{new}\ Temp();$ 5 $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

$L \rightarrow$ **id** [E]	{ $L.array = top.get(\textbf{id}.lexeme);$ $L.type = L.array.type.elem;$ 2 $L.addr = \textbf{new}\ Temp();$ $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }
| L_1 [E]	{ $L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \textbf{new}\ Temp();$ 4 $L.addr = \textbf{new}\ Temp();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Translating c + a[i][j]



Stack: E + E Reduce E + E to E

Generated code:

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

Outline

- Intermediate Representation
- Type and Declarations
- Type Checking
- Translation of Expressions
- Control Flow

Control Flow

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

- Grammar: $B \rightarrow B \parallel B \mid B \;\&\&\; B \mid !\, B \mid (B) \mid E$ **rel** $E \mid$ **true** $\mid$ **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;`

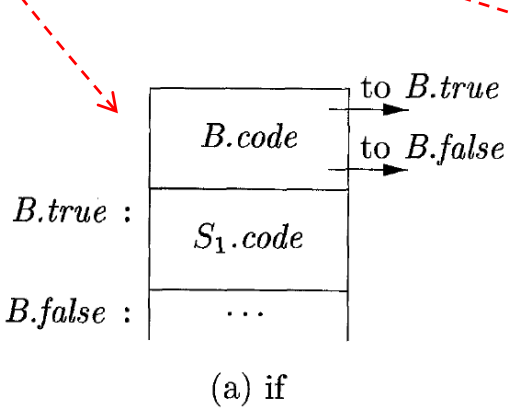
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Flow-of-Control Statements

- Grammar:
  - $S \rightarrow$ **if** (B) S_1
  - $S \rightarrow$ **if** (B) S_1 **else** S_2
  - $S \rightarrow$ **while** (B) S_1

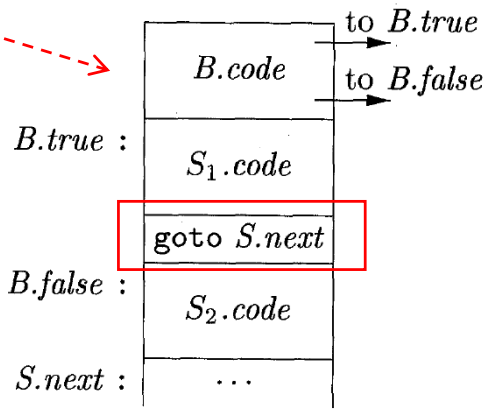
Inherited attributes:

- $B.true$: 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

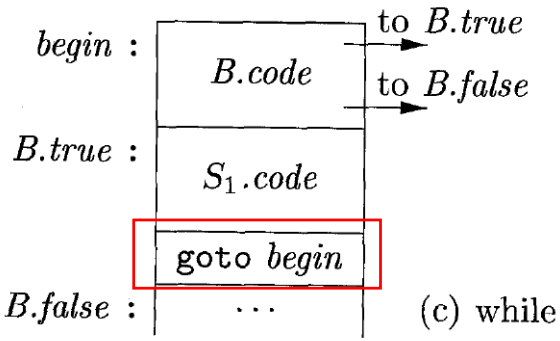


(a) if

$S.next$ is not needed



(b) if-else



(c) while

$S.next$ is not needed

SDD for Flow-of-Control Statements (1)

