



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 10

Yepang Liu

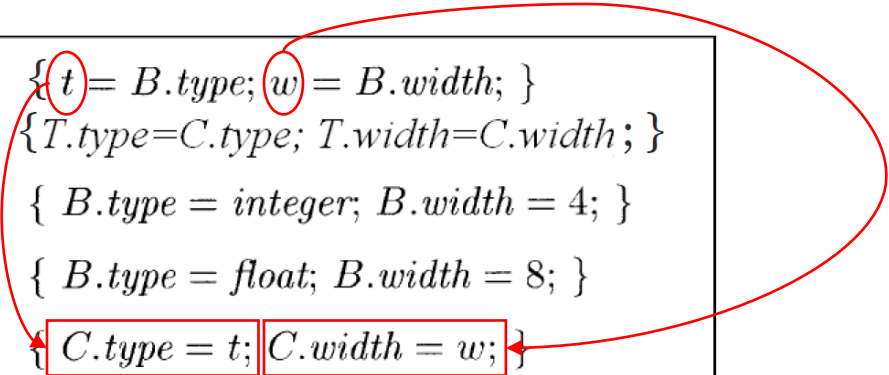
liuyp1@sustech.edu.cn

Outline

- Computing Type Information
- Type Checking
- Scope Checking

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)*

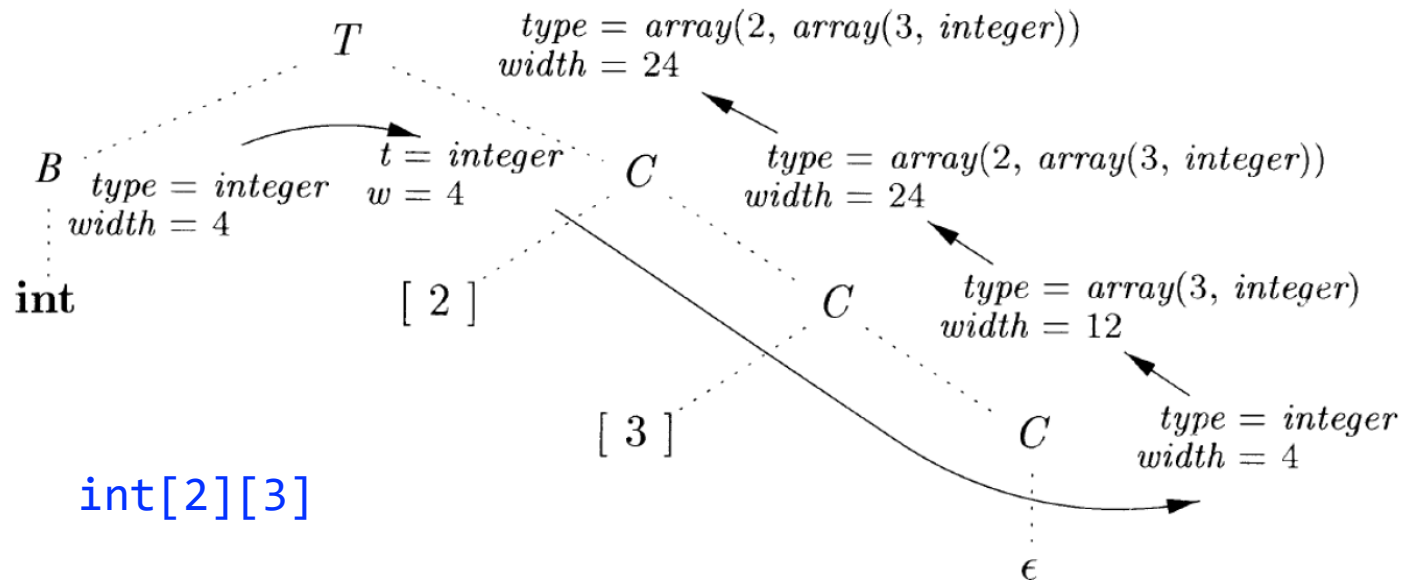


$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
C	$\{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$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

