南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 5: Intermediate-Code Generation

Yepang Liu

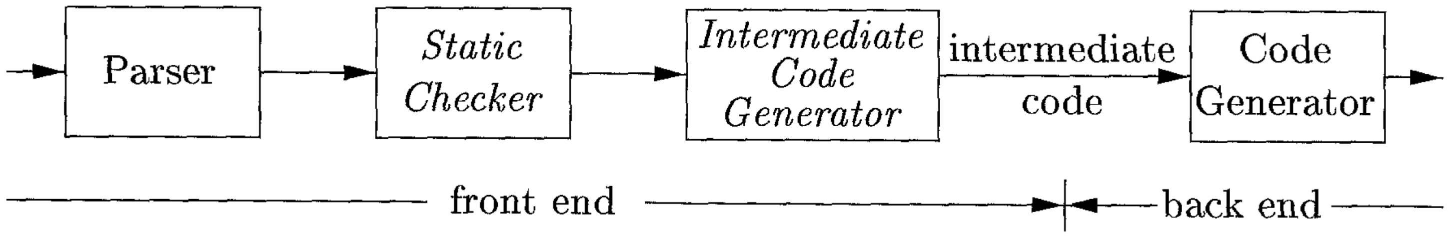
liuyp1@sustech.edu.cn

Outline

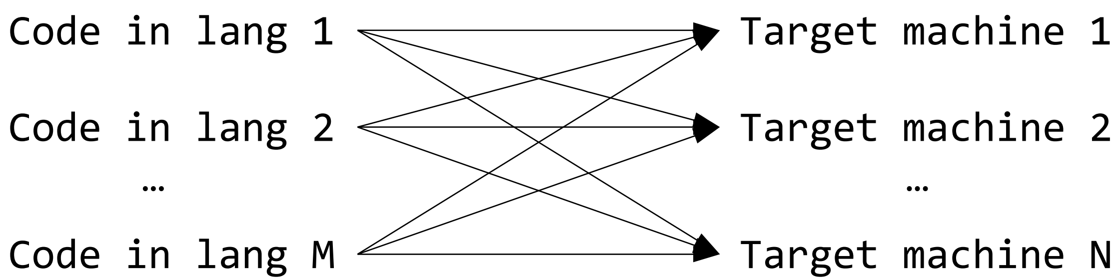
- Intermediate Representation
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

Compiler Front End

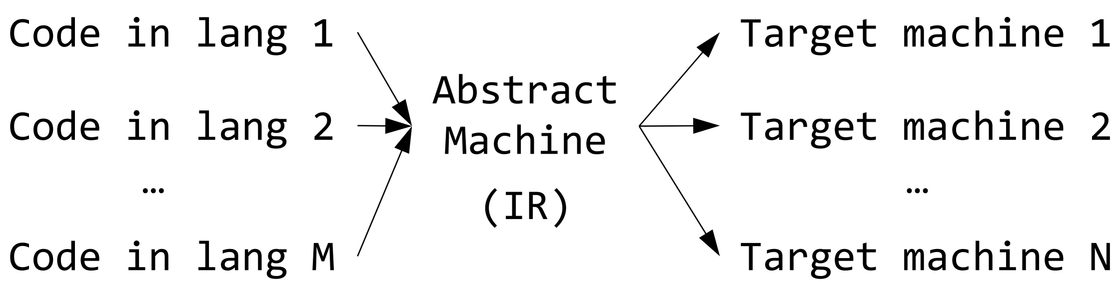
- The front end of a compiler analyzes a source program and creates an intermediate representation (IR, 中间表示), from which the back end generates target code
  - Details of the source language are confined to the front end, and details of the target machine to the back end



The Benefits of A Common IR



M * N compilers
without a common IR



M + N compilers
with a common IR

Different Levels of IRs

Source Program → High Level Intermediate Representation → ··· → Low Level Intermediate Representation → Target Code

- A compiler may construct a sequence of IR's
  - High-level IR's like syntax trees are close to the source language
    - They are suitable for machine-independent tasks like static type checking
  - Low-level IR's are close to the target machines
    - They are suitable for machine-dependent tasks like <u>register allocation</u> and <u>instruction selection</u>

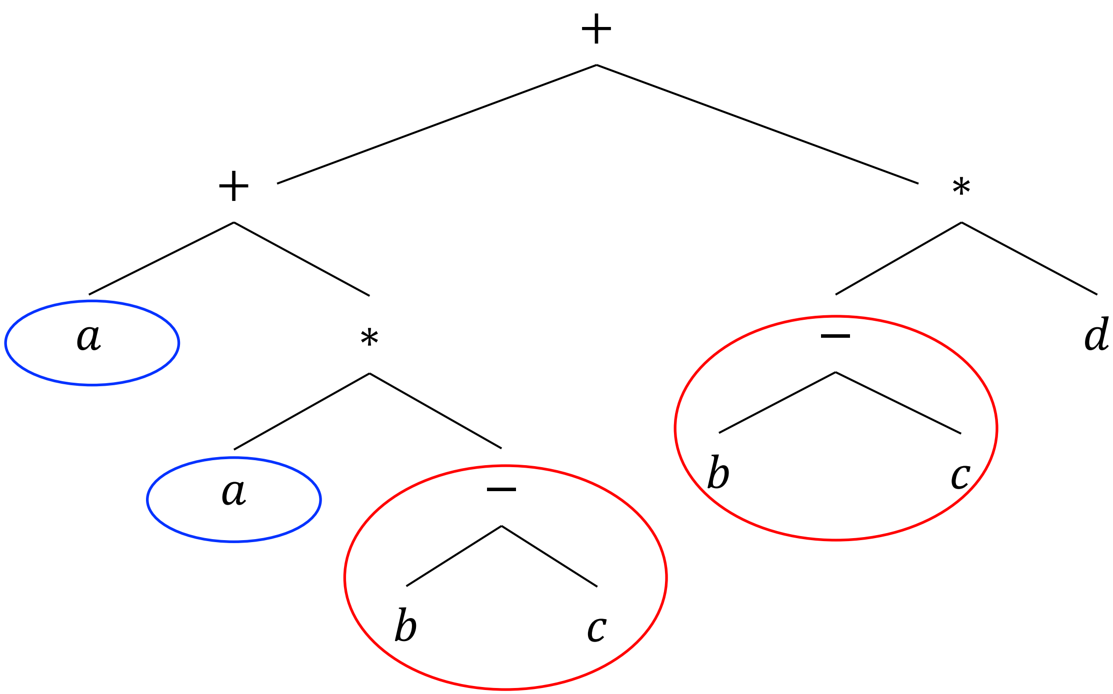
- Interesting fact: C is often used as an intermediate form. The first C++ compiler has a front end that generates C and a C compiler as a backend

Outline

- Intermediate Representation
  - DAG's for Expressions
  - Three-Address Code
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

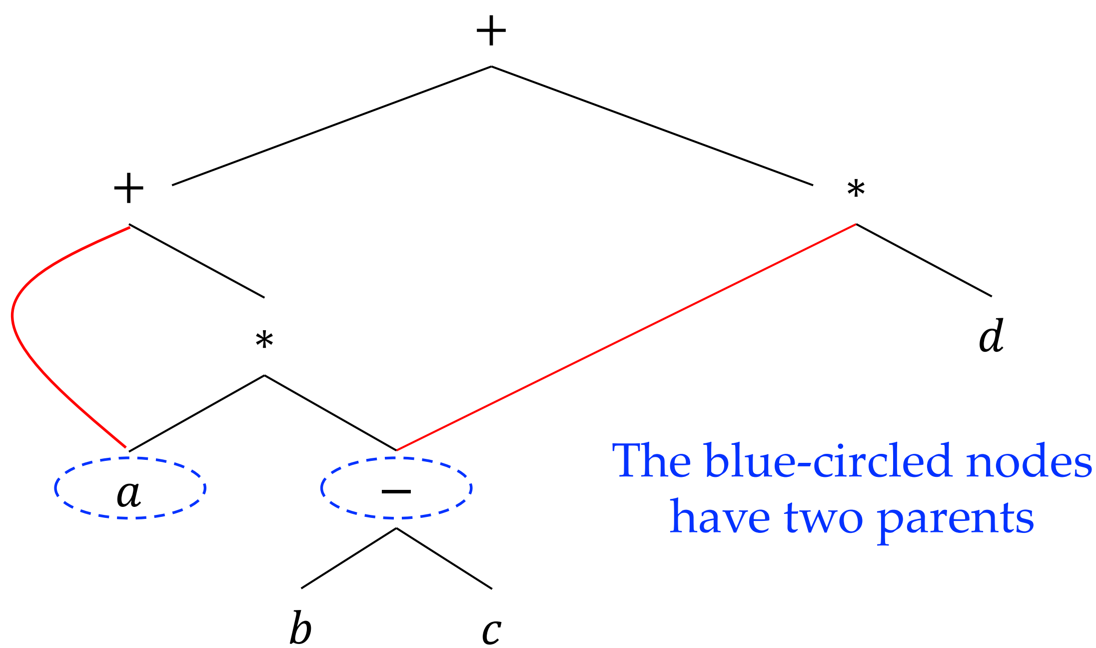
DAG's for Expressions

- In a syntax tree, the tree for a common subexpression would be replicated as many times as the subexpression appears
  - Example: $a + a * (b - c) + (b - c) * d$



DAG's for Expressions Cont.

- A *directed acyclic graph* (DAG, 有向无环图) identifies the common subexpressions and represents expressions succinctly
  - Example: $a + a * (b - c) + (b - c) * d$



Constructing DAG's

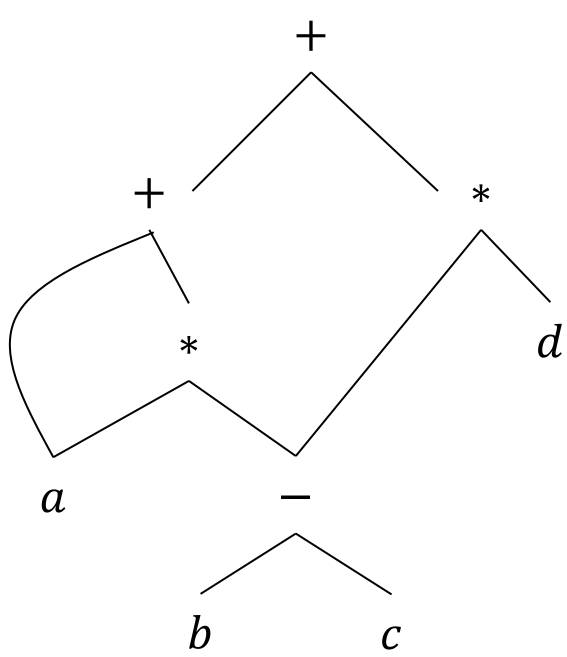
- DAG's can be constructed by the same SDD that constructs syntax trees
- **The difference:** When constructing DAG's, a new node is created if and only if there is no existing identical node

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

Special "new":
Reuse existing nodes when possible

Constructing DAG's Cont.

- The construction steps $a + a * (b - c) + (b - c) * d$



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

Node reuse

Outline

- Intermediate Representation
  - DAG's for Expressions
  - Three-Address Code
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

Three-Address Code (三地址代码)

- In three-address code, there is at most one operator on the right side of an instruction
  - Instructions are often in the form $x = y \ op \ z$
- Operands (or addresses) can be:
  - Names in the source programs
  - Constants: a compiler must deal with many types of constants
  - Temporary names generated by a compiler

Instructions (1)

1. Assignment instructions:
   - $x = y\ op\ z$, where op is a binary arithmetic/logical operation
   - $x = op\ y$, where op is a unary operation
2. Copy instructions: $x = y$
3. Unconditional jump instructions: goto L, where L is a label of the jump target
4. Conditional jump instructions:
   - if x goto L
   - ifFlase x goto L
   - if $x\ relop\ y$ goto L

Instructions (2)

5. Procedural calls and returns
   - param x_1
   - …
   - param x_n
   - call p, n (procedure call)
   - y = call p, n (function call)
   - return y

6. Indexed copy instructions: $x = y[i] \quad x[i] = y$
   - Here, $y[i]$ means the value in the location i memory units beyond location y

Instructions (3)

7. Address and pointer assignment instructions:
   - $x = \&y$ (set the r-value of x to be the l-value of y)
   - $x =* y$ (set the r-value of x to be the content stored at the location pointed to by y; y is a pointer whose r-value is a location)
   - $* x = y$ (set the r-value of the object pointed to by x to the r-value of y)

A variable has l-value and r-value:

- **L-value (location)** refers to the memory location, which identifies an object.
- **R-value (content)** refers to data value stored at some address in memory.

