# Chapter 5: Intermediate-Code Generation

Yepang Liu

liuyp1@sustech.edu.cn

# Outline

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

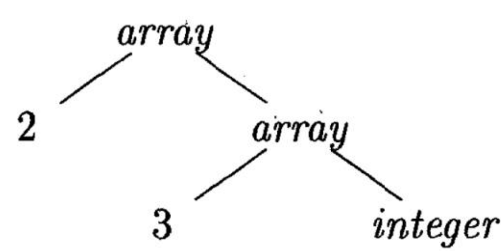
# Types and Type Checking

- *Data type* or simply *type* tells a compiler or interpreter how the programmers intend to use the data
- The usefulness of type information
  - Find faults in the source code
  - Determine the storage needed for a name at runtime
  - Calculate the address of an array element
  - Insert type conversions
  - Choose the right version of some arithmetic operator (e.g., `fadd`, `iadd`)
- *Type checking* (类型检查) uses logical rules to make sure that the types of the operands match the type expectation by an operator

# Type Expressions (类型表达式)

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

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



# The Definition of Type Expression

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

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

# Type Equivalence

- Type checking rules usually have the following form

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

Code under analysis:
```
a + b
```

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

# Name Equivalence (名等价)

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

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

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

```
Code under analysis:

Stack x, y;

Set r, s;

x = y;

r = s;

x = r;
```

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

# Structural Equivalence (结构等价)

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

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

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

```
Code under analysis:

Stack x, y;

Set r, s;

x = y;

r = s;

x = r;
```

# Declarations (变量声明)

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

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

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

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

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

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

# An SDT for Computing Types and Their Widths

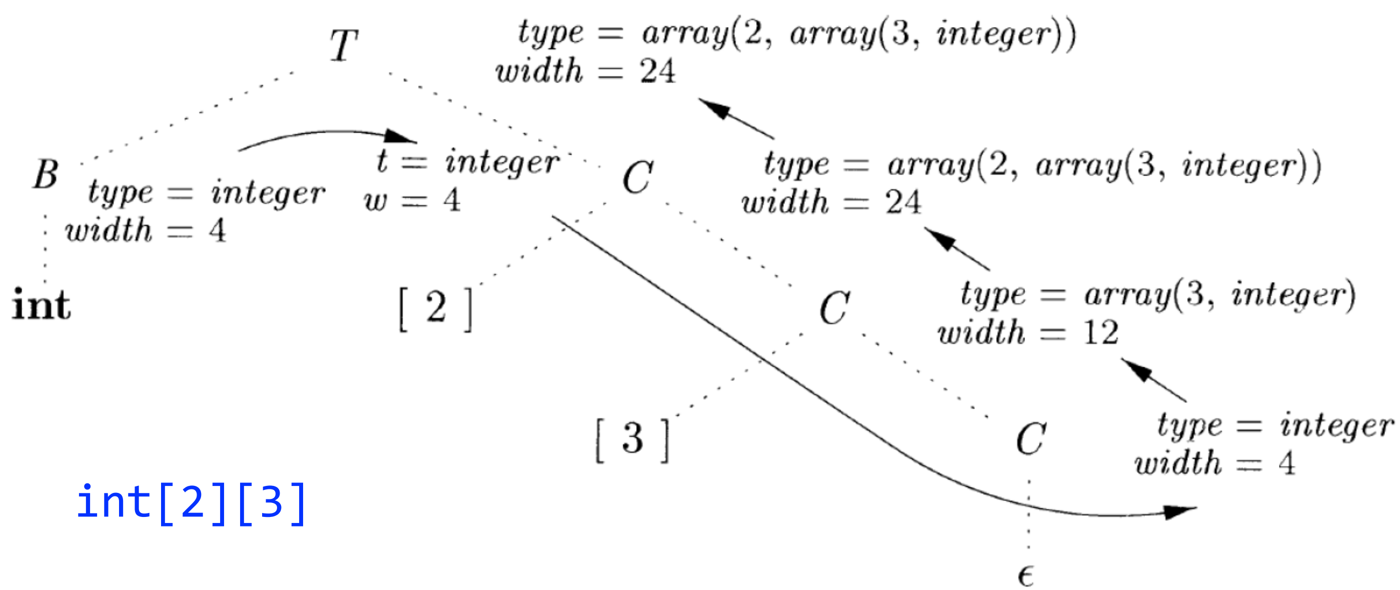
- Synthesized attributes: *type*, *width*
- Global variables $t$ and $w$ pass type and width information from a $B$ node in a parse tree to the node for the production $C \to \epsilon$
  - In an SDD, $t$ and $w$ would be $C$'s inherited attributes (the SDD is L-attributed)*

