南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 4: Intermediate-Code Generation

Yepang Liu

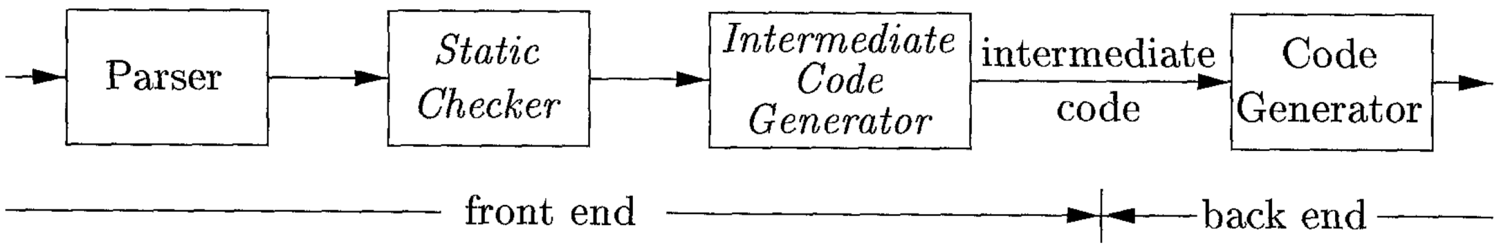
liuyp1@sustech.edu.cn

Outline

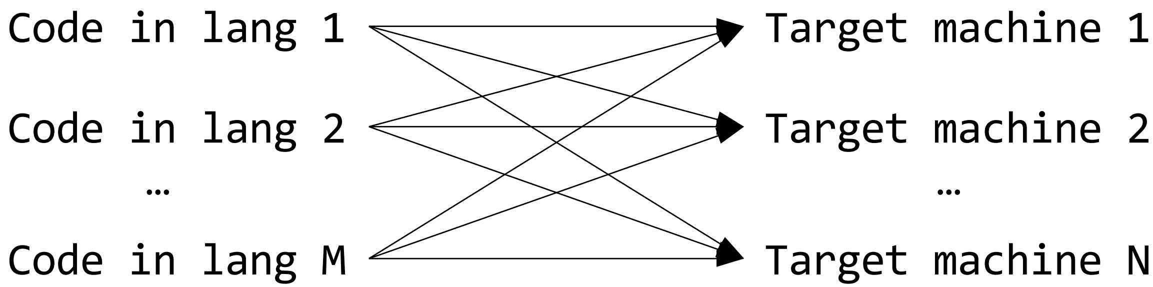
- Intermediate Representation
- Type and Declarations
- Type Checking
- Translation of Expressions
- Control Flow

Compiler Front End

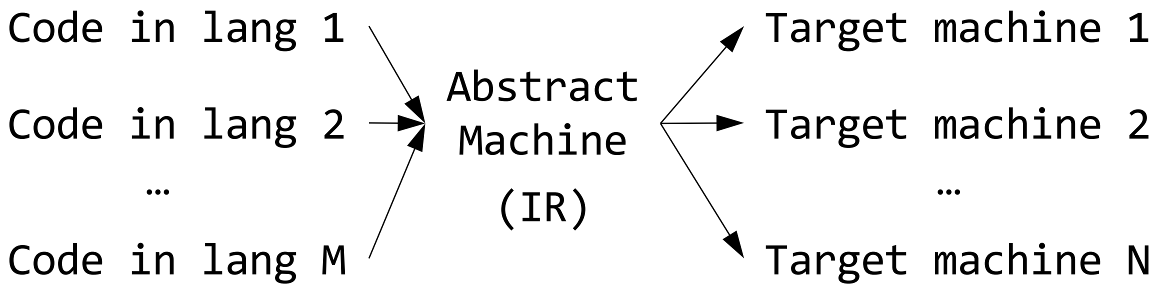
- The front end of a compiler analyzes a source program and creates an intermediate representation (IR, 中间表示), from which the back end generates target code
  - Details of the source language are confined to the front end, and details of the target machine to the back end



The Benefits of A Common IR

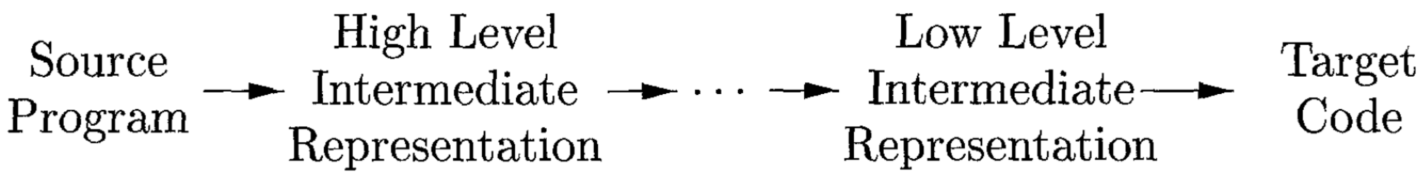


M * N compilers
without a common IR



M + N compilers
with a common IR

Different Levels of IRs



- 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 register allocation and instruction selection

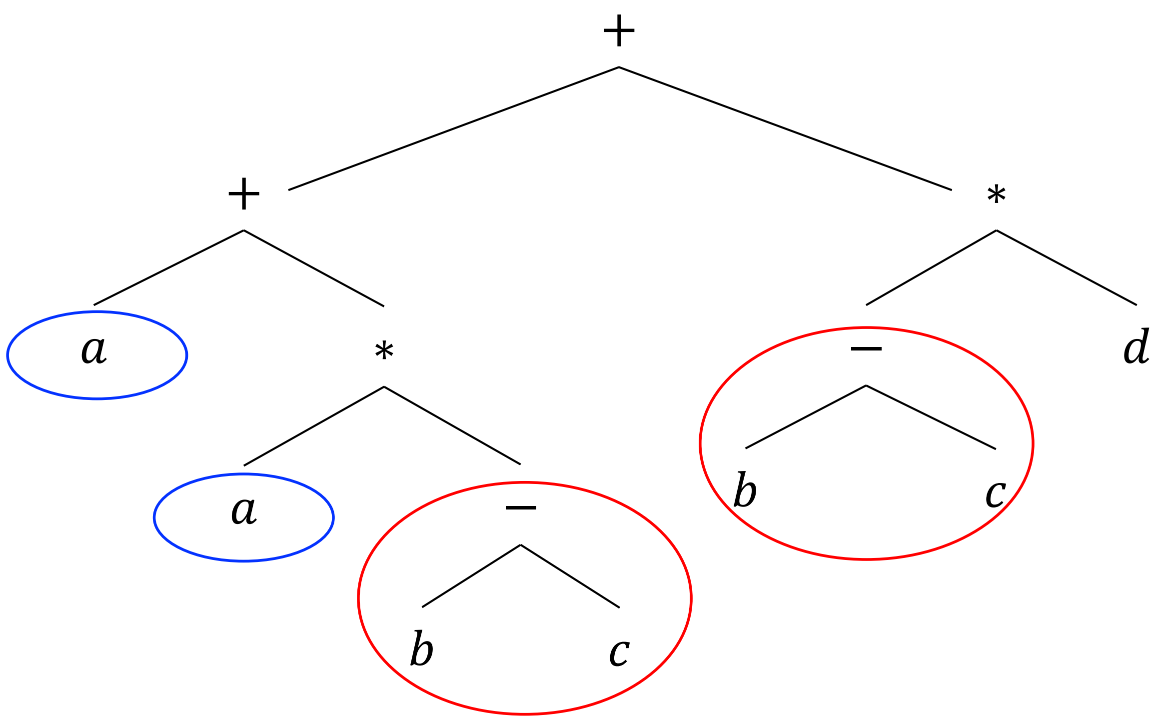
- Interesting fact: C is often used as an intermediate form. The first C++ compiler has a front end that generates C and a C compiler as a backend.