Example

- Source code: `do i = i + 1; while (a[i] < v);`

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

Assuming each array element takes 8 units of space

Representation of Instructions

- In a compiler, three-address instructions can be implemented as objects/records with fields for the operator and the operands

- Three typical representations:
  - Quadruples (四元式表示方法)
  - Triples (三元式表示方法)
  - Indirect triples (间接三元式表示方法)

Quadruples (四元式)

- A *quadruple* has four fields
  - General form: *op* arg_1 arg_2 *result*
  - *op* contains an internal code for the operator
  - arg_1, arg_2, *result* are addresses (operands)
  - Example: $x = y + z$ ➔ $+ \quad y \quad z \quad x$

- Some exceptions:
  - Unary operators like $x = minus\ y$ or $x = y$ do not use arg_2
  - *param* operators use neither arg_2 nor *result*
  - Conditional/unconditional jumps put the target label in *result*

Quadruples Example

- Assignment statement: $a = b * -c + b * -c$

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

(a) Three-address code

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

Temporaries

(b) Quadruples

The result field is used primarily for temporary names.
Temporary names waste space (symbol table entries)

Triples (三元式)

- A *triple* has only three fields: *op*, arg_1, arg_2
- We refer to the result of an operation *x op y* by its position without generating temporary names (an optimization over quadruples)

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

Three-address code

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

Quadruples

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

Triples

Quadruples vs. Triples

- In optimizing compilers, instructions are often moved around

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

Swap 1 and 2

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	minus	c		t_3
2	*	b	t_1	t_2
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

Quadruples' advantage

The instructions that use t_1 and t_3 are not affected

Quadruples vs. Triples

- In optimizing compilers, instructions are often moved around

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

Swap 1 and 2 →

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

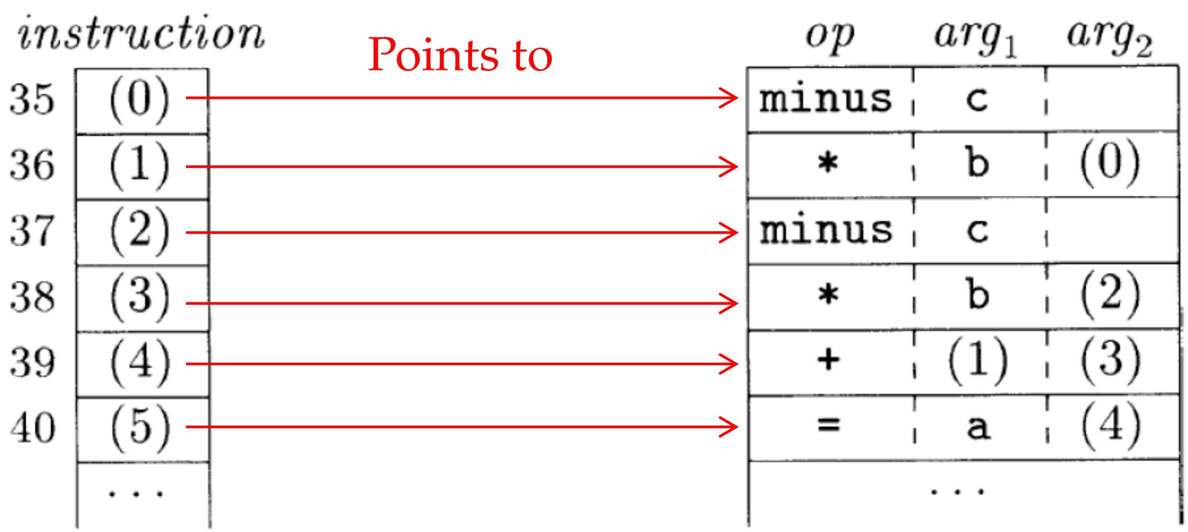
Are they still correct after swapping?

Triples' problem

The instructions now refer to wrong results; The positions need to be updated.

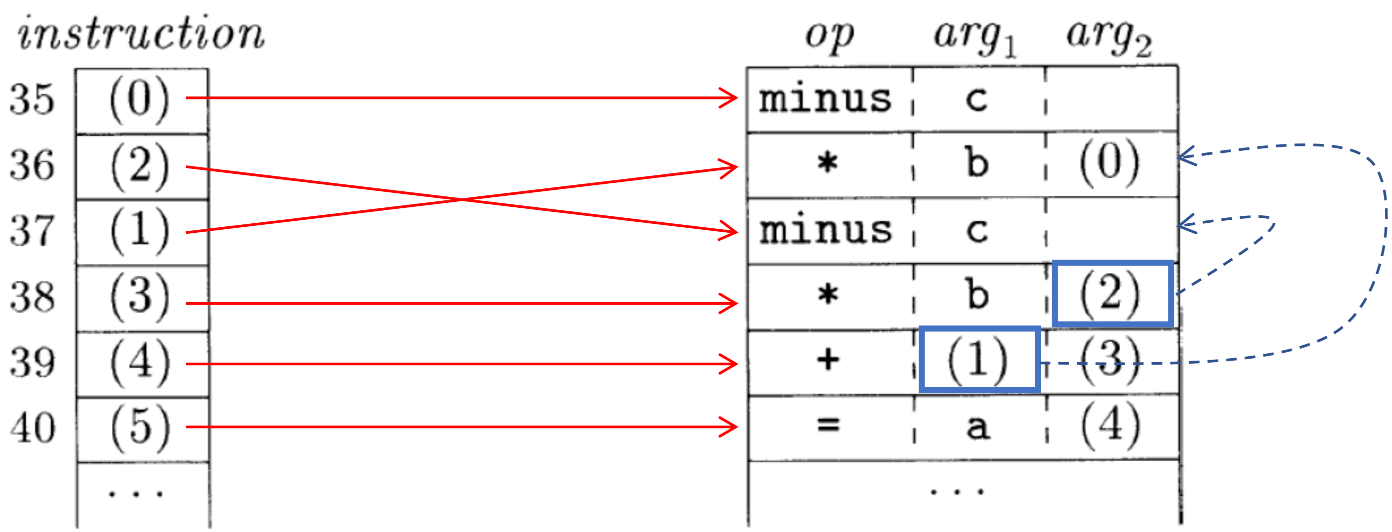
Indirect Triples (间接三元式)

- *Indirect triples* consist of a list of pointers to triples



Indirect Triples (间接三元式)

- An optimization can move an instruction by reordering the *instruction* list



Swapping pointers!

The triples are not affected.

Static Single-Assignment Form

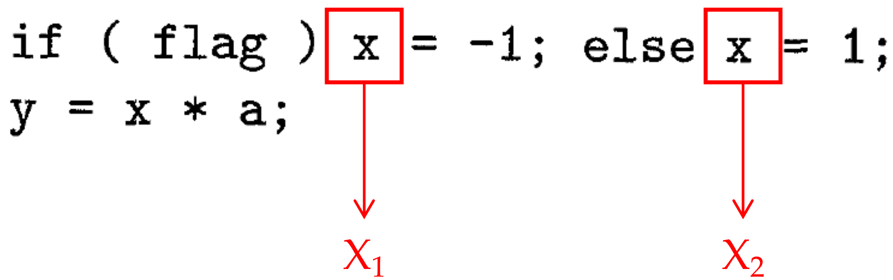
- Static single-assignment form (SSA, 静态单赋值形式) is an IR that facilitates certain code optimizations
- In SSA, each name receives a single assignment

$$
\begin{array}{ll}
\begin{array}{l}
\boxed{p} = a + b \\
q = p - c \\
\boxed{p} = q * d \\
p = e - p \\
q = p + q
\end{array}
&
\qquad
\begin{array}{l}
\boxed{p_1} = a + b \\
q_1 = p_1 - c \\
\boxed{p_2} = q_1 * d \\
p_3 = e - p_2 \\
q_2 = p_3 + q_1
\end{array}
\end{array}
$$

(a) Three-address code. (b) Static single-assignment form.

Static Single-Assignment Form

- The same variable may be defined in two control-flow paths



Which name should we use in y = x * a?

Static Single-Assignment Form

- The same variable may be defined in two control-flow paths

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

- SSA uses a notational convention called ϕ-function to combine the two definitions of x

```
if ( flag ) x₁ = -1; else x₂ = 1;
x₃ = φ(x₁,x₂);  // x1 if control flow passes through the true path; otherwise x2
y = x₃ * a;
```

Outline

- Intermediate Representation
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

Types and Type Checking

- *Data type* or simply *type* tells a compiler or interpreter how the programmers intend to use the data

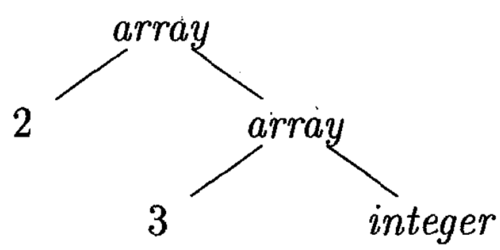
- The usefulness of type information
  - Find faults in the source code
  - Determine the storage needed for a name at runtime
  - Calculate the address of an array element
  - Insert type conversions
  - Choose the right version of some arithmetic operator (e.g., `fadd`, `iadd`)

- *Type checking* (类型检查) uses logical rules to make sure that the types of the operands match the type expectation by an operator

Type Expressions (类型表达式)

- Types have structure, which can be represented by *type expressions*
  - A type expression is either a basic type, or
  - Formed by applying a *type constructor* (类型构造算子) to a type expression

- $array(2, array(3, integer))$ is the type expression for `int[2][3]`
  - $array$ is a type constructor with two arguments: a number, a type expression



The Definition of Type Expression

- A basic type is a type expression
  - $boolean$, $char$, $integer$, $float$, and $void$, …
- A type name (e.g., name of a class) is a type expression
- A type expression can be formed
  - By applying the $array$ type constructor to a number and a type expression
  - By applying the $record$ type constructor to the field names and their types
  - By applying the $\rightarrow$ type constructor for function types

- If s and t are type expressions, then their Cartesian product $s{\times}t$ is a type expression (this is introduced for completeness, can be used to represent a list of types such as function parameters)
- Type expressions may contain type variables (e.g., those generated by compilers) whose values are type expressions

Type Equivalence

- Type checking rules usually have the following form

If two type expressions are equivalent
then return a given type
else return **type_error**

Code under analysis:
`a + b`

- The key is to define when two type expressions are equivalent
  - The main difficulty arises from the fact that most modern languages allow the naming of user-defined types
    - In C/C++, type naming is achieved by the `typedef` statement

Name Equivalence (名等价)

- Treat named types as basic types; names in type expressions are not replaced by the exact type expressions they define
- Two type expressions are name equivalent if and only if they are identical (represented by the same syntax tree, with the same labels)

```
typedef struct {
    int data[100];
    int count;
} Stack;
```

```
typedef struct {
    int data[100];
    int count;
} Set;
```

```
Code under analysis:

Stack x, y;

Set r, s;

x = y;

r = s;

x = r;
```

http://web.eecs.utk.edu/~bvanderz/teaching/cs365Sp14/notes/types.html

Structural Equivalence (结构等价)

- For named types, replace the names by the type expressions and recursively check the substituted trees

```
typedef struct {
    int data[100];
    int count;
} Stack;
```

```
typedef struct {
    int data[100];
    int count;
} Set;
```

```
Code under analysis:

Stack x, y;

Set r, s;

x = y;

r = s;

x = r;
```

Declarations (变量声明)

- The grammar below deals with basic, array, and record types
  - Nonterminal *D* generates a sequence of declarations
  - *T* generates basic, array, or record types
  - A record type is a sequence of declarations for the fields of the record, surrounded by curly braces
  - *B* generates one of the basic types: `int` and `float`
  - *C* generates sequences of one or more integers, each surrounded by brackets

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \ \mid\ \epsilon \\
T & \rightarrow & B\ C \ \mid\ \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \ \mid\ \mathbf{float} \\
C & \rightarrow & \epsilon \ \mid\ [\ \mathbf{num}\ ]\ C
\end{array}
$$

Storage Layout for Local Names (局部变量的存储布局)

- From the type of a name, we can decide the amount of memory needed for the name at run time
  - The *width* (宽度) of a type: # memory units needed for an object of the type
  - For data of varying lengths, such as strings, or whose size cannot be determined until run time, such as dynamic arrays, we only reserve a fixed amount of memory for a pointer to the data

- For local names of a function, we always assign contiguous bytes*
  - For each such name, at compile time, we can compute a relative address
  - Type information and relative addresses are stored in symbol table

* This follows the principle of proximity and is mainly for performance considerations.

An SDT for Computing Types and Their Widths

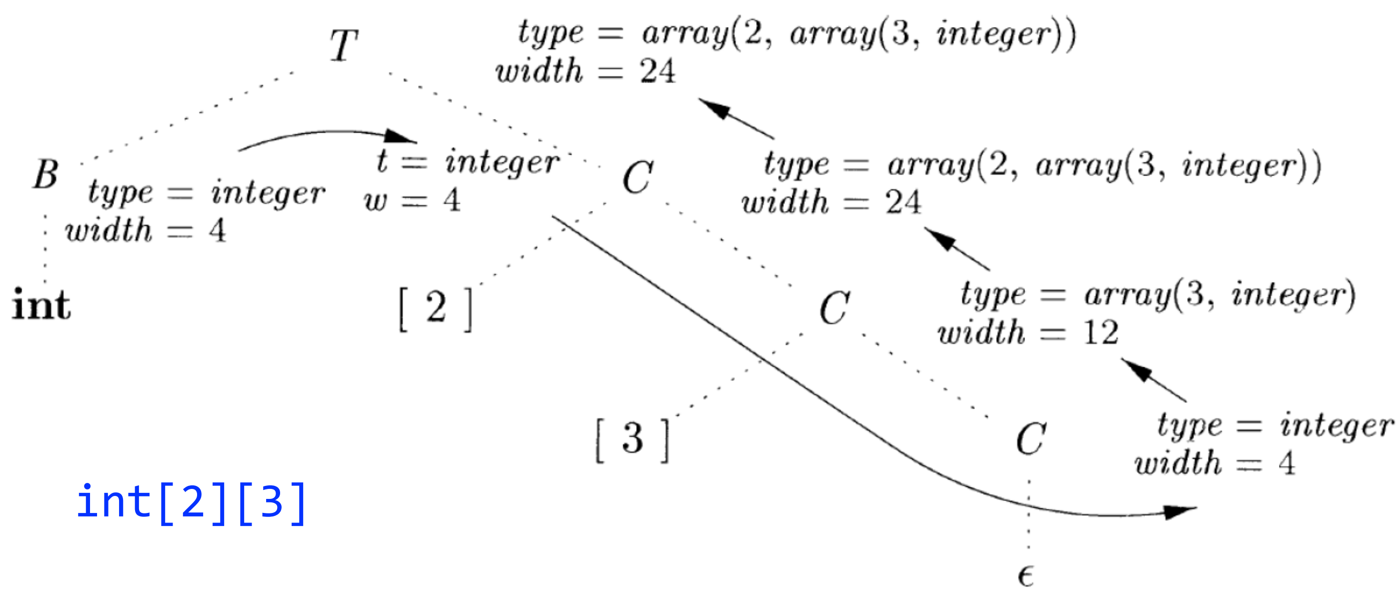
- Synthesized attributes: *type, width*
- Global variables t and w pass type and width information from a B node in a parse tree to the node for the production $C \rightarrow \epsilon$
  - In an SDD, t and w would be C's inherited attributes (the SDD is L-attributed)*