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

Illustrated by previous figures

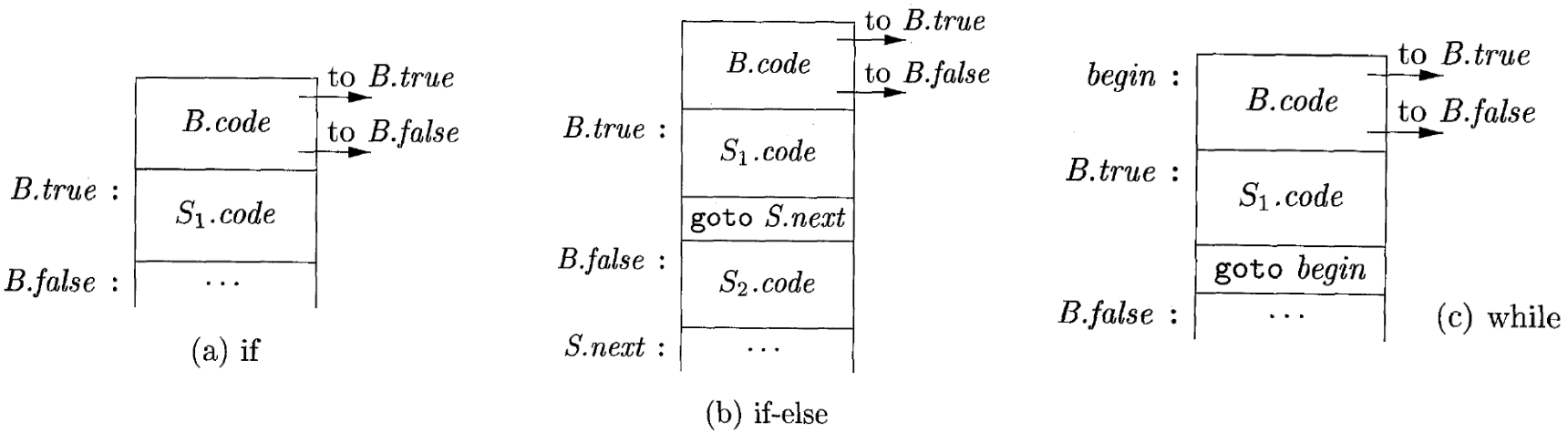
SDD for Flow-of-Control Statements (2)

Illustrated by previous figure

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

Translating Boolean Expressions in Flow-of-Control Statements

- A boolean expression B is translated into three-address instructions that evaluate B using conditional and unconditional jumps 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)

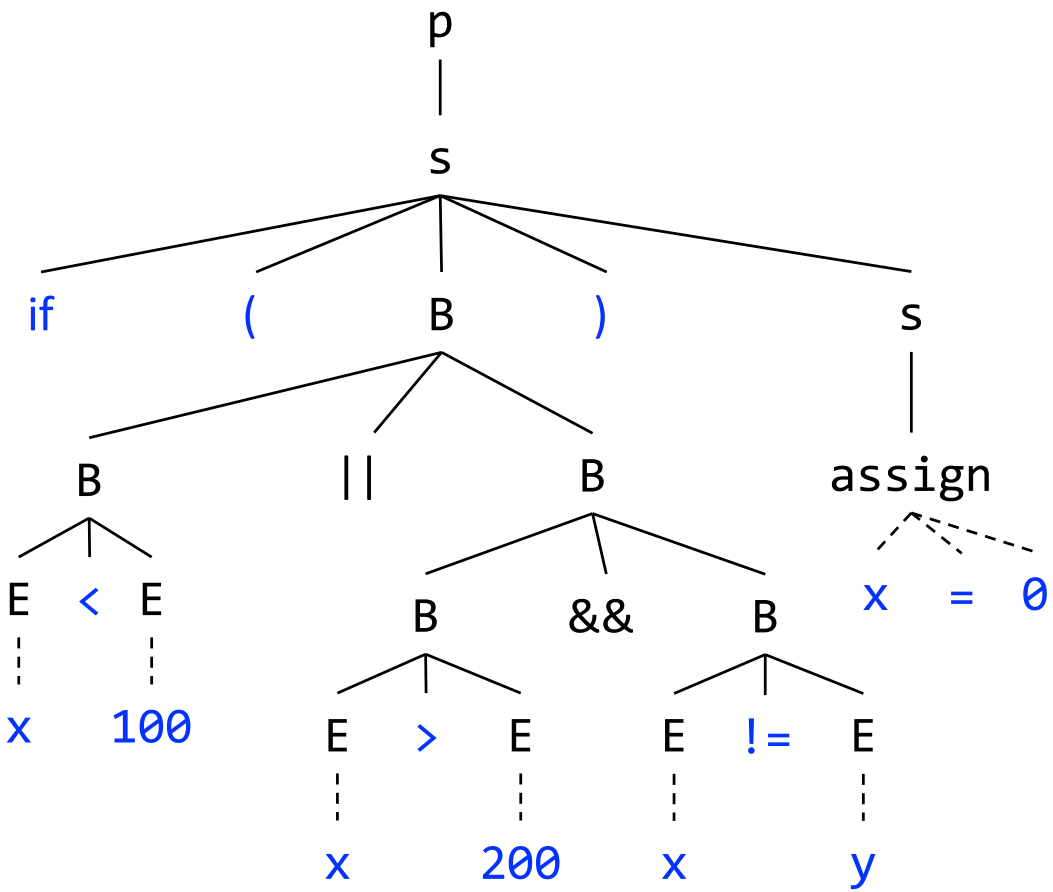
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'}\texttt{if}\text{'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'}\texttt{goto}\text{'}\ B.true)$ $\mid\mid gen(\text{'}\texttt{goto}\text{'}\ B.false)$
$B \rightarrow$ **true**	$B.code = gen(\text{'}\texttt{goto}\text{'}\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

