Chapter 5:
Intermediate-Code Generation
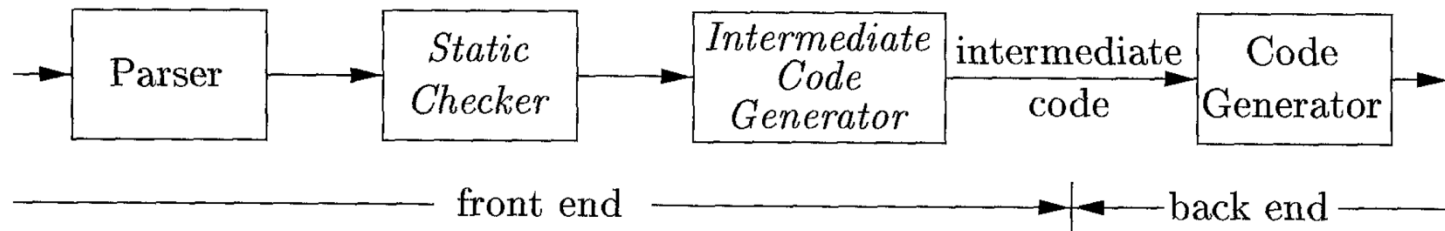
Yepang Liu

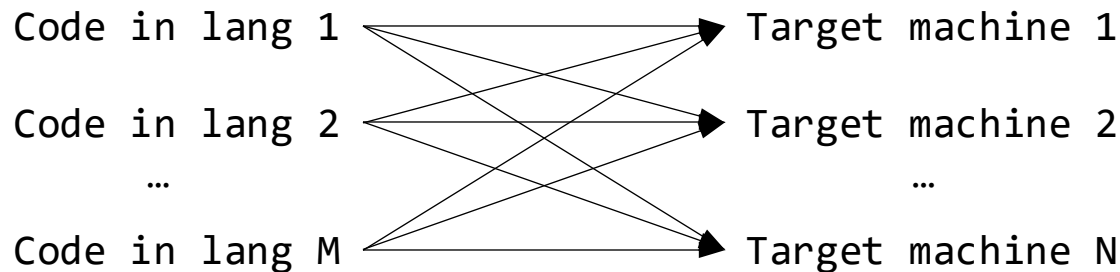liuyp1@sustech.edu.cn

Outline

- Intermediate Representation

- Type and Declarations

- Type Checking

- Translation of Expressions

- Control Flow (Lab)

- Backpatching (Lab)

Compiler Front End
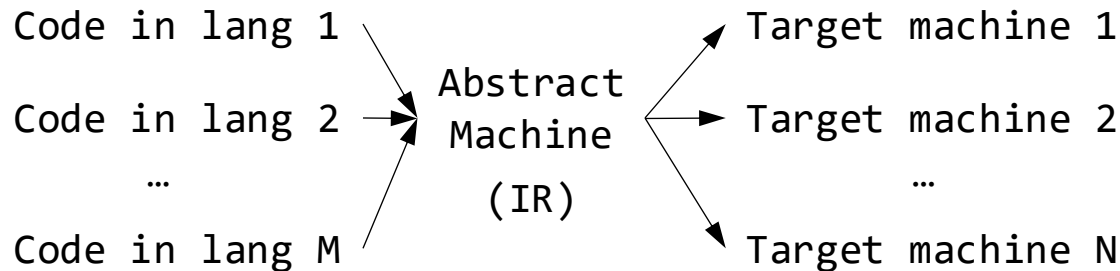
- The front end of a compiler analyzes a source program and creates an intermediate representation (IR, 中间表示), from which the back end generates target code

    - 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

Code in lang 1   Target machine 1

Code in lang 2   Target machine 2
…   …

Code in lang M   Target machine N

M * N compilers
without a common IR

Code in lang 1   Target machine 1

Code in lang 2   Abstract Machine   Target machine 2
…   (IR)   …

Code in lang M   Target machine N

M + N compilers
with a common IR

Different Levels of IRs

Source Program → High Level Intermediate Representation → · · · → Low Level Intermediate Representation → Target Code
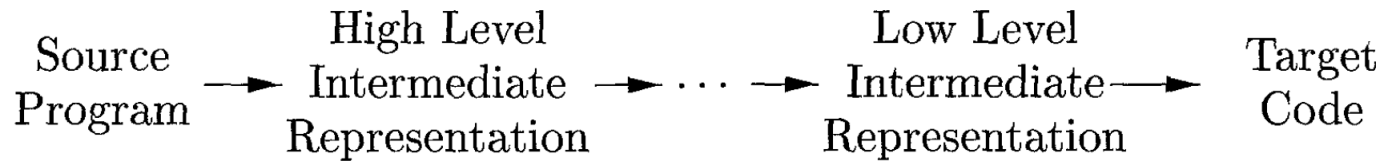
- A compiler may construct a sequence of IR's

  - High-level IR's like syntax trees are close to the source language

    o They are suitable for machine-independent tasks like static type checking

  - Low-level IR's are close to the target machines

    o They are suitable for machine-dependent tasks like register allocation and instruction selection

- 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
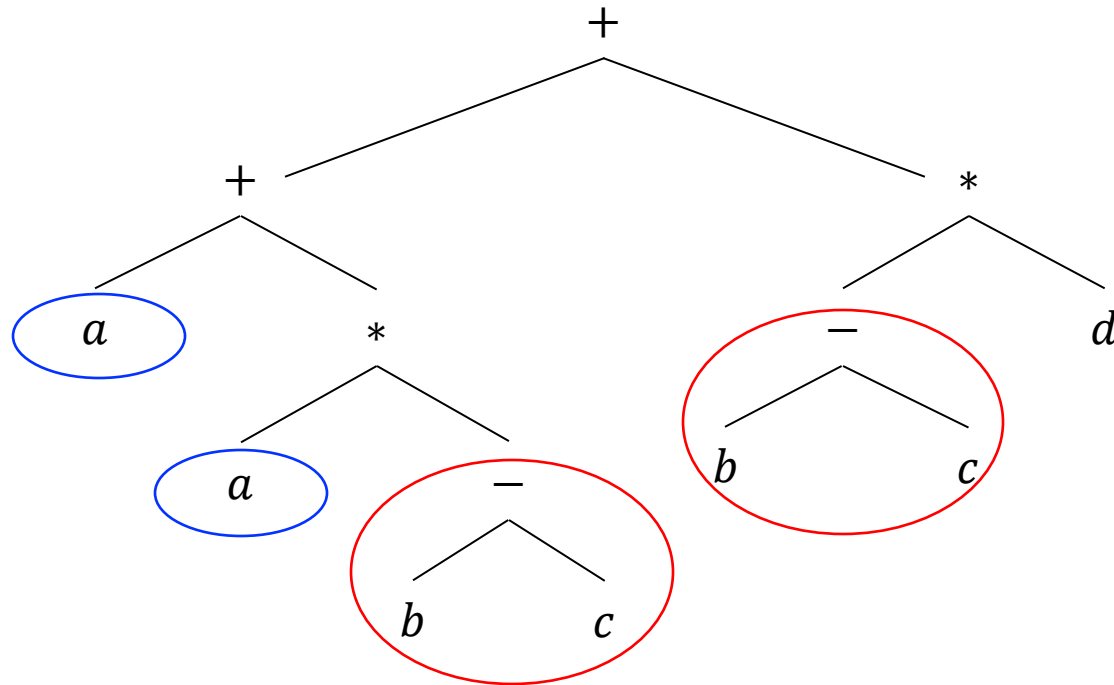
Outline

• **Intermediate Representation** →

• DAG's for Expressions
• Three-Address Code

• Type and Declarations

• Type Checking

• Translation of Expressions

• Control Flow (Lab)

• Backpatching (Lab)

DAG's for Expressions

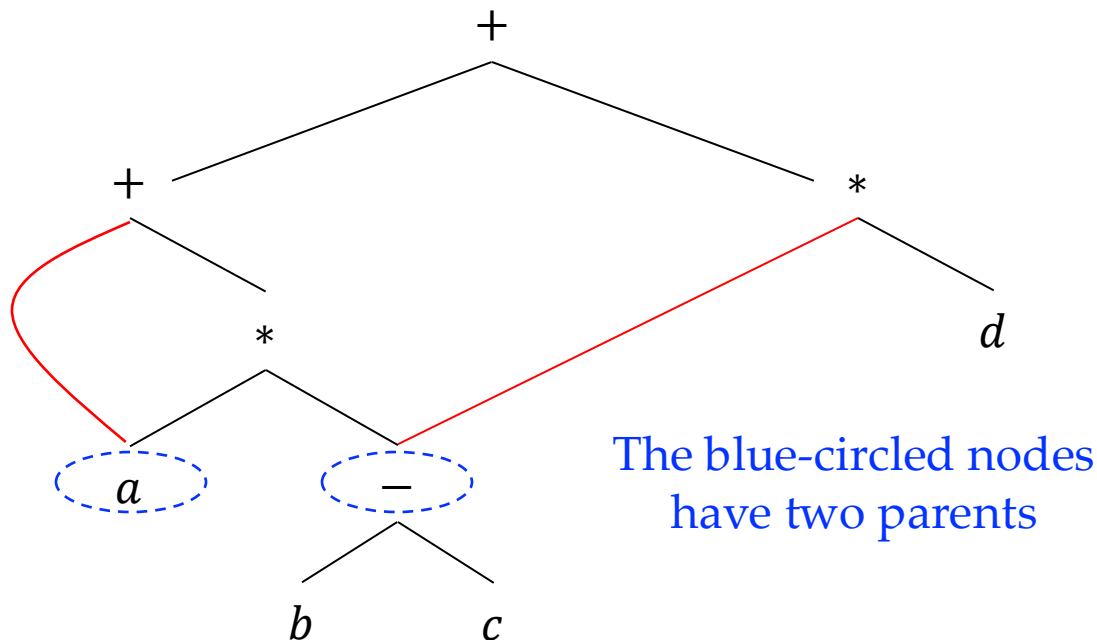- In a syntax tree, the tree for a common subexpression would be replicated as many times as the subexpression appears

  - 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: $a + a * (b - c) + (b - c) * d$

The blue-circled nodes have two parents

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

Special "new":

Reuse existing nodes when possible

Constructing DAG's Cont.