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

This SDT can be implemented during recursive-descent parsing

Translation Process Example

- **Recall the translation during recursive-descent parsing**
  - Use the arguments of function *A*() to pass nonterminal *A*'s inherited attributes*
  - Evaluate and Return the synthesized attributes of *A* when the *A*() completes

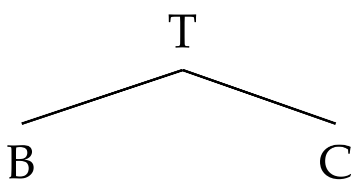


* In our example, we use global variables *t* and *w*

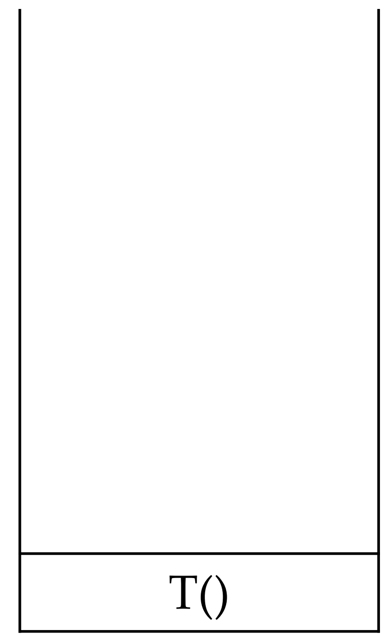
Translation Process Example

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	$\{T.type{=}C.type;\ T.width{=}C.width;\}$
B	$\rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	[**num**] C_1	$\{\ C.type{=}\ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$

Input string: `int[2][3]`



Step 1: Rewrite T using $T \rightarrow BC$

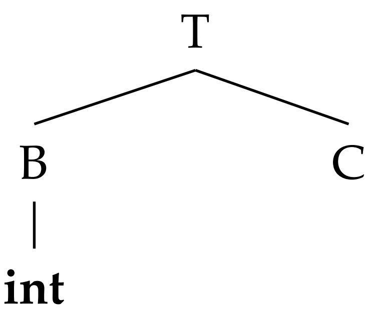


Call stack

Translation Process Example

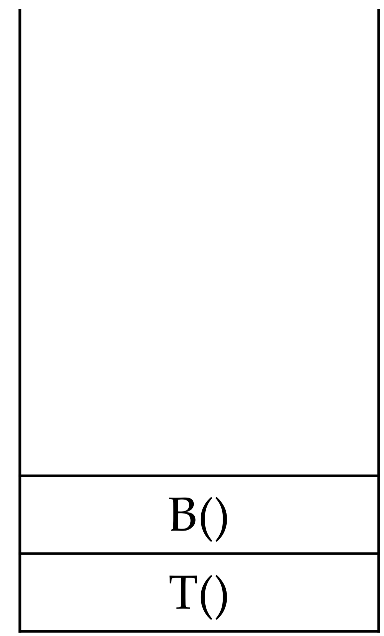
Step 2:
- Rewrite *B* using *B* → **int**
- Match input

Input string: `int[2][3]`



Step 2:

- Rewrite *B* using *B* → **int**
- Match input

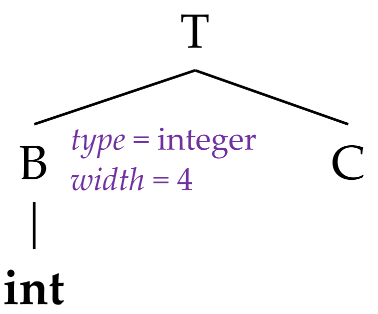


Call stack

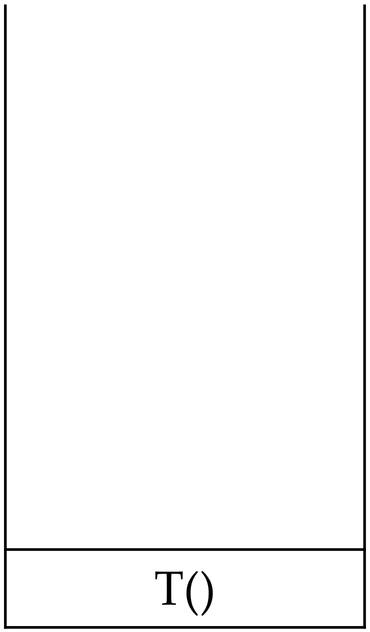
Translation Process Example

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	$\{T.type = C.type;\ T.width = C.width;\ \}$
B	$\rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	[**num**] C_1	$\{\ C.type = array(\mathbf{num}.value,\ C_1.type);\ C.width = \mathbf{num}.value \times C_1.width;\ \}$

Input string: `int[2][3]`



Step 3:
- B() returns
- Execute semantic action



Call stack

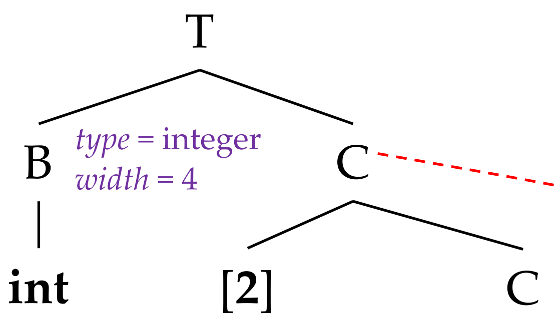
$B \rightarrow \mathbf{int} \qquad \{\ B.type = integer;\ B.width = 4;\ \}$

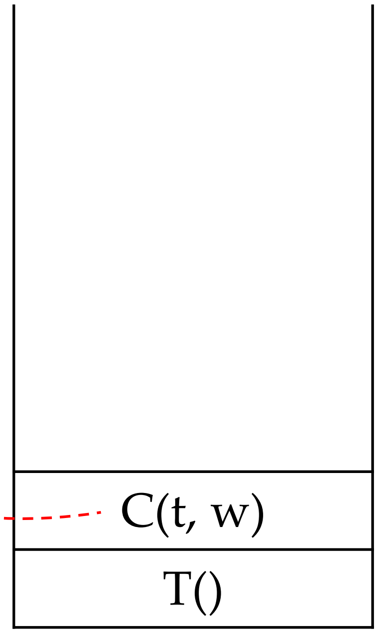
Translation Process Example

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	$\{T.type{=}C.type;\ T.width{=}C.width;\ \}$
B	$\rightarrow$	$\textbf{int}$	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	$\textbf{float}$	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	$[\ \textbf{num}\ ]\ C_1$	$\{\ C.type = array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width;\ \}$

Input string: `int[2][3]`

t = integer
$w = 4$





Call stack

Step 4:

- Execute semantic action
- Rewrite C using $C \rightarrow [\textbf{num}]C$
- Match input

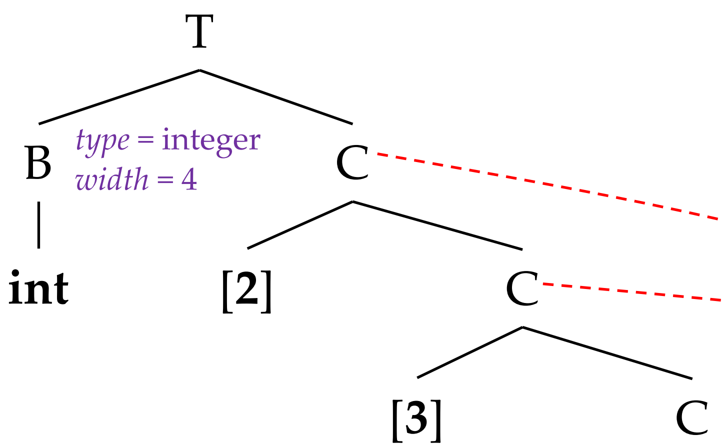
$$T \rightarrow B \qquad \boxed{\{\ t = B.type;\ w = B.width;\ \}}$$
$$\qquad C \qquad \{T.type{=}C.type;\ T.width{=}C.width;\ \}$$

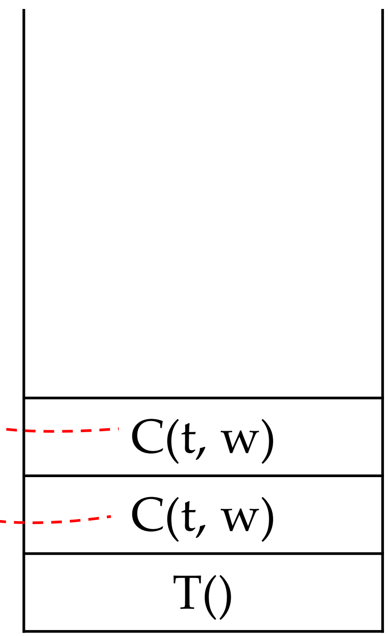
Translation Process Example

$T \rightarrow B$		$\{ t = B.type;\ w = B.width; \}$
C		$\{T.type{=}C.type;\ T.width{=}C.width; \}$
$B \rightarrow$ **int**		$\{ B.type = integer;\ B.width = 4; \}$
$B \rightarrow$ **float**		$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow \epsilon$		$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow$ [**num**] C_1		$\{ C.type{=}\ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

Input string: `int[2][3]`

t = integer
w = 4





Call stack

Step 5:

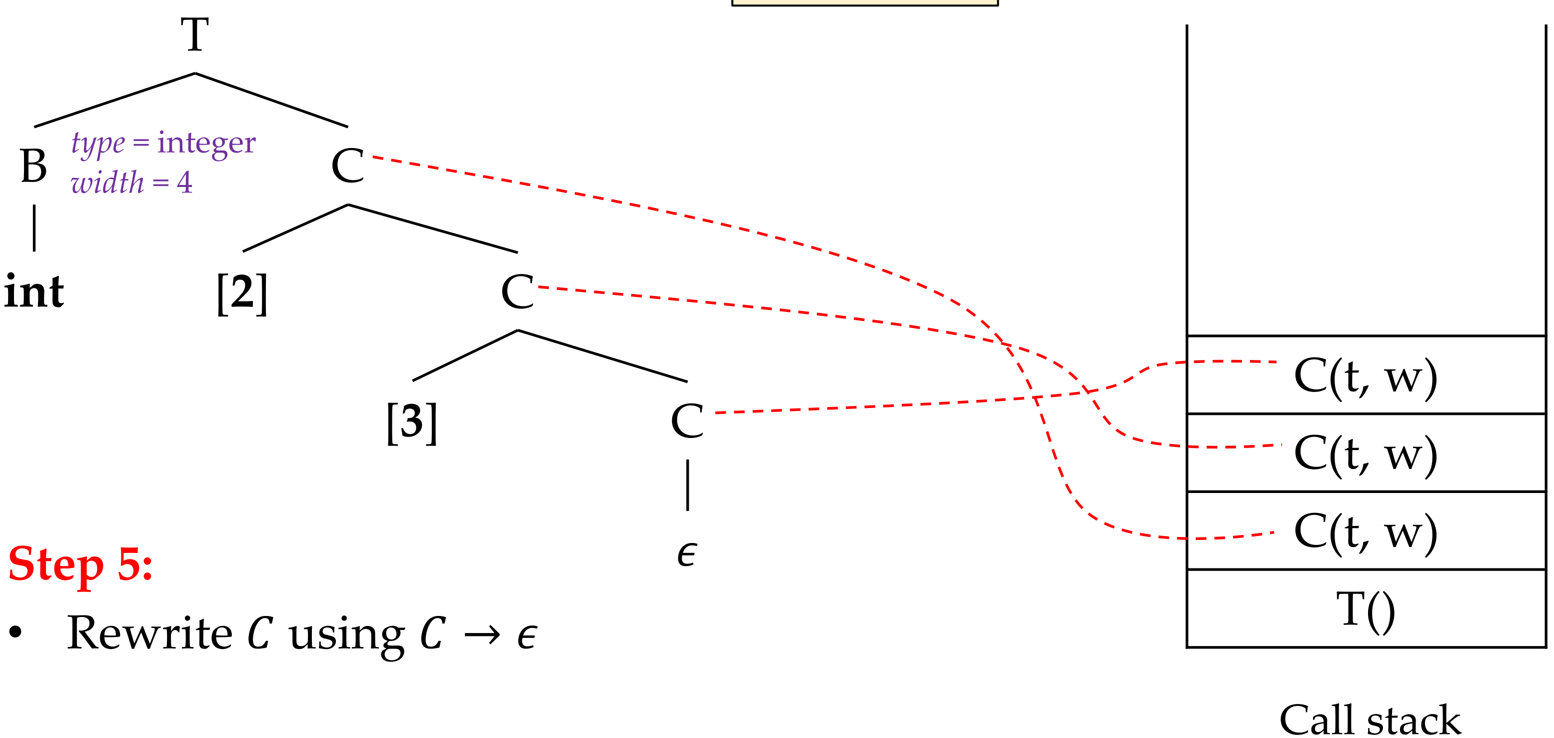
- Rewrite C using $C \rightarrow$ [**num**]C
- Match input