This SDT can be implemented during recursive-descent parsing

Translation Process Example

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

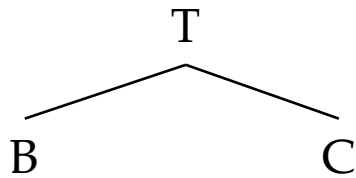


* In our example, we use global variables t and w

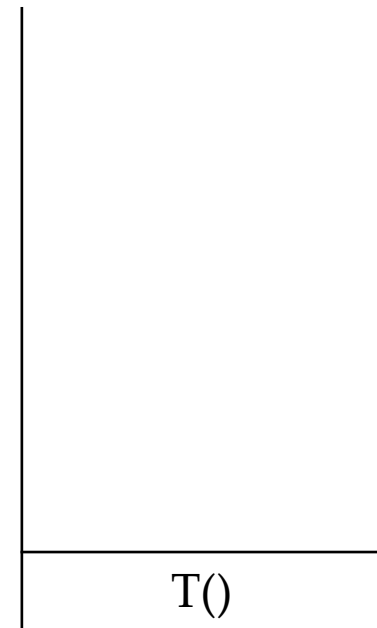
Translation Process Example

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

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



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

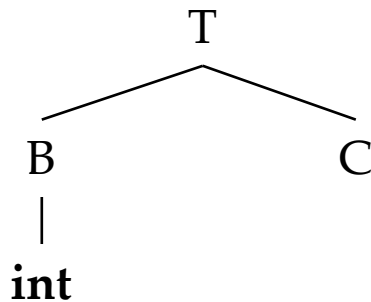


Call stack

Translation Process Example

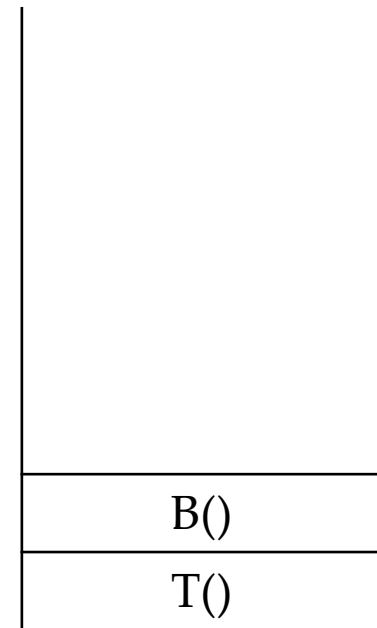
$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
C	$\{ T.type = C.type; T.width = C.width; \}$
$B \rightarrow \text{int}$	$\{ B.type = \text{integer}; B.width = 4; \}$
$B \rightarrow \text{float}$	$\{ B.type = \text{float}; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$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

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



Step 2:

- Rewrite B using $B \rightarrow \text{int}$
- Match input

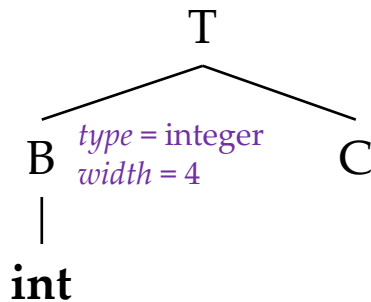


Call stack

Translation Process Example

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

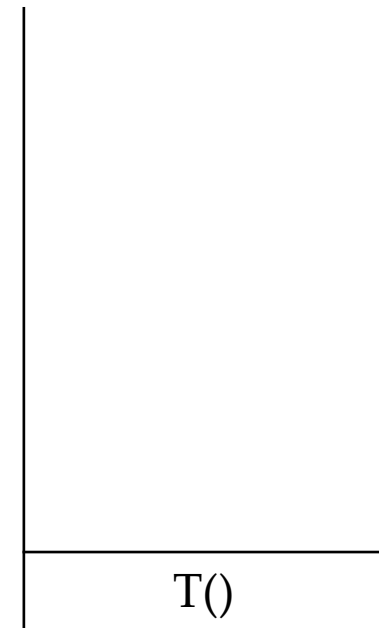
Input string: `int[2][3]`



Step 3:

- B() returns
- Execute semantic action

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



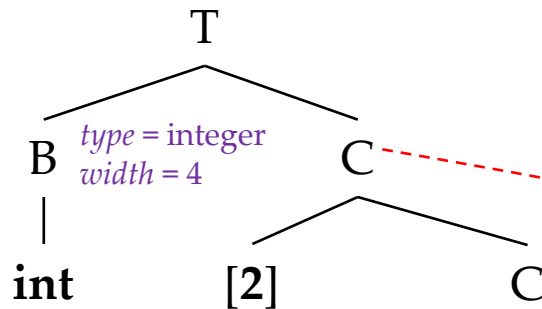
Call stack

Translation Process Example

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

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

$t = \text{integer}$
 $w = 4$

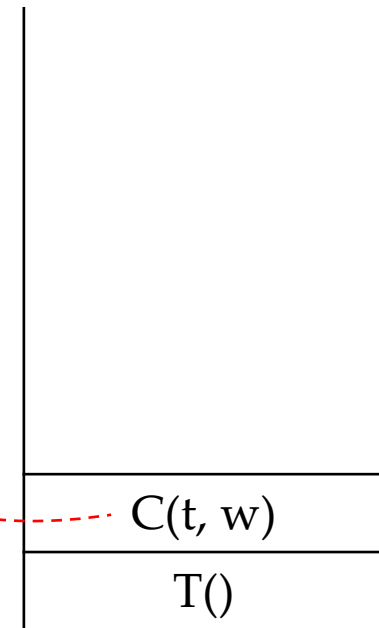


Step 4:

