COMP321 Compiler
Spring 2023

# Semantic Analysis

Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

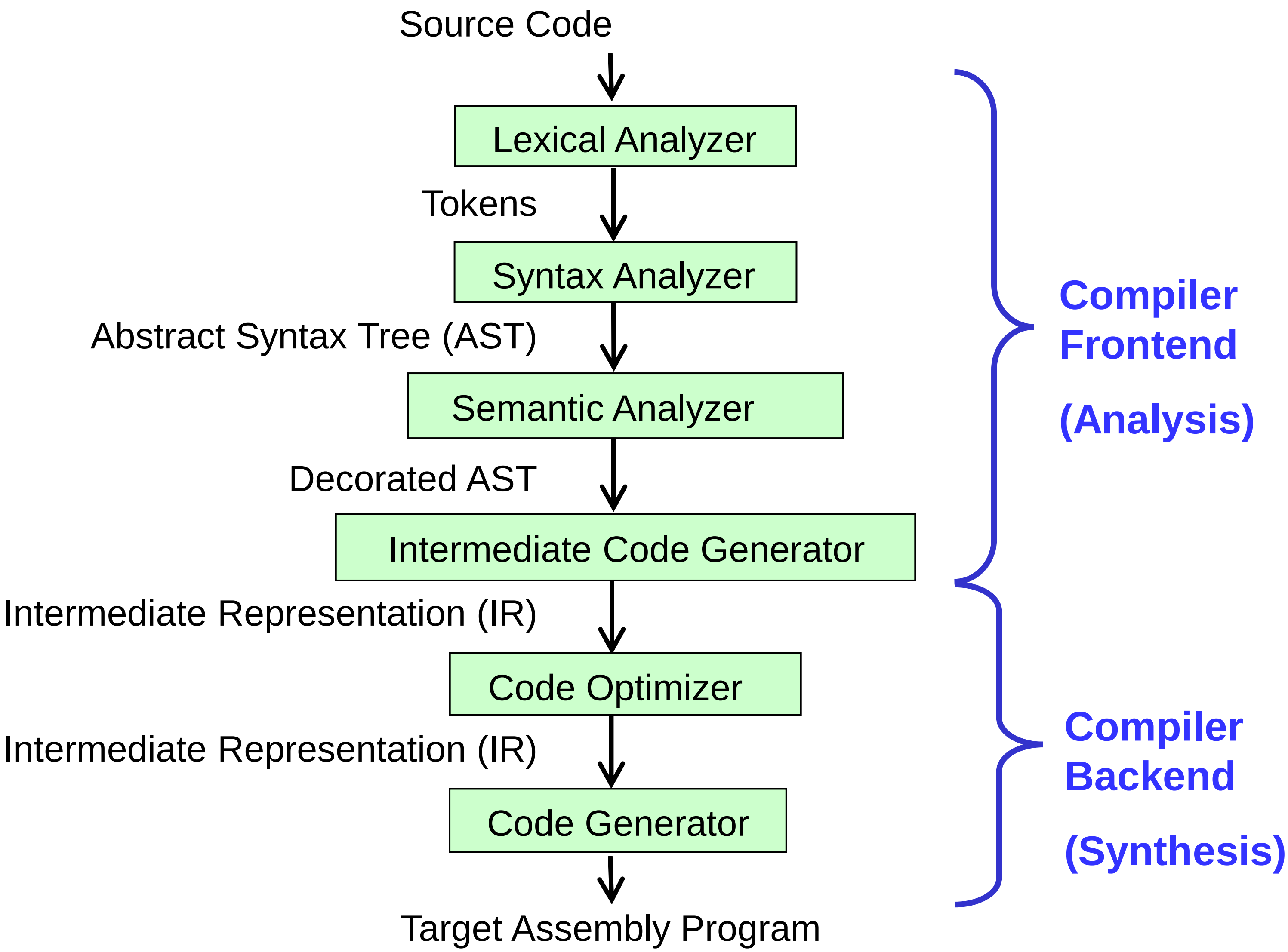
# Lecture 3: Semantic Analysis

1. Overview & Purpose
   a) Static semantics
   b) Dynamic semantics

2. Static semantics
   a) Two types of semantic constraints:
      - Scope rules
      - Type rules
   b) Two subphases in static semantic analysis
      - Identification (to enforce scope rules)
      - Type checking (to enforce type rules)
   c) Standard environments
   d) Assignment 2-(2)

3. Attribute Grammars

# Recapitulate: the structure of a compiler...



Source Code

Lexical Analyzer

Tokens

Syntax Analyzer

Abstract Syntax Tree (AST)

Semantic Analyzer

Decorated AST

Intermediate Code Generator

Intermediate Representation (IR)

Code Optimizer

Intermediate Representation (IR)

Code Generator

Target Assembly Program

**Compiler Frontend**

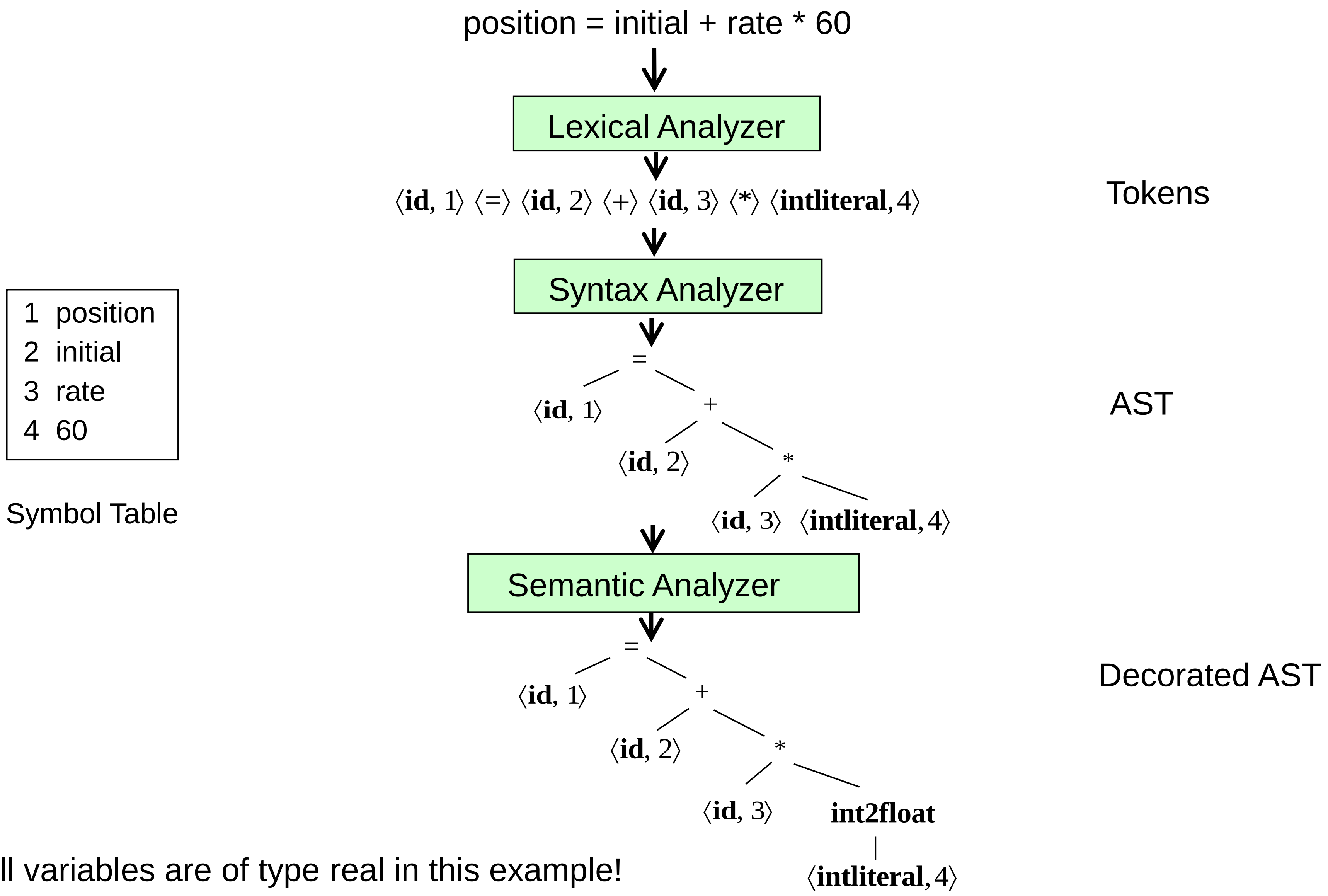
**(Analysis)**

**Compiler Backend**

**(Synthesis)**

…each phase transforms the program from one representation to the next…

# Recapitulate: the structure of a compiler...



position = initial + rate * 60

Lexical Analyzer

⟨**id**, 1⟩ ⟨=⟩ ⟨**id**, 2⟩ ⟨+⟩ ⟨**id**, 3⟩ ⟨*⟩ ⟨**intliteral**, 4⟩

Tokens

Syntax Analyzer

=

⟨**id**, 1⟩

+

⟨**id**, 2⟩

*

⟨**id**, 3⟩ ⟨**intliteral**, 4⟩

AST

| | |
|---|---|
| 1 | position |
| 2 | initial |
| 3 | rate |
| 4 | 60 |

Symbol Table

Semantic Analyzer

=

⟨**id**, 1⟩

+

⟨**id**, 2⟩

*

⟨**id**, 3⟩ **int2float**

⟨**intliteral**, 4⟩

Decorated AST

Note: all variables are of type real in this example!