Translation Process Example

$T \rightarrow B$		$\{ t = B.type;\ w = B.width; \}$
C		$\{T.type = C.type;\ T.width = C.width; \}$
$B \rightarrow$ **int**		$\{ B.type = integer;\ B.width = 4; \}$
$B \rightarrow$ **float**		$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow \epsilon$		$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$		$\{ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

Input string: `int[2][3]`

t = integer
w = 4



Step 5:

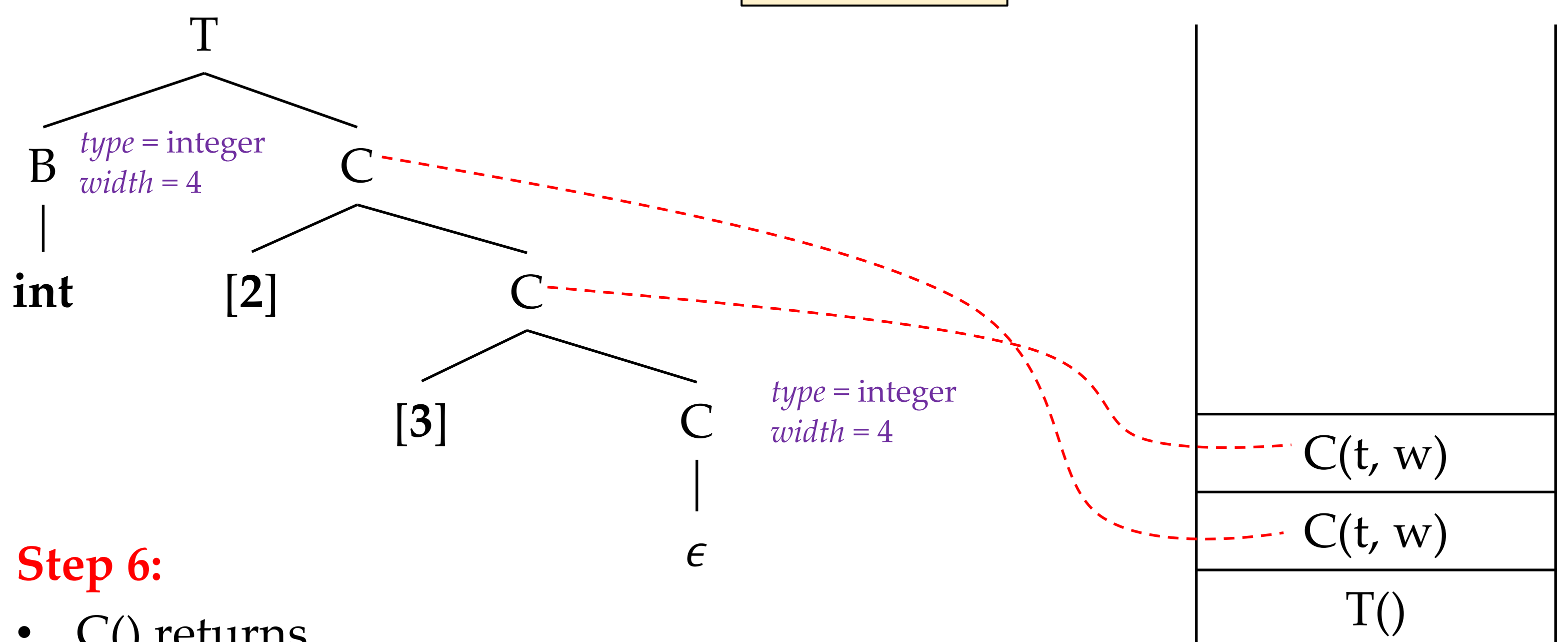
- Rewrite C using $C \rightarrow \epsilon$

Translation Process Example

T	$\rightarrow$	B C	$\{\ t = B.type;\ w = B.width;\ \}$ $\{T.type{=}C.type;\ T.width{=}C.width;\}$
B	$\rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	[**num**] C_1	$\{\ C.type{=}\ array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width;\ \}$

Input string: `int[2][3]`

t = integer
w = 4



Call stack

Step 6:

- C() returns
- Execute semantic action

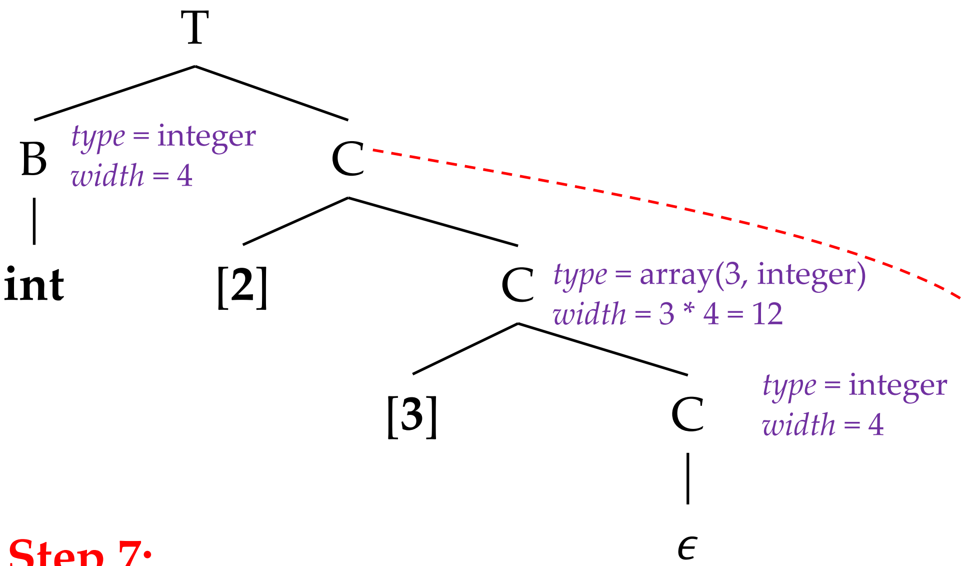
$C \rightarrow \epsilon$ $\boxed{\{\ C.type = t;\ C.width = w;\ \}}$

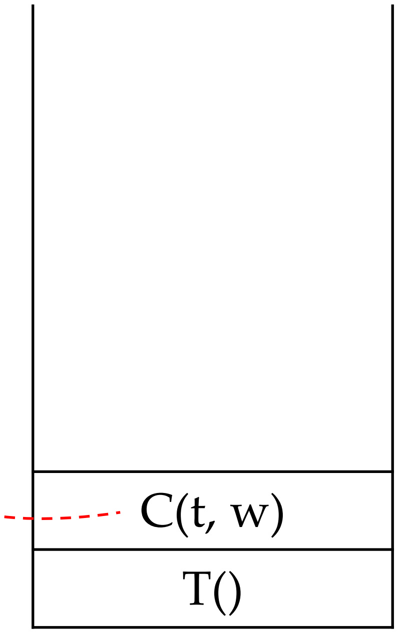
Translation Process Example

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	$\{T.type{=}C.type;\ T.width{=}C.width;\}$
B	$\rightarrow$	$\mathbf{int}$	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	$\mathbf{float}$	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	$[\ \mathbf{num}\ ]\ C_1$	$\{\ C.type{=}\ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width;\ \}$

Input string: `int[2][3]`

t = integer
w = 4





Call stack

Step 7:

- C() returns
- Execute semantic action

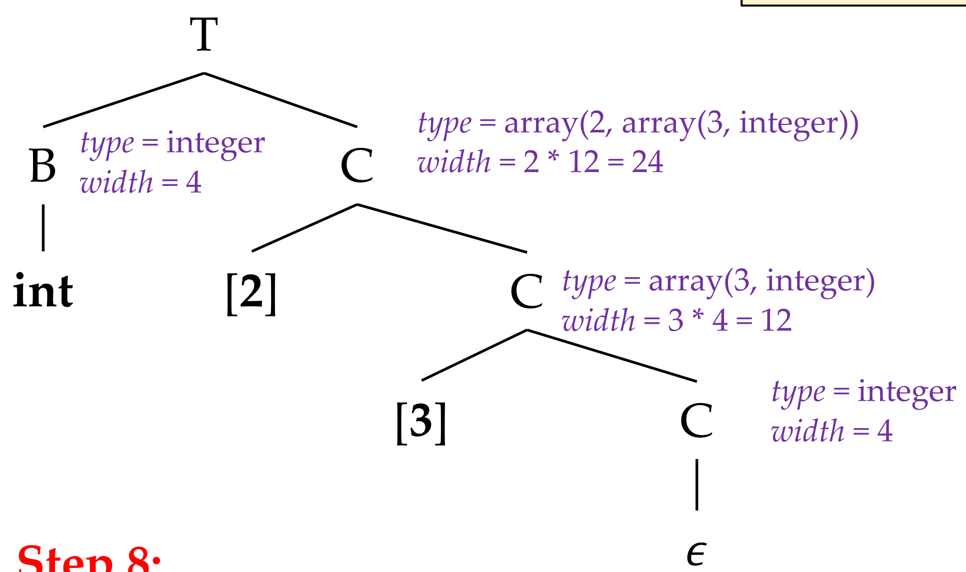
$$C \rightarrow [\ \mathbf{num}\ ]\ C_1 \quad \boxed{\{\ C.type{=}\ array(\mathbf{num}.value,\ C_1.type);\ C.width = \mathbf{num}.value \times C_1.width;\ \}}$$

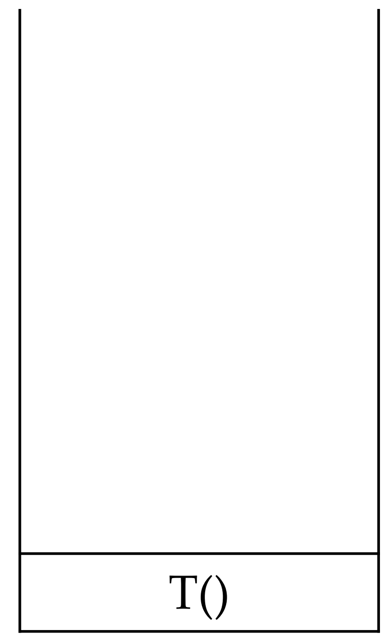
Translation Process Example

T	$\rightarrow$	B	$\{ t = B.type;\ w = B.width; \}$
		C	$\{T.type = C.type;\ T.width = C.width; \}$
B	$\rightarrow$	**int**	$\{ B.type = integer;\ B.width = 4; \}$
B	$\rightarrow$	**float**	$\{ B.type = float;\ B.width = 8; \}$
C	$\rightarrow$	ϵ	$\{ C.type = t;\ C.width = w; \}$
C	$\rightarrow$	[**num**] C_1	$\{ C.type = array(\textbf{num}.value,\ C_1.type);\ C.width = \textbf{num}.value \times C_1.width; \}$

Input string: `int[2][3]`

t = integer
w = 4





Call stack

Step 8:

- C() returns
- Execute semantic action

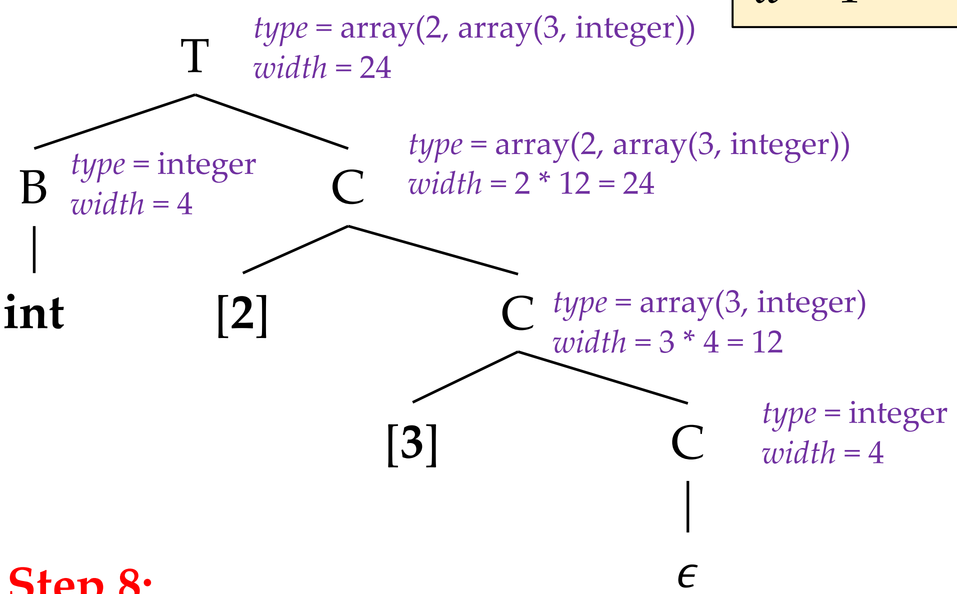
$$C \rightarrow [\ \textbf{num}\ ]\ C_1 \quad \boxed{\{\ C.type = array(\textbf{num}.value,\ C_1.type);\ C.width = \textbf{num}.value \times C_1.width;\ \}}$$

Translation Process Example

$T \rightarrow B$		$\{\ t = B.type;\ w = B.width;\ \}$
C		$\{T.type{=}C.type;\ T.width{=}C.width;\}$
$B \rightarrow$ **int**		$\{\ B.type = integer;\ B.width = 4;\ \}$
$B \rightarrow$ **float**		$\{\ B.type = float;\ B.width = 8;\ \}$
$C \rightarrow \epsilon$		$\{\ C.type = t;\ C.width = w;\ \}$
$C \rightarrow$ [**num**] C_1		$\{\ C.type = array(\mathbf{num}.value,\ C_1.type);\ C.width = \mathbf{num}.value \times C_1.width;\ \}$