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

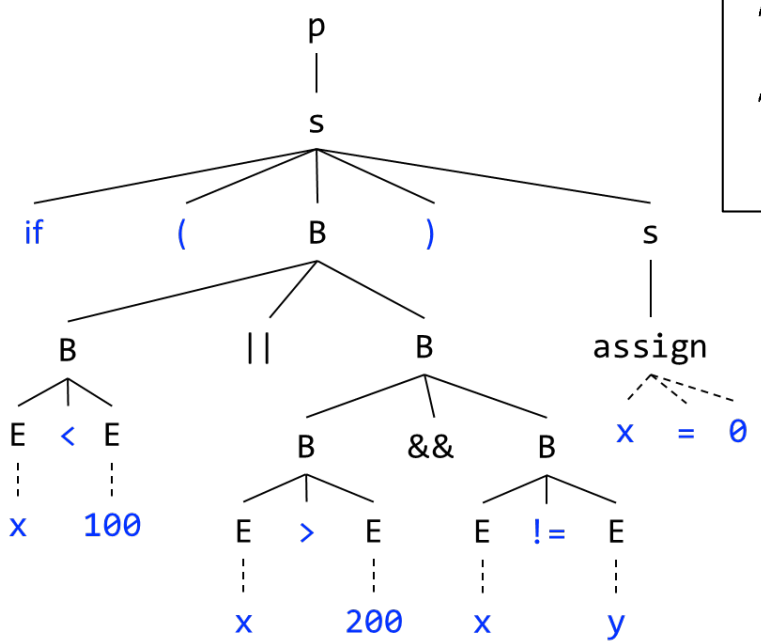
Example

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



Dashed lines mean that the reduction may consist of multiple steps

Example



PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

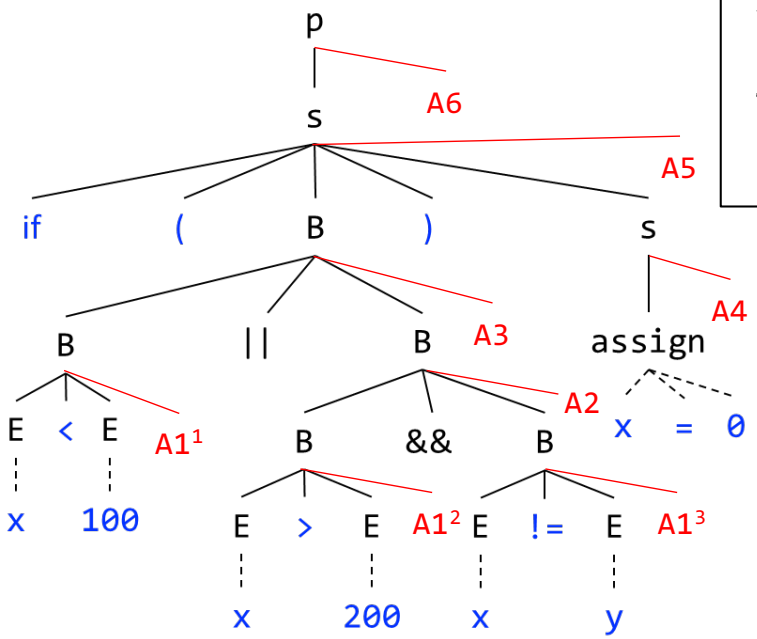
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr$ **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):