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

$T \rightarrow B$
 C

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



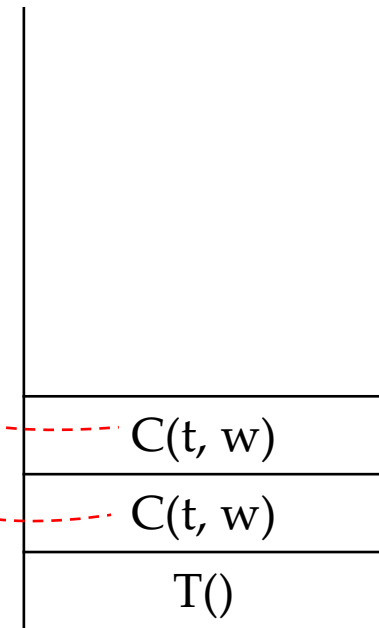
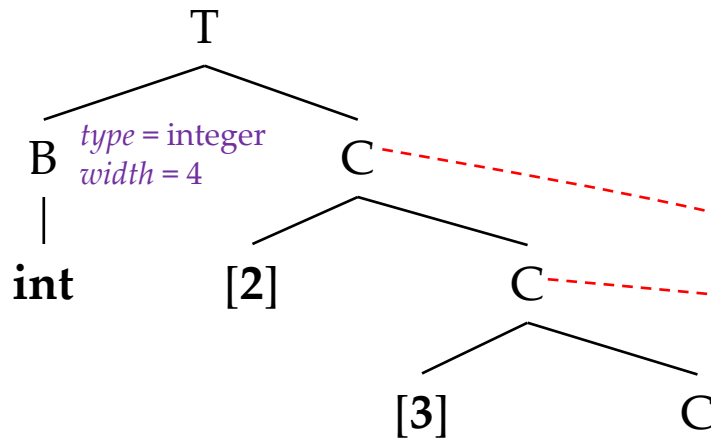
Call stack

Translation Process Example

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

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

$t = \text{integer}$
 $w = 4$



Step 5:

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

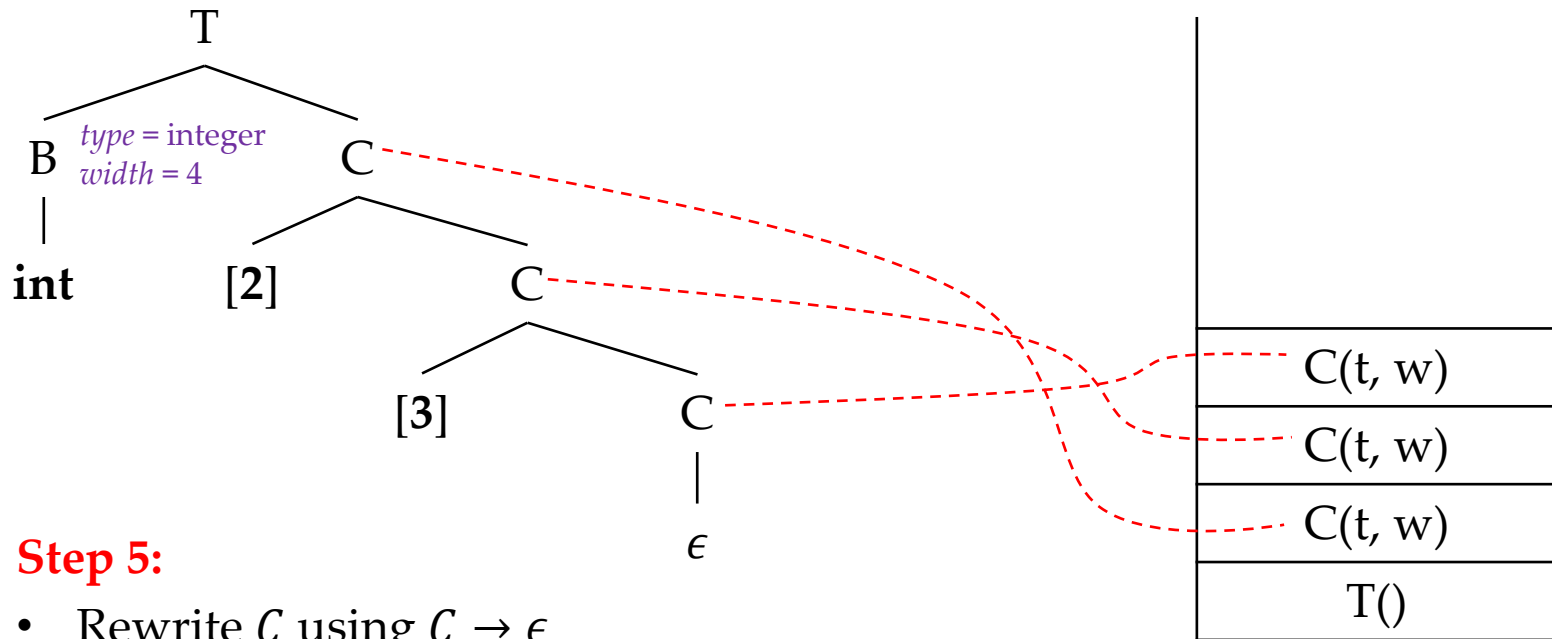
Call stack

Translation Process Example

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

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

$t = \text{integer}$
 $w = 4$



Step 5:

- Rewrite C using $C \rightarrow \epsilon$

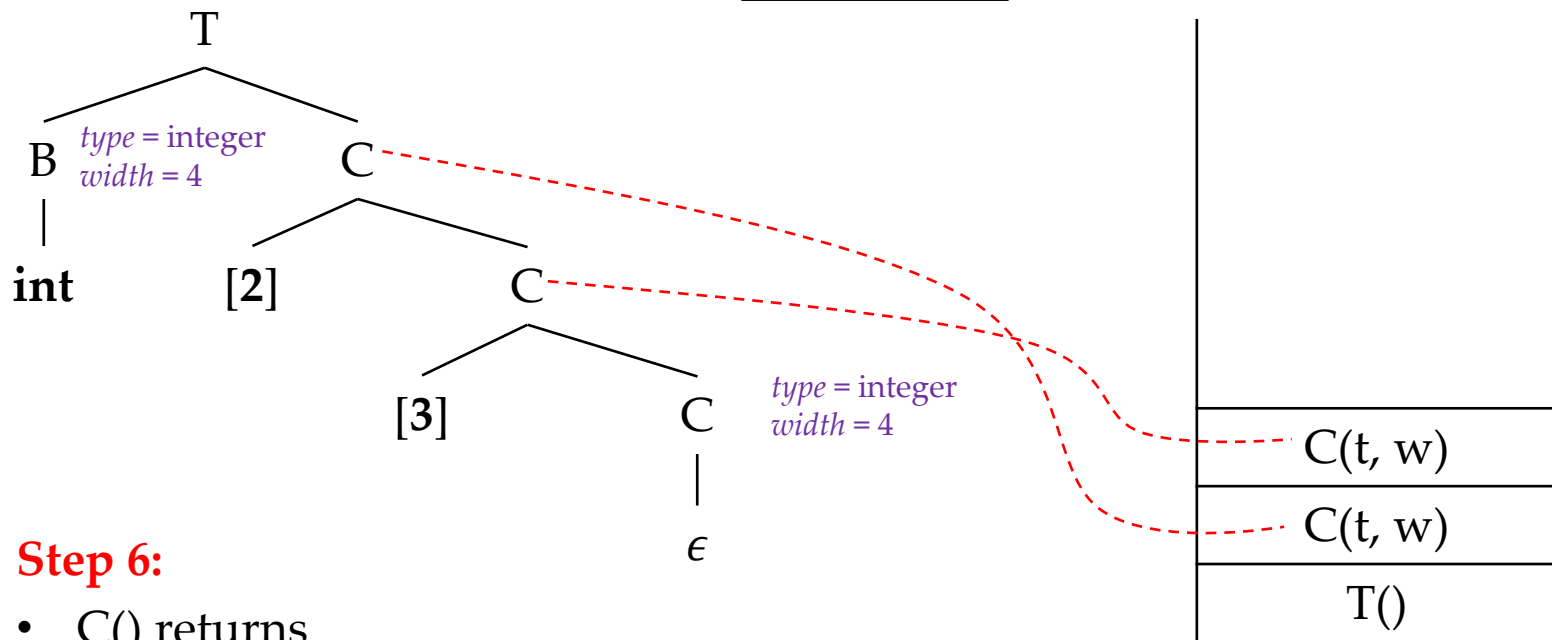
Call stack

Translation Process Example

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

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

$t = \text{integer}$
 $w = 4$



Step 6:

- $C()$ returns
- Execute semantic action

$C \rightarrow \epsilon$

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

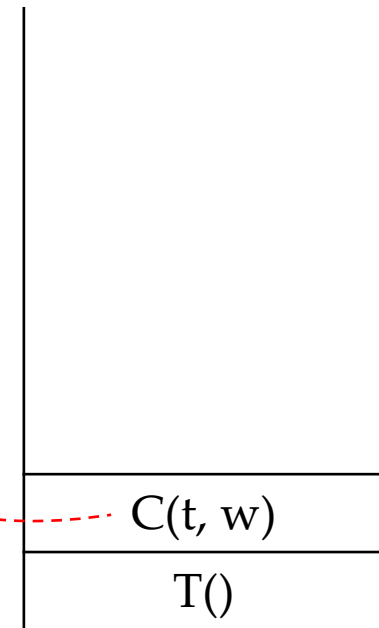
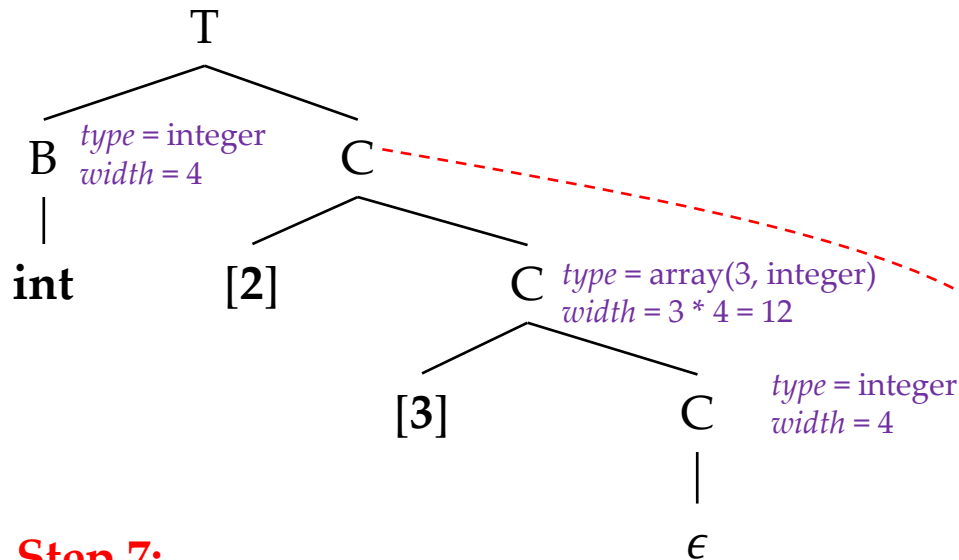
Call stack