Input string: `int[2][3]`

t = integer
w = 4



Step 8:

- T() returns
- Execute semantic action

$$T \rightarrow B \quad \{\ t = B.type;\ w = B.width;\ \}$$
$$C \quad \boxed{\{T.type{=}C.type;\ T.width{=}C.width;\}}$$

Call stack

Sequences of Declarations

- When dealing with a procedure, local variables should be put in a separate symbol table; their declarations can be processed as a group
  - Name, type, and relative address of each variable should be stored
- The translation scheme below handles a sequence of declarations
  - $offset$: the next available relative address; top: the current symbol table

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

Computing relative addresses of declared names

Fields in Records and Classes*

- Two assumptions:
  - The field names within a record must be distinct
  - The offset for a field name is relative to the data area (数据区) for that record
- For convenience, we use a symbol table for each record type
  - Store both type and relative address of fields
- A record type has the form $record(t)$
  - $record$ is the type constructor
  - t is a symbol table object, holding info about the fields of this record type

* Self-study materials

Fields in Records and Classes

$$
\begin{array}{llll}
T & \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env(); \\
 & & & \quad Stack.push(offset);\ offset = 0;\ \} \\
 & & D\ '\}' & \{\ T.type = record(top);\ T.width = offset; \\
 & & & \quad top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$

- The class Env implements symbol tables
- $Env.push(top)$ and $Stack.push(offset)$ save the current symbol table and offset; later, they will be popped to continue with other translation
- The translation scheme can be adapted to deal with classes

Outline

- Intermediate Representation
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

Type Checking

- To do type checking, a compiler needs to assign a type expression to each component of the source program

- The compiler then determines whether the type expressions conform to a collection of logical rules (i.e., the *type system*)
  - A *sound* type system allows us to determine statically that type errors cannot occur at run time

- A language is *strongly typed* if the compiler guarantees that the programs it accepts will run without type errors (sound type system)
  - **Strongly typed:** Java (`double a;` ~~`int b = a`~~`; //cannot compile`)
  - **Weakly typed:** C/C++ (`double a; int b = a; //implicit conversion`)

Rules for Type Checking

- **Type synthesis (类型合成)**
  - Build up the type of an expression from the types of subexpressions
    - **Typical form:** **if** f has type $s \rightarrow t$ **and** x has type s, **then** expression $f(x)$ has type t
    - **Example:** If x is of interger type, the function f has type $integer \rightarrow integer$, then the type of the expression $f(x) + x$ is also integer
- **Type inference (类型推导)**
  - Determine the type of a language construct from the way it is used
    - **Typical form:** **if** $f(x)$ is an expression, **then:** as f has type $\alpha \rightarrow \beta$ (α, β represent two types), x has type α
    - **Example:** let $null$ be a function that tests whether a list is empty, then from the usage $null(x)$, we can tell that x must be a list

Type Conversions

- Consider an expression $x * i$, where x is a float and i is an integer
  - The representation (the way of organizing 0/1 bits) of integers and floating-point numbers is different
  - Different machine instructions are used for operations on integers an floats
  - Convert integers to floats: $\mathrm{t_1 = (float)\ i}$ $\mathrm{t_2 = x\ fmul\ t_1}$

- Type conversion SDT for a simple case (using type synthesis)
  - $E \rightarrow E_1 + E_2$

$\{$ $\textbf{if}(E_1.type = integer \textbf{ and } E_2.type = integer)\ E.type = integer;$

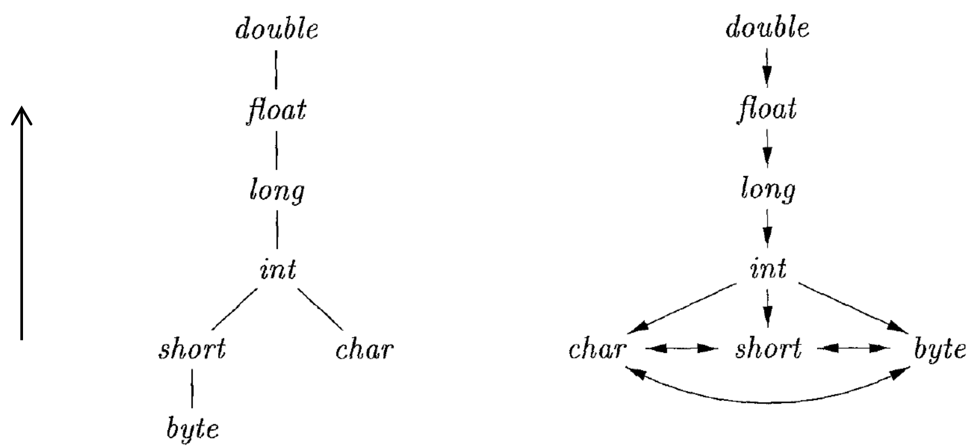
$\textbf{else if}(E_1.type = float \textbf{ and } E_2.type = integer)\ E.type = float;$

…

$\}$

Widening and Narrowing (1)

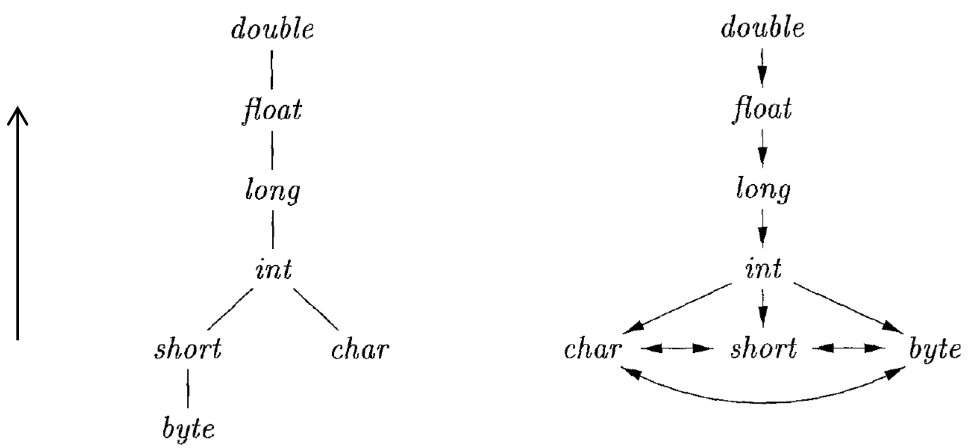
- Type conversion rules vary from language to language
- Java distinguishes between *widening* conversions (类型拓宽) and *narrowing* conversions (类型窄化)



(a) Widening conversions (b) Narrowing conversions

Widening and Narrowing (2)

- *Widening* conversions preserve information and can be done automatically by the compiler (*implicit* type conversions, or *coercions*)
- *Narrowing* conversions lose information and require programmers to write code to cause the conversion (*explicit* type conversions, or *casts*)



(a) Widening conversions (b) Narrowing conversions

SDT for Type Conversion

- $\max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy

- $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w

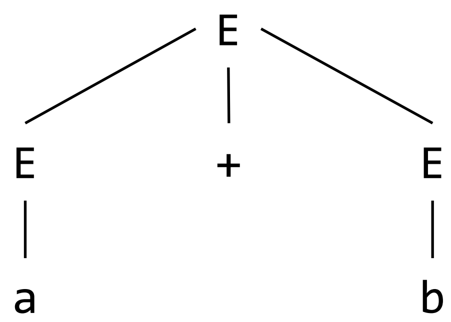
```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else  error;
}
```

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{ & E.type = max(E_1.type, E_2.type); \\
 & & a_1 = widen(E_1.addr, E_1.type, E.type); \\
 & & a_2 = widen(E_2.addr, E_2.type, E.type); \\
 & & E.addr = \textbf{new}\ Temp(); \\
 & & gen(E.addr\ '='\ a_1\ '+'\ a_2); \}
\end{array}
$$

Example

- a + b (suppose a is of *int* type and b is of *float* type)

$Addr\ widen(Addr\ a,\ Type\ t,\ Type\ w)$
$\quad \textbf{if}\ (\ t = w\ )\ \textbf{return}\ a;$ 3
$\quad \textbf{else if}\ (\ t = integer\ \textbf{and}\ w = float\ )\ \{$
$\quad\quad temp\ =\ \textbf{new}\ Temp();$
$\quad\quad gen(temp\ '='\ '(\textbf{float})'\ a);$ 2
$\quad\quad \textbf{return}\ temp;$
$\quad \}$
$\quad \textbf{else}\ \ \textbf{error};$
$\}$



Generated code:

```
temp = (float) a
temp2 = temp + b
```

2
5

$E \ \rightarrow\ E_1 + E_2 \quad \{\ E.type\ =\ max(E_1.type, E_2.type);$
$\quad a_1\ =\ widen(E_1.addr, E_1.type, E.type);$
$\quad a_2\ =\ widen(E_2.addr, E_2.type, E.type);$
$\quad E.addr = \textbf{new}\ \ Temp\ ();$
$\quad gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}$

E.type = *max*(int, float) = float 1

a_1 = *widen*(a, int, float) = temp 2

a_2 = *widen*(b, float, float) = b 3

E.addr = new *Temp*() = temp2 4

5

Outline

- Intermediate Representation
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

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

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

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

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

SDD for Expression Translation – Problem

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

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

$$
\begin{array}{lll}
S & \rightarrow \textbf{id} = E \;; & \{\; gen(\; top.get(\textbf{id}.lexeme)\; '=' \; E.addr); \;\} \\
\\
E & \rightarrow E_1 + E_2 & \{\; E.addr = \textbf{new}\; Temp\,(); \\
 & & \;\; gen(E.addr\; '='\; E_1.addr\; '+'\; E_2.addr); \;\} \\
\\
 & \mid \; - E_1 & \{\; E.addr = \textbf{new}\; Temp\,(); \\
 & & \;\; gen(E.addr\; '='\; '\textbf{minus}'\; E_1.addr); \;\} \\
\\
 & \mid \; (\; E_1 \;) & \{\; E.addr = E_1.addr; \;\} \\
\\
 & \mid \; \textbf{id} & \{\; E.addr = top.get(\textbf{id}.lexeme); \;\}
\end{array}
$$

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

Incremental Translation Scheme

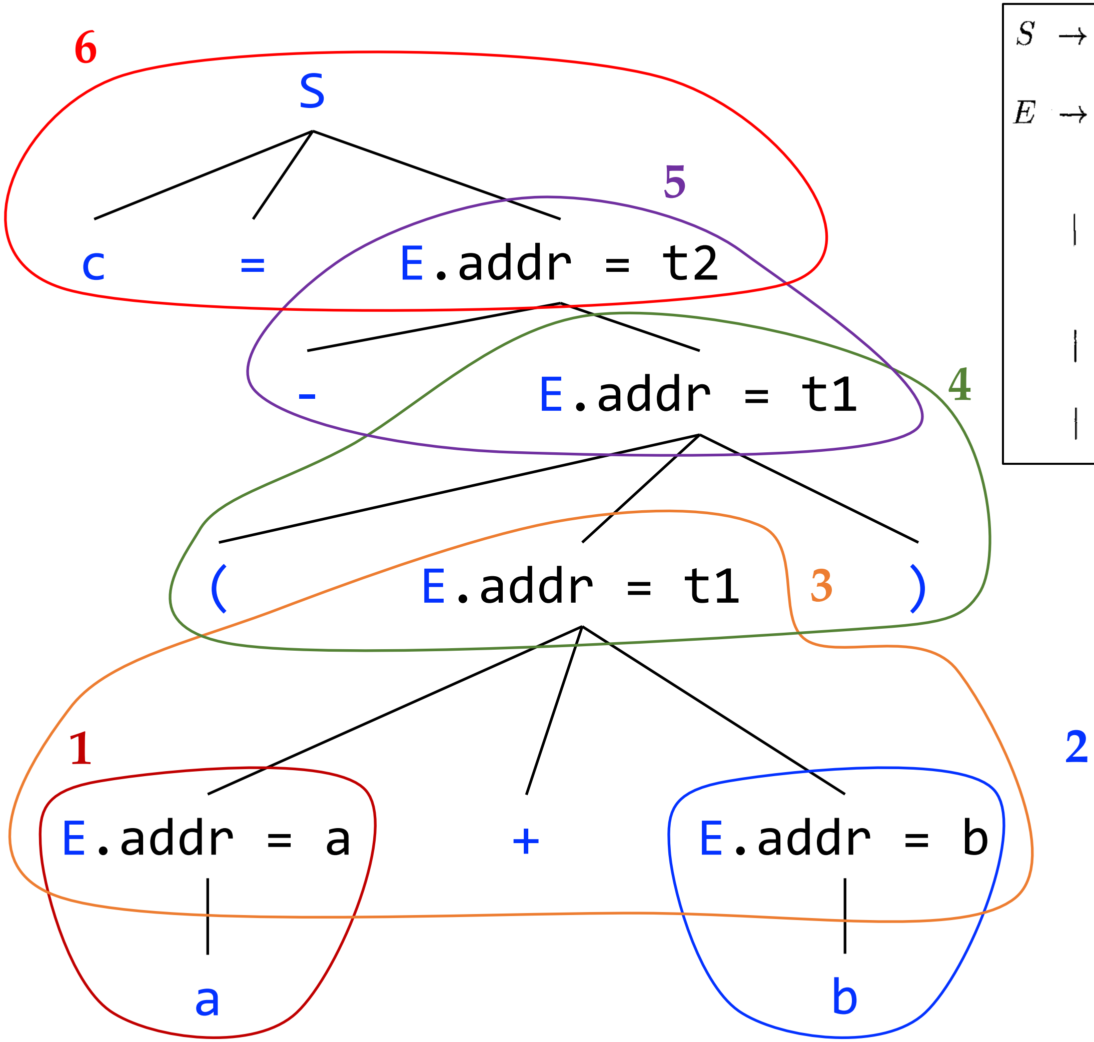
- In the SDT below, *gen* not only generates a three-address instruction, but also appends it to the sequence of instructions generated so far
  - In comparison, in the previous SDD, the *code* attribute can be long strings after concatenations