# Syntax versus Semantics

- Syntax determines the structure or form of a valid program.

```
expr ::=  ID  |  INT  |  "-" expr  |  "(" expr ")"  |  expr op expr
op   ::=  "+"  |  "-"  |  "*"  |  "/"
```

- 101 + 1 is a syntactically correct sentential form.
- But what is its meaning?
  - In binary:  101 + 1 = 110
  - In decimal: 101 + 1 = 102
  - As strings: "101" + "1" = "1011"
- What is the meaning of the + operator?
  - Addition?
  - String concatenation?
  - ?
- Semantics of a programming language determines the meaning of sentences.

# Semantic Analysis

The compilation process is driven by the **syntactic structure** of the program as discovered by the parser.

Semantic routines:

- Interpret the meaning of the program based on its syntactic structure.
- Two purposes:
  - finish analysis by deriving **context-sensitive** information
  - begin synthesis by generating IR or target code.
- Semantic analysis is associated with the individual productions of a context-free grammar or subtrees of a syntax tree.
  - Called **syntax-directed translation**.

# Context-Sensitive Information

- Is `x` a variable, method, array, class or package?
  Example: `a(17)` in Ada
- Is `x` declared before it is used?
- To which declaration of `x` does a reference refer to (identification)?
  Example:

  ```
  int a;
   {
      int a;
      a = 5;}
  ```

- Is an expression type-consistent ?
  Example: `2 + "mystring"`
- Does the dimension of an array **access** match with the declaration?
  Example: `int a[255]; a[1][2][3] = 0;`
- Is an array reference within the bounds of the array?
  Example: `int a[255]; a[x] = 0;`

# Context-Sensitive Information (cont.)

- Is a method called with the right number and types of arguments?
  Example:

```
int foo(int a, char b);
foo(1.0, 'c', 2);
```

- Is a break or continue statement enclosed in a loop construct?

All those context-sensitive requirements cannot be specified using CFGs!

Some are so difficult to check that the compiler cannot manage at **compile-time**.

→ need run-time checks

→ we distinguish between static and dynamic semantics

- static semantics are checked at **compile-time**
- dynamic semantics are checked at **run-time**

# Static Semantics

- Static semantic rules are enforced by the compiler at compile-time. They are part of the semantic analysis phase of the compiler.

- Examples:
  - Type checking
    - in **statically typed languages** (C, C++, Java, Ada, C#, ...)
  - Check of subroutine call arguments
  - Use of identifiers in appropriate contexts

# Dynamic Semantics

- Certain semantic rules too complex to be caught at compile-time. For these rules, the **compiler inserts code that performs these checks at run-time**.
- Examples:
  - Array subscript values are within bounds
  - Arithmetic errors (division by zero, overflow/underflow, …)
  - Dereferencing of pointers to invalid objects.
  - Use of a variable that has not been initialized.
- Some languages (Euclid, Eiffel, C) allow programmers to insert explicit semantic checks in the form of assertions, e.g.,

```
assert (buffer_space_left > 0);
```

- If a check fails at run-time, a run-time error (exception) is raised.

# Dynamic Semantics (cont.)

- **Dynamically typed programming languages** conduct type-checking at **run-time**.
  - Example: Python

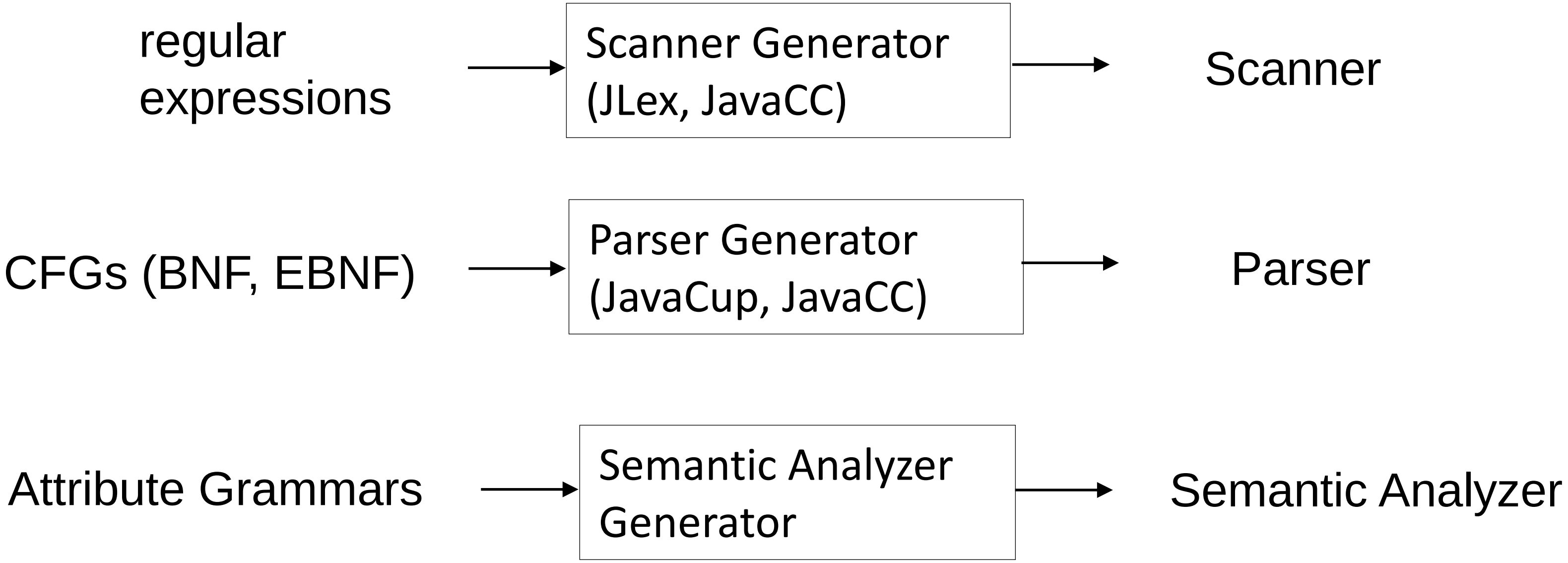
```
a = 5
b = 4.3
print(a + b)
```

- If an operation shall be applied to a value of an inappropriate type, an error (exception) will be raised.
  - The notion of ``inappropriate'' is checked at run-time.

# Context-Sensitive Analysis (for static semantics)

- **Ad-hoc** techniques
  - Symbol tables and code
  - ``Action routines'' in parser generators (e.g., JavaCC)
- **Formal** methods
  - Attribute grammars
  - Type systems
- Our approach for MiniC:
  - Static semantics specified
    - in English (see the MiniC language spec and the Assignment 2-(2) spec), and
    - partly by an attribute grammar.
  - We’ll build a semantic analyzer by hand.

# Automatic Construction of the Front-End



regular expressions → Scanner Generator (JLex, JavaCC) → Scanner

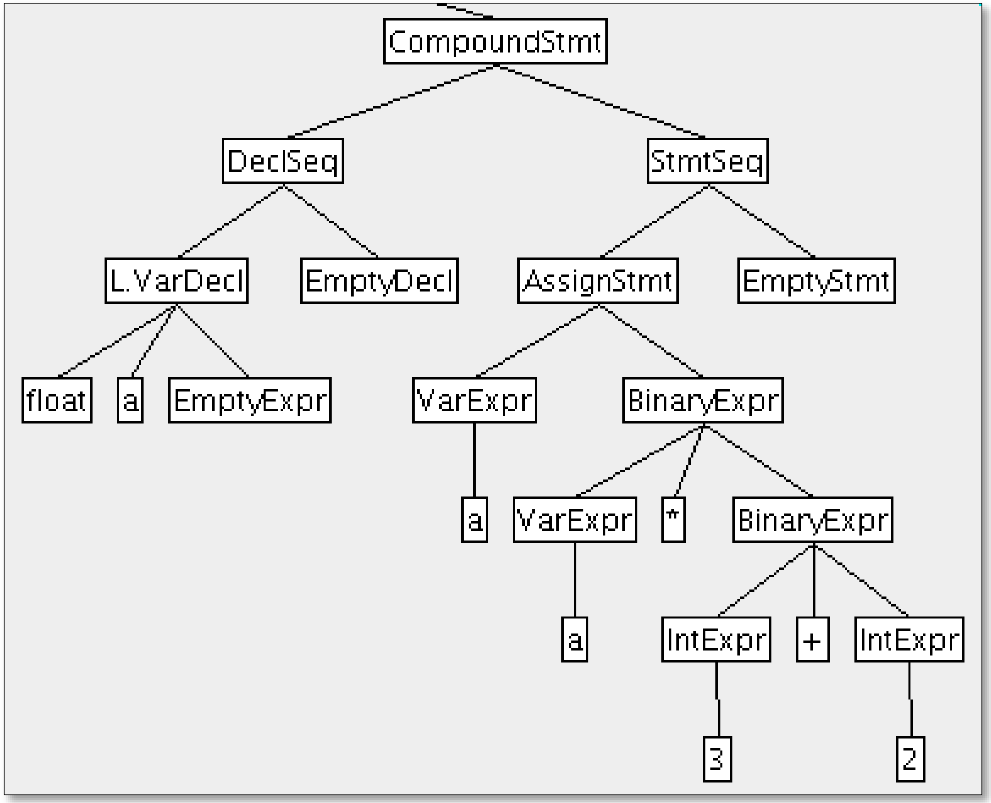
CFGs (BNF, EBNF) → Parser Generator (JavaCup, JavaCC) → Parser

Attribute Grammars → Semantic Analyzer Generator → Semantic Analyzer

There exist no widely accepted semantic analyzer generators.