Outline

- Intermediate Representation
  - DAG's for Expressions
  - Three-Address Code
- Type and Declarations
- Type Checking
- Translation of Expressions
- Control Flow

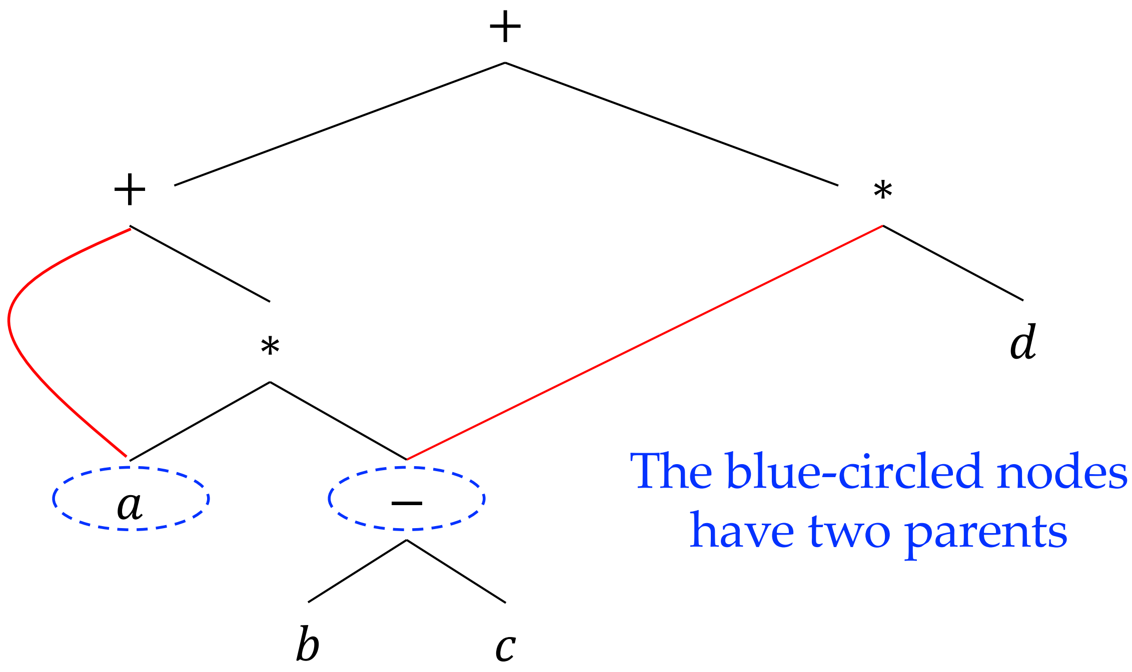
DAG's for Expressions

- 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

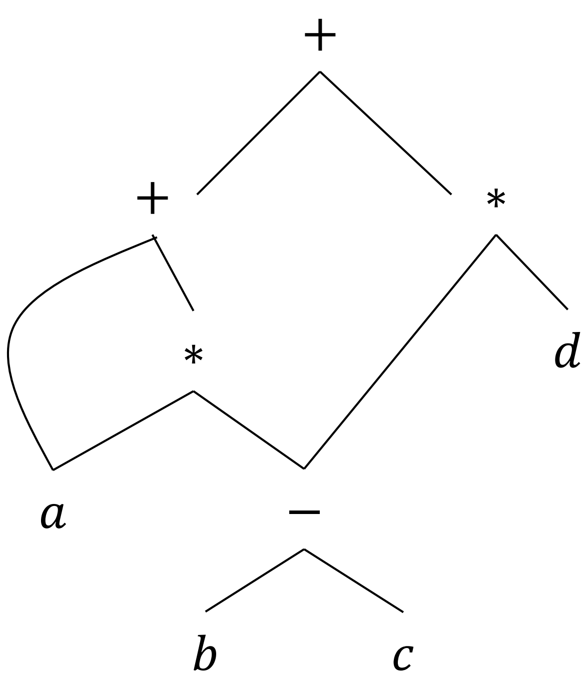
	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \boxed{\textbf{new}}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \boxed{\textbf{new}}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \boxed{\textbf{new}}\ Leaf(\textbf{id}, \textbf{id}.entry)$
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) $p_1 = \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}a)$
2) $p_2 = \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}a) = p_1$
3) $p_3 = \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}b)$
4) $p_4 = \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}c)$
5) $p_5 = \mathit{Node}('-', p_3, p_4)$
6) $p_6 = \mathit{Node}('*', p_1, p_5)$
7) $p_7 = \mathit{Node}('+', p_1, p_6)$
8) $p_8 = \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}b) = p_3$
9) $p_9 = \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}c) = p_4$
10) $p_{10} = \mathit{Node}('-', p_3, p_4) = p_5$
11) $p_{11} = \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}d)$
12) $p_{12} = \mathit{Node}('*', p_5, p_{11})$
13) $p_{13} = \mathit{Node}('+', p_7, p_{12})$

Node reuse

Outline

- Intermediate Representation
  - DAG's for Expressions
  - Three-Address Code
- Type and Declarations
- Type Checking
- Translation of Expressions
- Control Flow

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);`

(a) Symbolic labels.		(b) Position numbers.	
L:	t_1 = i + 1	100:	t_1 = i + 1
	i = t_1	101:	i = t_1
	t_2 = i * 8	102:	t_2 = i * 8
	t_3 = a [t_2]	103:	t_3 = a [t_2]
	if t_3 < v goto L	104:	if t_3 < v goto 100

(a) Symbolic labels. (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: $x = y + z \rightarrow +\quad y\quad z\quad x$
- 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: $a = b * -c + b * -c$

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

(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
		. . .		