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

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**$.lexeme)$ $'='$ $E.addr)$; } 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

Generated code:

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

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

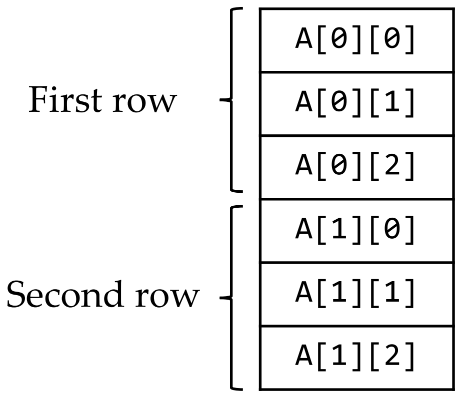
```
c + a[i][j]
```

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

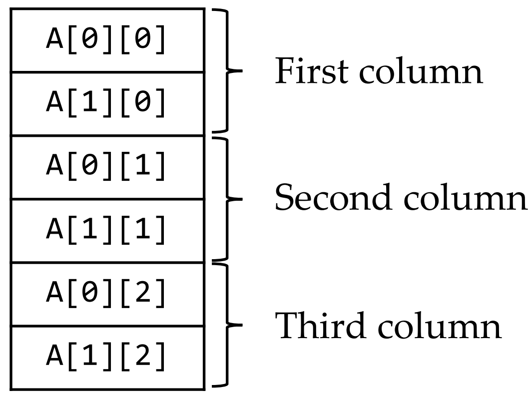
calculate address

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

SDT for Array References (1)

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

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 $base + offset$

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

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

Array references can be part of an expression

SDT for Array References (3)

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

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

Instruction of form $a[i] = x$

Array references can appear at the LHS of an assignment statement

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

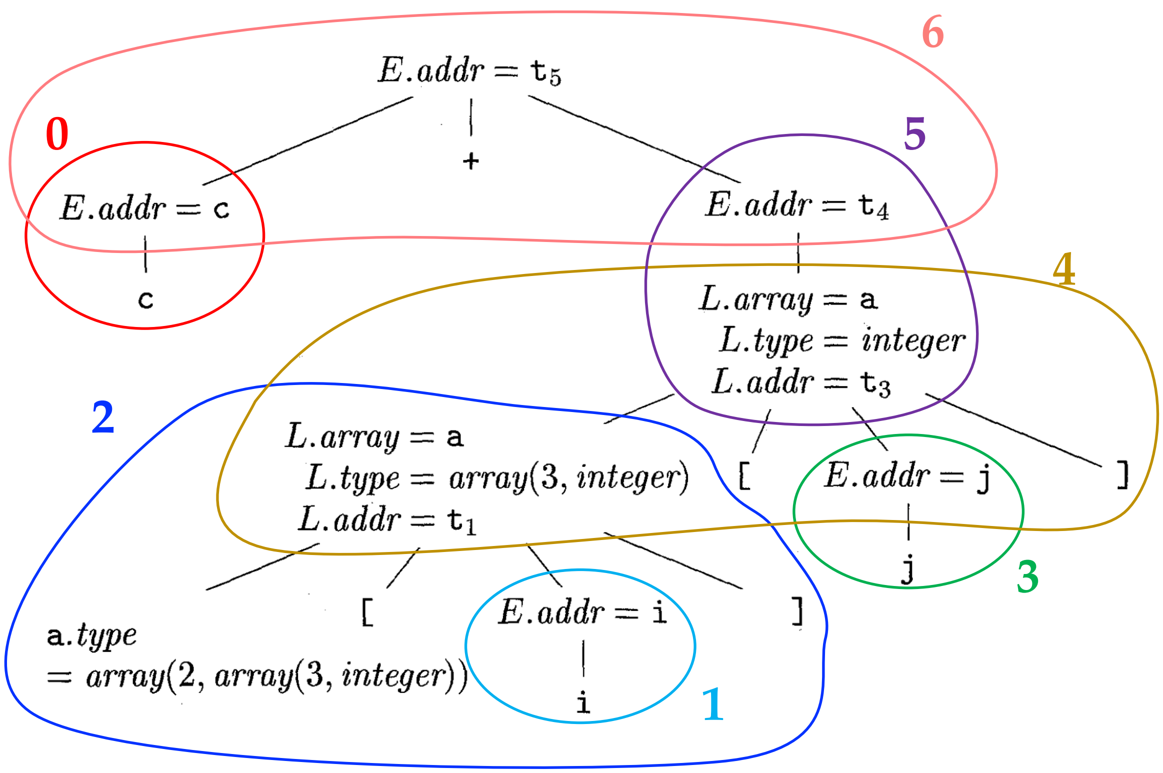
| **id** { $E.addr$ = $top.get($**id**$.lexeme)$; } 0 1 3

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

$L \rightarrow$ **id** [E] { $L.array$ = $top.get($**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

- Intermediate Representation
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