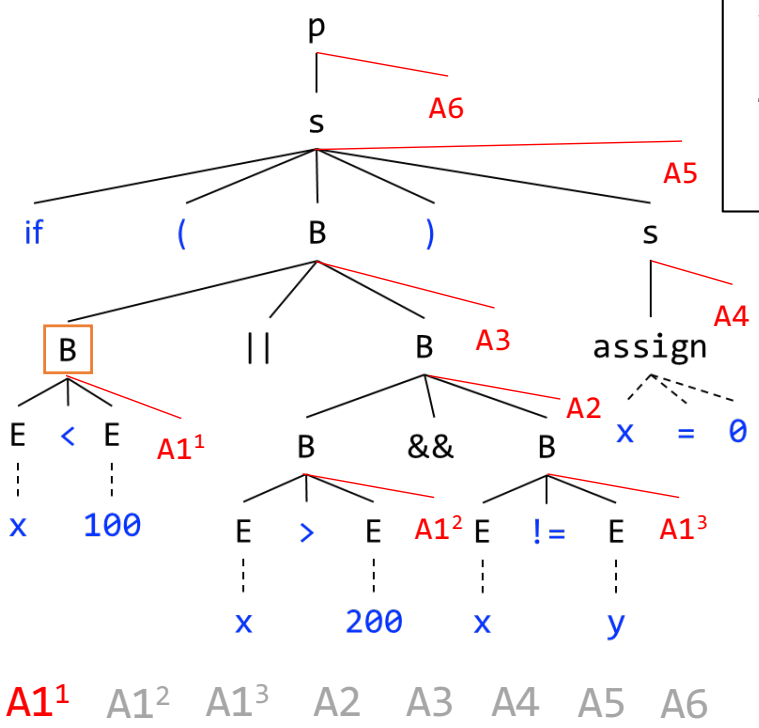
A1¹ A1² A1³ A2 A3 A4 A5 A6

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

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

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

Example



Generated code:

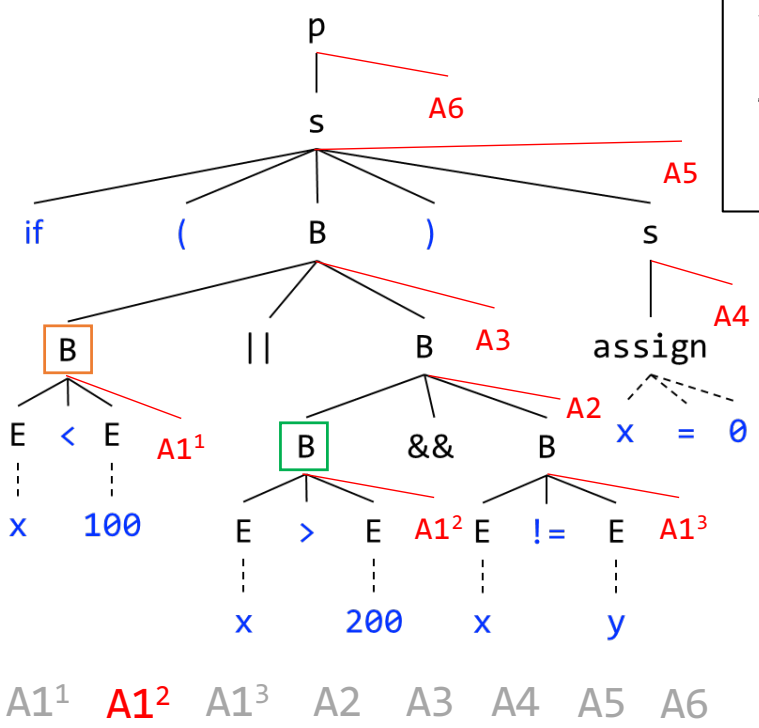
```
if x < 100 goto B.true
goto B.false
```