(b) Quadruples

Temporaries

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	

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?

b*-c

-c

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

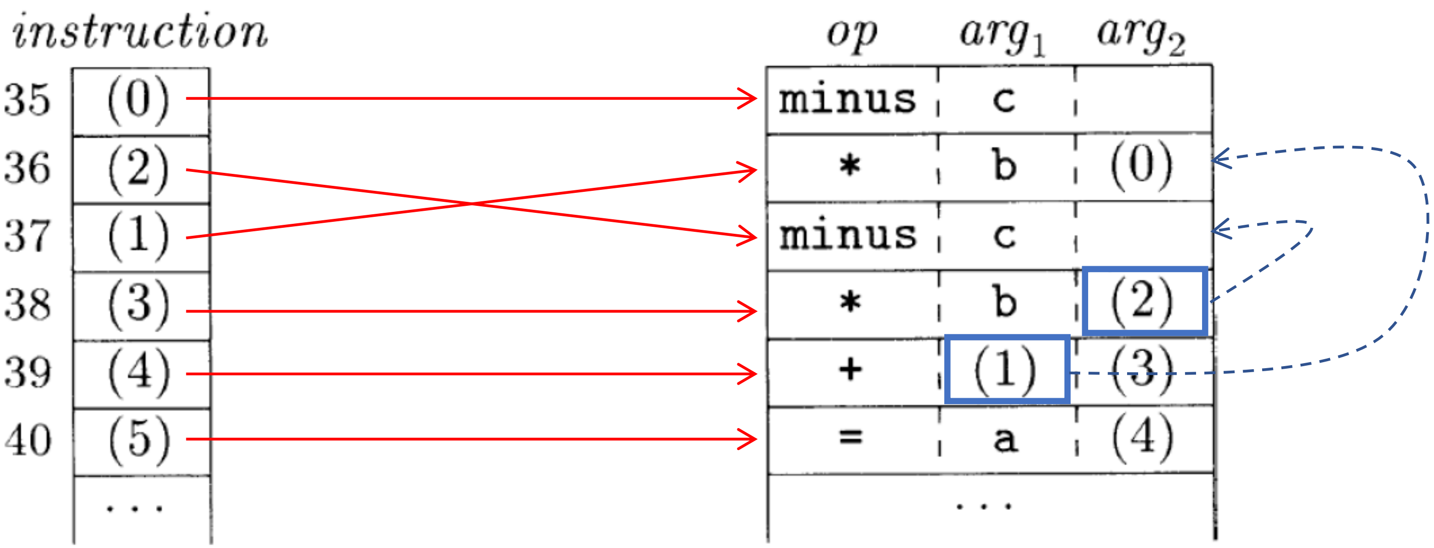
	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	…

Points to

op	arg_1	arg_2
minus	c	
*	b	(0)
minus	c	
*	b	(2)
+	(1)	(3)
=	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

$$
\begin{aligned}
\boxed{p} &= a + b \\
q &= p - c \\
\boxed{p} &= q * d \\
\boxed{p} &= e - p \\
q &= p + q
\end{aligned}
$$

(a) Three-address code.

$$
\begin{aligned}
\boxed{p_1} &= a + b \\
q_1 &= p_1 - c \\
\boxed{p_2} &= q_1 * d \\
\boxed{p_3} &= e - p_2 \\
q_2 &= p_3 + q_1
\end{aligned}
$$

(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

$$
\begin{aligned}
&\texttt{if (flag) } x_1 \texttt{ = -1; else } x_2 \texttt{ = 1;}\\
&x_3 = \phi(x_1, x_2); \quad \text{// x1 if control flow passes through the true path; x2 otherwise}\\
&\texttt{y = } x_3 \texttt{ * a;}
\end{aligned}
$$

Outline

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

Types and Type Checking

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

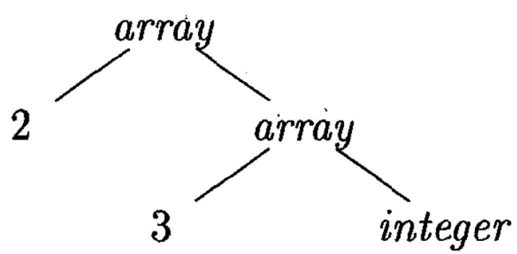
- The usefulness of type information
  - Find faults in the source code
  - Determine the storage needed for a name at runtime
  - Calculate the address of an array element
  - Insert type conversions
  - Choose the right version of some arithmetic operator (e.g., `fadd`, `iadd`)

- *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 structures, 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 zero or more integers, each surrounded by brackets

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

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 During Recursive-Descent Parsing

- It is possible to extend a recursive-descent parser to implement L-attributed SDD's.
  - Recall that a recursive-decent parser has a function *A* for each nonterminal *A*