- The construction steps $a + a * (b - c) + (b - c) * d$
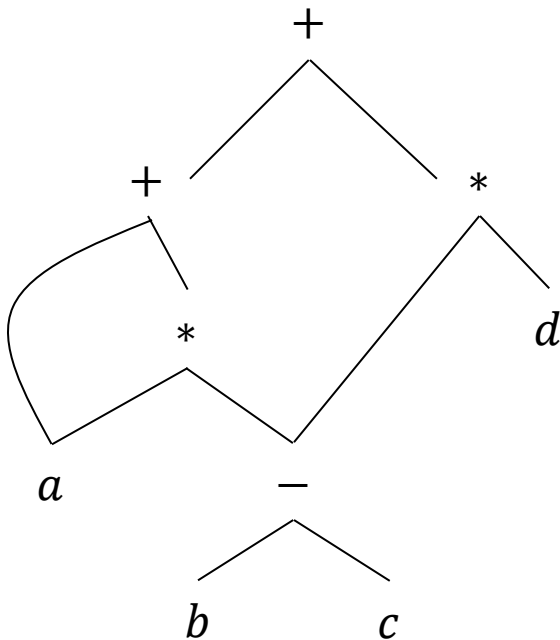
1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
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

* Translation of Expressions

* Control Flow (Lab)

* 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: $x = y[i]$ $x[i] = y$

    - 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);`

$$
\boxed{\text{L:}} \quad \begin{aligned}
& t_1 = i + 1 \\
& i = t_1 \\
& t_2 = i * \textcircled{8} \\
& t_3 = a [\ t_2\ ] \\
& \text{if } t_3 < v \text{ goto L}
\end{aligned}
$$

(a) Symbolic labels.

$$
\begin{aligned}
\boxed{100:} & \quad t_1 = i + 1 \\
101: & \quad i = t_1 \\
102: & \quad t_2 = i * 8 \\
103: & \quad t_3 = a [\ t_2\ ] \\
104: & \quad \text{if } t_3 < v \text{ goto } 100
\end{aligned}
$$

(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 <u>fields for the operator and the operands</u>

- 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* ➜ *+   y   z   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	$\cdots$		op	arg_1	arg_2	$result$	
	t_1 = minus c		0	minus	c		t_1	
	t_2 = b * t_1		1	*	b	t_1	t_2	
	t_3 = minus c		2	minus	c		t_3	Temporaries
	t_4 = b * t_3		3	*	b	t_3	t_4	
	t_5 = t_2 + t_4		4	+	t_2	t_4	t_5	
	a = t_5		5	=	t_5		a	

(a) Three-address code   (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)

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

	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
	. . .			

	op	arg$_1$	arg$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	. . .		

Three-address code   Quadruples   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 →

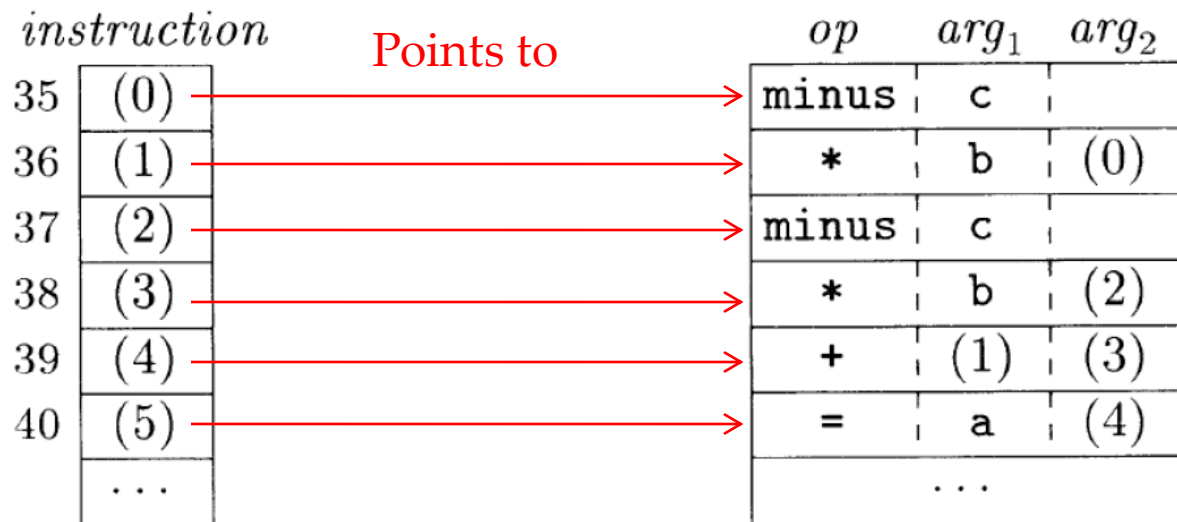|	op	arg_1	arg_2
0	minus	c	
1	minus	c	
2	*	b	(0)
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Are they still correct after swapping?

Triples' problem

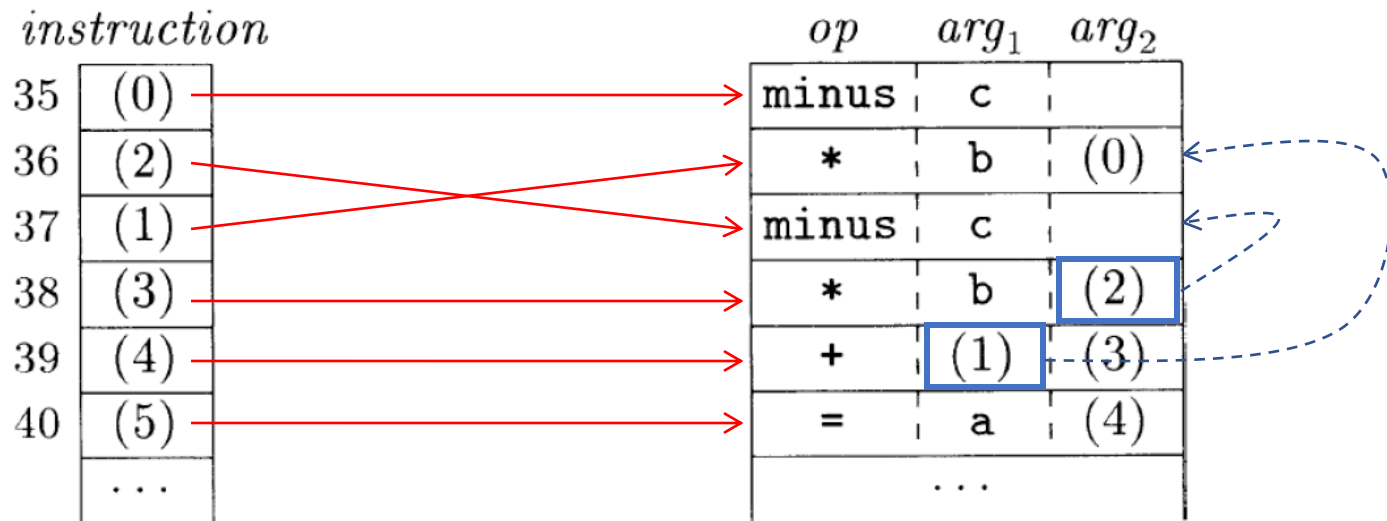The instructions now refer to wrong results; The positions need to be updated.

Indirect Triples (间接三元式)

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

instruction			Points to		op	arg_1	arg_2
35	(0)		→		minus	c	
36	(1)		→		*	b	(0)
37	(2)		→		minus	c	
38	(3)		→		*	b	(2)
39	(4)		→		+	(1)	(3)
40	(5)		→		=	a	(4)
	. . .					. . .	

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

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

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

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;
```

X_1 X_2

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;
```

$x_3 = \phi(x_1, x_2);$ // x1 if control flow passes through the true path; otherwise x2

```
y = x₃ * a;
```

Outline

- Intermediate Representation

- **Type and Declarations**

- Type Checking

- Translation of Expressions

- Control Flow (Lab)

- 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 <u>two arguments</u>: a number, a type expression
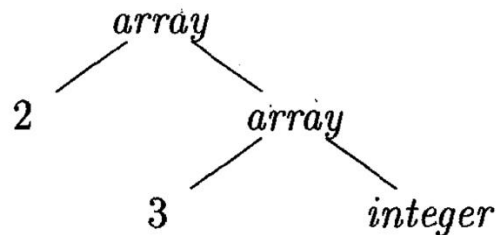
The Definition of Type Expression

- 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 → type constructor for function types

- If *s* and *t* are type expressions, then their Cartesian product *s×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

<div>

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

</div>

<div>

Code under analysis:
`a + b`

</div>

- 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 zero or more integers, each surrounded by brackets

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

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.
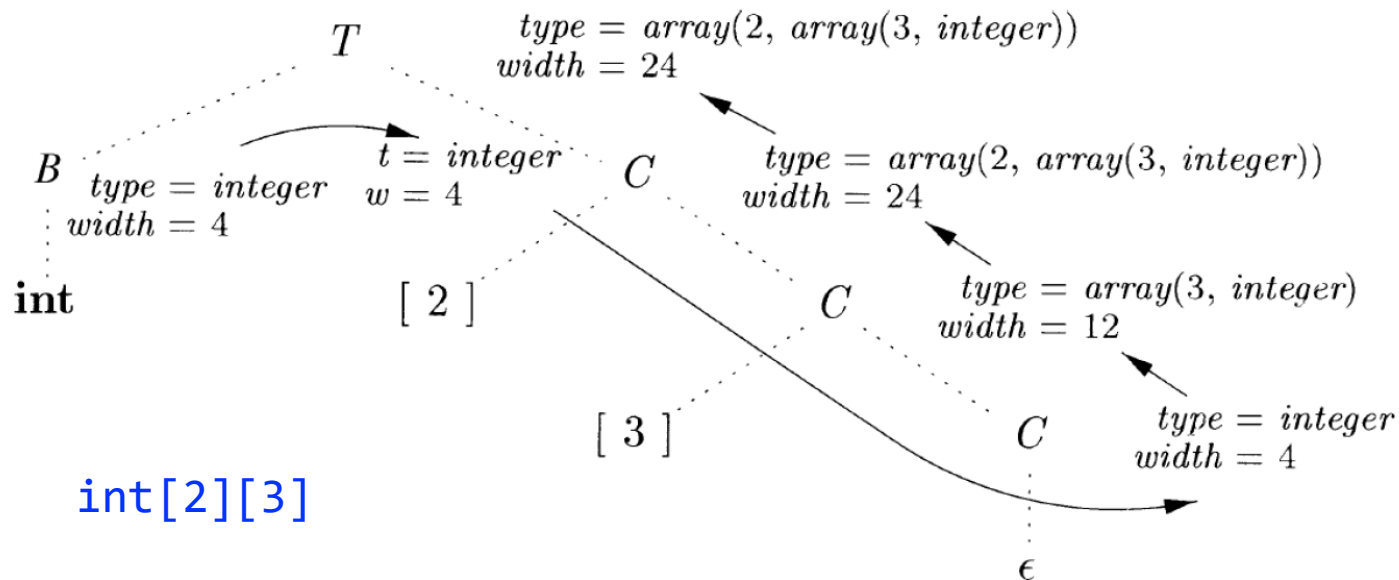
An SDT for Computing Types and Their Widths

- 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 $C \to \epsilon$

  - In an SDD, t and w would be C's inherited attributes (the SDD is L-attributed)*

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

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

T $type = array(2,\ array(3,\ integer))$
$width = 24$

B $type = integer$ $t = integer$ C $type = array(2,\ array(3,\ integer))$
$width = 4$ $w = 4$ $width = 24$

int   [2] C $type = array(3,\ integer)$
$width = 12$

[3] C $type = integer$
$width = 4$

int[2][3]

ϵ

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

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

```
        T
      /   \
    B       C
```

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

```
|          |
|          |
|   T()    |
```

Call stack

Translation Process Example

$$\begin{array}{lll} T & \to & B & \{\, t = B.type;\ w = B.width;\,\} \\ & & C & \{T.type{=}C.type;\ T.width{=}C.width\,;\,\} \\ B & \to & \mathbf{int} & \{\, B.type = integer;\ B.width = 4;\,\} \\ B & \to & \mathbf{float} & \{\, B.type = float;\ B.width = 8;\,\} \\ C & \to & \epsilon & \{\, C.type = t;\ C.width = w;\,\} \\ C & \to & [\,\mathbf{num}\,]\ C_1 & \{\ C.type{=}\ array(\mathbf{num}.value,\ C_1.type); \\ & & & \quad C.width = \mathbf{num}.value \times C_1.width;\,\} \end{array}$$