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

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

Example



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

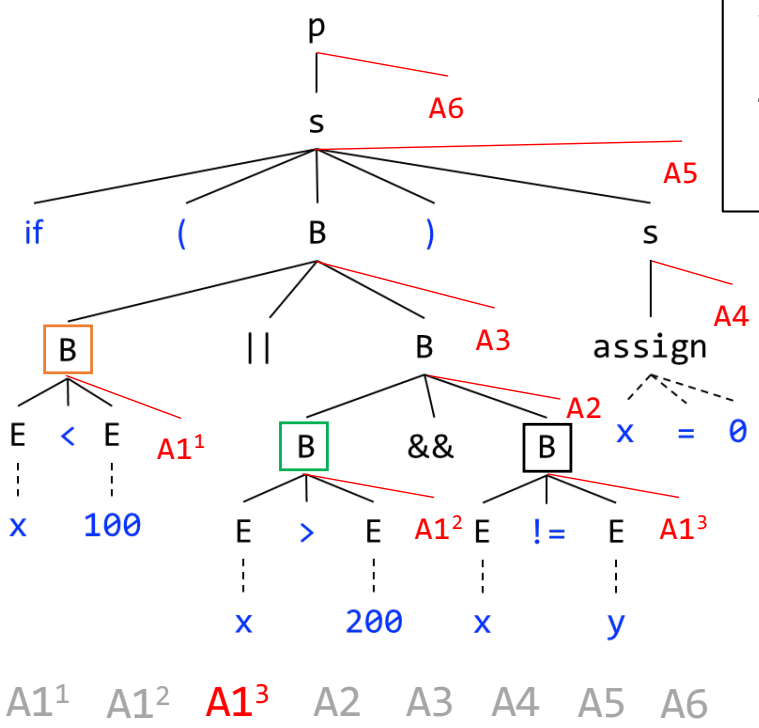
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$ A3
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

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

Generated code:

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

Example



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

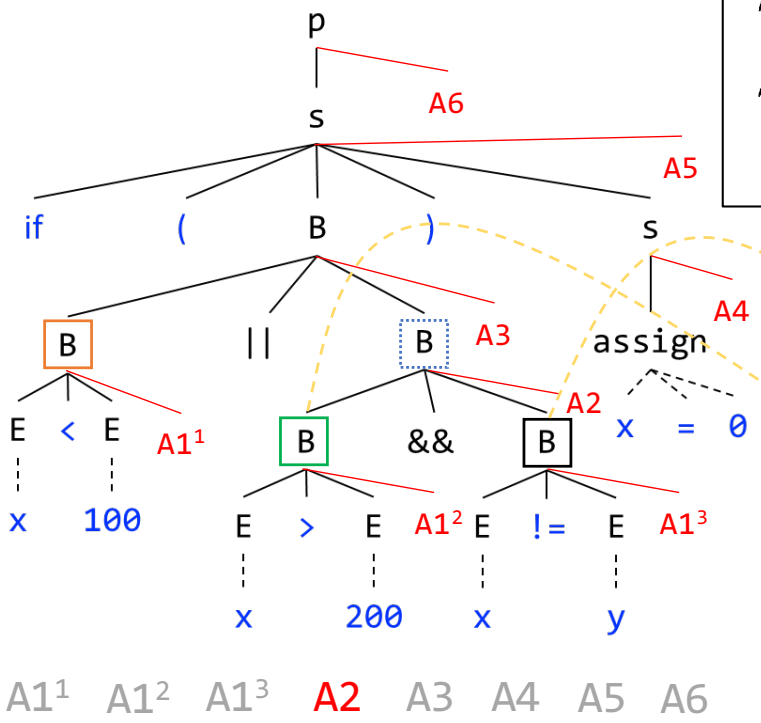
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$ A3
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