# Example Information Flow in the AST

```
{
   float a;
   a = a * (3 + 2);
}
```



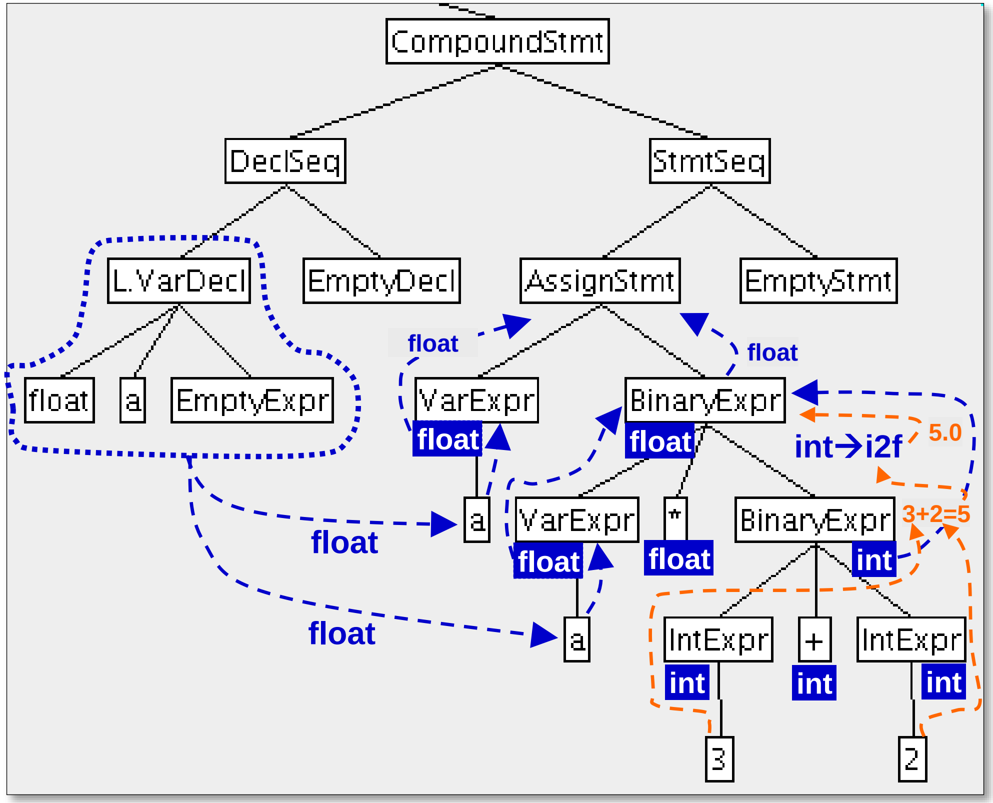
So far our AST contains only the structure, but no meaning of the input program.

- We will attach the ``meaning'' to AST nodes...

- See next slide.

# Example Information Flow in the AST (cont.)

```
{
  float a;
  a = a * (3 + 2);
}
```



So far our AST contains only the structure, but no meaning of the input program.

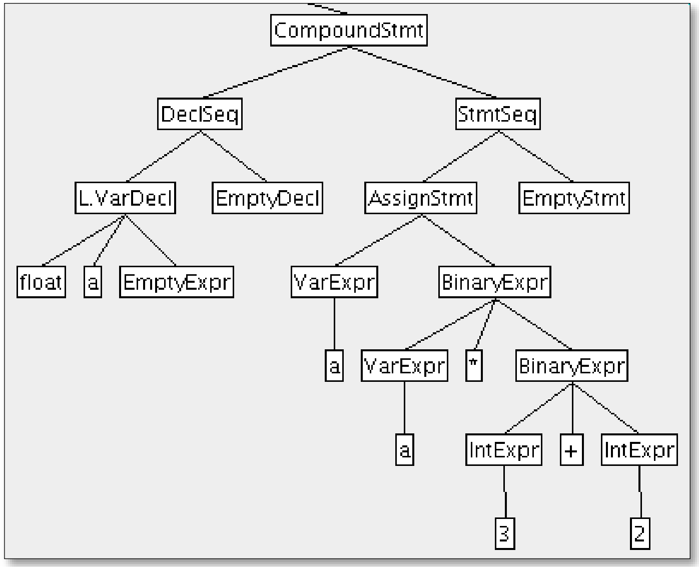
- We will **attach meaning** to nodes of the AST.
  → ``decorating a syntax tree''.
- We **propagate** this information to other places in the AST, to compute more ``meaning''
  → ``tree traversals''

Attribute grammars extend CFGs to allow decoration and traversal.

# AST Traversals

Semantic analysis and code generation typically involve a **depth-first left-to-right traversal** of the AST:

```
void traverse (AST n){
  visit (n); //pre-order
  for each child m of n:
      traverse (m);
  visit (n); //post-order
}
```



A node can be visited or processed

- **before** its children (pre-order traversal)
- **after** its children (post-order traversal)
- **in between** the visits to its children (in-order traversal)

Such traversals are used in all MiniC **tree** packages.
We will use them for semantic analysis and code generation.

# Lecture 3: Semantic Analysis

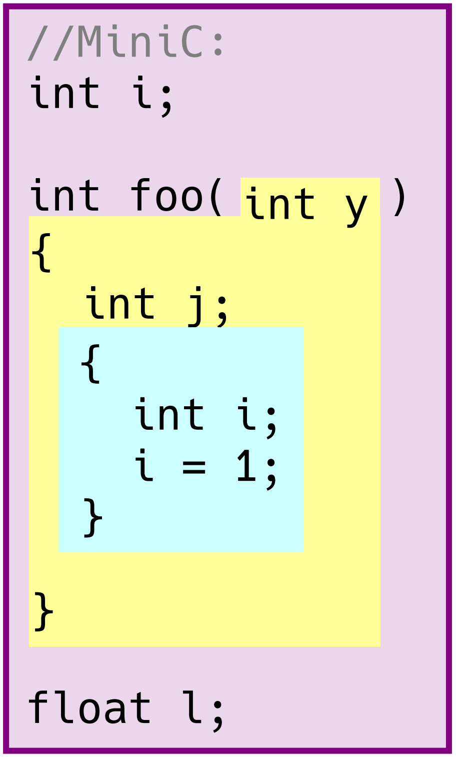
1. Overview & Purpose ✓
   a) Static semantics ✓
   b) Dynamic semantics ✓

2. Static semantics
   a) Two types of semantic constraints:
      - Scope rules
      - Type rules
   b) Two subphases in static semantic analysis
      - Identification (to enforce scope rules)
      - Type checking (to enforce type rules)
   c) Standard environments
   d) Assignment 2-(2)

3. Attribute Grammars

# Blocks

- Block: a language construct that can contain declarations:
  - the compilation units (i.e., the files containing source code)
  - procedures, functions (or methods)
  - compound statements
- Blocks in MiniC:
  - The **entire file** is a block (i.e., the outermost block)
  - **Functions**
  - **Compound statements** { ... }
- Block structured languages
  - permit nesting of blocks
    (blocks within blocks)
  - Examples: Ada, Pascal, Modula-2
  - C: compound statements { ... }, but no nesting of functions within functions.



```
//MiniC:
int i;

int foo( int y )
{
  int j;
  {
    int i;
    i = 1;
  }

}

float l;
```

# Scope

- The scope of a declaration is the part of the program where the declaration is visible.
  - MiniC Example 1: `int i` is visible in the whole program.
  - MiniC Example 2: `int k` is only visible in the innermost block {...}.
- A declaration is in scope at a program point, if the scope of the declaration includes that program point.
- Defining occurrence: declaration of a variable or function.
  - MiniC Example: `int k;`
- Applied occurrence: reference to a declaration.
  - MiniC Example: “`k`” in `k = 1*i;`

```
//MiniC:
int i;

int main() {
  int j;

  {
    int k;
    k = 1*i;
  }
}

float l;
```

# Scope

The scope rules of a language tell us how

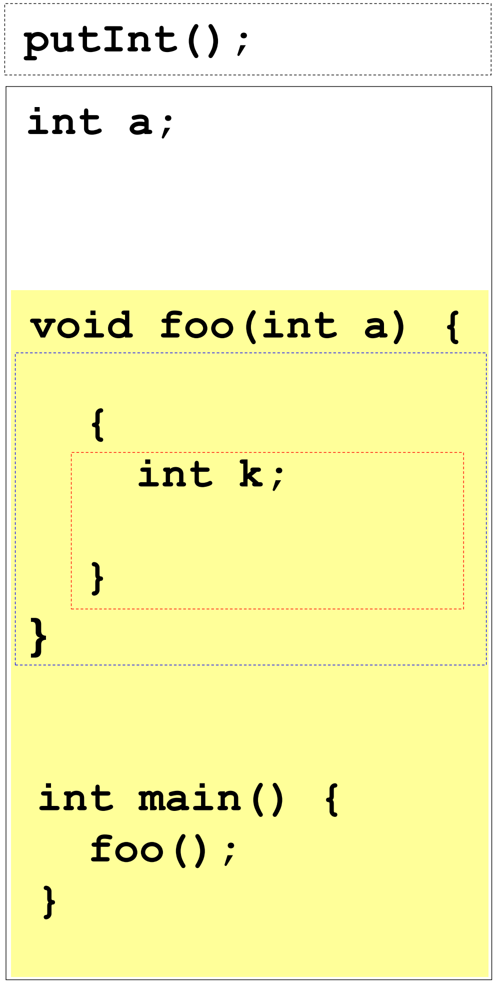
- to find the defining occurrence for an applied occurrence in the program.