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

```
        T
       / \
      B   C
      |
     int
```

Step 2:

- Rewrite B using $B \to \mathbf{int}$
- Match input

```
|          |
|          |
|   B()    |
|   T()    |
|_____|
```

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`$[2][3]$

```
        T
       / \
   B       C
   |  type = integer
   |  width = 4
  int
```

Step 3:

- B() returns
- Execute semantic action

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

```
┌──────────────┐
│              │
│              │
│              │
│              │
├──────────────┤
│     T()      │
└──────────────┘
```
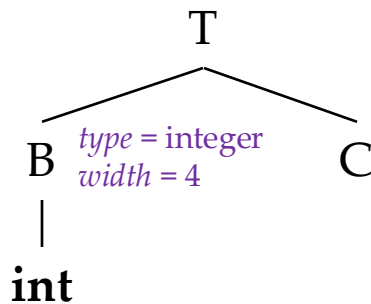
Call stack

Translation Process Example

$T \rightarrow B$ $\{ t = B.type; \ w = B.width; \}$
C $\{T.type=C.type; \ T.width=C.width; \}$
$B \rightarrow \textbf{int}$ $\{ B.type = integer; \ B.width = 4; \}$
$B \rightarrow \textbf{float}$ $\{ B.type = float; \ B.width = 8; \}$
$C \rightarrow \epsilon$ $\{ C.type = t; \ C.width = w; \}$
$C \rightarrow [\ \textbf{num}\ ] \ C_1$ $\{ \ C.type= array(\textbf{num}.value, \ C_1.type); \ C.width = \textbf{num}.value \times C_1.width; \}$
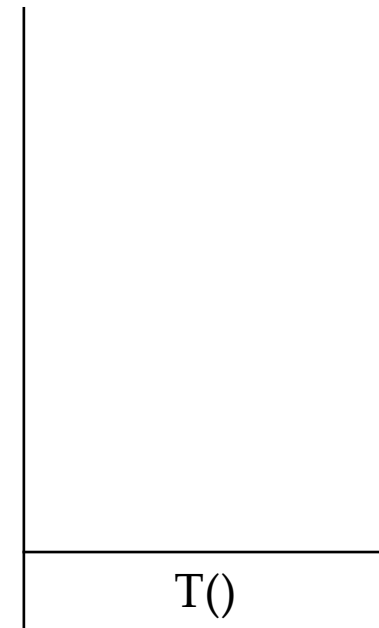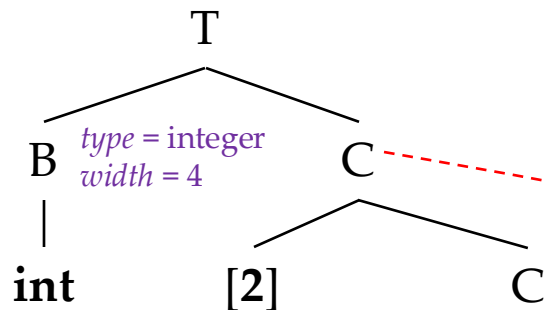
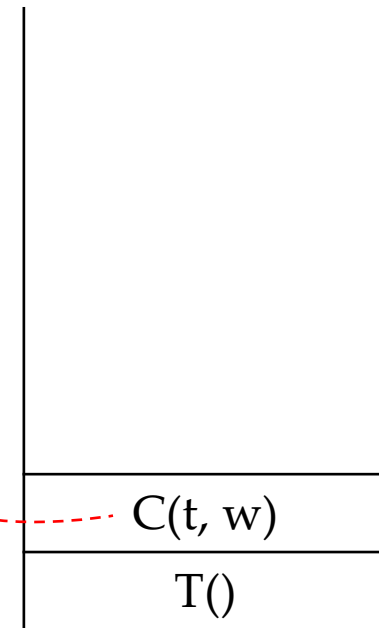Input string: `int[2][3]`

t = integer
w = 4

Step 4:

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

$T \rightarrow B$
$\quad\quad C$

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

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

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

t = integer
w = 4

T

B   *type* = integer   C
    width = 4

int   [**2**]   C

[**3**]   C

Step 5:

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

C(t, w)

C(t, w)

T()

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

T

B $\quad$ *type* = integer
$\qquad$ *width* = 4

C

int $\qquad$ [2]

C

[3] $\qquad$ C

C

ϵ

Step 5:

• Rewrite C using $C \rightarrow \epsilon$

C(t, w)

C(t, w)

C(t, w)

T()

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	$\{ C.type= array(\textbf{num}.value, C_1.type);$
		$C.width = \textbf{num}.value \times C_1.width; \}$

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

t = integer
w = 4

T

B *type* = integer
   width = 4

int   [2]   C

[3]   C   *type* = integer
                     width = 4

C

ϵ

Step 6:

- C() returns
- Execute semantic action

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

C(t, w)

C(t, w)

T()

Call stack

Translation Process Example

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

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

t = integer
w = 4

T

B $\quad$ *type* = integer
$\quad\quad$ *width* = 4

|
int

C

$\quad\quad$ **[2]**

C $\quad$ *type* = array(3, integer)
$\quad\quad\quad$ *width* = 3 * 4 = 12

[3] $\qquad\qquad$ C $\quad$ *type* = integer
$\qquad\qquad\qquad\qquad$ *width* = 4

|
ϵ

Step 7:

- C() returns
- Execute semantic action

$$
C \ \to \ [\ \textbf{num}\ ]\ C_1 \quad \left\{\ \begin{array}{l} C.type= \ array(\textbf{num}.value,\ C_1.type); \\ C.width = \textbf{num}.value \times C_1.width;\ \} \end{array}\right.
$$

C(t, w)

T()

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	$\{\ C.type=\ array(\textbf{num}.value,\ C_1.type);$
		$C.width = \textbf{num}.value \times C_1.width;\ \}$

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

t = integer
w = 4

T

B *type* = integer
   width = 4

C *type* = array(2, array(3, integer))
   width = 2 * 12 = 24

int

[2]

C *type* = array(3, integer)
   width = 3 * 4 = 12

[3]

C *type* = integer
   width = 4

ϵ

Step 8:

- C() returns
- Execute semantic action

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

T()

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	$\{ C.type= array(\textbf{num}.value, C_1.type); C.width = \textbf{num}.value \times C_1.width; \}$

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

t = integer
w = 4

T *type* = array(2, array(3, integer))
   width = 24

B *type* = integer
   width = 4

C *type* = array(2, array(3, integer))
   width = 2 * 12 = 24

int   [2]

C *type* = array(3, integer)
   width = 3 * 4 = 12

[3]   C *type* = integer
             width = 4

ϵ

Call stack

Step 8:

- T() returns
- Execute semantic action

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

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

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

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

$$
\begin{aligned}
T \;\to\; &\mathbf{record} \; '\{' & \{ \; &Env.push(top); \; top = \mathbf{new} \; Env(); \\
& & &Stack.push(offset); \; offset = 0; \; \} \\
&D \; '\}' & \{ \; &T.type = record(top); \; T.width = offset; \\
& & &top = Env.pop(); \; offset = Stack.pop(); \; \}
\end{aligned}
$$

- 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

- Translation of Expressions

- Control Flow (Lab)

- 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 <u>the way it is used</u>
    - **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: $t_1 = (\text{float})\ i \quad t_2 = x\ \text{fmul}\ t_1$

- Type conversion SDT for a simple case (using type synthesis)

  - $E \rightarrow E_1 + E_2$

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

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

  $\quad\quad \dots$

  $\}$

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
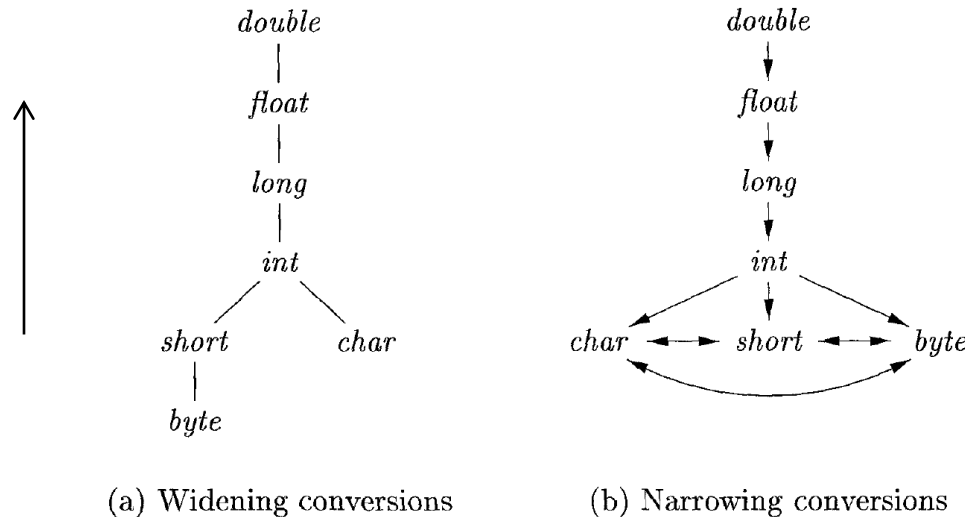
Widening and Narrowing (2)

- *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
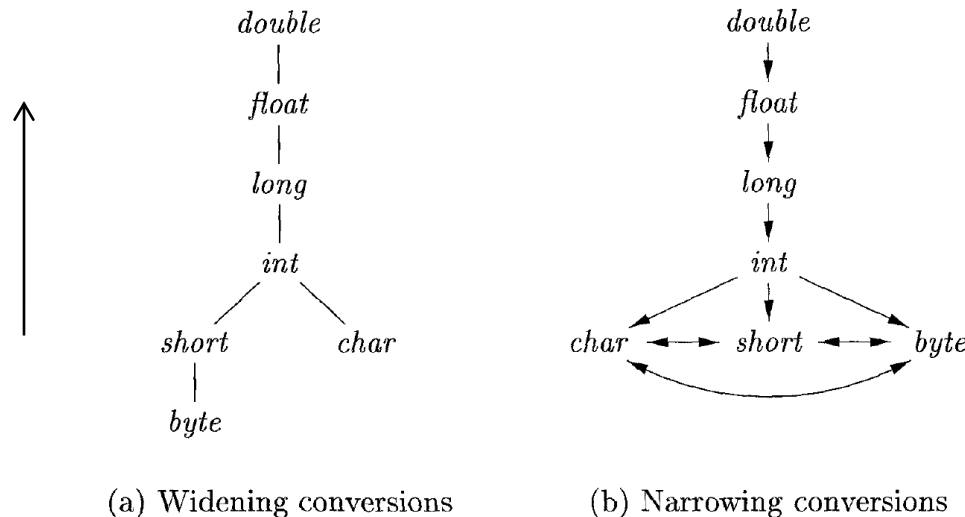
SDT for Type Conversion

- $\max(t_1, t_2)$ 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;
}
```

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

Example

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

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

Tree:
```
        E
       /|\
      E + E
      |   |
      a   b
```
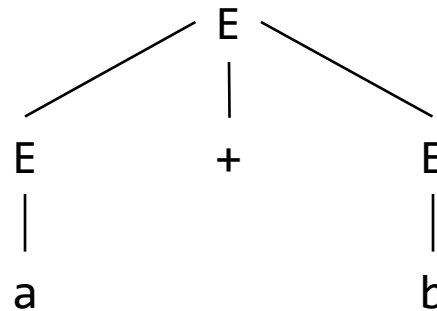
Generated code:

```
temp = (float) a   ---2
temp2 = temp + b   ---5
```

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

$E.type = max(\text{int}, \text{float}) = \text{float}$ 1

$a_1 = widen(\text{a}, \text{int}, \text{float}) = \text{temp}$ 2

$a_2 = widen(\text{b}, \text{float}, \text{float}) = \text{b}$ 3

$E.addr = \text{new } Temp() = \text{temp2}$ 4

5

Outline

- Intermediate Representation

- Type and Declarations

- Type Checking

- **Translation of Expressions**

- Control Flow (Lab)

- 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

$$a + b * c$$

$$t_1 = b * c$$
$$t_2 = a + t_1$$

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

SDD for Expression Translation – Attributes

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

$S.code$ 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
$S \rightarrow \mathbf{id} = E ;$	$S.code = E.code \;\|\|$ $\quad gen(top.get(\mathbf{id}.lexeme) \;'=' \; E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new} \; Temp\,()$ $E.code = E_1.code \;\|\| \; E_2.code \;\|\|$ $\quad gen(E.addr \;'=' \; E_1.addr \;'+' \; E_2.addr)$
$\quad\mid \; - E_1$	$\boxed{E.addr = \mathbf{new} \; Temp\,()}$ $\longrightarrow$ Temporary name generated by compiler $E.code = E_1.code \;\|\|$ $\quad gen(E.addr \;'=' \; '\mathbf{minus}' \; E_1.addr)$
$\quad\mid \; (\,E_1\,)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad\mid \; \mathbf{id}$	$\boxed{E.addr = top.get(\mathbf{id}.lexeme)}$ $\longrightarrow$ Check the symbol-table entry for **id** and save its address in *E.addr* $E.code = '\,'$

SDD for Expression Translation – Rules

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

- Generate instructions when seeing operations.

- Then concatenate instructions.

SDD for Expression Translation – Problem

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

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

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

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

    $| \quad - E_1$ $\{ E.addr = \mathbf{new} \ Temp \ (); \\ gen(E.addr \ '=' \ '\mathbf{minus}' \ E_1.addr); \}$

    $| \quad (E_1)$ $\{ E.addr = E_1.addr; \}$

    $| \quad \mathbf{id}$ $\{ E.addr = top.get(\mathbf{id}.lexeme); \}$

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

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

$$E \;\to\; E_1 + E_2 \quad \{\; E.addr = \textbf{new } Temp\,();$$
$$gen(E.addr\; '=' \; E_1.addr\; '+'\; E_2.addr);\; \}$$

$$\mid \;-\; E_1 \qquad \{\; E.addr = \textbf{new } Temp\,();$$
$$gen(E.addr\; '=' \; '\textbf{minus}'\; E_1.addr);\; \}$$

$$\mid \;(\,E_1\,) \qquad \{\; E.addr = E_1.addr;\; \}$$

$$\mid \;\textbf{id} \qquad \{\; E.addr = top.get(\textbf{id}.lexeme);\; \}$$

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)
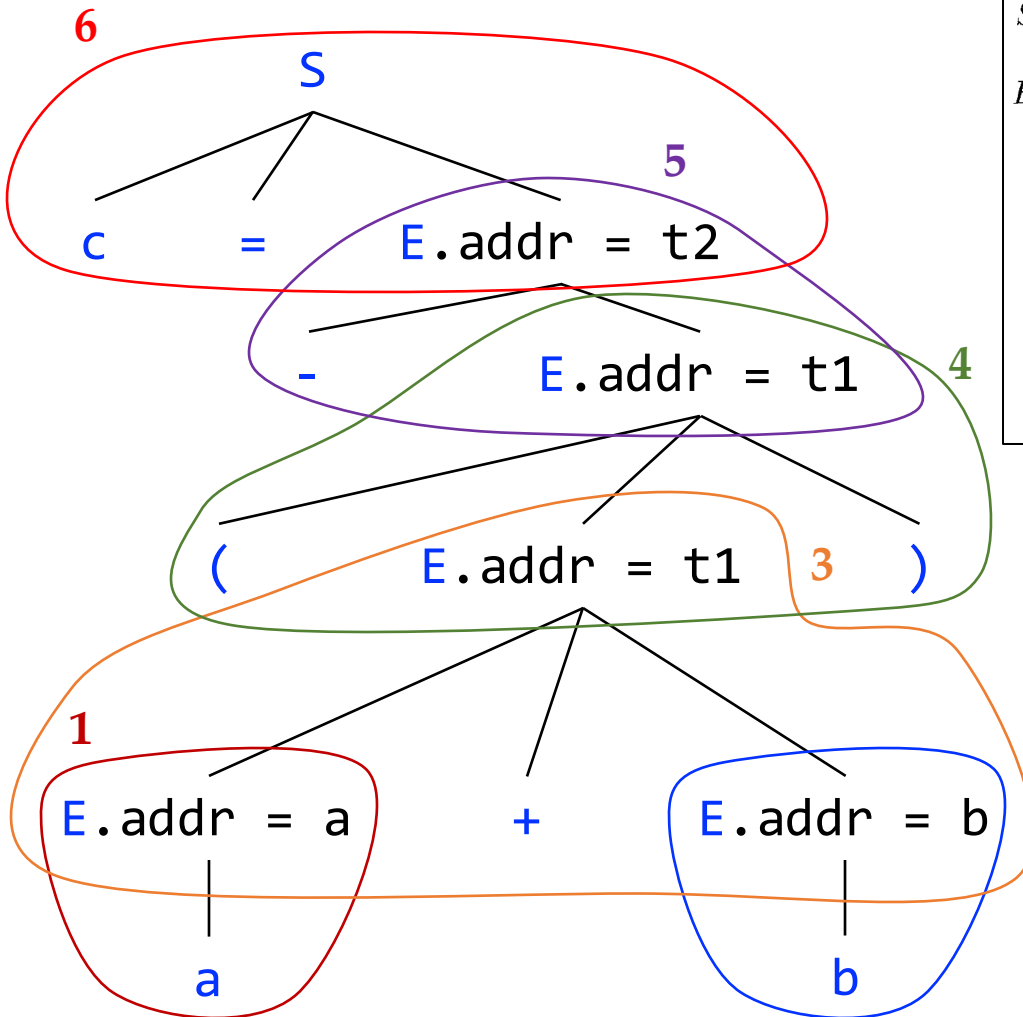
$S \rightarrow \textbf{id} = E \; ; \quad \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \}$ **6**

$E \rightarrow E_1 + E_2 \quad \{ \; E.addr = \textbf{new} \; Temp \; (); \quad$ **3**
$\qquad\qquad\qquad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \}$

$\quad | \;\; – E_1 \qquad\quad \{ \; E.addr = \textbf{new} \; Temp \; (); \quad$ **5**
$\qquad\qquad\qquad gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr); \; \}$

$\quad | \;\; (\; E_1 \;) \qquad \{ \; E.addr = E_1.addr; \; \}$ **4**

$\quad | \;\; \textbf{id} \qquad\qquad \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \}$ **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
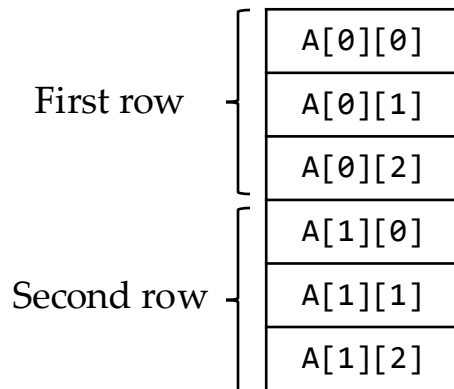
`c + a[i][j]` ➡️

```
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:

  - $base + i * w$ ($base$ 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:

  - $base + i_1 * w_1 + i_2 * w_2$ (w_1 is the width of a row, w_2 is the width of an element)

First row	A[0][0] / A[0][1] / A[0][2]
Second row	A[1][0] / A[1][1] / A[1][2]

Row-major (C)

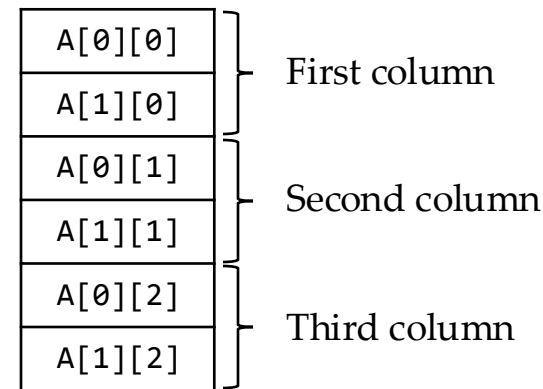|	
First column	A[0][0] / A[1][0]
Second column	A[0][1] / A[1][1]
Third column	A[0][2] / A[1][2]

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:
    - ***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:
    - ***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 $A[i_1][i_2] \dots [i_k]$ is:
    - ***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
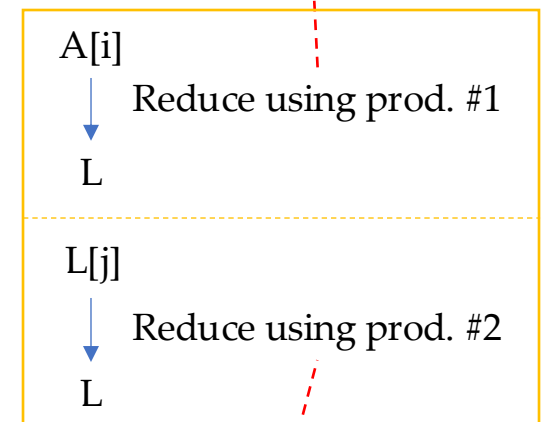
- The main problem in generating code for array references is to relate the address-calculation formula to the grammar

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

SDT for Array References (1)

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

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

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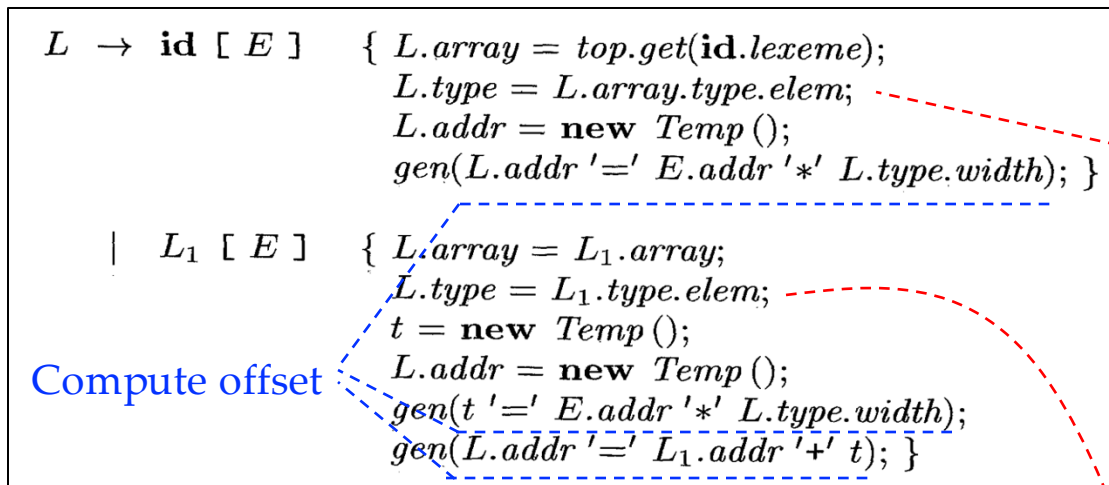
A is a 2*3 array of integers
Translate A[i][j]