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

This SDT can be implemented during recursive-descent parsing

# Translation Process Example

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

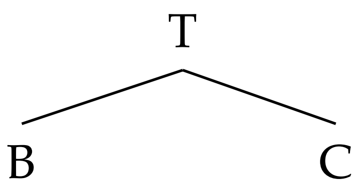



* In our example, we use global variables *t* and *w*

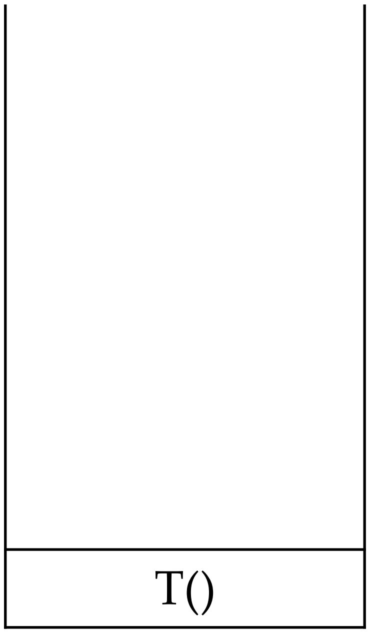
# Translation Process Example

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

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




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

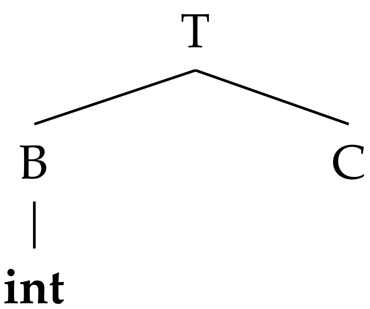



Call stack

# Translation Process Example

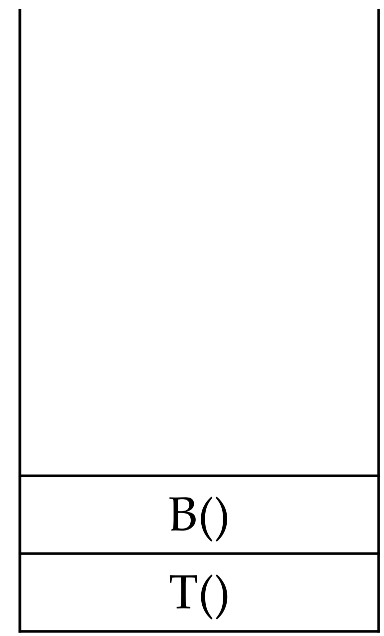
$$\begin{array}{lcl} T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\ B & \rightarrow & \textbf{int} \mid \textbf{float} \\ C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C \end{array}$$