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

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
```

Example



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

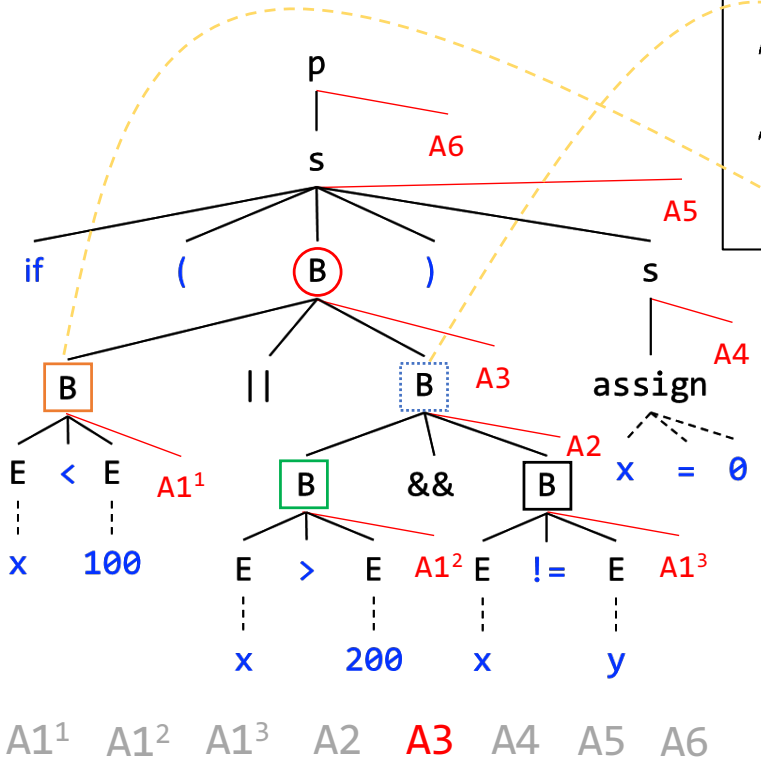
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$ A3
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

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

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



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

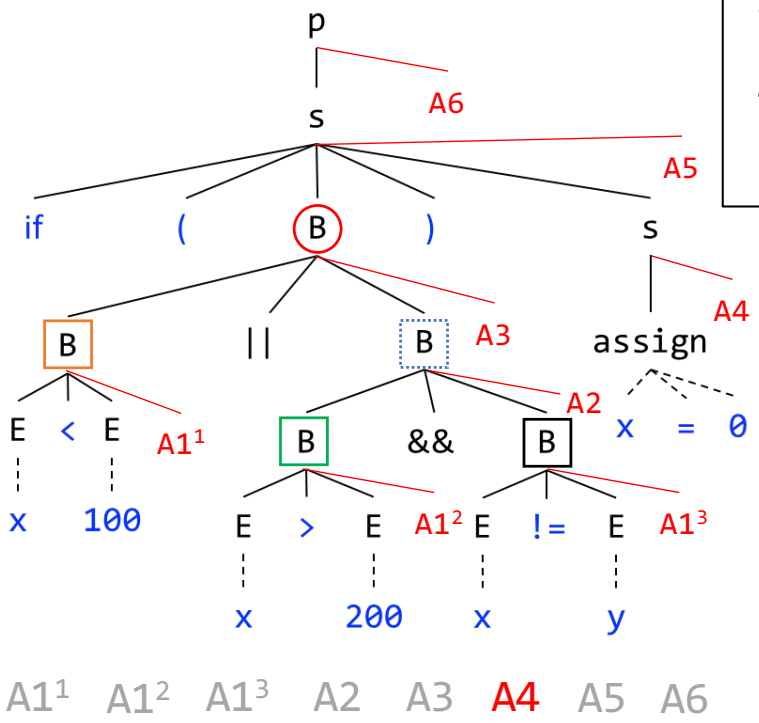
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$ A3
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

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

Generated code:

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

Example



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

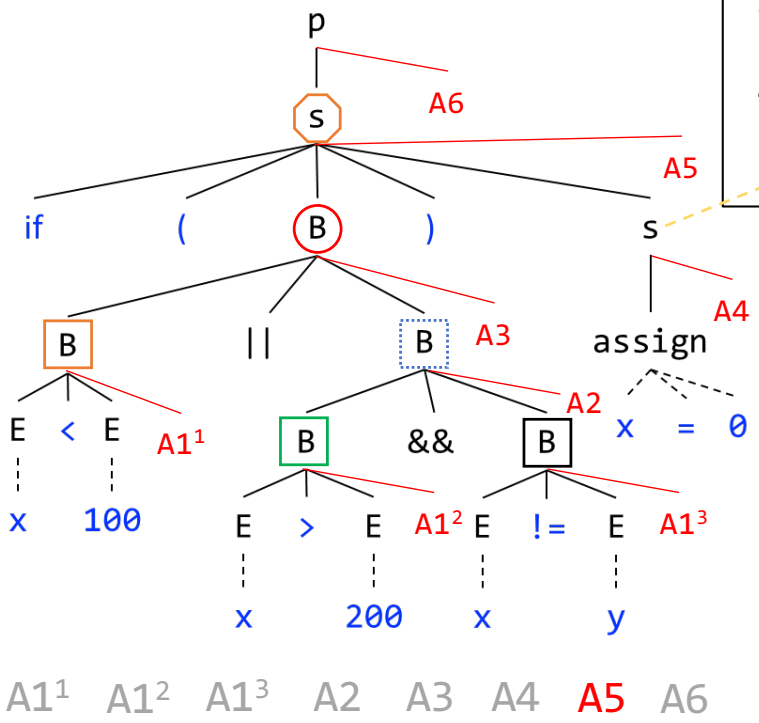
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$ A3
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

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