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

$L.array$: 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

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

SDT for Array References (2)

- The semantic actions of L-productions compute offsets

- The address of an array element is $base + offset$

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

$$| \quad \textbf{id} \qquad\qquad \{ E.addr = top.get(\textbf{id}.lexeme); \}$$

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

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

Array references can be part of an expression

SDT for Array References (3)

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

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

Instruction of form $a[i] = x$

Array references can appear at the LHS of an assignment statement

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

$$\mid \quad \mathbf{id} \qquad \{ E.addr = top.get(\mathbf{id}.lexeme); \} \quad \mathbf{0} \; \mathbf{1} \; \mathbf{3}$$

$$\mid \quad L \qquad \{ E.addr = \mathbf{new} \; Temp\,(); \quad \mathbf{5}$$
$$gen(E.addr \; '=' \; L.array.base \; '[' \; L.addr \; ']'); \}$$

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

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

Translating c + a[i][j]
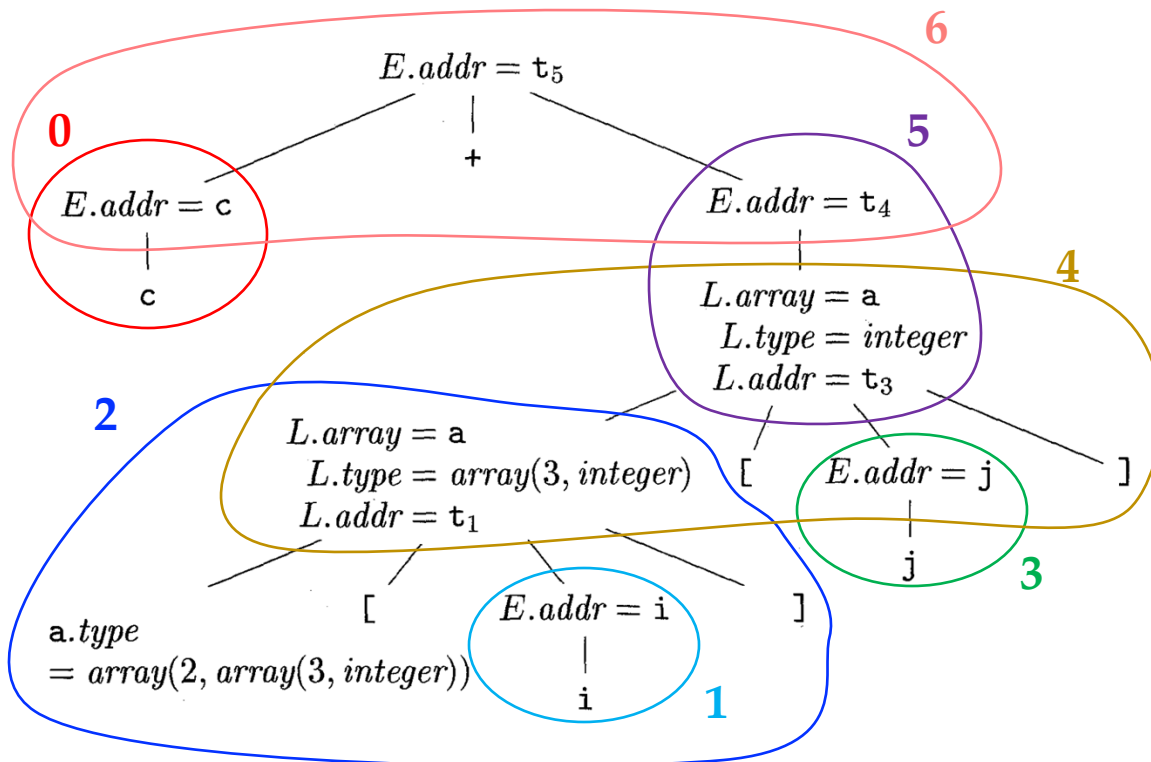
Generated code:

```
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 (Lab)**

- Backpatching (Lab)

Control Flow

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

- Grammar: $B \rightarrow B \parallel B \mid B \mathrel{\&\&} B \mid !\,B \mid (B) \mid E$ **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:
```
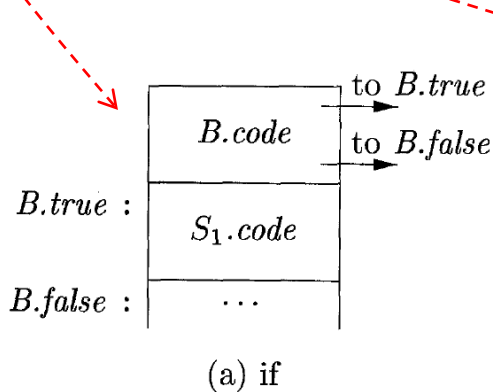
Flow-of-Control Statements

- 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
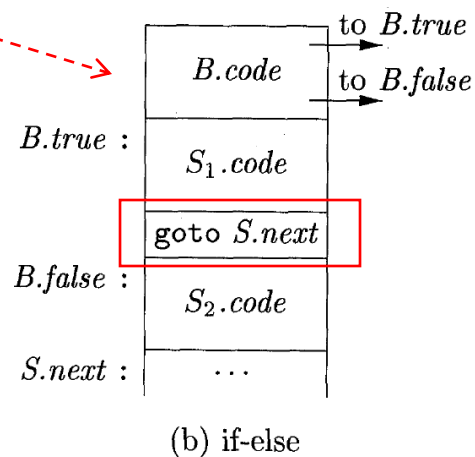
(b) if-else

(c) while

$S.next$ is not needed

$S.next$ is not needed

SDD for Flow-of-Control Statements (1)

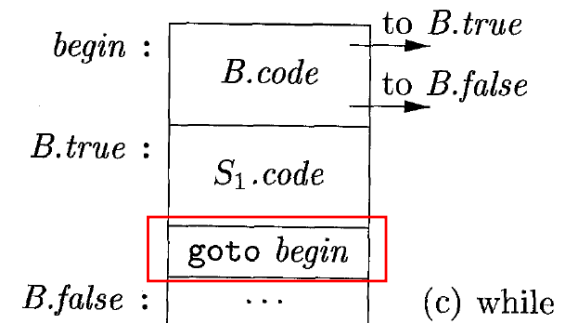| PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mathbin{\|} label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if } (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mathbin{\|} label(B.true)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

Illustrated by previous figures

SDD for Flow-of-Control Statements (2)

Illustrated by previous figure

$$S \rightarrow \textbf{while} \ (\ B \ ) \ S_1$$

$$begin \ = \ newlabel()$$
$$B.true \ = \ newlabel()$$
$$B.false \ = \ S.next$$
$$S_1.next \ = \ begin$$
$$S.code \ = \ label(begin) \ || \ B.code$$
$$|| \ label(B.true) \ || \ S_1.code$$
$$|| \ gen('\texttt{goto}' \ begin)$$

$$S \rightarrow S_1 \ S_2$$

$$S_1.next \ = \ newlabel()$$
$$S_2.next \ = \ S.next$$
$$S.code \ = \ S_1.code \ || \ label(S_1.next) \ || \ S_2.code$$

Translating Boolean Expressions in Flow-of-Control Statements

- A boolean expression B is translated into <u>three-address instructions</u> <u>that evaluate B using conditional and unconditional jumps</u> to one of two labels: $B.true$ and $B.false$

  - $B.true$ and $B.false$ are two inherited attributes. Their value depends on the context of B (e.g., if statement, $if\text{-}else$ statement, $while$ statement)

(a) if

(b) if-else

(c) while
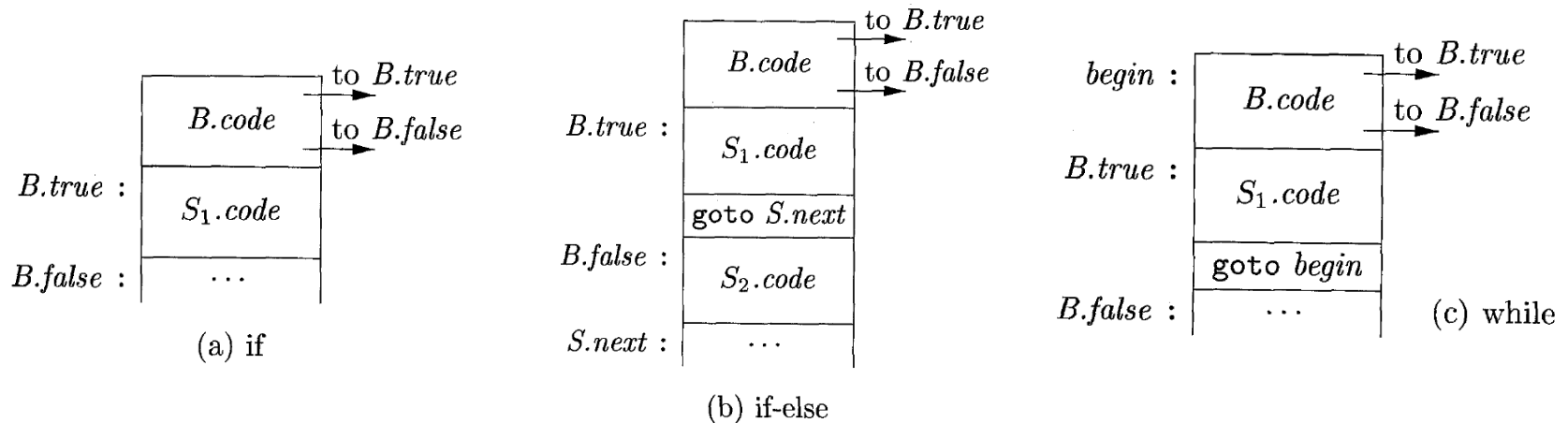
Generating Three-Address Code for Booleans (1)

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

$$B \rightarrow \text{true} \quad | \quad B.code = gen('\text{goto}'\; B.true)$$

$$B \rightarrow \text{false} \quad | \quad B.code = gen('\text{goto}'\; B.false)$$

Generating Three-Address Code for Booleans (2)