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




**Step 2:**

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

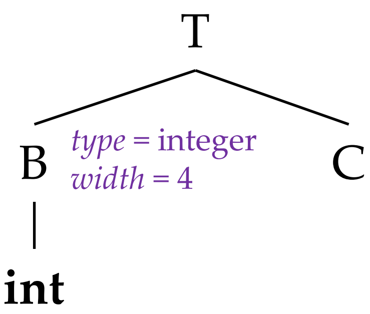



Call stack

# Translation Process Example

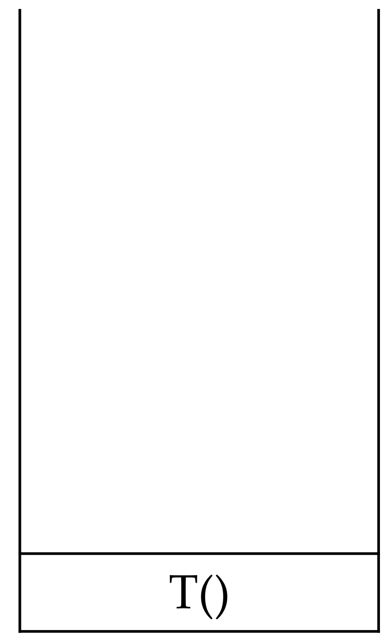
$$
\begin{array}{lcl}
T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int} \mid \textbf{float} \\
C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C
\end{array}
$$

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




**Step 3:**

- B() returns
- Execute semantic action




Call stack

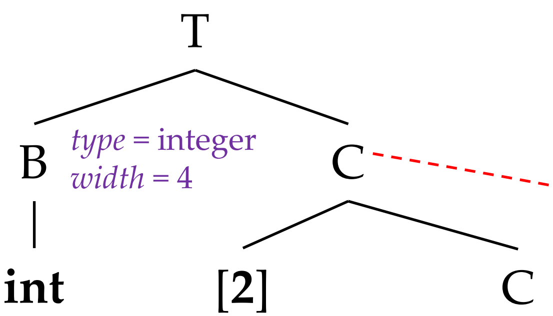
$$B \rightarrow \textbf{int} \qquad \{\ B.type = integer;\ B.width = 4;\ \}$$

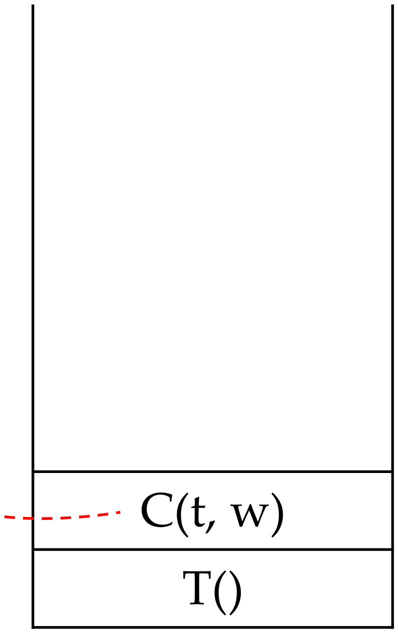
# Translation Process Example

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

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

$t$ = integer
$w$ = 4




Call stack

**Step 4:**

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

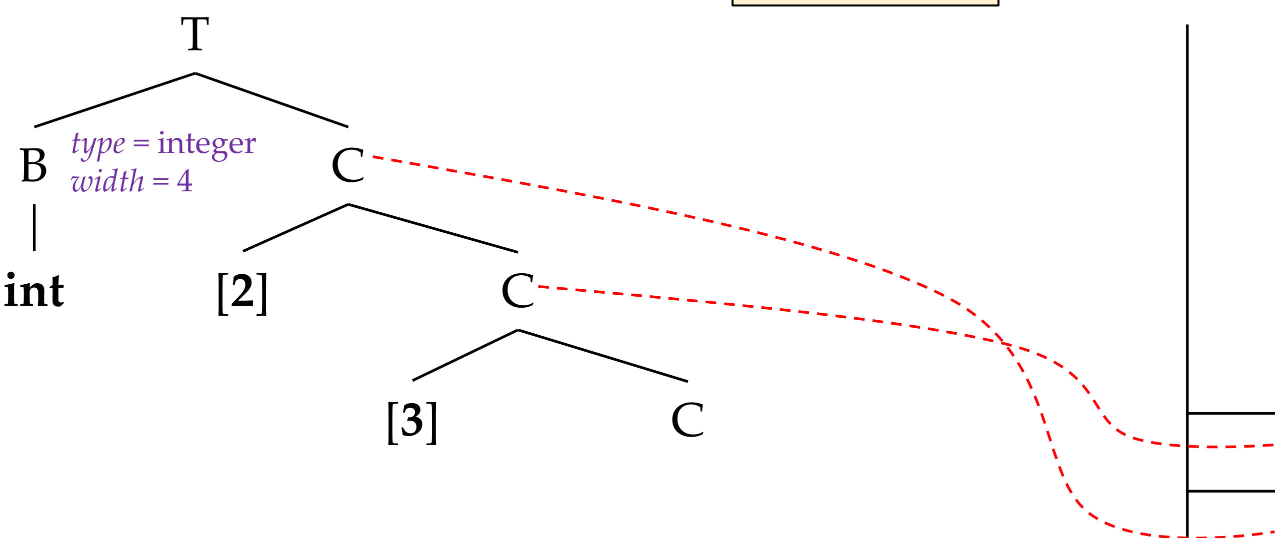
$$
\begin{array}{lcll}
T & \rightarrow & B & \boxed{\{\ t = B.type;\ w = B.width;\ \}} \\
 & & C & \{T.type{=}C.type;\ T.width{=}C.width;\}
\end{array}
$$