Generated code:

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

Example



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

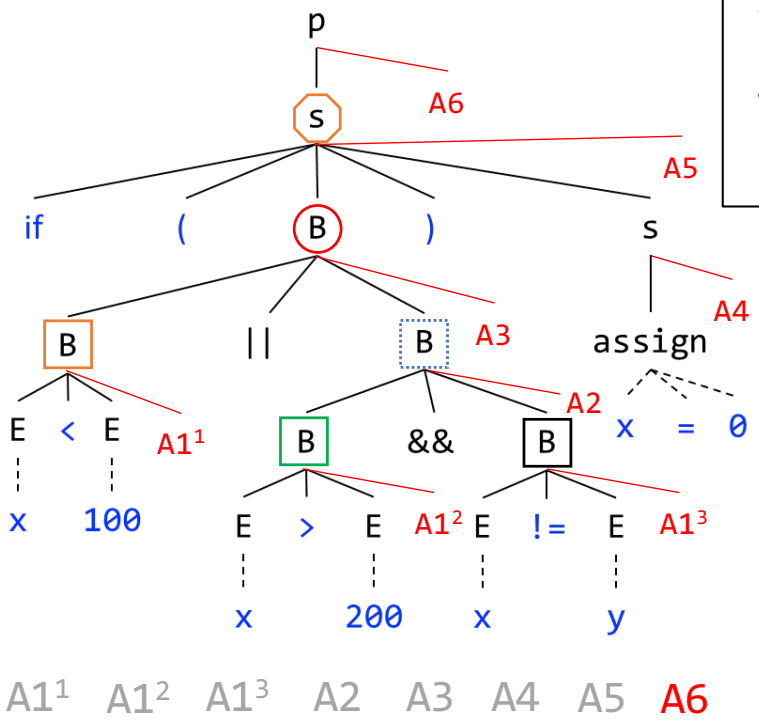
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$ A3
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

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

Generated code:

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

Example



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

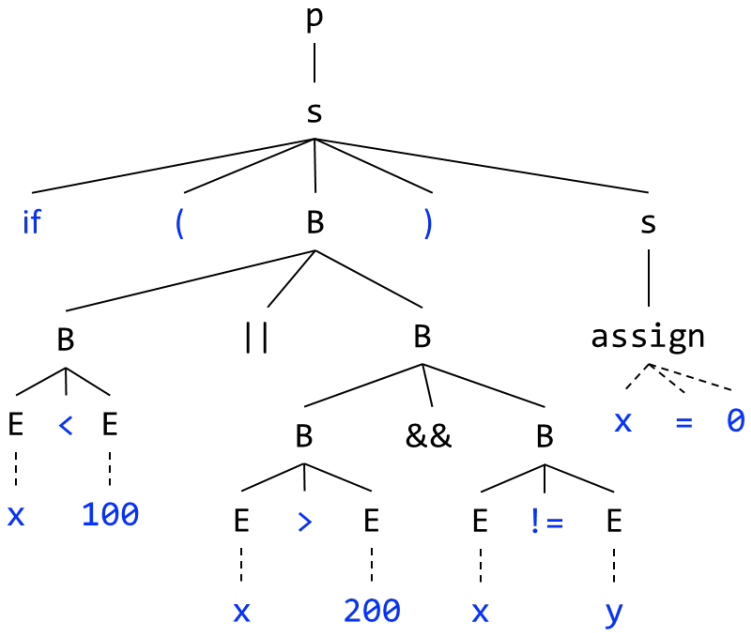
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\text{'if'}\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ \text{'goto'}\ B.true)$ $\mid\mid gen(\text{'goto'}\ B.false)$ A1

Generated code:

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

Reading Tasks

- Chapter 6 of the dragon book
  - 6.1.1 Directed Acyclic Graphs for Expressions
  - 6.2 Three-Address Code
  - 6.3 Types and Declarations
  - 6.4 Translation of Expressions
  - 6.5 Type Checking (6.5.1 – 6.5.2)
  - 6.6 Control Flow (6.6.1 – 6.6.4)