That is: ``Given this reference, which is the corresponding declaration?''

```
//MiniC:
int i;

int main() {
  int i,j;

  {
    int k;
    j = 1;
  }
}

float l;
```

# Scope Rules in MiniC

1. Scope of a function declaration: from the point of declaration to the end of the file.
   - Example: scope of function **`foo`** is the yellow area.
2. The scope of a variable in a block: from the point where it is declared to the end of the block.
   - Example: the scope of k is the area inside the dashed red rectangle.
3. The scope of a formal parameter: same as a local variable in the function body (from the point of declaration to the end of the function body).
   - Example: the scope of **`a`** is the area inside the dashed blue rectangle.
4. The scope of a built-in function: the entire program.
   - Example: **`putInt()`** in MiniC.



```
putInt();
```

```
int a;


void foo(int a) {

   {
      int k;

   }
}



 int main() {
   foo();
 }
```

# Scope Rules in MiniC (cont.)

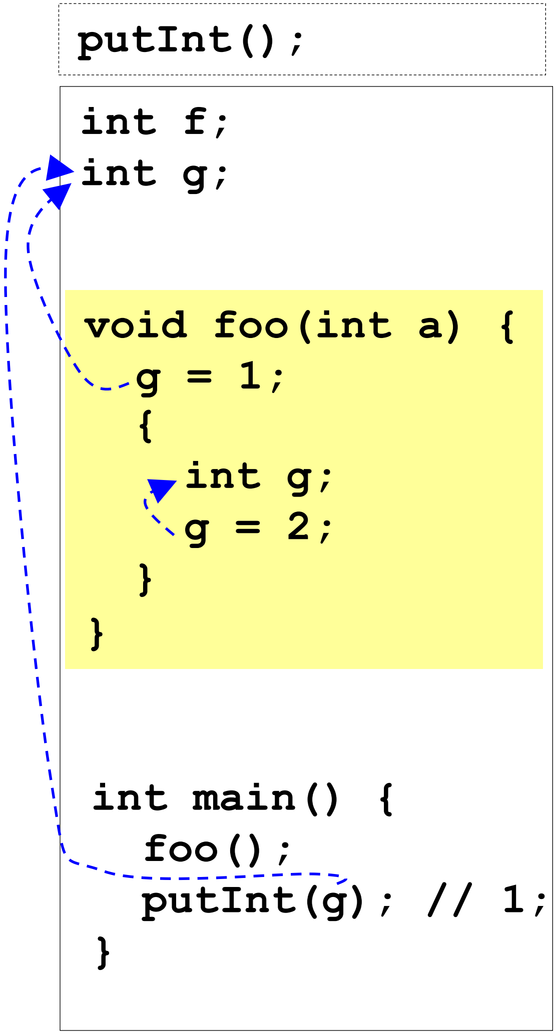
5. No identifier can be declared more than once in a single block.
   - Example: the second declaration of **int f;** in the global block is illegal.
6. Most closely nested rule: for every applied occurrence of a variable, there must be a corresponding declaration.
   - Declarations are searched from the innermost enclosing block to the outermost enclosing block.
   - The first declaration found (i.e., the one in the most ``innermost'' block) is taken.
   - Example: to find the declaration for the applied occurrence of **g** in **g = 1**, we search the following blocks:
     1. The innermost enclosing braces { ... }
        → no declaration of **g** found.
     2. The block of function foo
        → no declaration of **g** found
     3. The global block (file-level).
        → found! (so **int g** in line 3 is the corresponding declaration.

```
putInt();
```

```
int f;
int f;
int g;

void foo(int a) {

   {
     int k;
     g = 1;
   }
}


int main() {
   foo();
}
```

# Scope Rules in MiniC (cont.)

7. Due to Rule 6, the scope of a declaration defined by Rule 1 .. 4 excludes the scope of a declaration in an inner block that uses the same name.
   - Such a gap is known as a scope hole.
   - The inner declaration hides the outer declaration.
   - The outer declaration is not visible in the scope of the inner declaration.
   - Example: the second declaration of **int g;** hides the declaration of **g** in the global block.



```
putInt();
```

```
int f;
int g;

void foo(int a) {
  g = 1;
  {
    int g;
    g = 2;
  }
}


int main() {
  foo();
  putInt(g); // 1;
}
```

# Implication of Scope Rule 1 in MiniC

- A *syntactically* legal MiniC program:

```
int f() {
  g(); // not in scope
}

int g() {
  f();
}
```

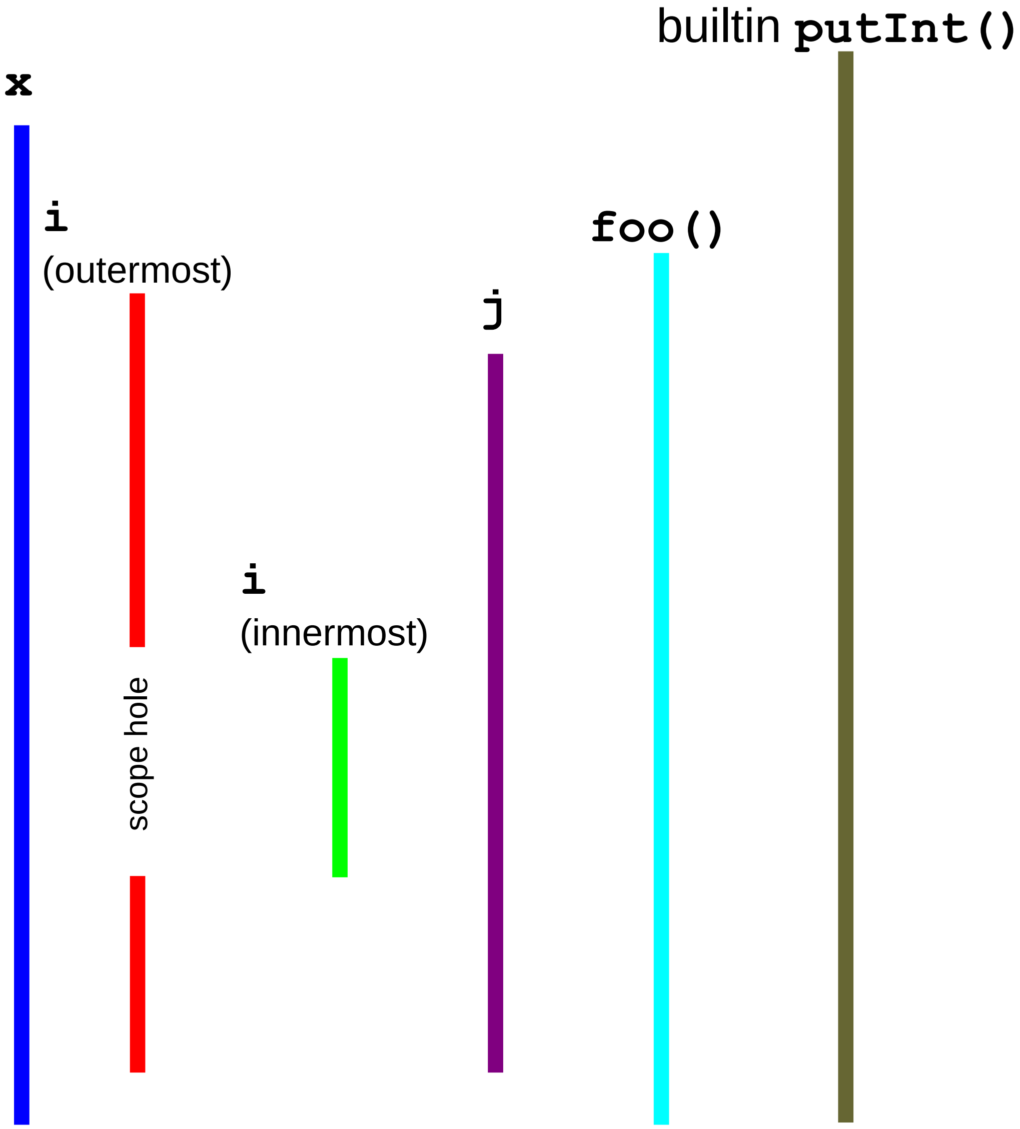
Semantically, this MinC program is illegal.

- Disallowing the above program allows identification and type checking in one pass.
- ANSI C and C++ solve this scoping problem using function prototypes.