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;`

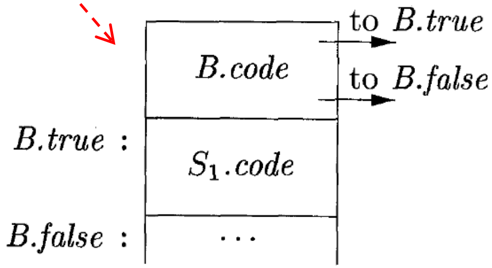
```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Flow-of-Control Statements

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

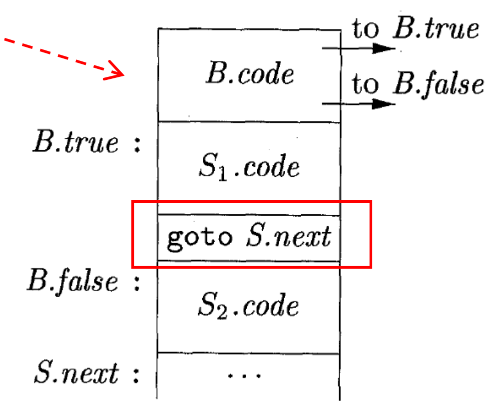
Inherited attributes:

- $B.true$: 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

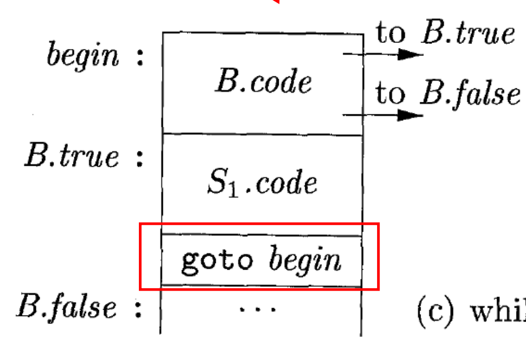


(a) if

$S.next$ 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

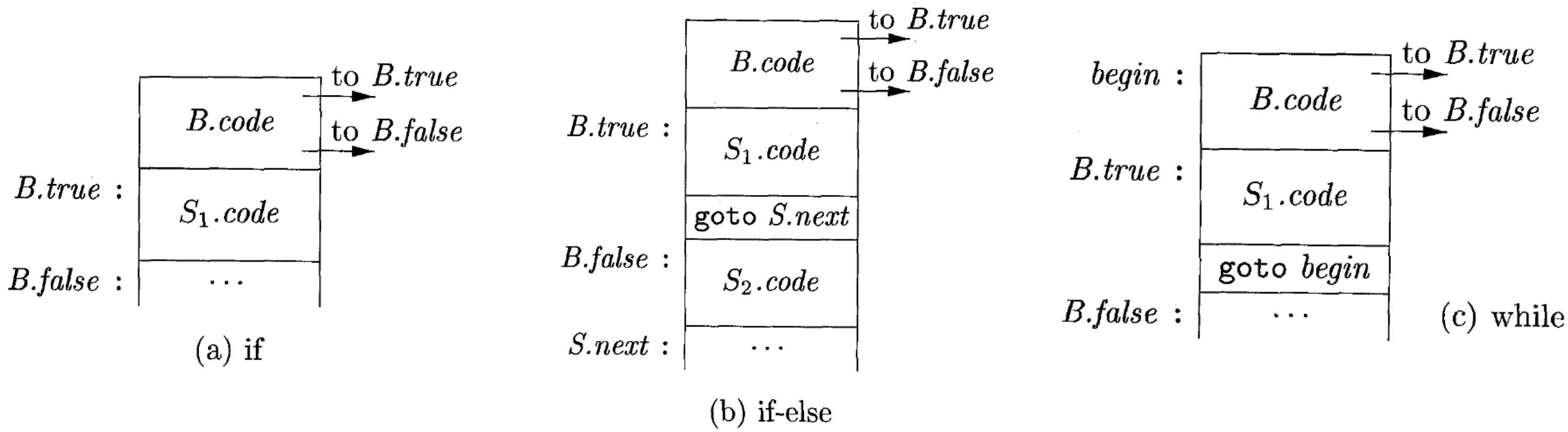
SDD for Flow-of-Control Statements (2)

Illustrated by previous figure

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

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)

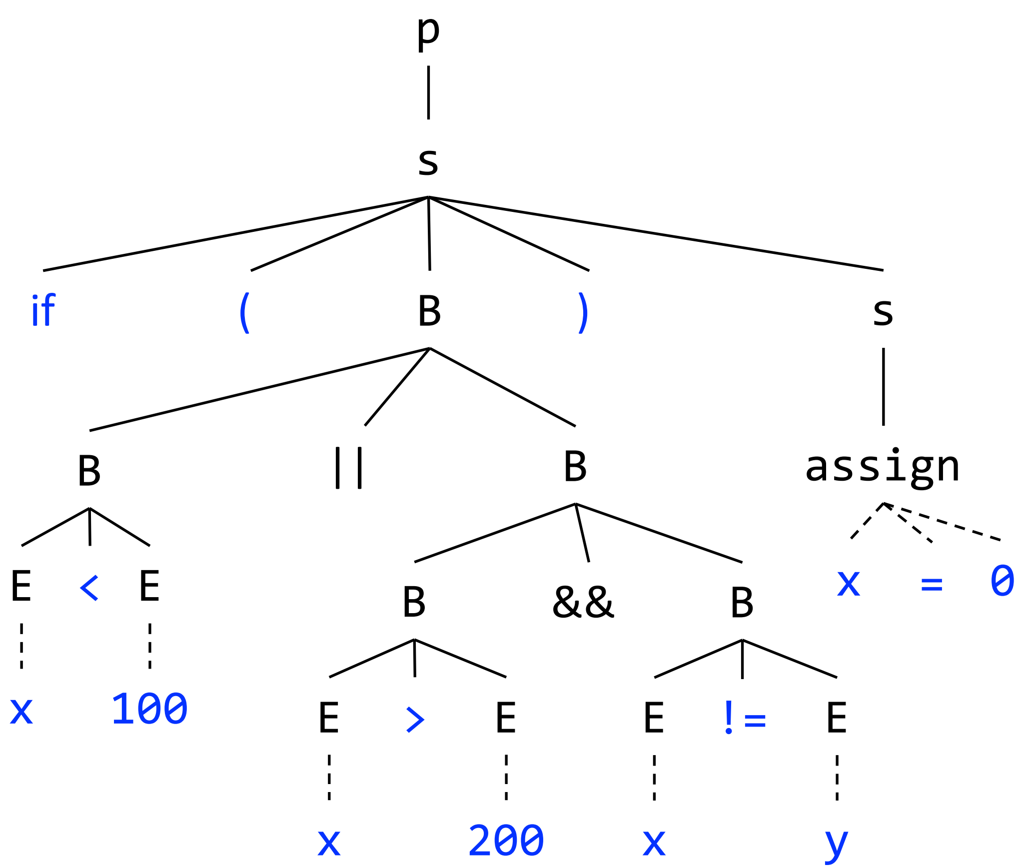
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \parallel E_2.code$ $\parallel gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\parallel gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code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

Generating Three-Address Code for Booleans (2)

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

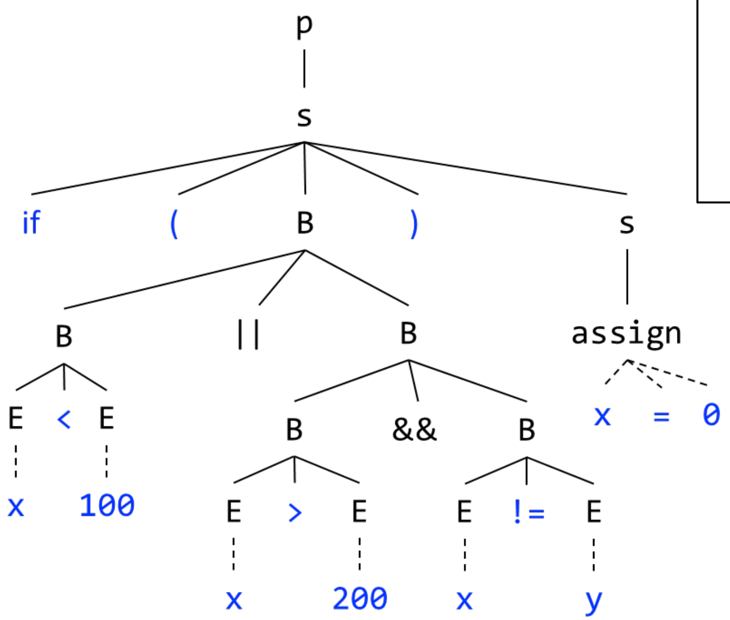
Example

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



Dashed lines mean that the reduction may consist of multiple steps

Example



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

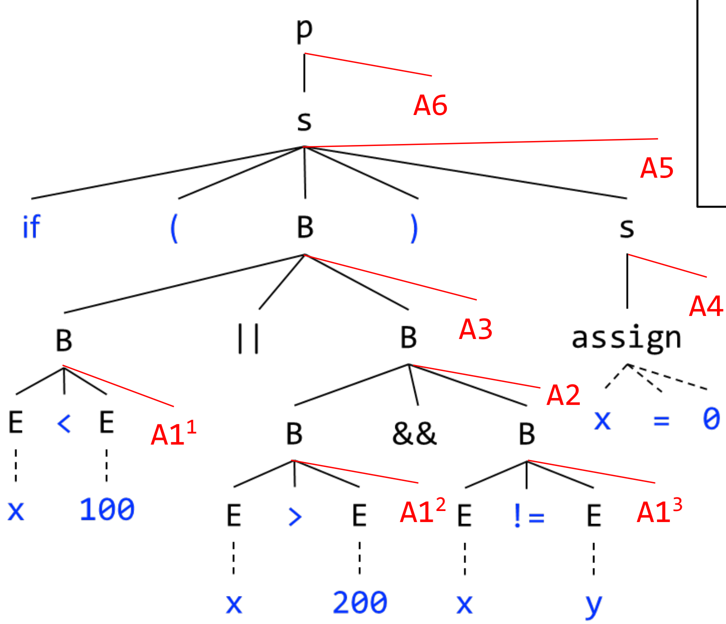
$B \rightarrow B_1 \;\|\|\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

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

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 intermidate code

Example



Virtual nodes are in red color

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

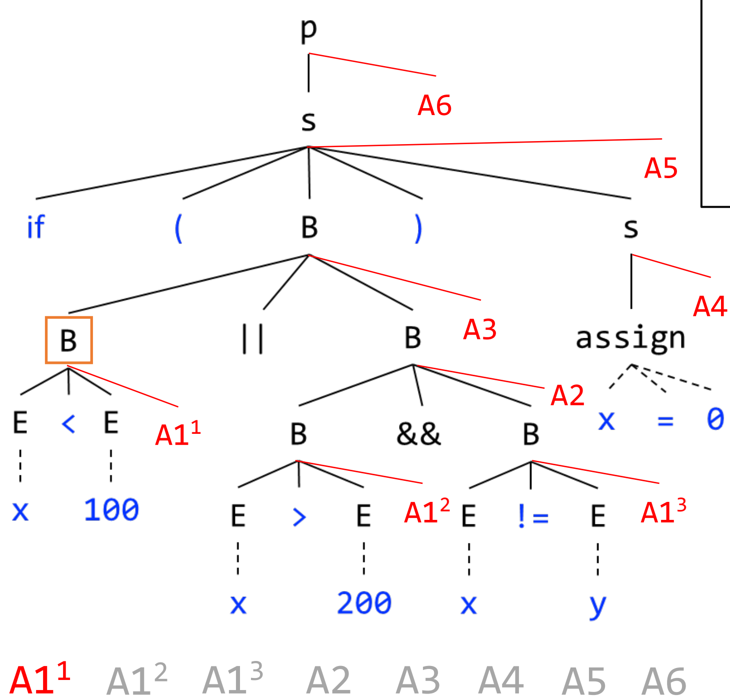
A1[1] A1[2] A1[3] A2 A3 A4 A5 A6

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



Generated code:

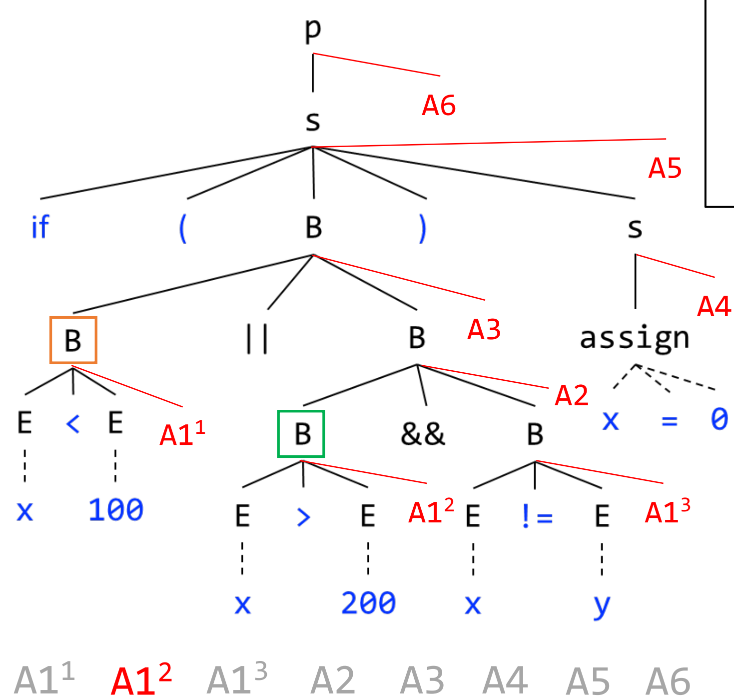
```
if x < 100 goto B.true
goto B.false
```

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$ A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$ 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 $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$

Example



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

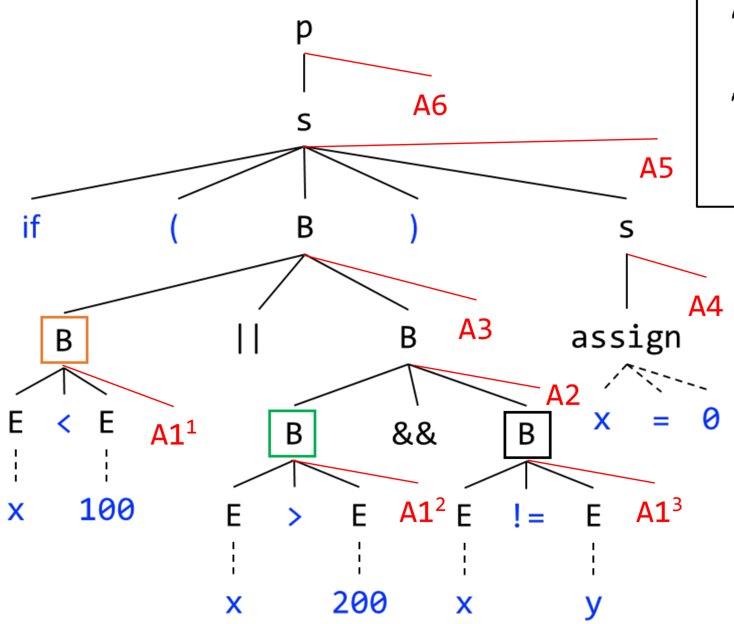
$B \to B_1 \ \|\|\ B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \parallel label(B_1.false) \parallel B_2.code$	A3
$B \to B_1 \ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code =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

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

Generated code:

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

Example



A1[1] A1[2] A1[3] 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
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ A5 $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

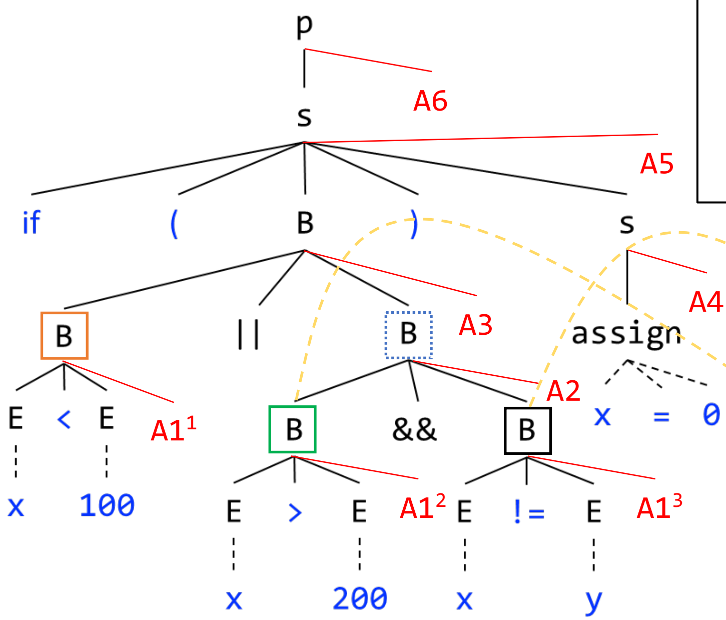
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ A3 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ A2 $B_2.false =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

$B \rightarrow E_1 \ \textbf{rel} \ E_2$	$B.code = E_1.code \mid\mid E_2.code$ 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
goto B.false
if x > 200 goto B.true
goto B.false
if x != y goto B.true
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
$S \rightarrow$ **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

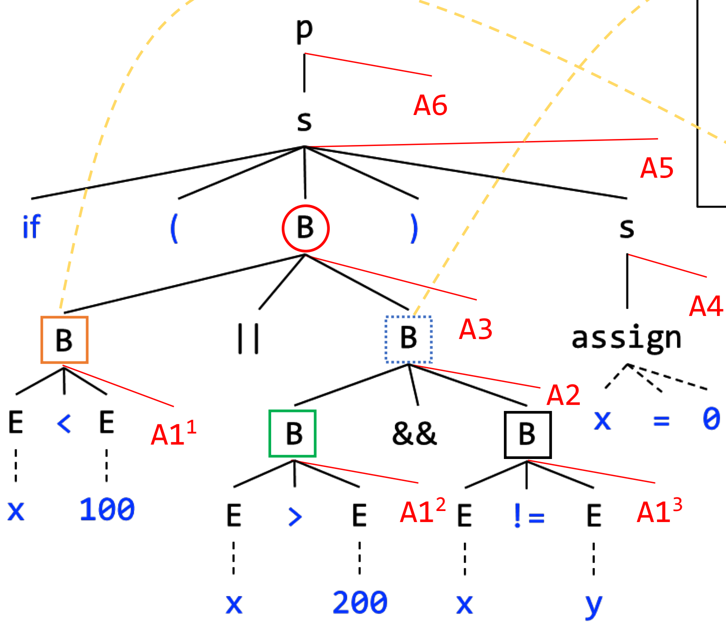
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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

$B \rightarrow E_1$ **rel** E_2	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



A1[1] A1[2] A1[3] 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
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ A5 $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$

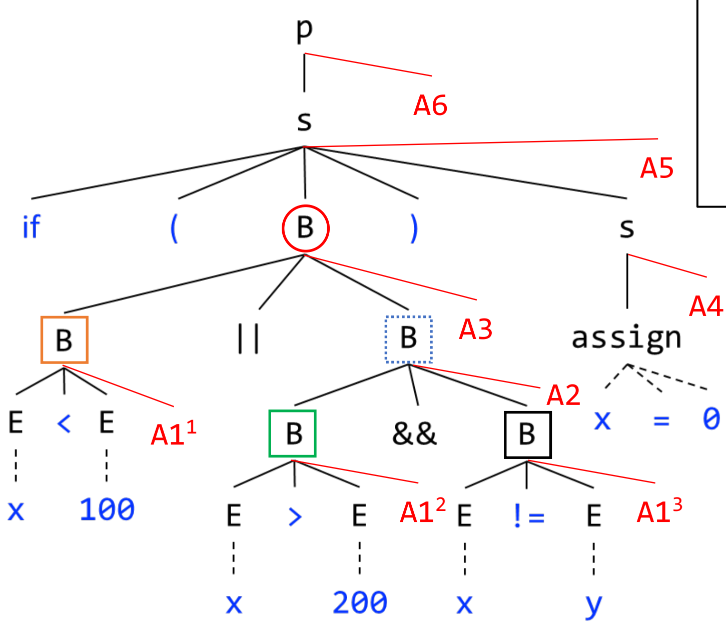
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ A3 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ A2 $B_2.false = B.false$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

$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ 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
    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
```

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

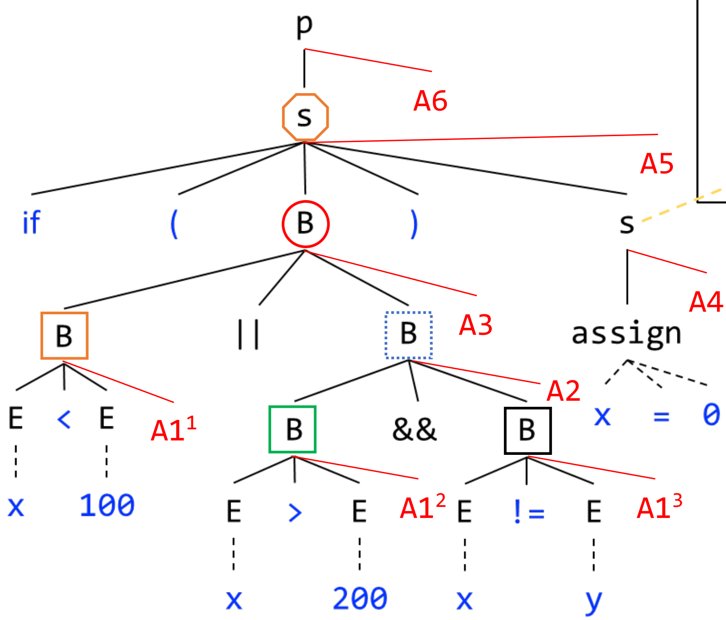
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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