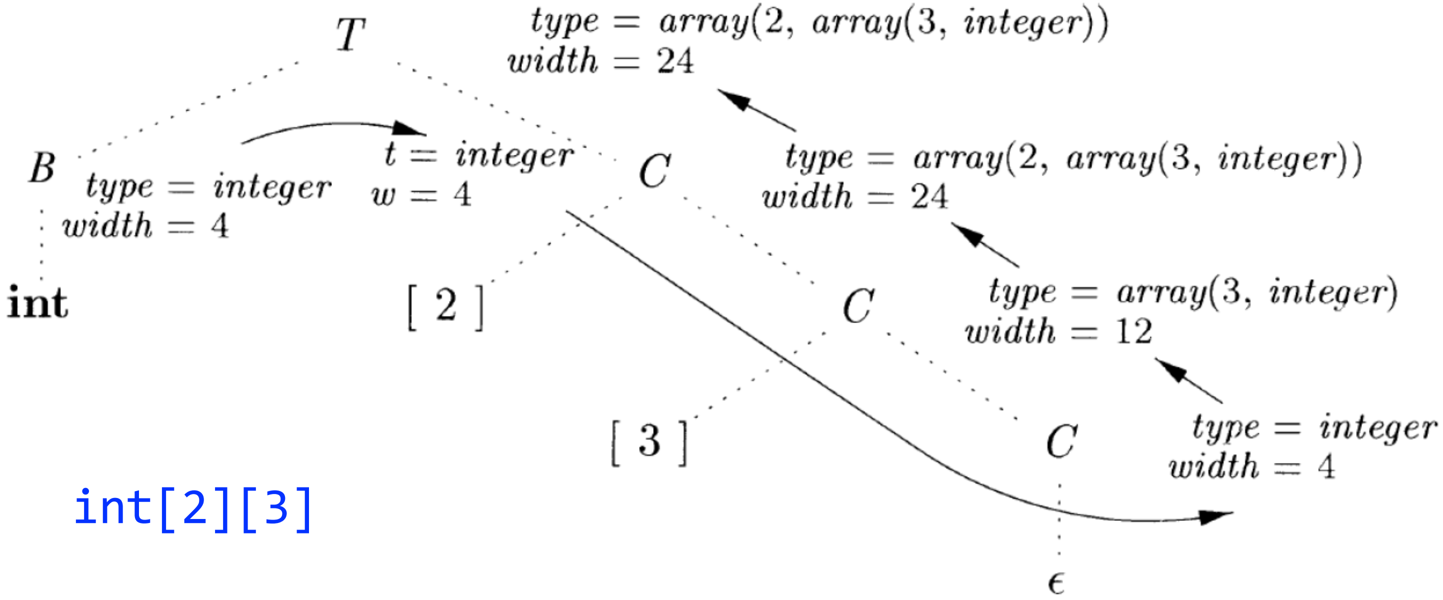
```
void A() {
    Choose an A-production, A → X1 X2 ··· Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Translation During Recursive-Descent Parsing

- Generally, we can extend a recursive-descent parser to implement L-attributed SDD's as follows:
  - A recursive-decent parser has a function *A* for each nonterminal *A*
  - Use the arguments of function *A* to **pass** *A*'s inherited attributes so that children nodes on the parse tree can use the attributes
  - **Return** the synthesized attributes of *A* when the function *A* completes so that parent node on the parse three can use the attributes
- With the above extension, in the body of the function *A*, we need to both parse and handle attributes

Translation Process Example

- **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

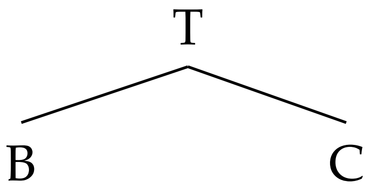


* In our example, we use global variables *t* and *w*

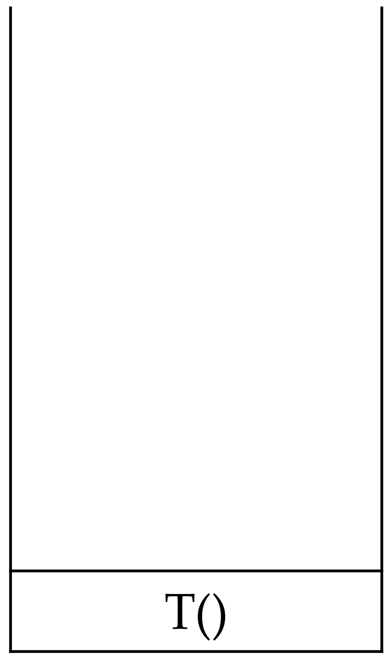
Translation Process Example

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

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



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

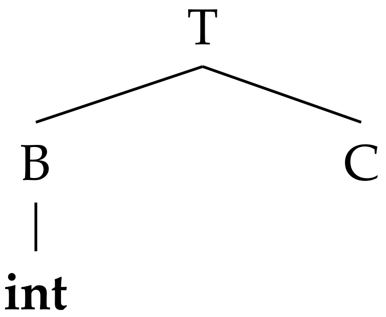


Call stack

Translation Process Example