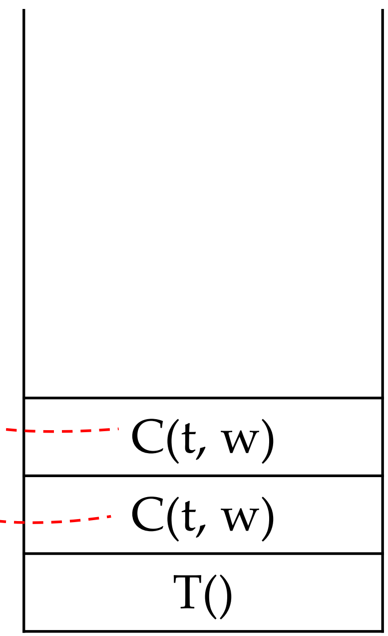
# Translation Process Example

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

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

$t$ = integer
$w$ = 4




Call stack

**Step 5:**

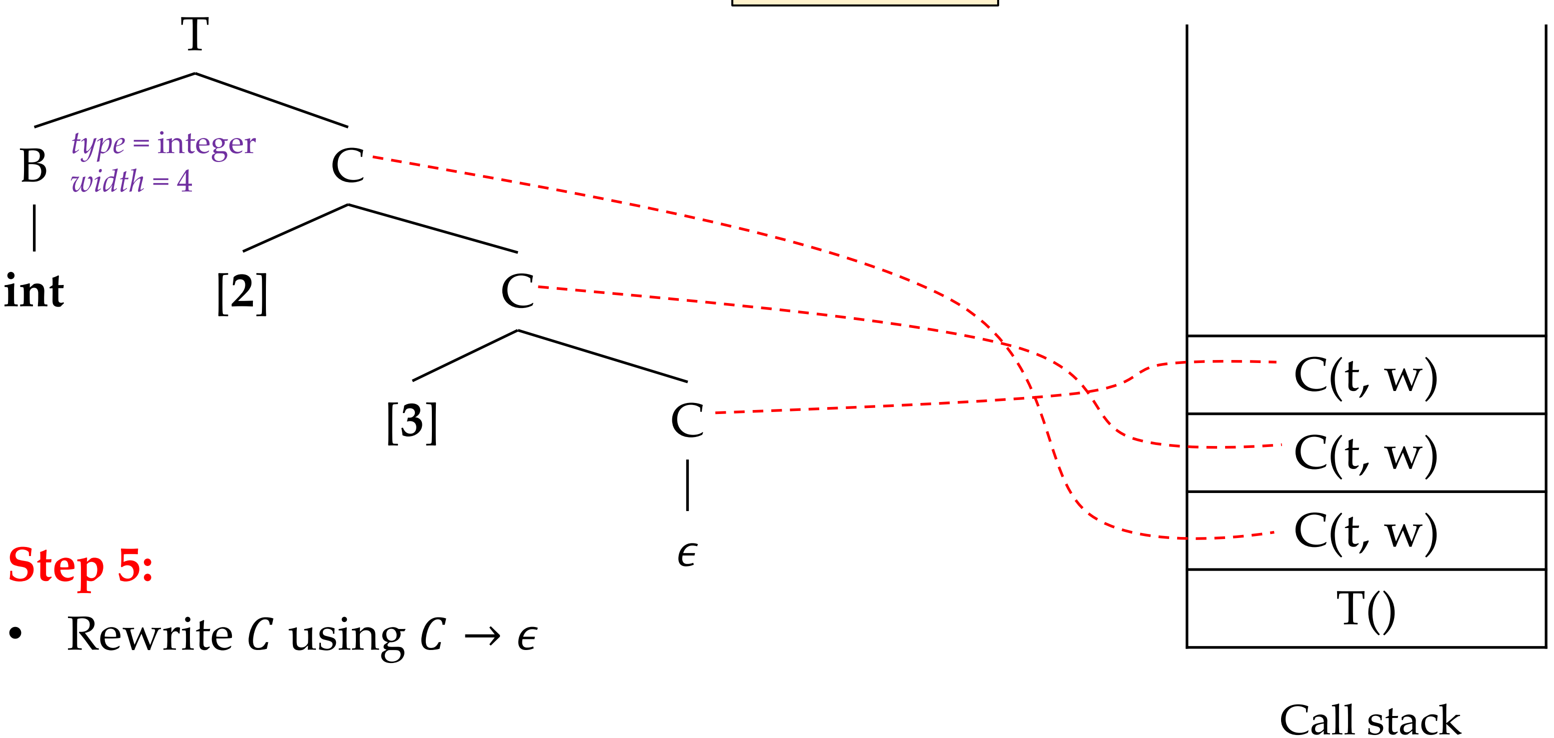
- Rewrite $C$ using $C \rightarrow [\textbf{num}]C$