A1[1] A1[2] A1[3] 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

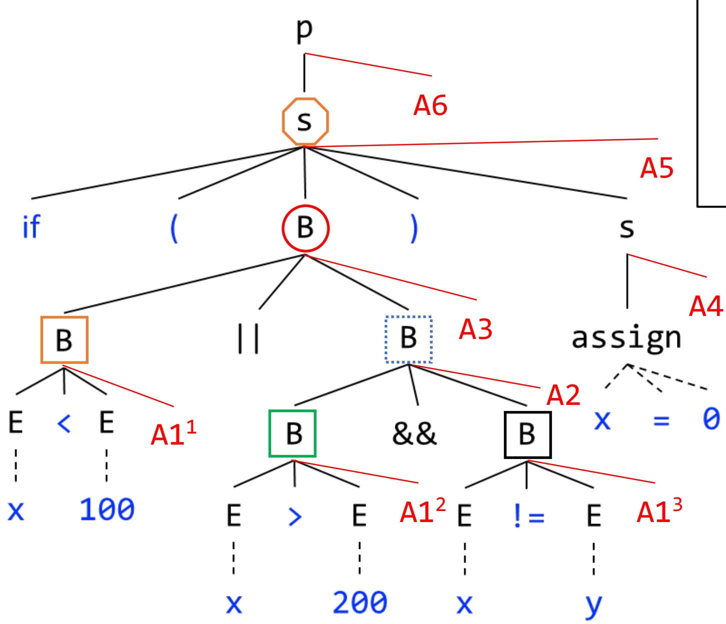
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$	A3
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$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



A1[1] A1[2] A1[3] 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

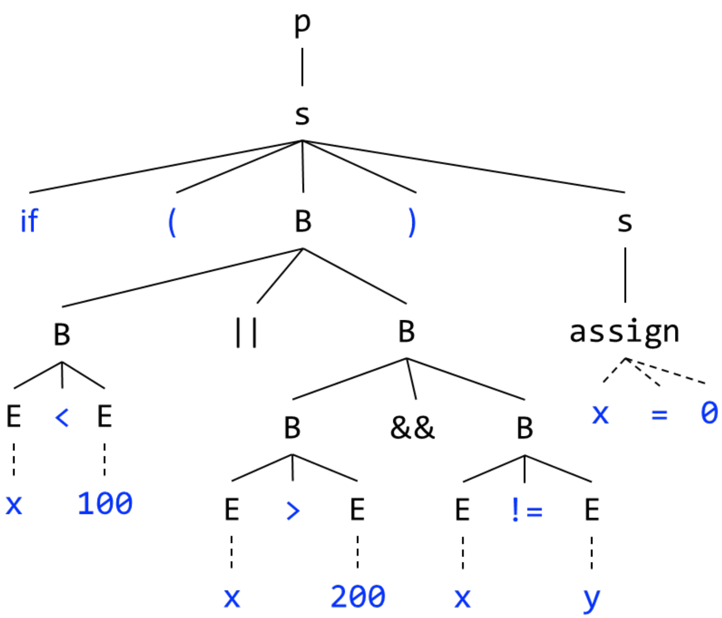
$B \rightarrow E_1\ \textbf{rel}\ E_2$	A1	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid\ gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid\ gen('\texttt{goto}'\ B.false)$

Generated code:

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

- Intermediate Representation
- Type and Declarations
- Type Checking (Lab)
- Translation of Expressions (Lecture + Lab)
- Control Flow
- Backpatching (Lab)

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 <u>another separate pass (traversing the parse tree)</u> 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?

	1)	$B \rightarrow B_1 \;\|\|\; M\; B_2$	$\{\; backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist; \;\}$
2	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)	$B \rightarrow (\; B_1 \;)$	$\{\; B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist; \;\}$
1	5)	$B \rightarrow E_1 \;\mathbf{rel}\; E_2$	$\{\; B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ gen('if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto _'); gen('goto _'); }
	6)	$B \rightarrow \mathbf{true}$	$\{\; B.truelist = makelist(nextinstr);$ gen('goto _'); }
	7)	$B \rightarrow \mathbf{false}$	$\{\; B.falselist = makelist(nextinstr);$ gen('goto _'); }
	8)	$M \rightarrow \epsilon$	$\{\; M.instr = nextinstr; \;\}$

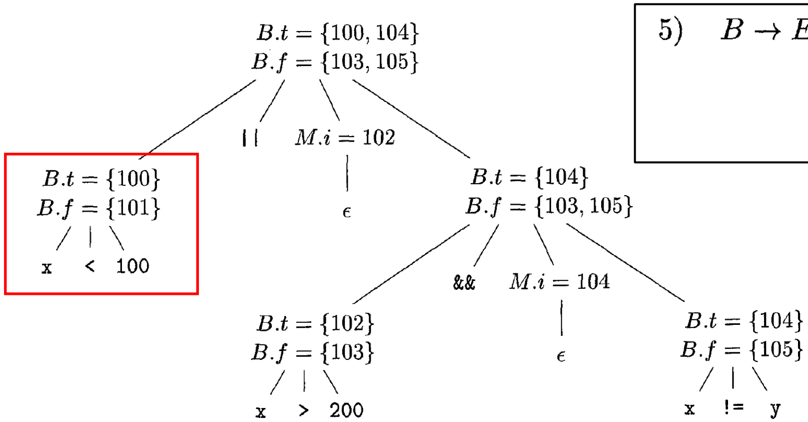
When finishing processing B1 && B2, we know the jump target for B1.truelist

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

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)



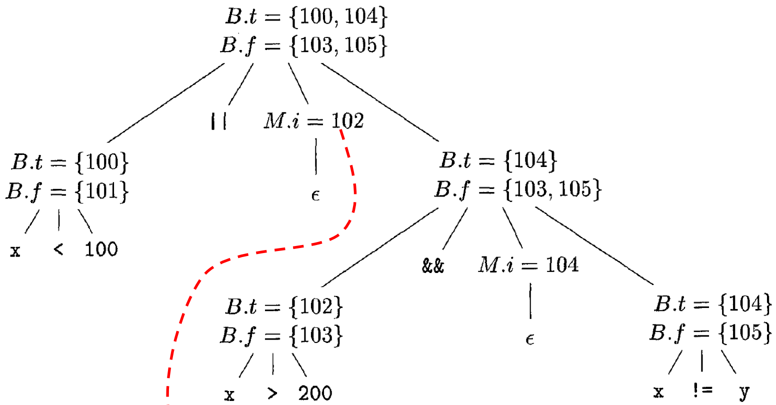
5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
gen('`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto _`');
gen('`goto _`'); }

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)

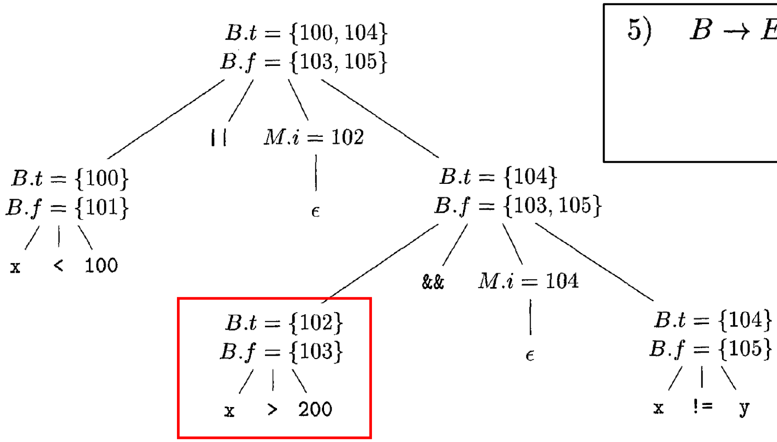


$$\boxed{8) \quad M \rightarrow \epsilon \qquad \{\ M.instr = nextinstr;\ \}}$$

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

Example – Boolean Expressions

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



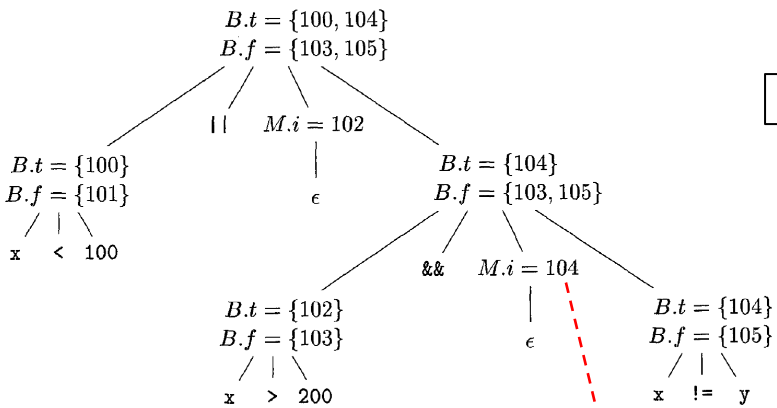
5)	$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ *gen*('`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto _`'); *gen*('`goto _`'); }

Generated code:

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

Example – Boolean Expressions

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

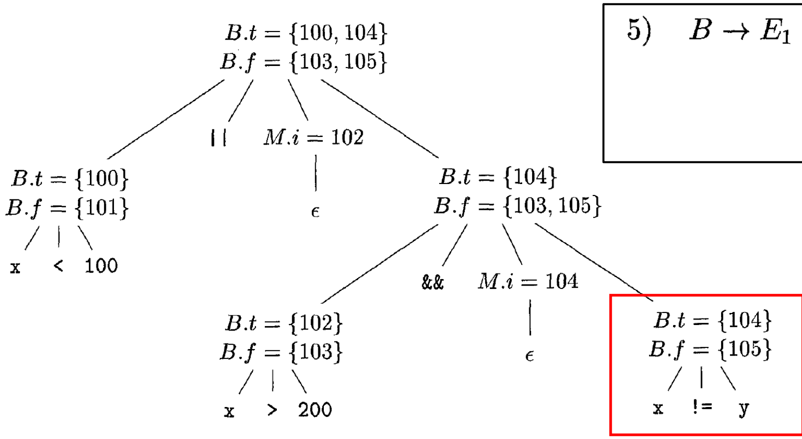


8)	$M \to \epsilon$	$\{\ M.instr = nextinstr;\ \}$

The marker nonterminal records the value of *nextinstr*, 104

Example – Boolean Expressions

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



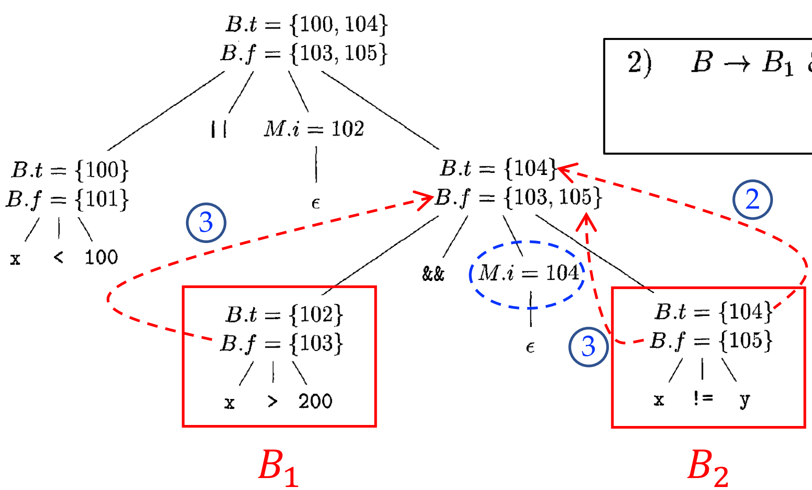
5) $B \to E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$gen($'`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto _`'$);$
$gen($'`goto _`'$);$ }

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)



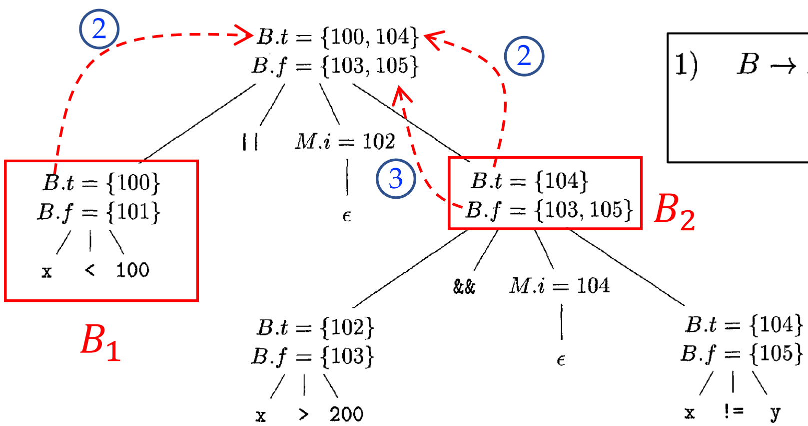
2) $B \rightarrow B_1\ \&\&\ M\ B_2$ { $backpatch(B_1.truelist, M.instr)$; 1
$B.truelist = B_2.truelist$; 2
$B.falselist = merge(B_1.falselist, B_2.falselist)$; } 3

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: $x < 100 \parallel x > 200 \;\&\&\; x\,! = y$
- **Step 7:** reduce $B_1 \parallel MB_2$ to B by production (1)



1) $B \rightarrow B_1$ `||` $M\ B_2$	$\{$ $backpatch(B_1.falselist, M.instr);$ 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:

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

(2) Backpatching scheme:

$$B \rightarrow E_1 \textbf{ rel } E_2 \qquad \begin{aligned} \{\; & B.truelist = makelist(nextinstr); \\ & B.falselist = makelist(nextinstr + 1); \\ & gen('\texttt{if}'\; E_1.addr\; \textbf{rel}.op\; E_2.addr\; '\texttt{goto \_}'); \\ & gen('\texttt{goto \_}');\; \} \end{aligned}$$

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:

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

(2) Backpatching scheme:

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

Comparison:

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

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)
  - 6.7 Backpatching (6.7.1 – 6.7.3)