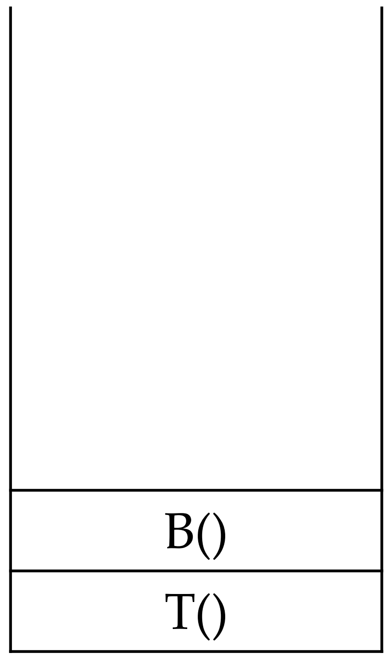
$T \rightarrow B$		$\{ t = B.type;\ w = B.width; \}$
C		$\{T.type = C.type;\ T.width = C.width; \}$
$B \rightarrow$ **int**		$\{ B.type = integer;\ B.width = 4; \}$
$B \rightarrow$ **float**		$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow \epsilon$		$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow$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

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



Step 2:

- Rewrite B using $B \rightarrow$ **int**
- Match input

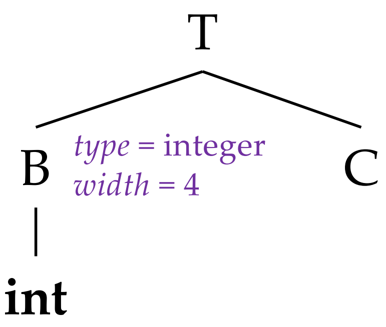


Call stack

Translation Process Example

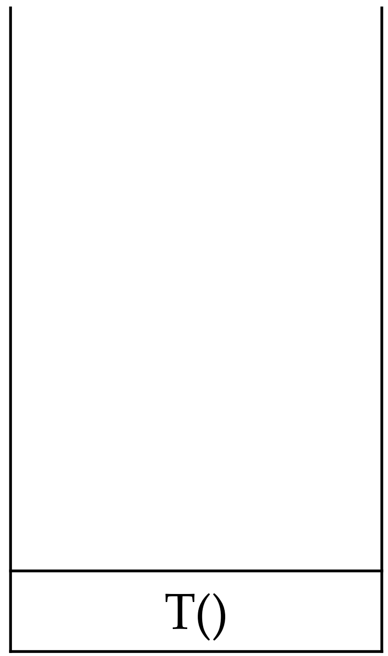
T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	$\{T.type = C.type;\ T.width = C.width;\}$
B	$\rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	[**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

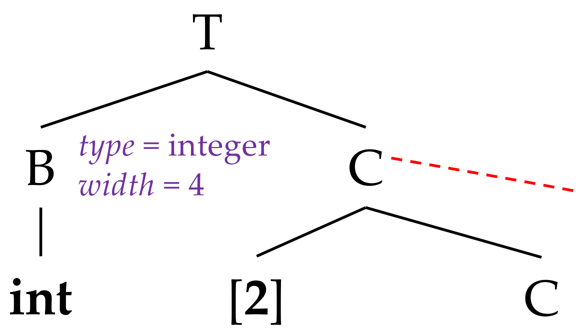
$$B \ \rightarrow \ \mathbf{int} \qquad \{\ B.type = integer;\ B.width = 4;\ \}$$

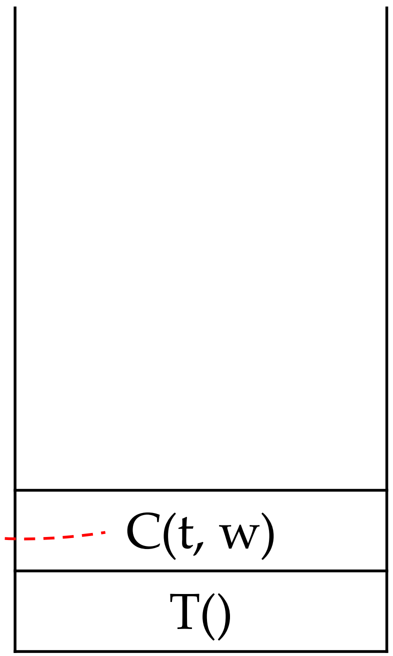
Translation Process Example

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	$\{T.type{=}C.type;\ T.width{=}C.width;\}$
B	$\rightarrow$	**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 = integer
w = 4





Call stack

Step 4:

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

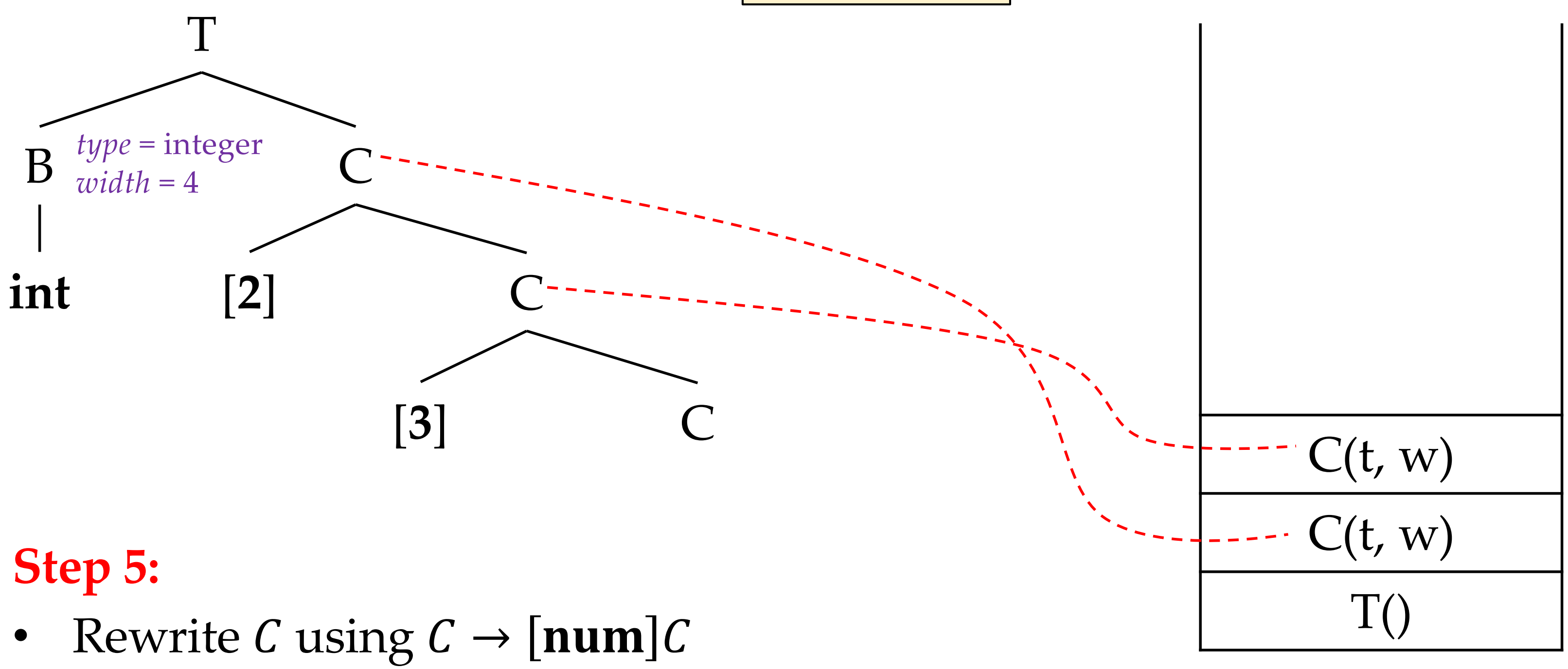
$$T \rightarrow B \quad \boxed{\{\ t = B.type;\ w = B.width;\ \}}$$
$$C \quad \{T.type{=}C.type;\ T.width{=}C.width;\}$$

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

Translation Process Example

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

t = integer
w = 4



Call stack

Step 5:

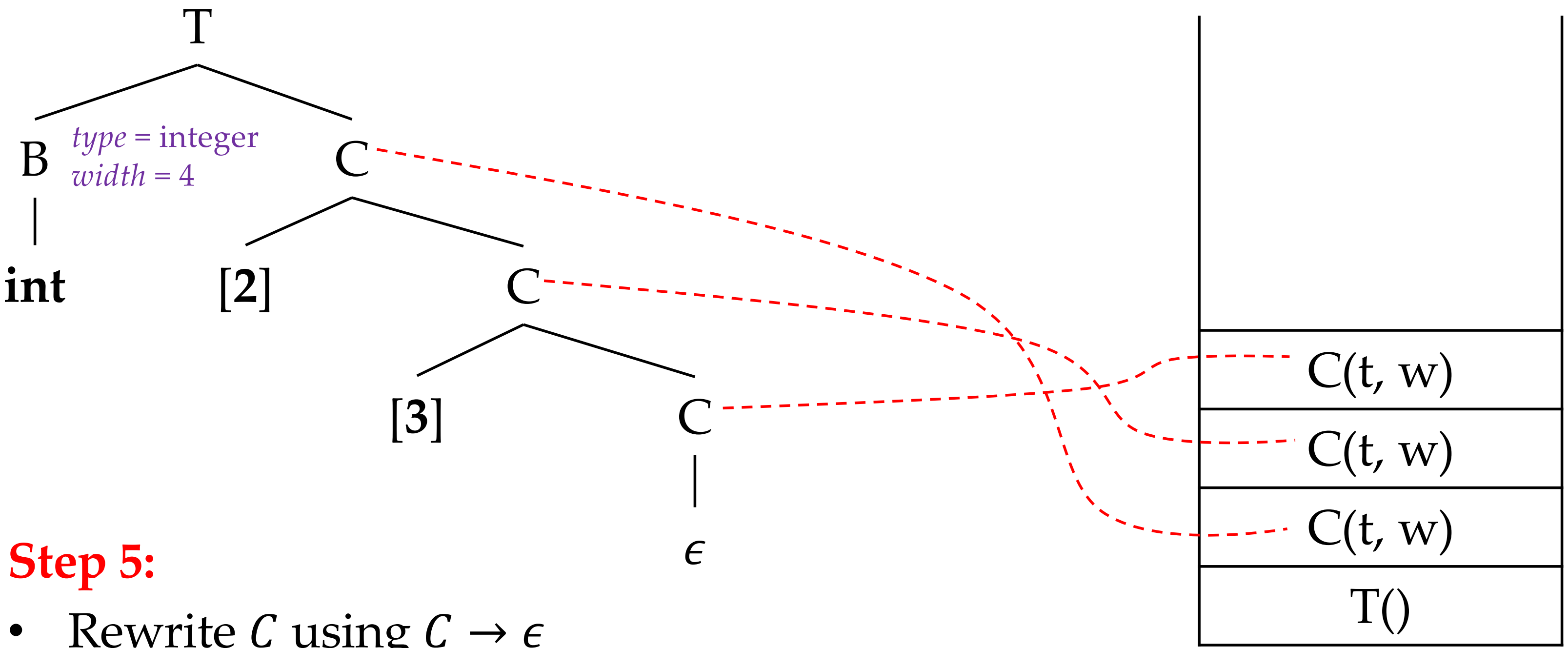
- Rewrite C using $C \rightarrow [\textbf{num}]C$
- Match input

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**	$\{\, B.type = float;\ B.width = 8;\, \}$
$C \rightarrow \epsilon$	$\{\, C.type = t;\ C.width = w;\, \}$
$C \rightarrow$ [**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:

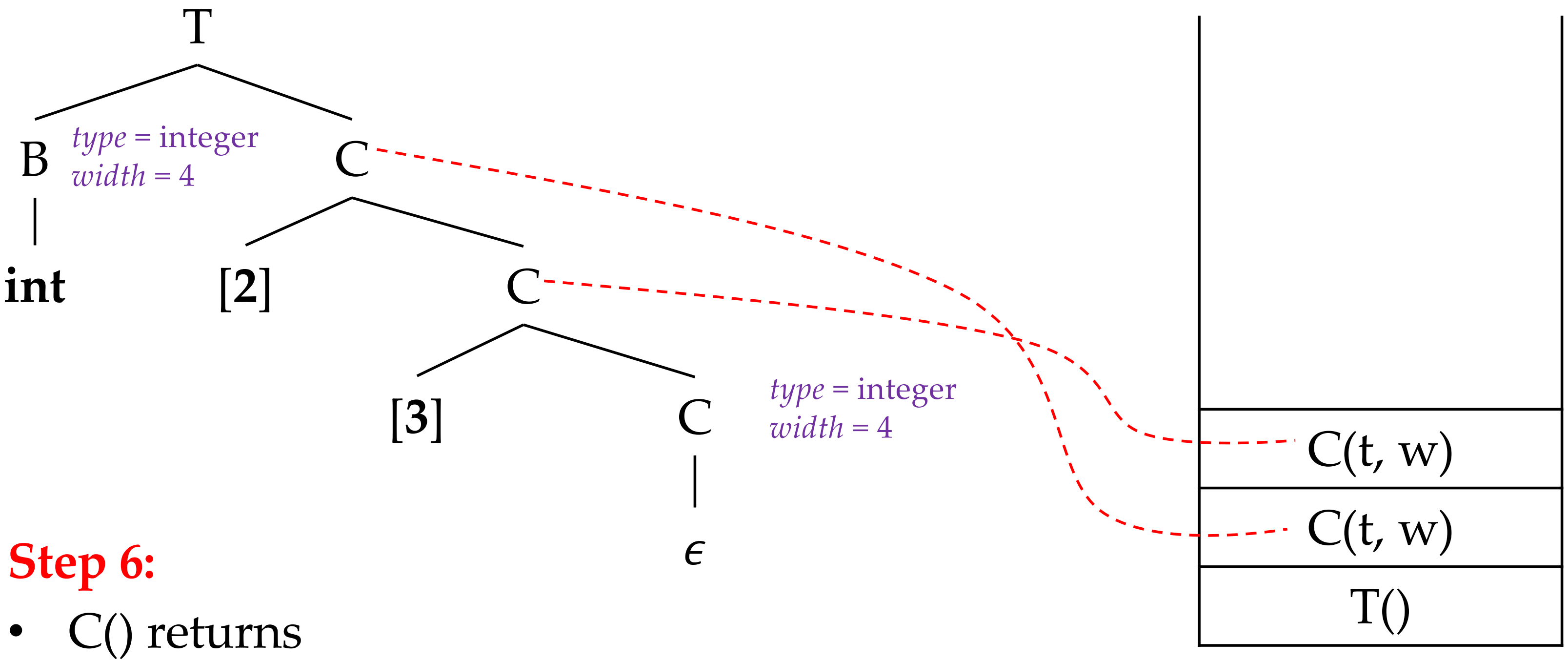
- Rewrite C using $C \rightarrow \epsilon$

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 = integer
w = 4



Call stack

Step 6:

- C() returns
- Execute semantic action

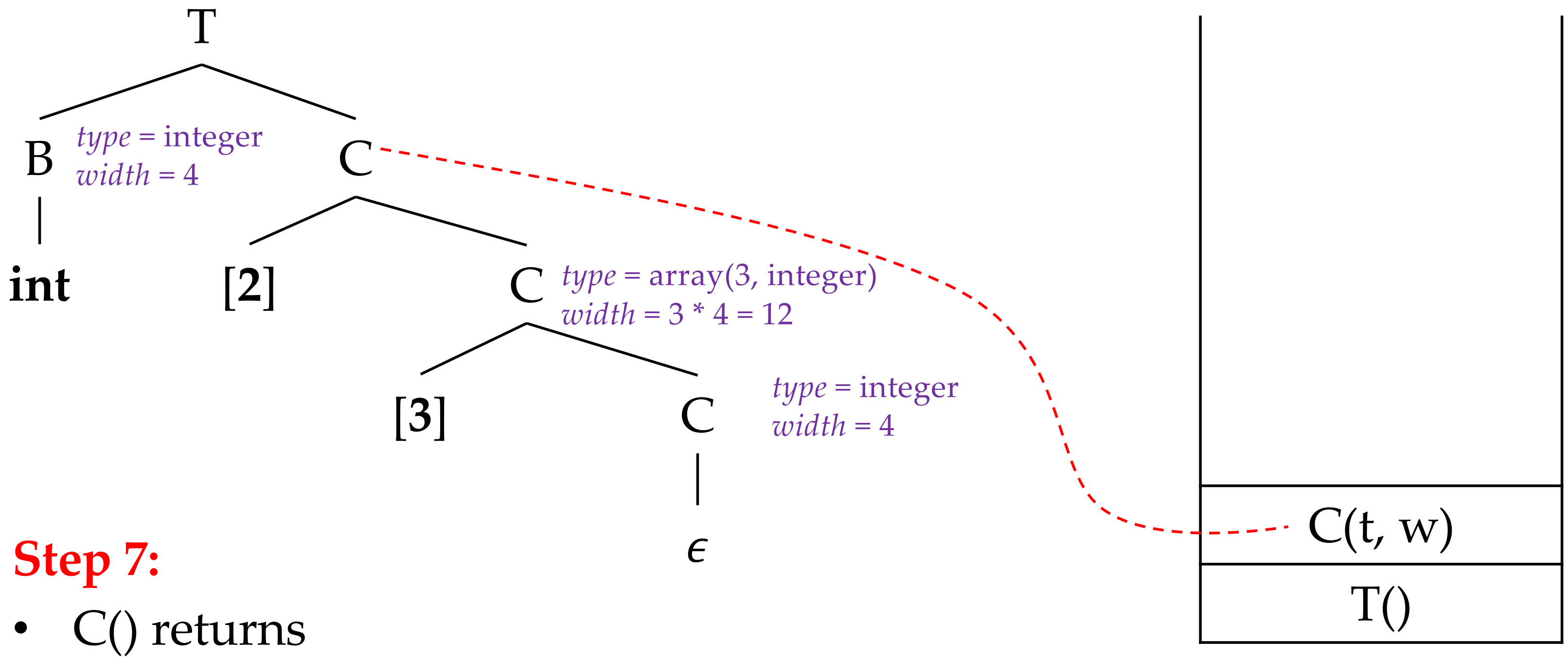
$$C \rightarrow \epsilon \qquad \boxed{\{ C.type = t;\ C.width = w; \}}$$

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**		$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow \epsilon$		$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow$ [**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 7:

- C() returns
- Execute semantic action

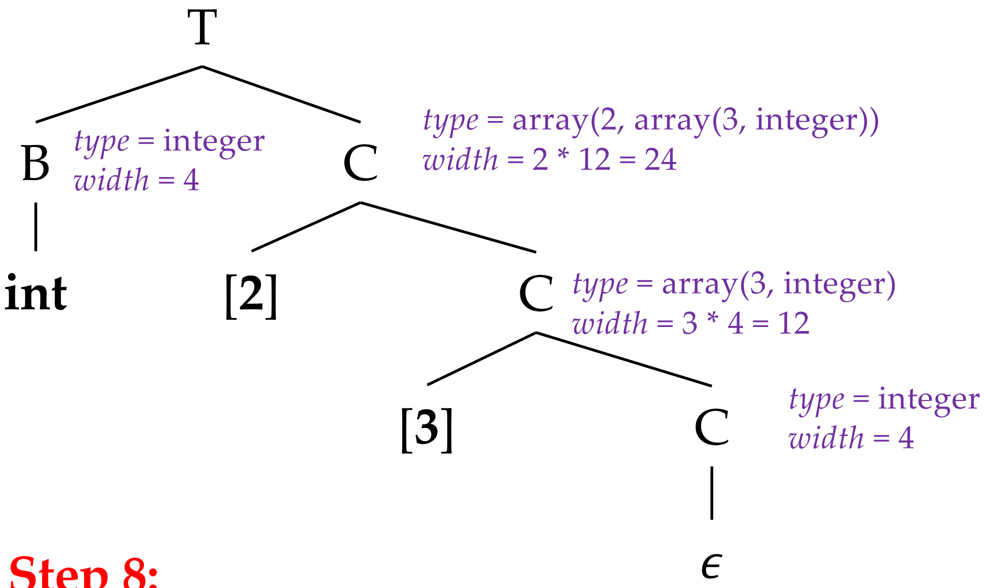
$C \rightarrow [\ \mathbf{num}\ ]\ C_1$ $\boxed{\{\ C.type{=}\ array(\mathbf{num}.value,\ C_1.type);\ C.width = \mathbf{num}.value \times C_1.width;\ \}}$

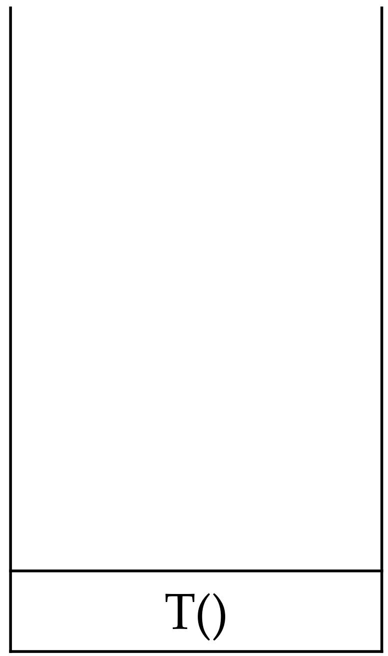
Translation Process Example

T	$\rightarrow$	B C	$\{\ t = B.type;\ w = B.width;\ \}$ $\{T.type{=}C.type;\ T.width{=}C.width;\}$
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 8:

- C() returns
- Execute semantic action

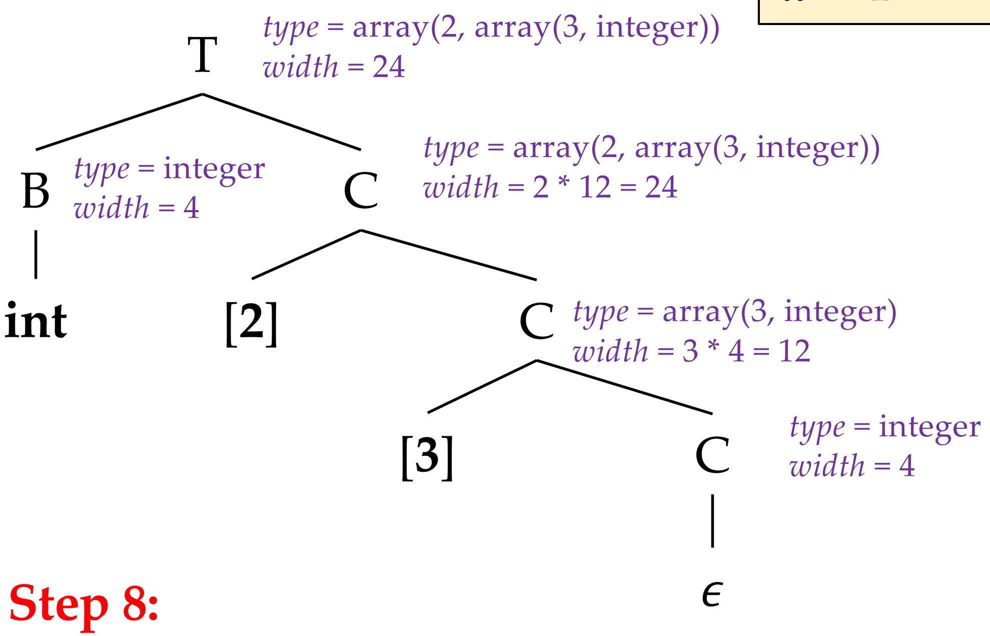
$$C \rightarrow [\ \mathbf{num}\ ]\ C_1 \quad \boxed{\{\ C.type{=}\ array(\mathbf{num}.value,\ C_1.type);\ C.width = \mathbf{num}.value \times C_1.width;\ \}}$$

Translation Process Example

T	$\rightarrow$	B 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	$\{ C.type{=}\ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

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

t = integer
w = 4



Call stack

Step 8:

- T() returns
- Execute semantic action

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

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

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

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

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