Translation Process Example

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

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

$t = \text{integer}$
 $w = 4$



Call stack

Step 7:

- $C()$ returns
- Execute semantic action

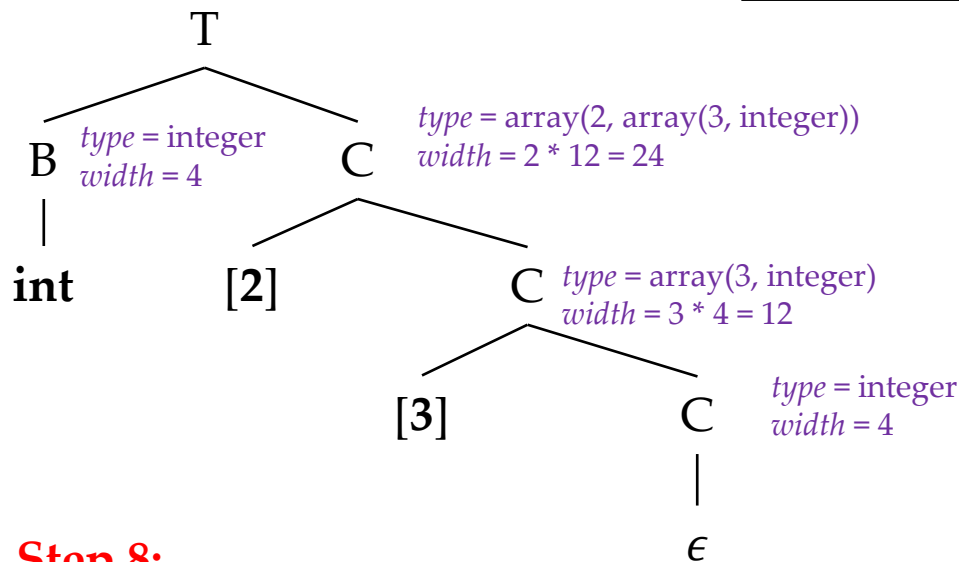
$C \rightarrow [\text{num}] C_1$
 $\{ C.type = \text{array}(\text{num.value}, C_1.type);$
 $C.width = \text{num.value} \times C_1.width; \}$

Translation Process Example

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

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

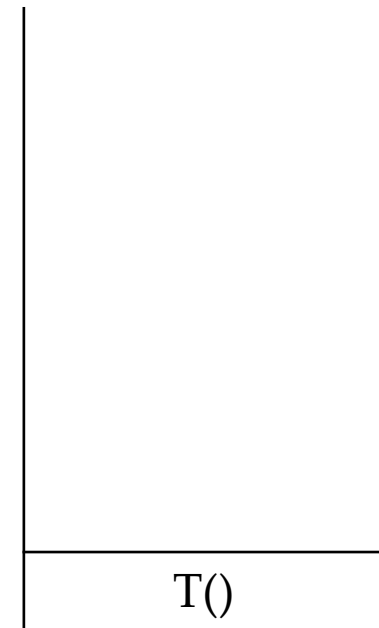
$t = \text{integer}$
 $w = 4$



Step 8:

- $C()$ returns
- Execute semantic action

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



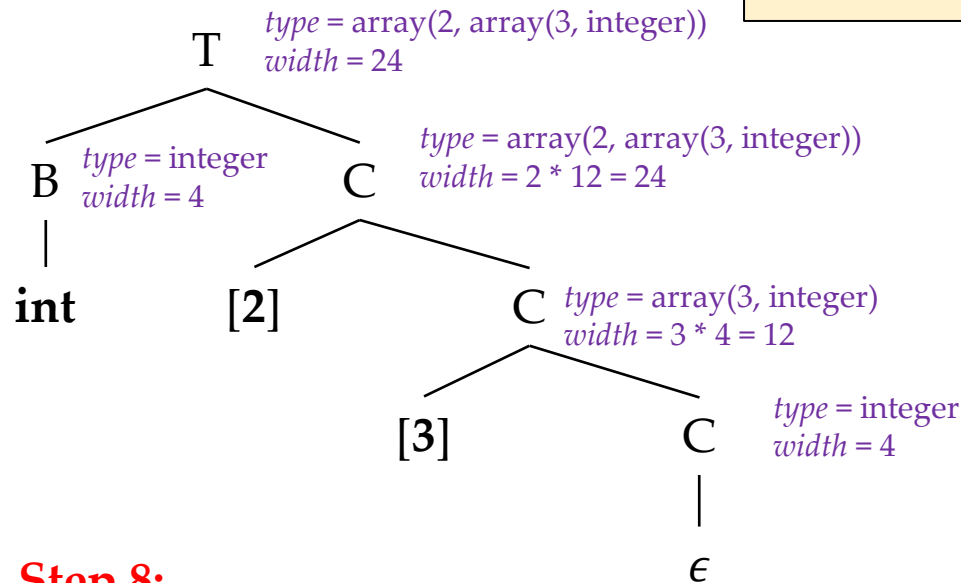
Call stack

Translation Process Example

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

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

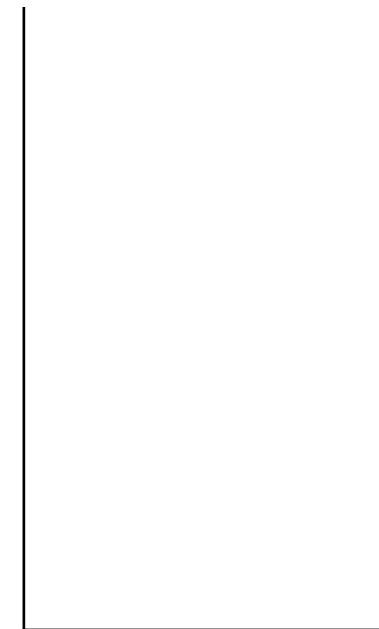
$t = \text{integer}$
 $w = 4$



Step 8:

- $T()$ returns
- Execute semantic action

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



Call stack

Sequences of Declarations

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

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

Computing relative addresses of declared names

Outline

- Computing Type Information
- Type Checking
- Scope Checking

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

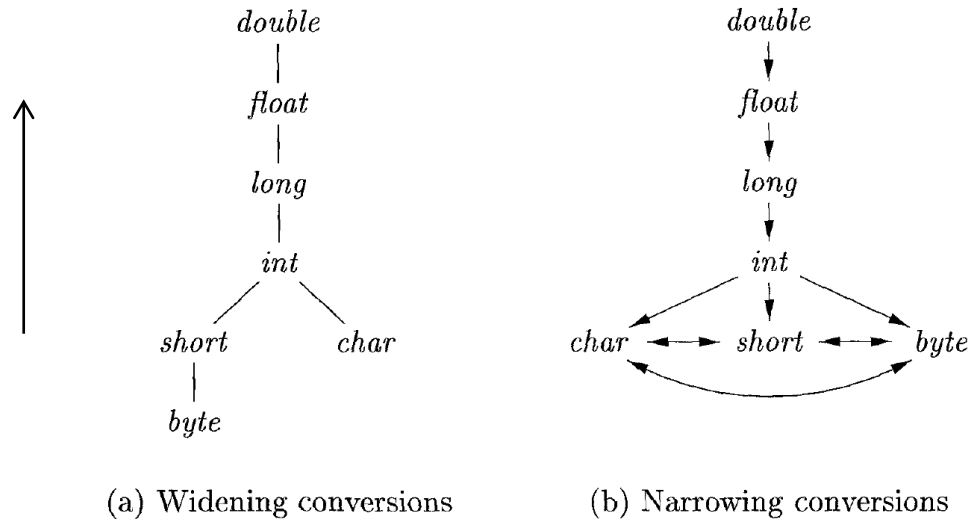
- Type synthesis (类型合成)
 - Build up the type of an expression from the types of subexpressions
 - **Typical form:** if f has type $s \rightarrow t$ and x has type s , then expression $f(x)$ has type t
 - **Example:** If x is of integer type, the function f has type $integer \rightarrow integer$, then 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 and floats
 - Convert integers to floats: $t_1 = (\text{float}) i$ $t_2 = x \text{ fmul } t_1$
- **Type conversion SDT** for a simple case (using type synthesis)
 - $E \rightarrow E_1 + E_2$
 - $\{$ **if**($E_1.\text{type} = \text{integer}$ **and** $E_2.\text{type} = \text{integer}$) $E.\text{type} = \text{integer};$
 else if($E_1.\text{type} = \text{float}$ **and** $E_2.\text{type} = \text{integer}$) $E.\text{type} = \text{float};$
 ...
 $\}$