- Match input

# Translation Process Example

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

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

$t$ = integer
$w = 4$




**Step 5:**

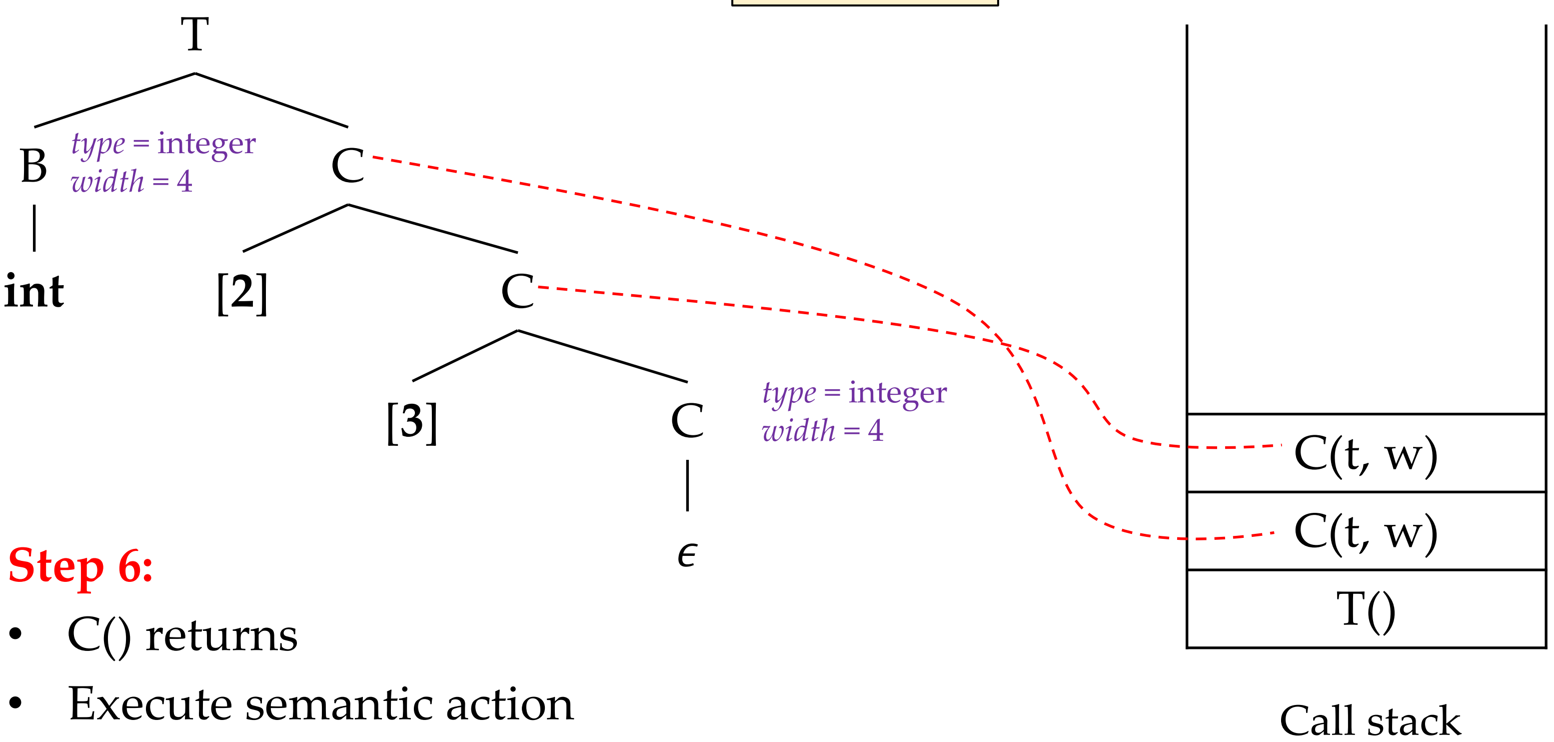
- Rewrite $C$ using $C \rightarrow \epsilon$