# Example: Scope Rules

```
putInt();
```

```
int x;

void foo() {
  int i;
  int j;
  i = 2;
  j = 3;
  putInt(i); // 2
  putInt(j); // 3
  {
    int i;
    i = 4;
    putInt(i); // 4
    putInt(j); // 3
  }
  putInt(i); // 2
  putInt(j); // 3
}
```



# Scope Levels in Block-Structured Languages

**Scope levels** correspond to the **scope nesting-depth** of a declaration.

- Scope levels in general:
  1. The declarations in the outermost block are on level 1.
  2. Increment the level every time when we move from an enclosing to an enclosed block.
  3. The built-in functions and constants of a language are on level 0 or 1.
- Scope levels in MiniC:
  - All function- and global variable declarations are on level 1.
  - Rule 2 as above.
  - All built-in functions are on level 1.
    Consequence: built-in functions cannot be re-declared as user-functions or global variables (MiniC Scope Rule 5).

# Example: Scope Levels

level 1:

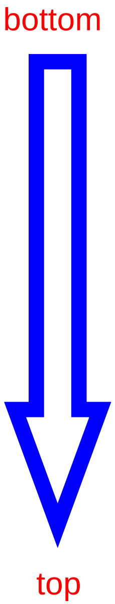
```
putInt();
```

```
int x;

void foo() { // level 2:
  int i;
  int j;
  i = 2;
  j = 3;
  putInt(i);
  putInt(j);
  { // level 3:
     int i;
     i = 4;
     putInt(i);
     putInt(j);
  }
  putInt(i);
  putInt(j);
}
```

lookup i

## Scope Stack

| Identifier | Scope Level |
|---|---|
| putInt() | 1 |
| x | 1 |
| foo | 1 |
| i | 2 |
| j | 2 |
| i | 3 |



Scope stack lookups are from top to bottom…

# Lecture 3: Semantic Analysis

1. Overview & Purpose ✓
   a) Static semantics ✓
   b) Dynamic semantics ✓

2. Static semantics
   a) Two types of semantic constraints:
      - Scope rules ✓
      - Type rules

   b) Two subphases in static semantic analysis
      - Identification (to enforce scope rules)
      - Type checking (to enforce type rules)

   coming next

   c) Standard environments

   d) Assignment 2-(2)

3. Attribute Grammars

# Identification

- Identification:
  - Find the declaration for each applied occurrence.
  - Applied occurrences are identifiers in MiniC!
  - Report an error if no declaration exists.
- The attributes of an identifier:
  - for a variable: the type of the variable
  - for a function: the functions return type and the types of the formal parameters (``signature'' of the function).
- In our MiniC compiler: for each identifier we store a pointer to its declaration (Assignment 2-(2) & see next slide).

# The Inherited Attribute declAST from MiniC.AstGen.ID.java for Decorating ASTs during Identification

```
package MiniC.AstGen;

import MiniC.Scanner.SourcePos;

public class ID extends Terminal {

    public AST declAST;

    public ID (String Lexeme, SourcePos pos) {
      super (pos);
      this.Lexeme = Lexeme;
      declAST = null;
    }

    public void accept(Visitor v) {
      v.visit(this);
    }

}
```

Here the word ``inherited" is not an OOP-term, but a term belonging to attribute grammars (coming soon..).

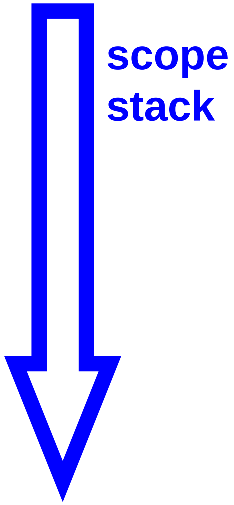
# Two Tasks with Identification

1. Processing declarations:
   - Call ScopeStack.openScope() at the start of a block
   - Call ScopeStack.closeScope() at the end of block
   - Call ScopeStack.enter() to push the ID of a declaration together with a pointer to its declaration on the scope stack.
2. Processing applied occurrences—decorating ID AST nodes
   - Call ScopeStack.retrieve(ID) to fetch a pointer to the innermost declaration for ID from the scope stack.
   - This pointer is stored in the declAST field of the AST node for ID.
   - declAST is set to null if no matching declaration is found on the scope stack.
     → used to report errors.

# The MiniC Standard Environment

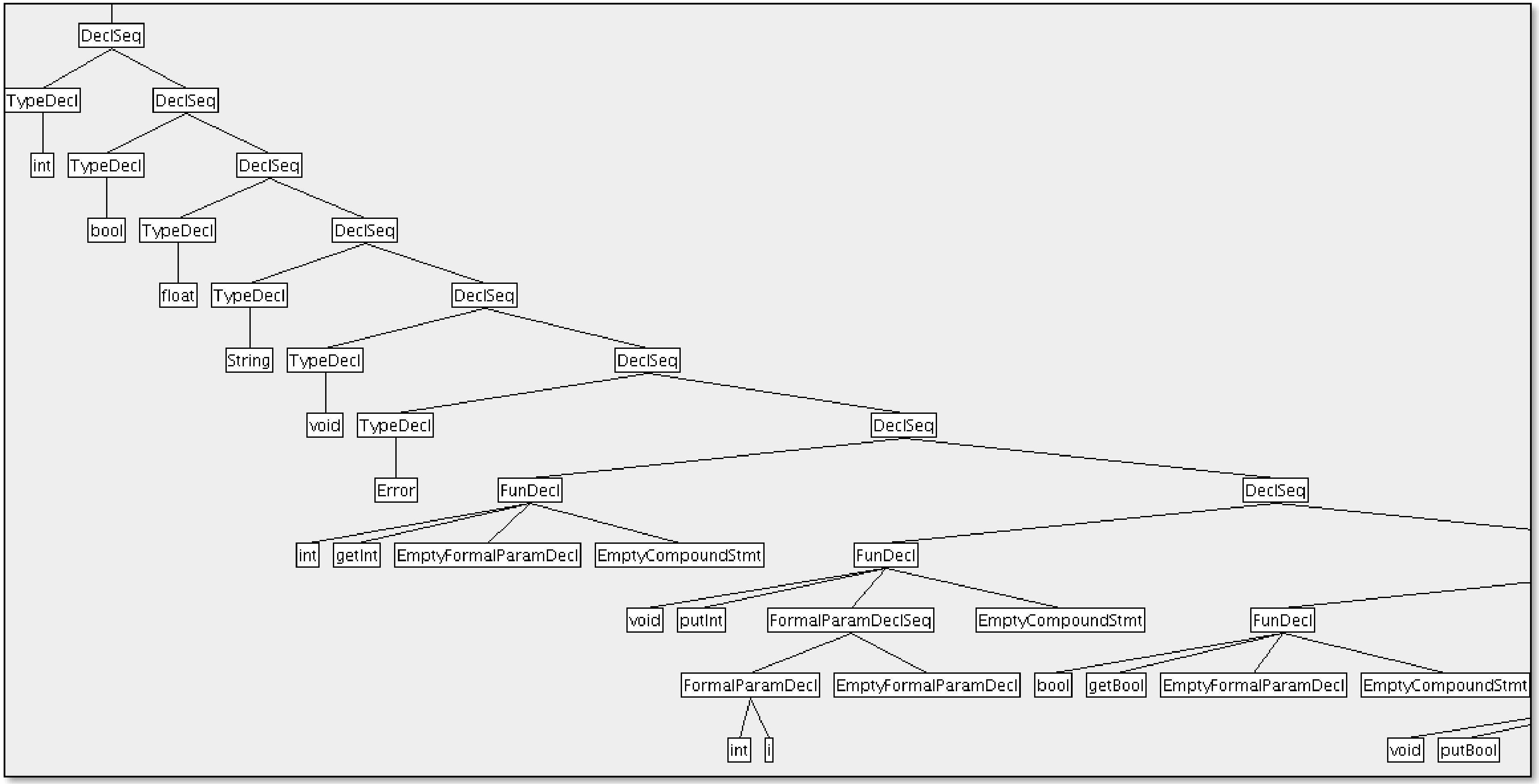
- Most languages contain a set of predefined types, functions, variables and constants.
  - Java: **`java.lang`**
  - MiniC: 9 built-in functions and several primitive types (**`int`**, **`void`**,...).
- At the start of identification, the ScopeStack is pre-loaded with the 9 built-in functions of MiniC:

| Identifier | Scope Level | Attribute |
|---|---|---|
| **`putInt()`** | 1 | ptr to the putInt AST |
| **`getInt()`** | 1 | ptr to getInt AST |
| **`putFloat()`** | 1 | ptr to putFloat AST |
| **`getFloat()`** | 1 | ptr to getFloat AST |
| *entries for the other 4 built-in functions* | | |
| **`putLn()`** | 1 | ptr to putLn AST |