- Translation of Expressions
- Control Flow

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: $\mathrm{t}_1 = (\mathrm{float})\ \mathrm{i} \quad \mathrm{t}_2 = \mathrm{x\ fmul\ t}_1$

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

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

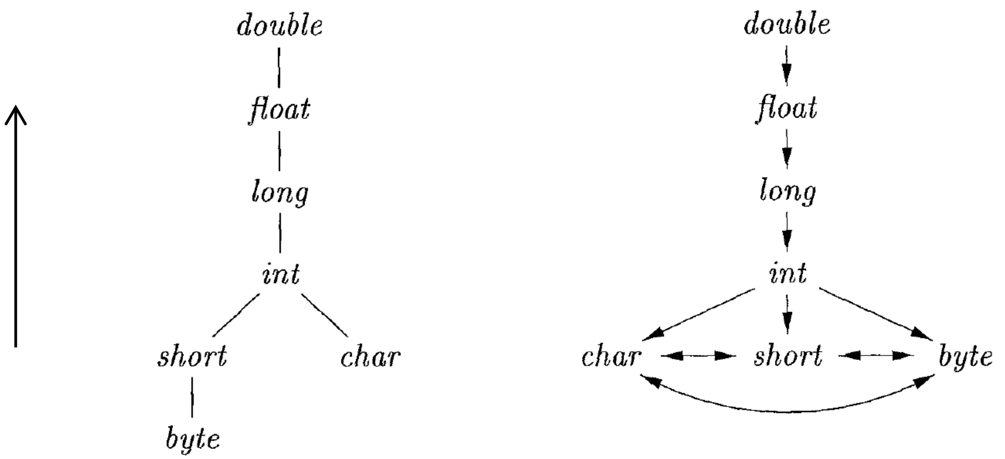
$\quad \mathbf{else\ if}(E_1.type = float\ \mathbf{and}\ E_2.type = integer)\ E.type = float;$

$\quad \ldots$

$\}$

Widening and Narrowing (1)

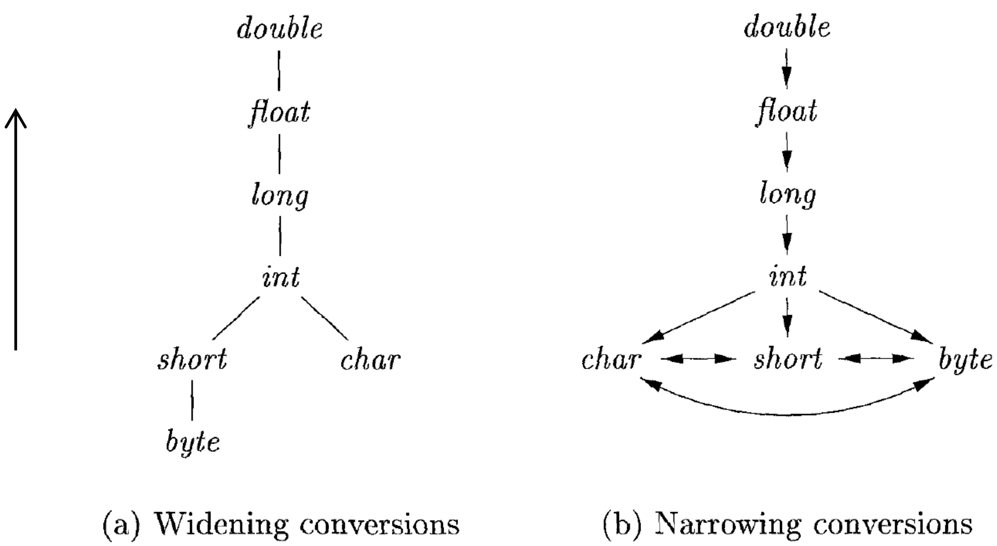
- Type conversion rules vary from language to language
- Java distinguishes between *widening* conversions (类型拓宽) and *narrowing* conversions (类型窄化)



(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*)



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

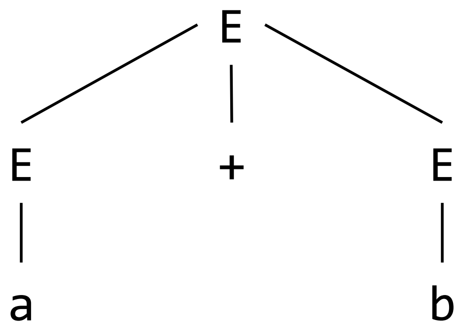
```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

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

Example

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

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



Generated code:

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

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

$E.type = max($`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