# Translation Process Example

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

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

$t$ = integer
$w$ = 4




**Step 6:**

- C() returns
- Execute semantic action

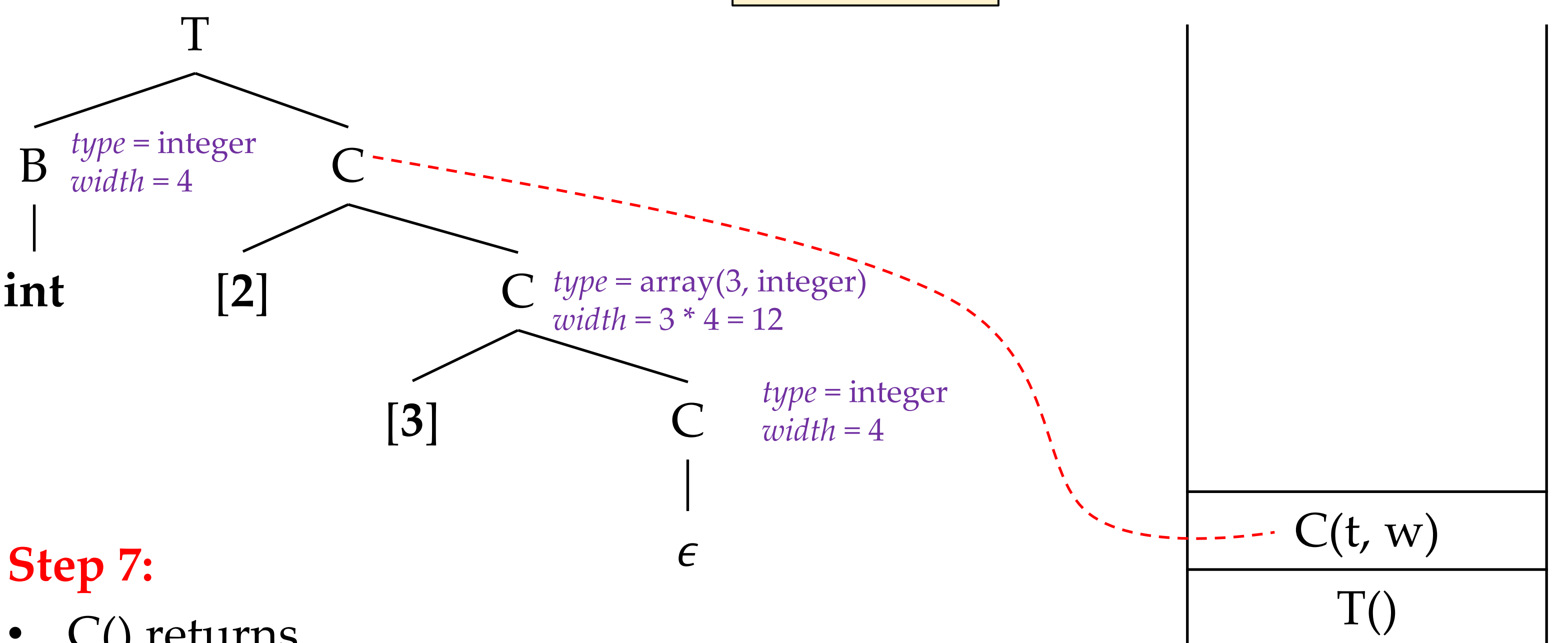
$C \rightarrow \epsilon \qquad \boxed{\{\ C.type = t;\ C.width = w;\ \}}$