# The MiniC Standard Environment (cont.)

- You can print the ASTs for the MiniC Standard Environment
  - option `-envast`

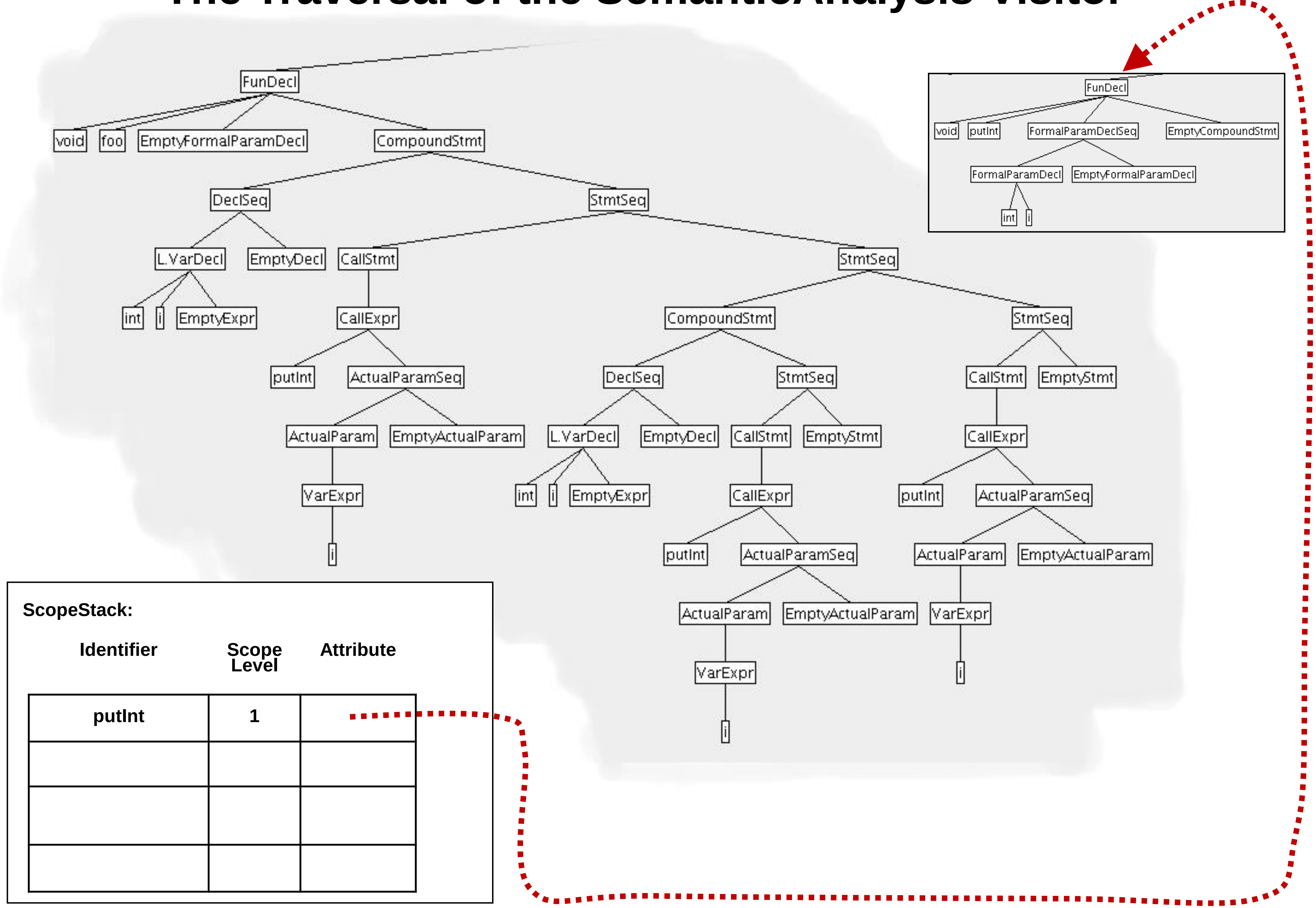


# Example

```
void foo() {
   int i;
   putInt(i);
   {
      int i;
      putInt(i);
   }
   putInt(i);
}
```

The next slides shows the AST for foo’s compound statements, the traversal of the AST, the calls to the scope stack, the scope stack and the decorated AST...

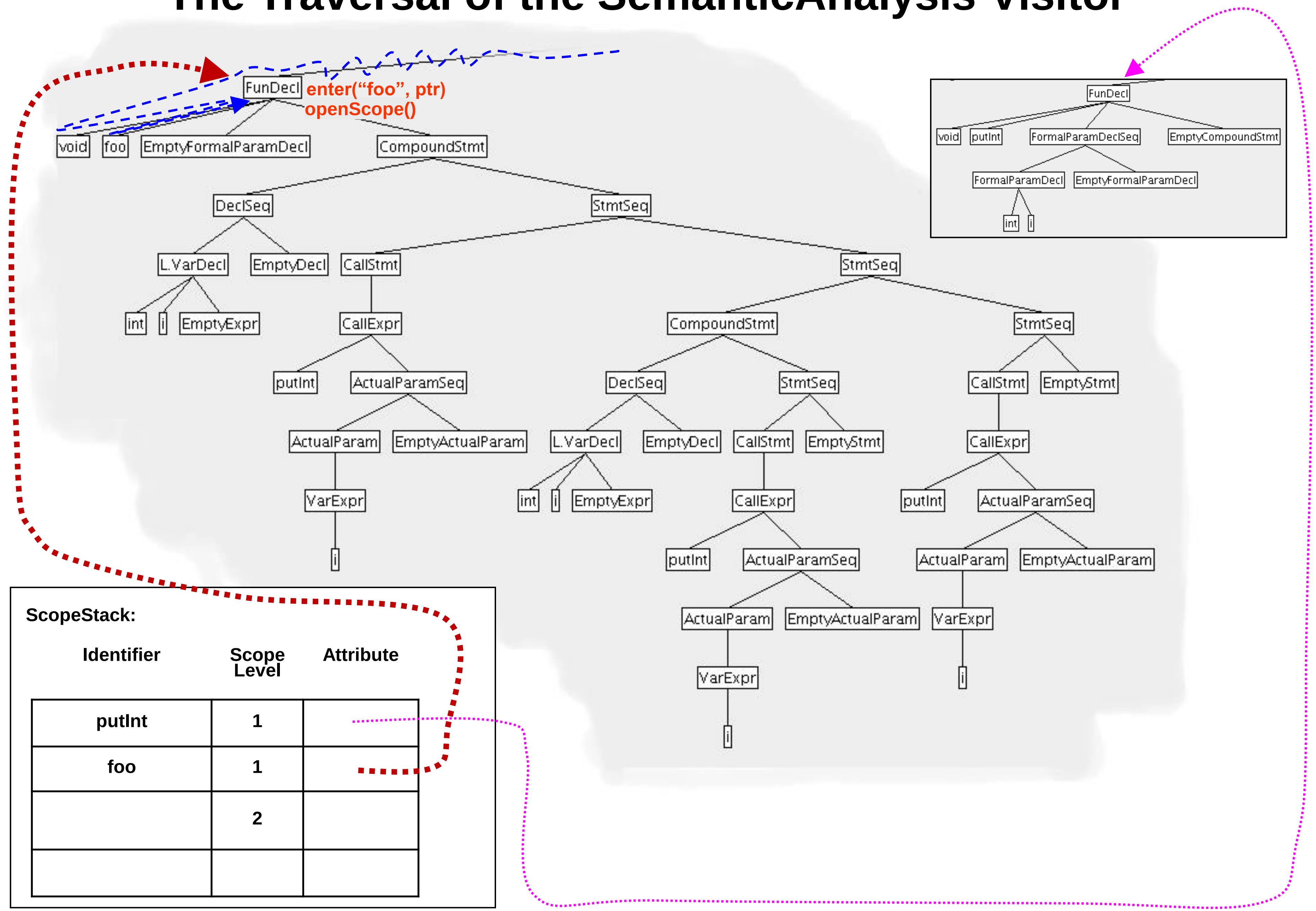
# The Traversal of the SemanticAnalysis Visitor