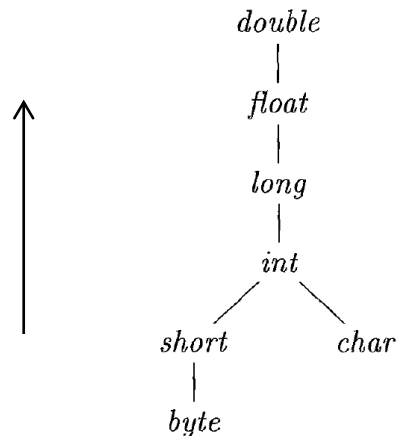
Widening and Narrowing (1)

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

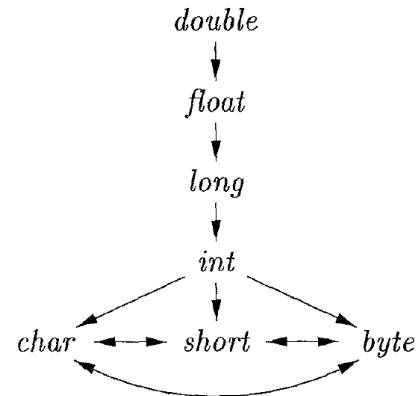


Widening and Narrowing (2)

- *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

- $\text{max}(t_1, t_2)$ takes two types t_1 and t_2 and returns the **maximum** (or **least upper bound**) of the two types in the widening hierarchy
- $\text{widen}(a, t, w)$ 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 → E1 + E2 { E.type = max(E1.type, E2.type);
                  a1 = widen(E1.addr, E1.type, E.type);
                  a2 = widen(E2.addr, E2.type, E.type);
                  E.addr = new Temp();
                  gen(E.addr '=' a1 '+' a2); }
```

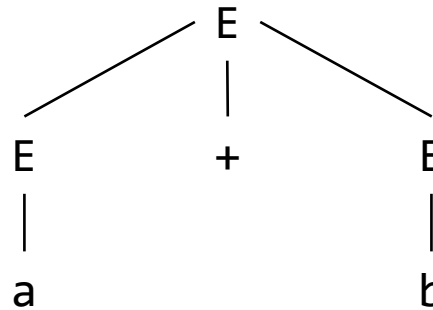
Example

- $a + b$ (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

```

$E \rightarrow E_1 + E_2$	$\{ E.type = \max(E_1.type, E_2.type);$	$E.type = \max(int, float) = float$	1
	$a_1 = widen(E_1.addr, E_1.type, E.type);$	$a_1 = widen(a, int, float) = temp$	2
	$a_2 = widen(E_2.addr, E_2.type, E.type);$	$a_2 = widen(b, float, float) = b$	3
	$E.addr = new Temp();$	$E.addr = new Temp() = temp2$	4
	$gen(E.addr '=' a_1 '+' a_2); \}$		5

Outline

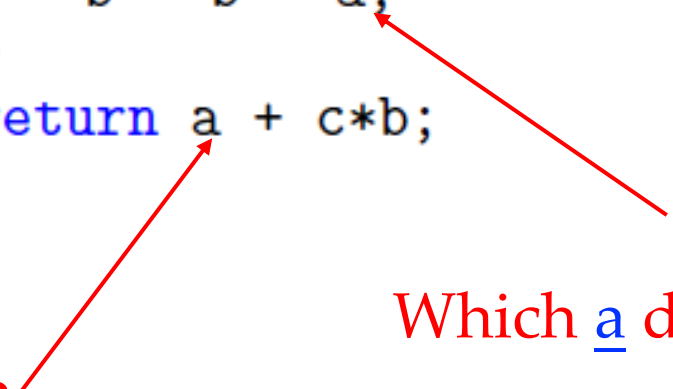
- Computing Type Information
- Type Checking
- Scope Checking

Scope Checking

- Variables in a program are only visible within certain sections, called *scope*
- For program without scopes, we say there is only global scope (the assumption of our SPL)
- *Scope checking* refers to the process of determining if an identifier (symbol) is accessible at a program location

Scope Example

```
int test_2_o01(){  
    int a, b, c;  
    a = a + b;  
    if(b > 0){  
        int a = c * 7;  
        b = b - a;  
    }  
    return a + c*b;  
}
```



Which a does it refer to?

Which a does it refer to?

Scope Checking


- What if there is only one symbol table?

```
→ int test_2_o01(){  
    int a=0, b=1, c=2;  
    a = a + b;  
    if(b > 0){  
        int a = c * 7;  
        b = b - a;  
    }  
    return a + c*b;  
}
```

id	type	Declaration location
a	int	Line 2
b	int	Line 2
c	int	Line 2

Scope Checking


- What if there is only one symbol table?



```
int test_2_o01(){  
    int a=0, b=1, c=2;  
    a = a + b;  
    if(b > 0){  
        int a = c * 7;  
        b = b - a;  
    }  
    return a + c*b;  
}
```

Shall we update it to line 5?