# Translation Process Example

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

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

$t$ = integer
$w$ = 4




Call stack

**Step 7:**

- C() returns
- Execute semantic action

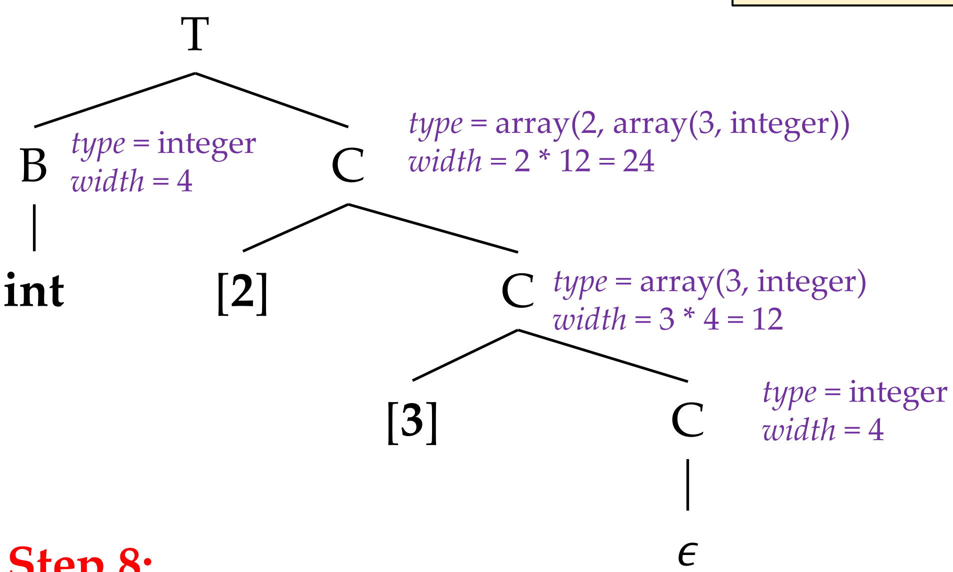
$$C \rightarrow [\ \textbf{num}\ ]\ C_1 \quad \boxed{\{\ C.type = array(\textbf{num}.value,\ C_1.type); \quad C.width = \textbf{num}.value \times C_1.width;\ \}}$$