**ScopeStack:**

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| | | |
| | | |
| | | |

# The Traversal of the SemanticAnalysis Visitor

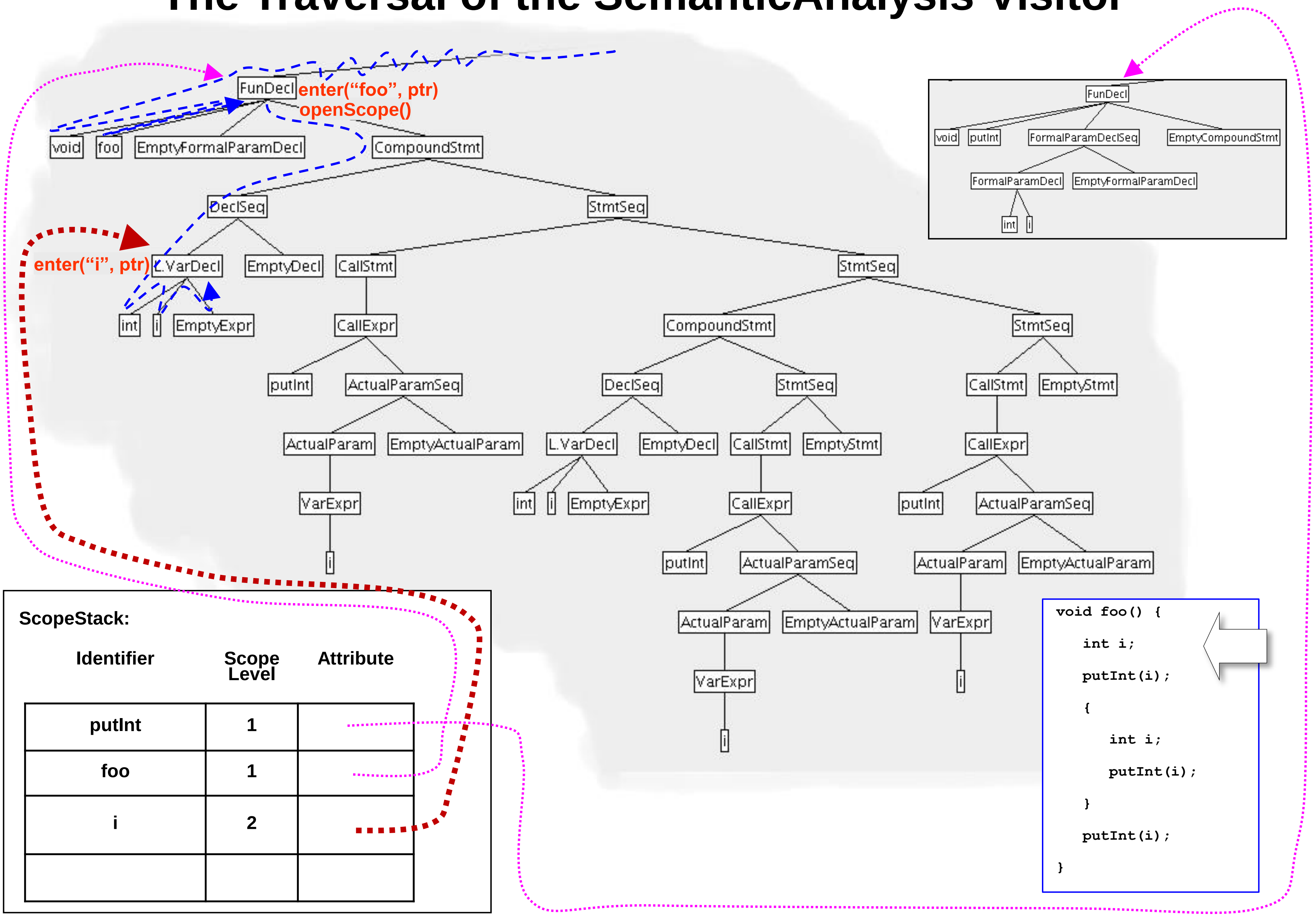


enter("foo", ptr)
openScope()

ScopeStack:

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| | 2 | |
| | | |

# The Traversal of the SemanticAnalysis Visitor

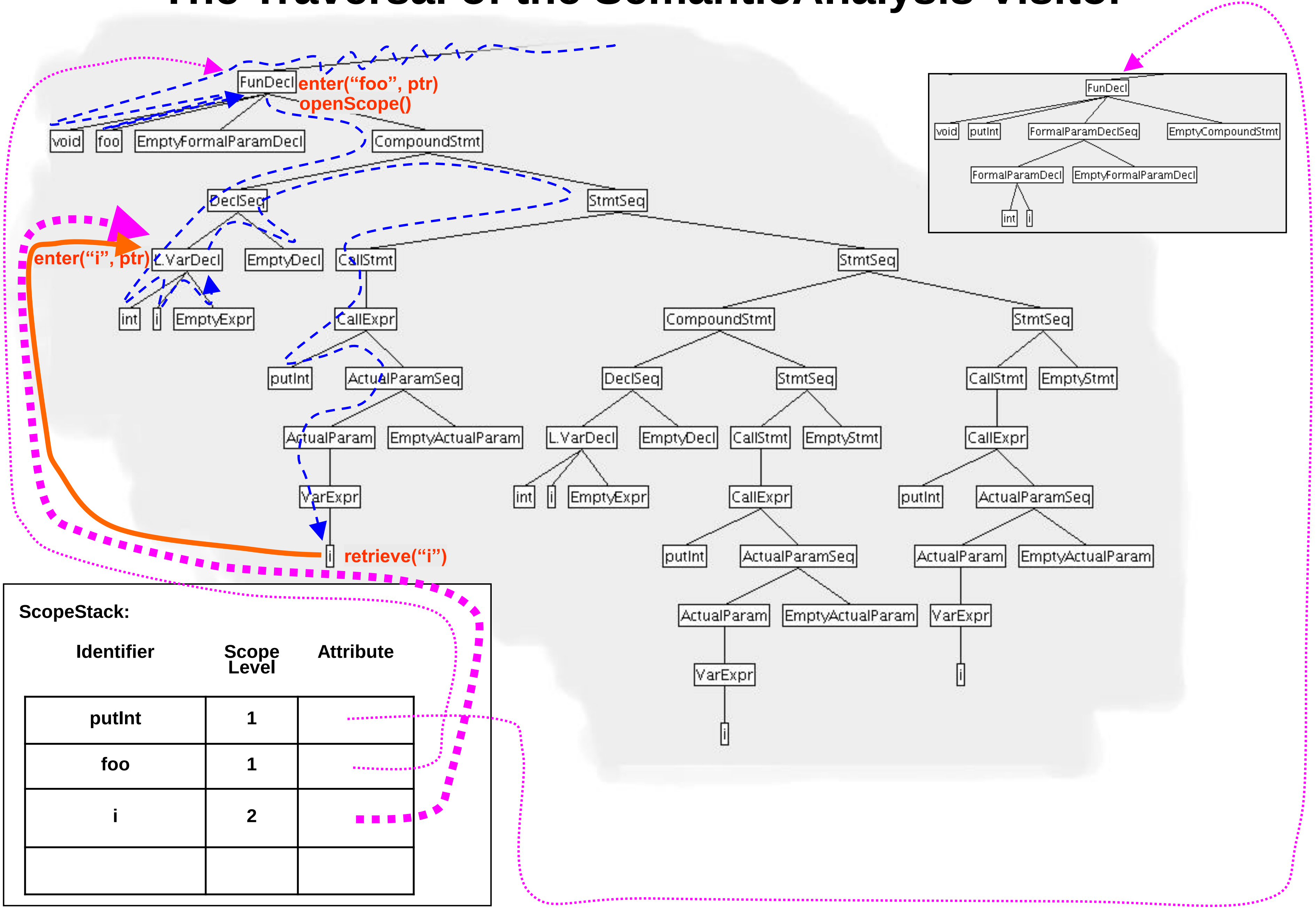


ScopeStack:

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| i | 2 | |
| | | |

```
void foo() {
   int i;
   putInt(i);
   {
      int i;
      putInt(i);
   }
   putInt(i);
}
```

# The Traversal of the SemanticAnalysis Visitor



enter("foo", ptr)
openScope()

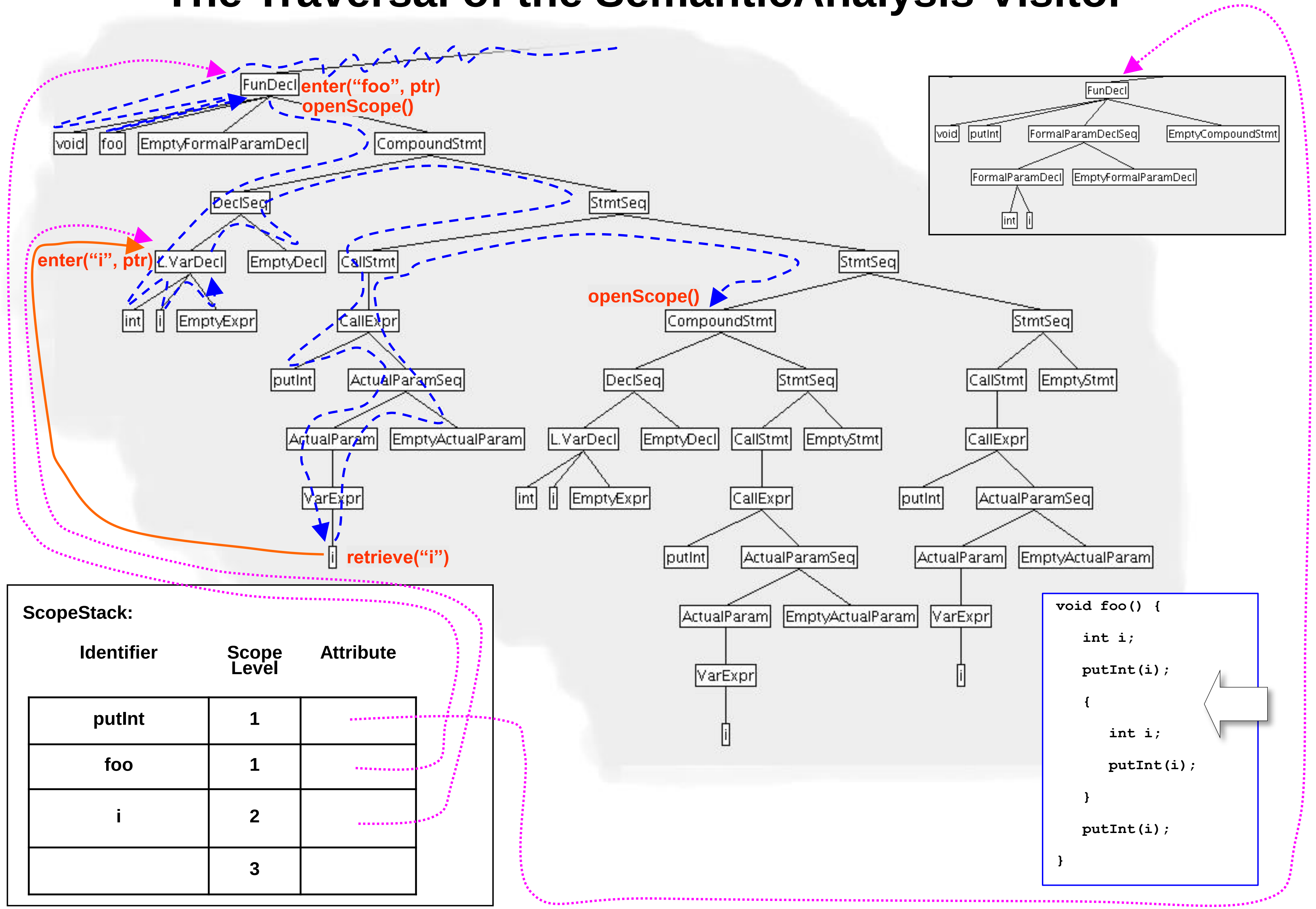
enter("i", ptr)

retrieve("i")