id	type	Declaration location
a	int	Line 2
b	int	Line 2
c	int	Line 2



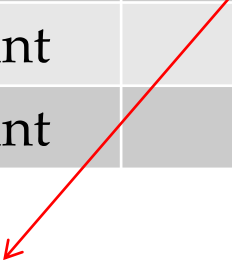
Scope Checking

- What if there is only one symbol table?

```
int test_2_o01(){  
    int a=0, b=1, c=2;  
    a = a + b;  
    if(b > 0){  
        int a = c * 7;  
        b = b - a;  
    }  
    return a + c*b;  
}
```



id	type	Declaration location
a	int	Line 5
b	int	Line 2
c	int	Line 2



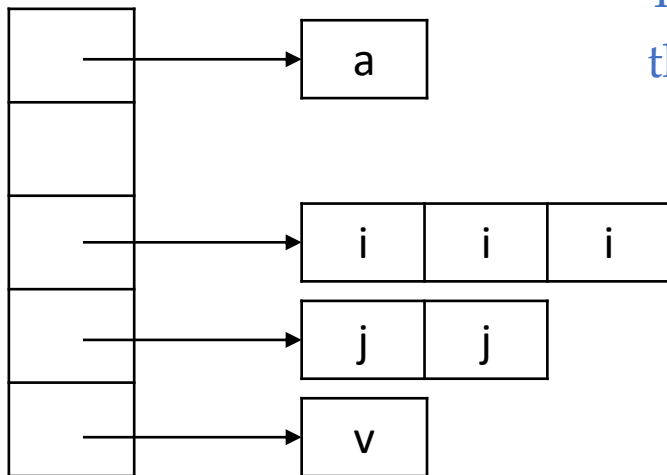
If we update to line 5 earlier, then at line 7, the compiler would consider *a* to be defined at line 5, which is not correct...

Implementing Scope Checking

- Two common strategies
 - Single table (also known as “imperative style”)
 - Multiple tables (also known as “functional style”)
- Both need to delete symbols when leaving a scope

Single Table Strategy

- The naïve implementation:
 - Use a hash table to implement the symbol table
 - Use separated chaining to address conflicts
 - Insert duplicate keys at the head of the corresponding list



The innermost definition always appears at the head of list (easy to find 😊)

Disadvantage:

When the current scope is closed, we need to remove symbols, which is not easy:

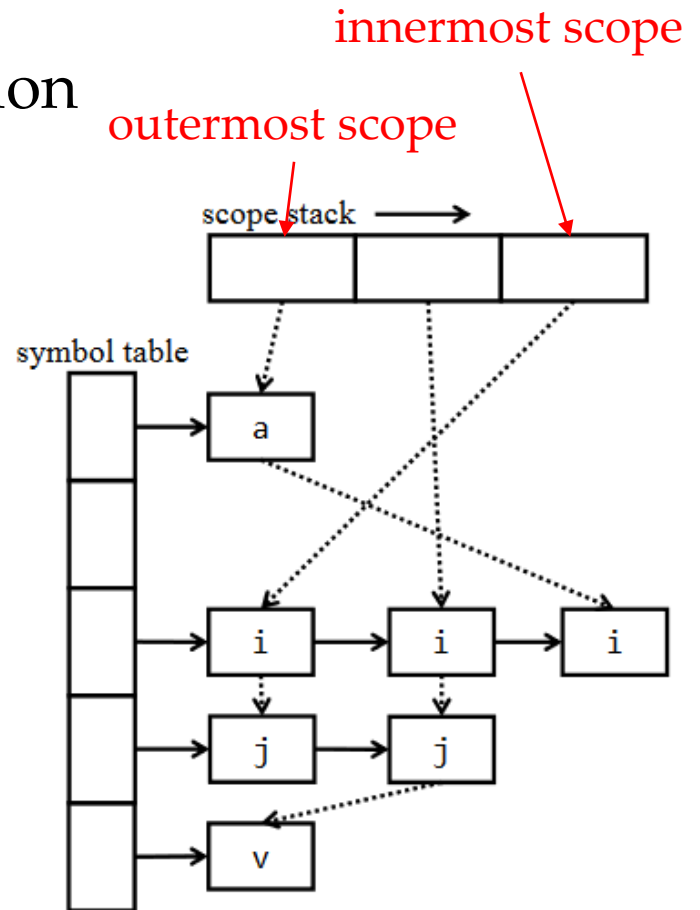
- Need to traverse all linked lists to check if the symbol at the head is available only for the current scope (i.e., defined there)

Single Table Strategy

- The orthogonal list implementation

Advantages:

- Still easy to find the innermost definition (at the head of each list)
- When closing the current scope, removing symbols is easy:
 - Tracing through the list corresponding to the stack top can efficiently locate the “to be removed” symbols



Multiple Tables Strategy

- Maintaining scope stack, each element is a symbol table
- Push new table when entering a new scope
- Pop the topmost table when leaving a scope

Disadvantage: When analyzing the scope for a variable, one may need to search all the way down the stack (from the symbol table at the top of the stack to the symbol table at the bottom of the stack)