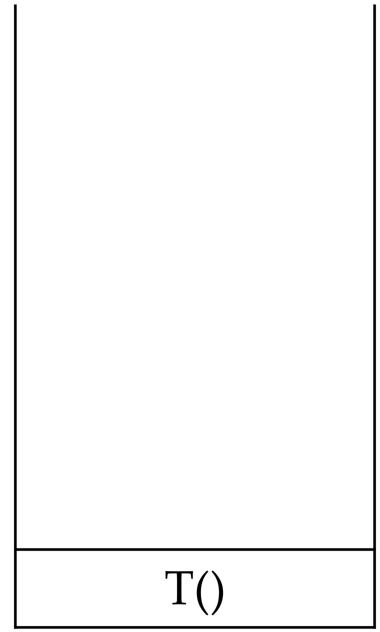
# Translation Process Example

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

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

$t$ = integer
$w$ = 4




Call stack

**Step 8:**

- C() returns
- Execute semantic action

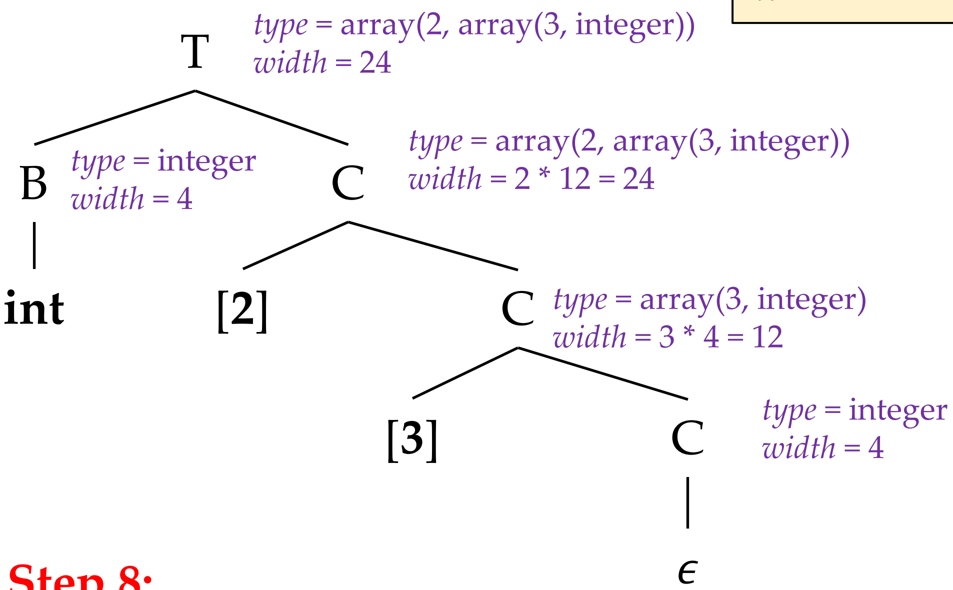
$C \rightarrow [\ \textbf{num}\ ]\ C_1$ $\{\ C.type = array(\textbf{num}.value,\ C_1.type);\ C.width = \textbf{num}.value \times C_1.width;\ \}$

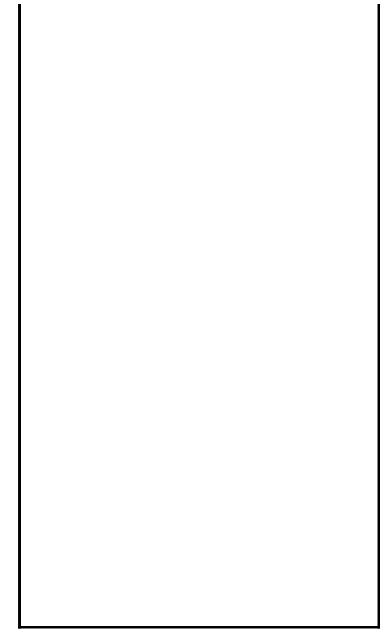
# Translation Process Example

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

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

$t$ = integer
$w$ = 4



Call stack

**Step 8:**

- T() returns
- Execute semantic action

$$\begin{array}{lcll} T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\ & & C & \boxed{\{T.type{=}C.type;\ T.width{=}C.width;\}} \end{array}$$

# 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

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

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

* Self-study materials

# Fields in Records and Classes

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

- 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