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

Example

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

```
                        p
                        |
                        s
        ┌───────┬───────┼───────┬───────────────┐
       if       (       B       )               s
                   ┌────┼────┐                   |
                   B    ||    B              assign
                 ┌─┴─┐      ┌─┴──┐          ╱  ┆  ╲
                 E < E      B && B         x   =   0
                 ┆   ┆    ┌─┴─┐  ┌─┴──┐
                 x  100   E > E  E != E
                          ┆   ┆  ┆    ┆
                          x  200  x    y
```
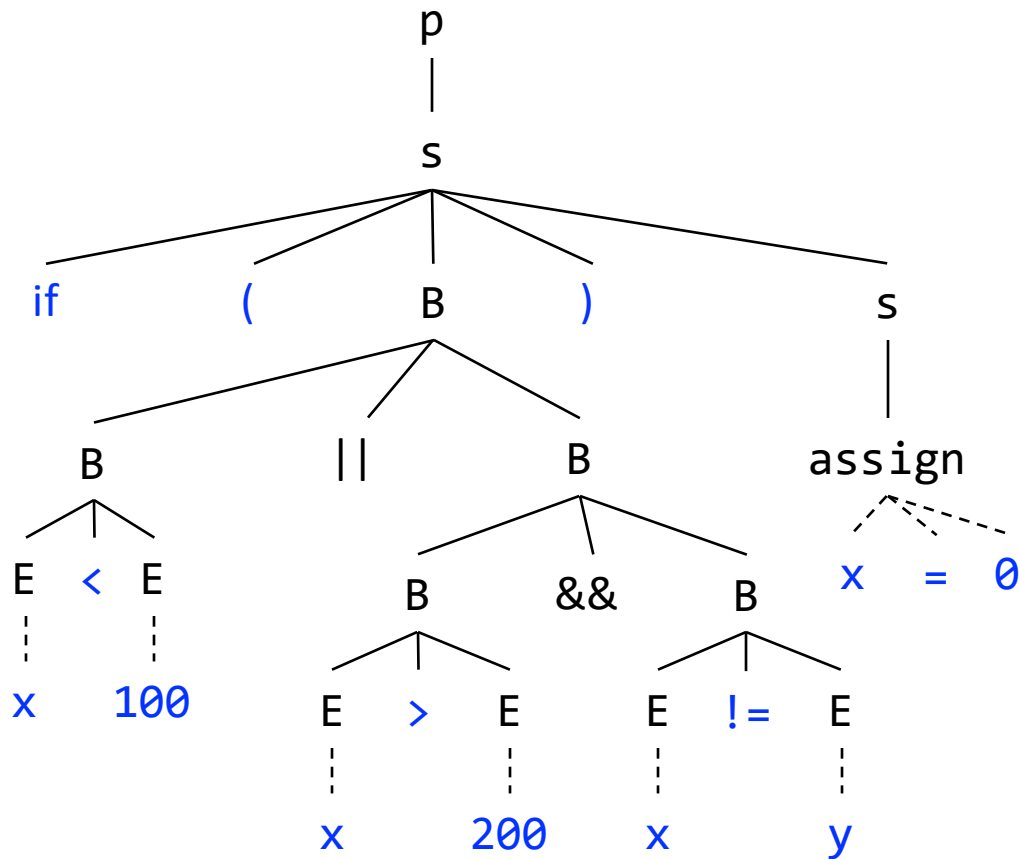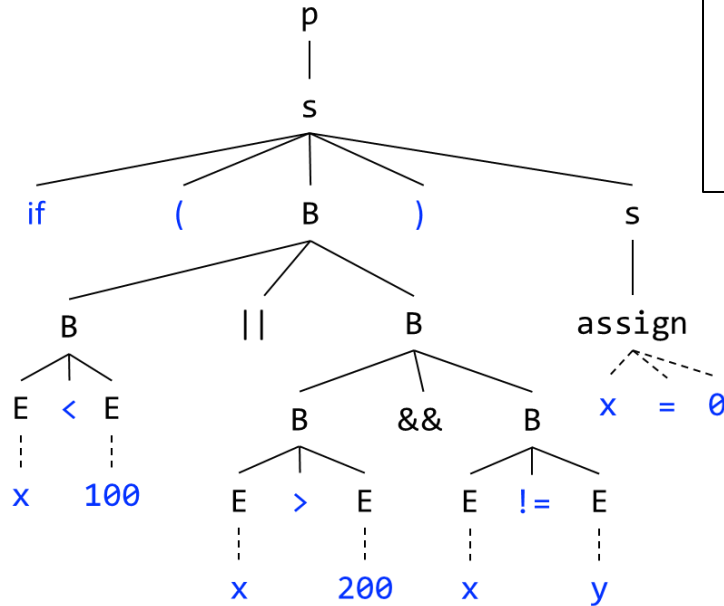
Dashed lines mean that
the reduction may consist
of multiple steps

Example

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

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

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

This SDD is L-attributed, not S-attributed. The grammar is not LL. There is no way to implement the SDD directly during parsing.
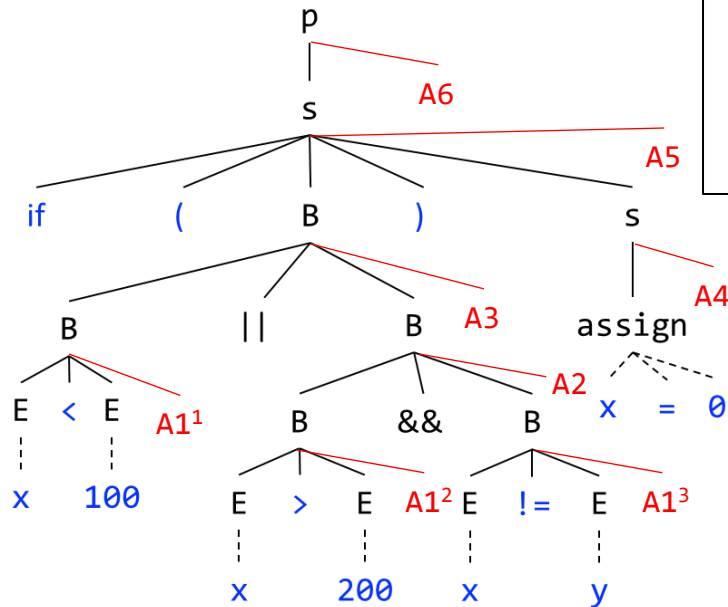
Traversing the parse tree to evaluate the attributes helps generate intermidate code

Example

p

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
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \| label(S.next)$ 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 $S.code = B.code \| label(B.true) \| S_1.code$

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

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

Example

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

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

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

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

Generated code:

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

Example

p
s
A6
if (B) s
A5
B || B A3
assign A4
B E < E A1¹
x 100
|| B && B A2
E > E A1² E != E A1³
x 200 x y
B x = 0

A1¹ **A1²** A1³ A2 A3 A4 A5 A6

Generated code:

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

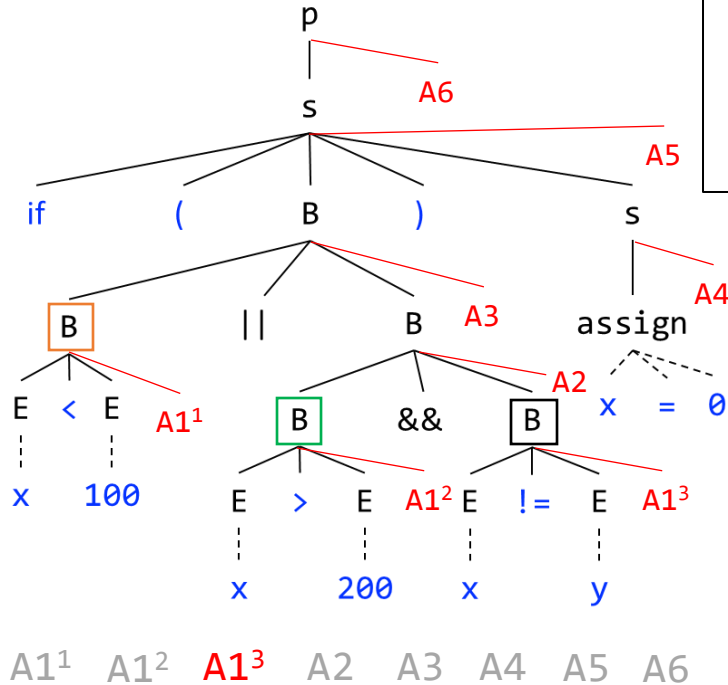| PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$ A6
$S \rightarrow \mathbf{assign}$	$S.code = \mathbf{assign}.code$ A4
$S \rightarrow \mathbf{if}\;(\;B\;)\;S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ A5 $S.code = B.code \;\|\|\;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

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

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

Example

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

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

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

A1¹ A1² **A1³** A2 A3 A4 A5 A6

Generated code:

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

Example

p

s   A6

A5

if   (B   )   s   A4

B || B   A3   assign

E < E   A1[1]   B   &&   B   A2   x = 0

x   100   E > E   A1[2]   E != E   A1[3]

x   200   x   y

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

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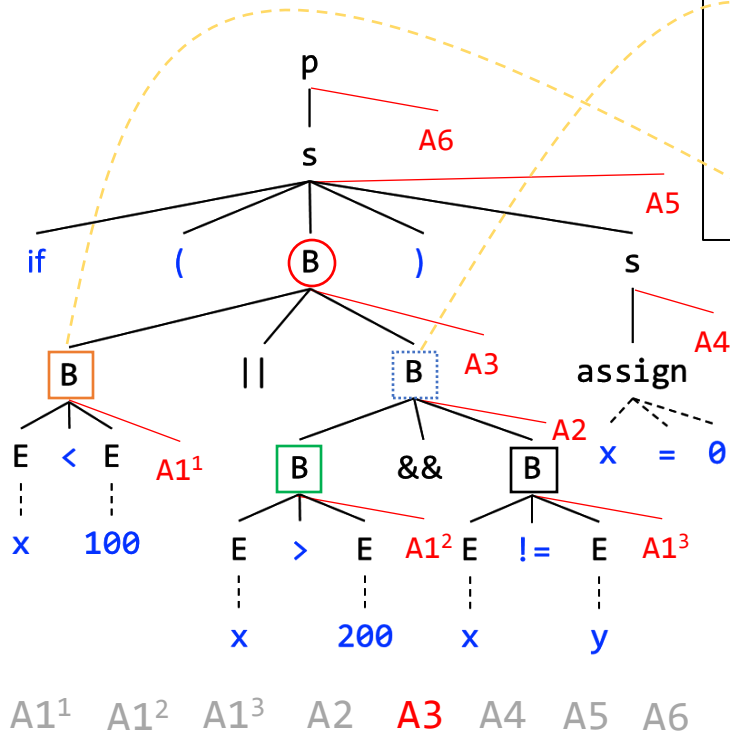
```
    if x < 100 goto B.true
    goto B.false
    if x > 200 goto B.true = L4
    goto B.false = B.false
L4: if x != y goto B.true = B.true
    goto B.false = B.false
```

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

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

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

Example

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

p
s   A6
if (B) s   A5
   A4
B || B   A3   assign
E < E   A1¹
x   100
B   &&   B   A2   x = 0
E > E   A1²   E != E   A1³
x   200   x   y

A1¹   A1²   A1³   A2   **A3**   A4   A5   A6

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

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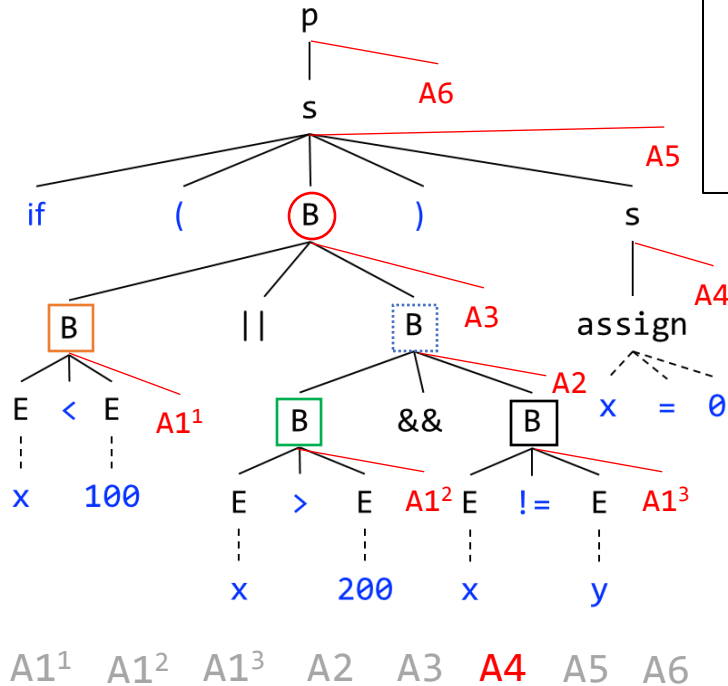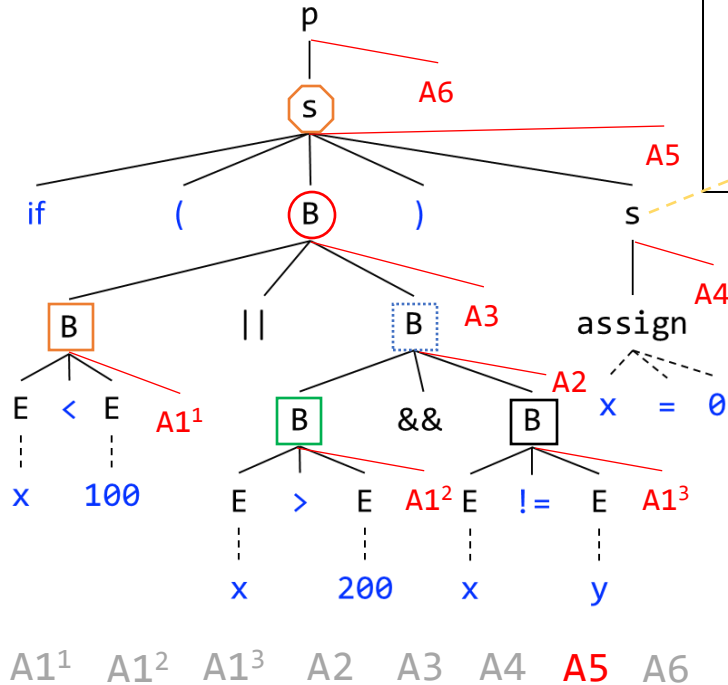
$B \rightarrow E_1 \ \textbf{rel} \ E_2$	$B.code = E_1.code \parallel E_2.code$ $\parallel gen('\texttt{if}' \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ A1 $\parallel gen('\texttt{goto}' \ B.false)$

Example

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

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

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

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

Generated code:

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

Example

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

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

$A1^1 \quad A1^2 \quad A1^3 \quad A2 \quad A3 \quad A4 \quad \textbf{A5} \quad A6$

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

Generated code:

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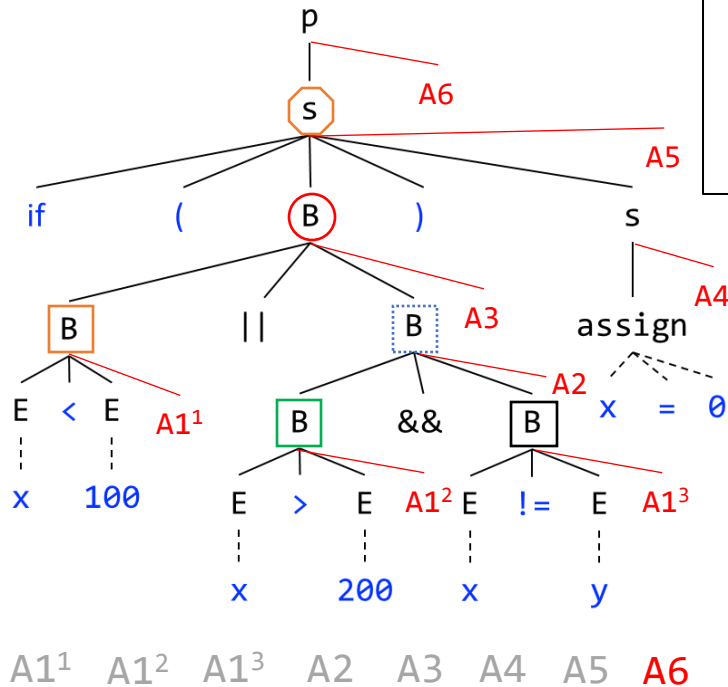
Example

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

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

A1¹ A1² A1³ A2 A3 A4 A5 **A6**

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

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: ···
```
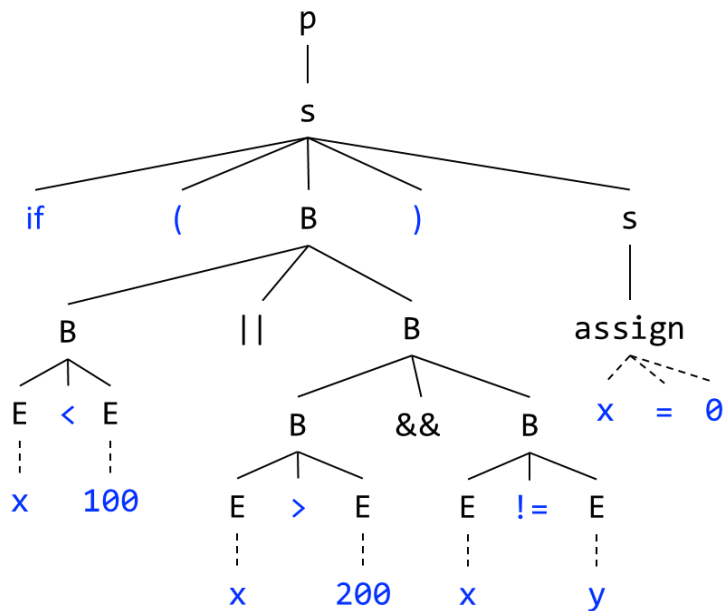
Example

- `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

- Translation of Expressions

- Control Flow (Lab)

- **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 another separate pass (traversing the parse tree) after parsing

How to address the problem in one pass?

One-Pass Code Generation Using Backpatching

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

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

One-Pass Code Generation Using Backpatching

- **The technique (技术细节) Cont.:**

  - $makelist(i)$: create a new list containing only i, the index of a jump instruction, and return the pointer to the list

  - $merge(p_1, p_2)$: concatenate the lists pointed by p_1 and p_2, and return a pointer to the concatenated list

  - $backpatch(p, i)$: insert i as the target for each of the jump instructions on the list pointed by p

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

- An SDT suitable for generating code for boolean expressions during bottom-up parsing

- Grammar:

  - $B \rightarrow B_1 \, \| \, M B_2 \mid B_1 \,\&\&\, M B_2 \mid \, ! \, B_1 \mid (B_1) \mid E_1 \text{ \bf rel } E_2 \mid \textbf{true} \mid \textbf{false}$

  - $M \rightarrow \epsilon$

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

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

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

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

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

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

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

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

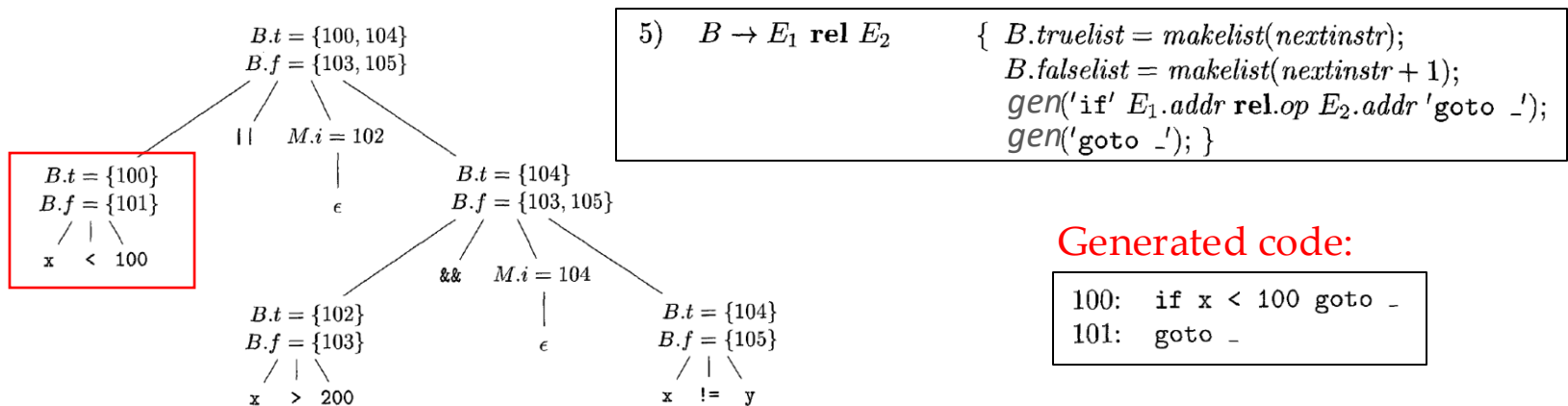8) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

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

Example – Boolean Expressions

- The earlier SDT is a postfix SDT. The semantic actions can be performed during a bottom-up parse.

- Boolean expression: $x < 100 \parallel x > 200 \;\&\&\; x \,!= y$

- **Step 1:** reduce $x < 100$ to B by production (5)

$B.t = \{100, 104\}$
$B.f = \{103, 105\}$

$||$ $M.i = 102$

$B.t = \{100\}$
$B.f = \{101\}$

$x \;<\; 100$

ϵ

$B.t = \{104\}$
$B.f = \{103, 105\}$

$\&\&$ $M.i = 104$

$B.t = \{102\}$
$B.f = \{103\}$

$x \;>\; 200$

ϵ

$B.t = \{104\}$
$B.f = \{105\}$

$x \;!= \; y$

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

Generated code:

```
100:    if x < 100 goto _
101:    goto _
```
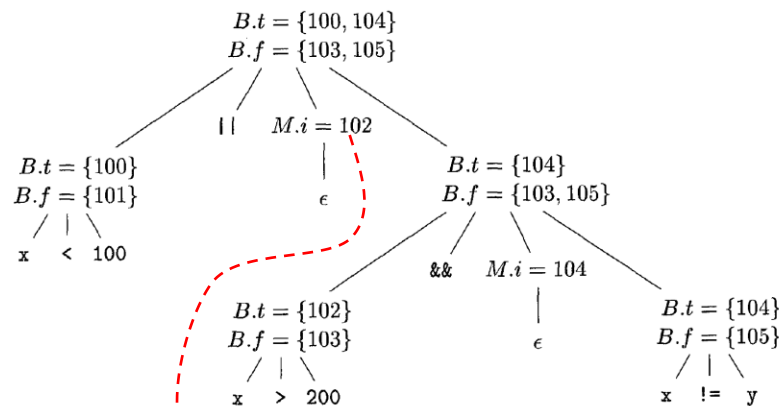
Example – Boolean Expressions

- The earlier SDT is a postfix SDT. The semantic actions can be performed during a bottom-up parse.

- Boolean expression: $x < 100 \parallel x > 200 \ \&\& \ x \mathbin{!=} y$
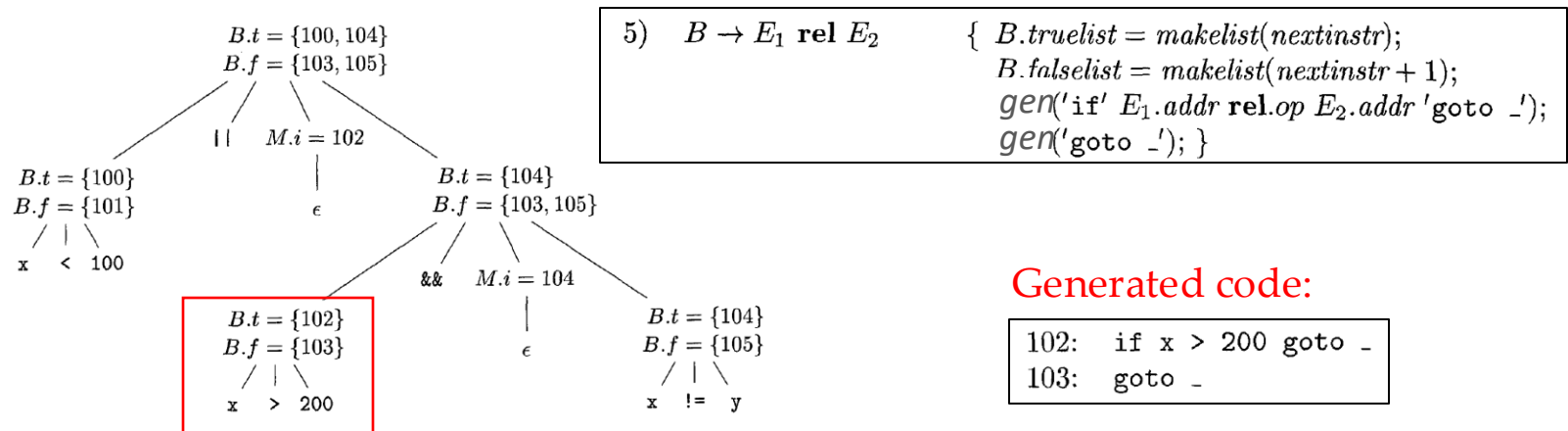
- **Step 2:** reduce ϵ to M by production (8)

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

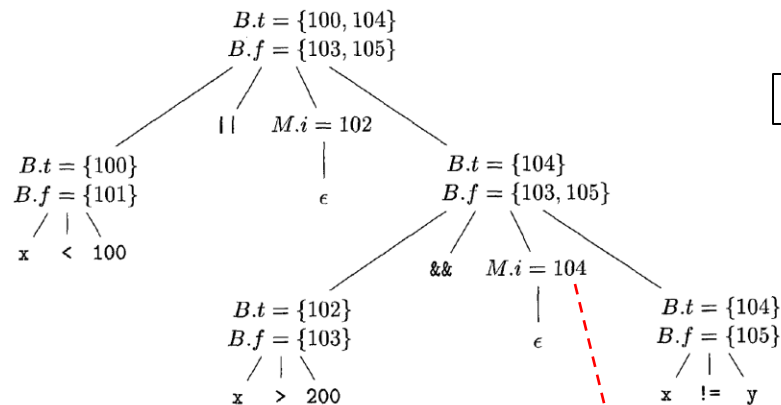The marker nonterminal records the value of *nextinstr*, 102

Example – Boolean Expressions

- Boolean expression: $x < 100 \parallel x > 200 \,\&\&\, x\,!=y$

- **Step 3:** reduce $x > 200$ to B by production (5)

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

Generated code:

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

Example – Boolean Expressions

- Boolean expression: $x < 100 \parallel x > 200 \,\&\&\, x\,! = y$

- **Step 4:** reduce ϵ to M by production (8)

$$B.t = \{100, 104\}$$
$$B.f = \{103, 105\}$$

$$\parallel \quad M.i = 102$$

$$B.t = \{100\}$$
$$B.f = \{101\}$$

$$x < 100$$

$$\epsilon$$

$$B.t = \{104\}$$
$$B.f = \{103, 105\}$$

$$B.t = \{102\}$$
$$B.f = \{103\}$$

$$x > 200$$

$$\&\& \quad M.i = 104$$

$$\epsilon$$

$$B.t = \{104\}$$
$$B.f = \{105\}$$

$$x \; != \; y$$

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

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

Example – Boolean Expressions

- Boolean expression: $x < 100 \,\|\, x > 200 \,\&\&\, x\,!=y$

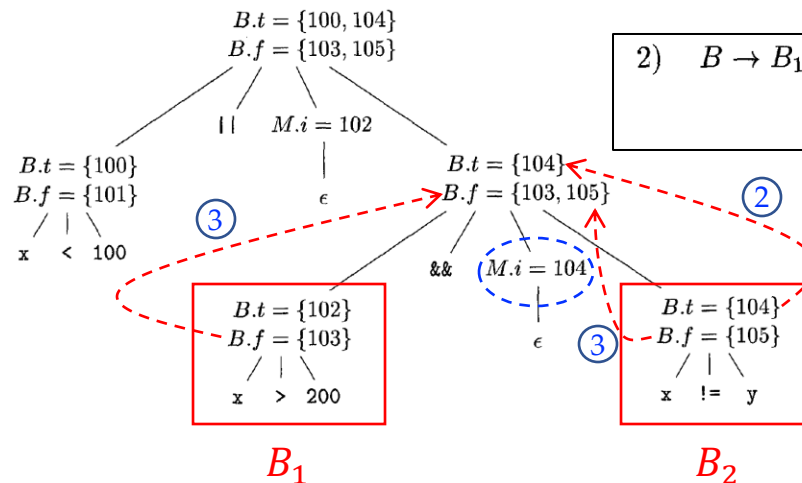- **Step 5:** reduce $x! = y$ to B by production (5)
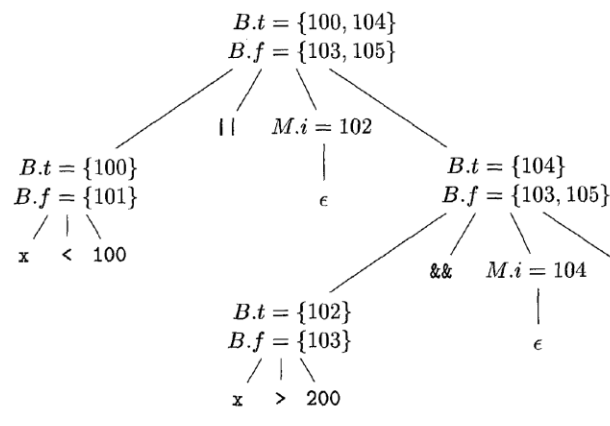
5) $\quad B \rightarrow E_1 \text{ \bf rel } E_2 \qquad \{ \ B.truelist = makelist(nextinstr);$
$\qquad\qquad\qquad\qquad\qquad\qquad B.falselist = makelist(nextinstr + 1);$
$\qquad\qquad\qquad\qquad\qquad\qquad gen('\texttt{if}'\ E_1.addr \text{ \bf rel}.op\ E_2.addr\ '\texttt{goto \_}');$
$\qquad\qquad\qquad\qquad\qquad\qquad gen('\texttt{goto \_}'); \ \}$

Generated code:

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

Example – Boolean Expressions

- Boolean expression: $x < 100 \;||\; x > 200 \;\&\&\; x\,!=\,y$

- **Step 6:** reduce $B_1 \;\&\&\; M B_2$ to B by production (2)

$$2) \quad B \rightarrow B_1 \;\&\&\; M \; B_2 \quad \{\; backpatch(B_1.truelist, M.instr); \;\;①$$
$$B.truelist = B_2.truelist; \;\;②$$
$$B.falselist = merge(B_1.falselist, B_2.falselist); \;\;③$$
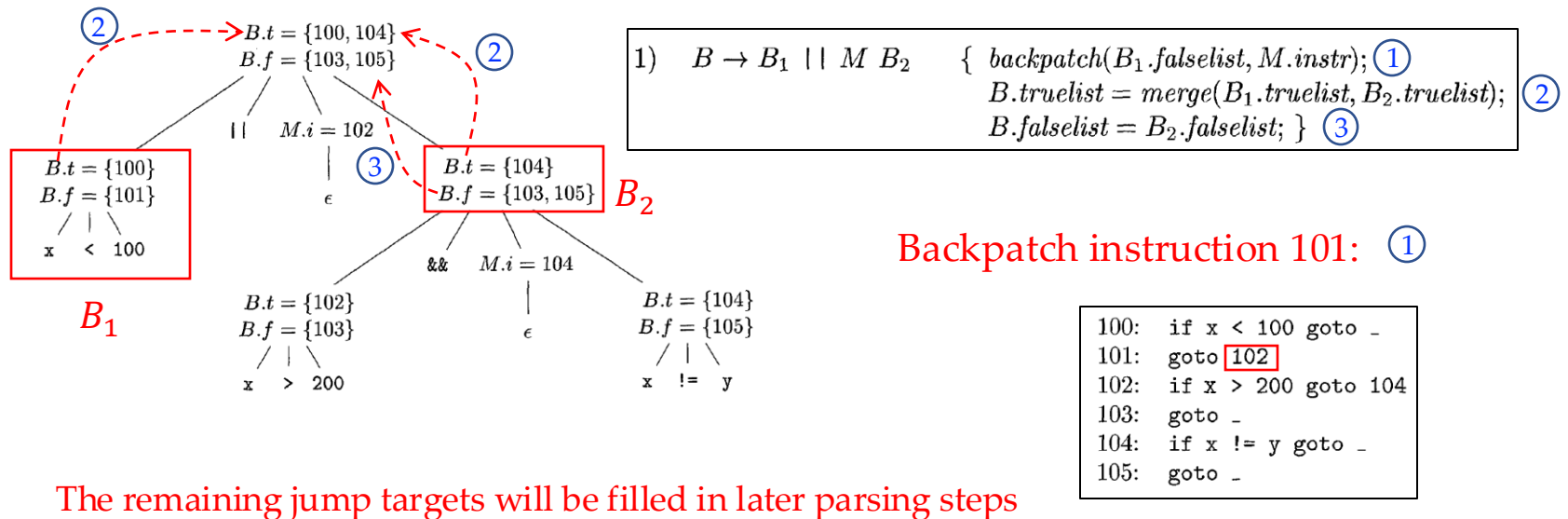
Backpatch instruction 102: ①

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

B_1 B_2

Example – Boolean Expressions

- Boolean expression: $x < 100 \| x > 200 \&\& x != y$

- **Step 7:** reduce $B_1 \| MB_2$ to B by production (1)
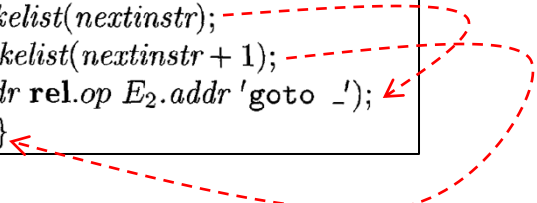

1) $B \to B_1 \mid\mid M\ B_2$ { $backpatch(B_1.falselist, M.instr)$; ①
$B.truelist = merge(B_1.truelist, B_2.truelist)$; ②
$B.falselist = B_2.falselist;$ } ③

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:

$$
\begin{array}{l|l}
B \rightarrow E_1 \ \textbf{rel} \ E_2 & B.code = E_1.code \ || \ E_2.code \\
& \quad || \ gen(\text{'if'} \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ \text{'goto'} \ B.true) \\
& \quad || \ gen(\text{'goto'} \ B.false)
\end{array}
$$

(2) Backpatching scheme:

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

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:

$$
\begin{array}{l|l}
B \;\rightarrow\; B_1 \;||\; B_2 & B_1.true = B.true \\
& B_1.false = newlabel() \\
& B_2.true = B.true \\
& B_2.false = B.false \\
& B.code = B_1.code \;||\; label(B_1.false) \;||\; B_2.code
\end{array}
$$

(2) Backpatching scheme:

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

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)