ScopeStack:

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| i | 2 | |
| | | |

# The Traversal of the SemanticAnalysis Visitor

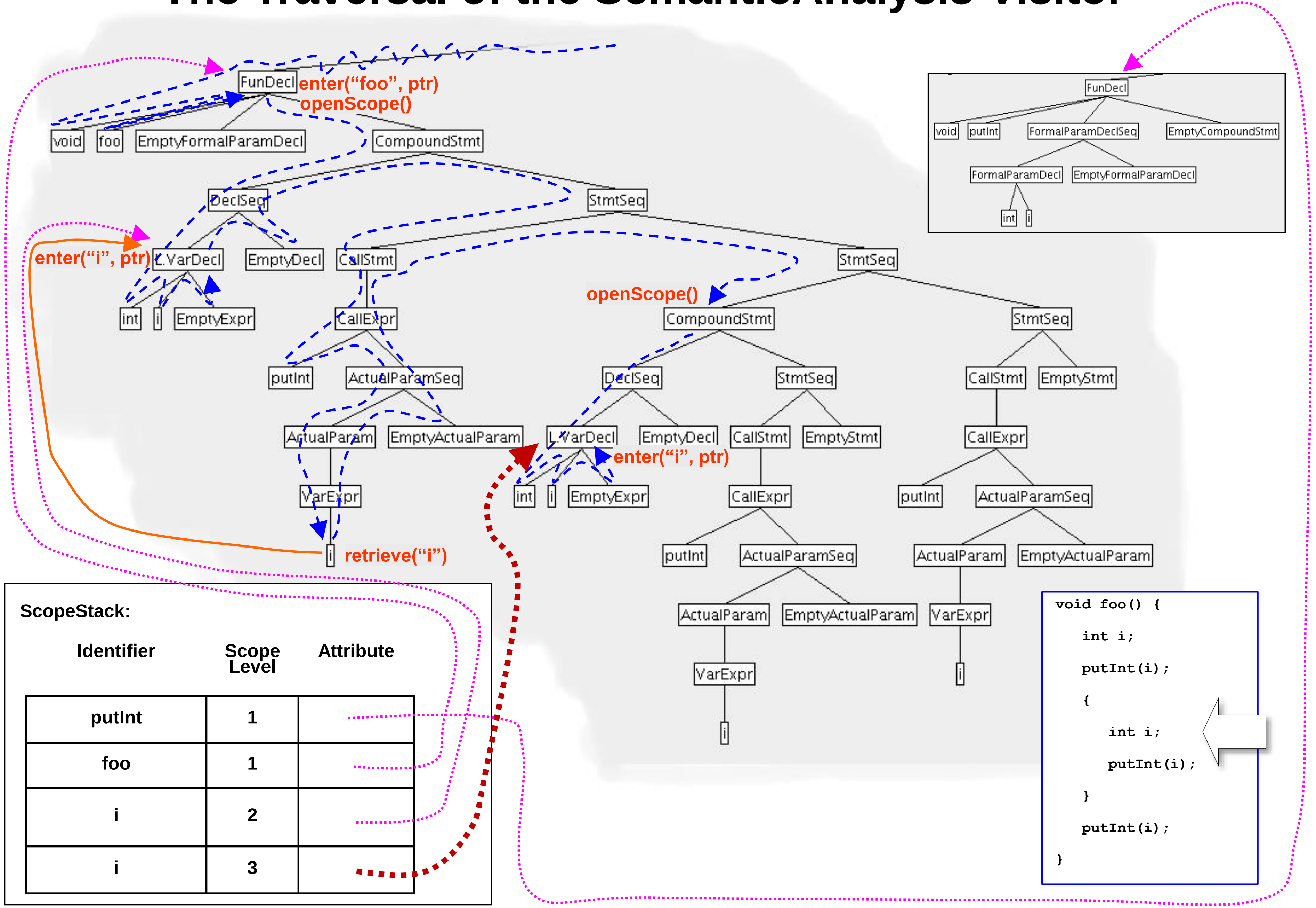


ScopeStack:

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| i | 2 | |
| | 3 | |

```
void foo() {
   int i;
   putInt(i);
   {
      int i;
      putInt(i);
   }
   putInt(i);
}
```

# The Traversal of the SemanticAnalysis Visitor

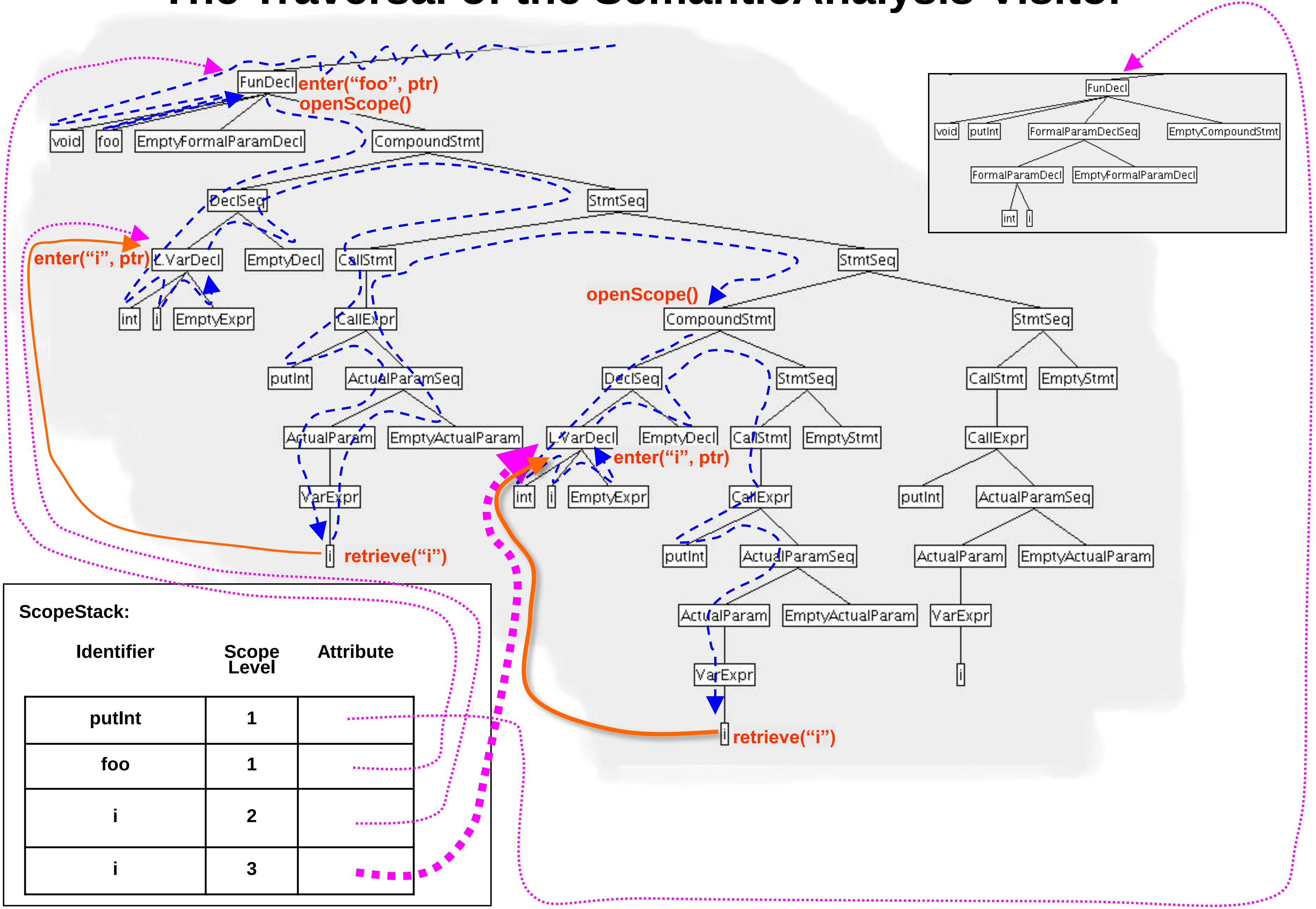


ScopeStack:

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| i | 2 | |
| i | 3 | |

```
void foo() {
   int i;
   putInt(i);
   {
      int i;
      putInt(i);
   }
   putInt(i);
}
```

# The Traversal of the SemanticAnalysis Visitor



enter(“foo”, ptr)
openScope()

enter(“i”, ptr)

retrieve(“i”)

openScope()

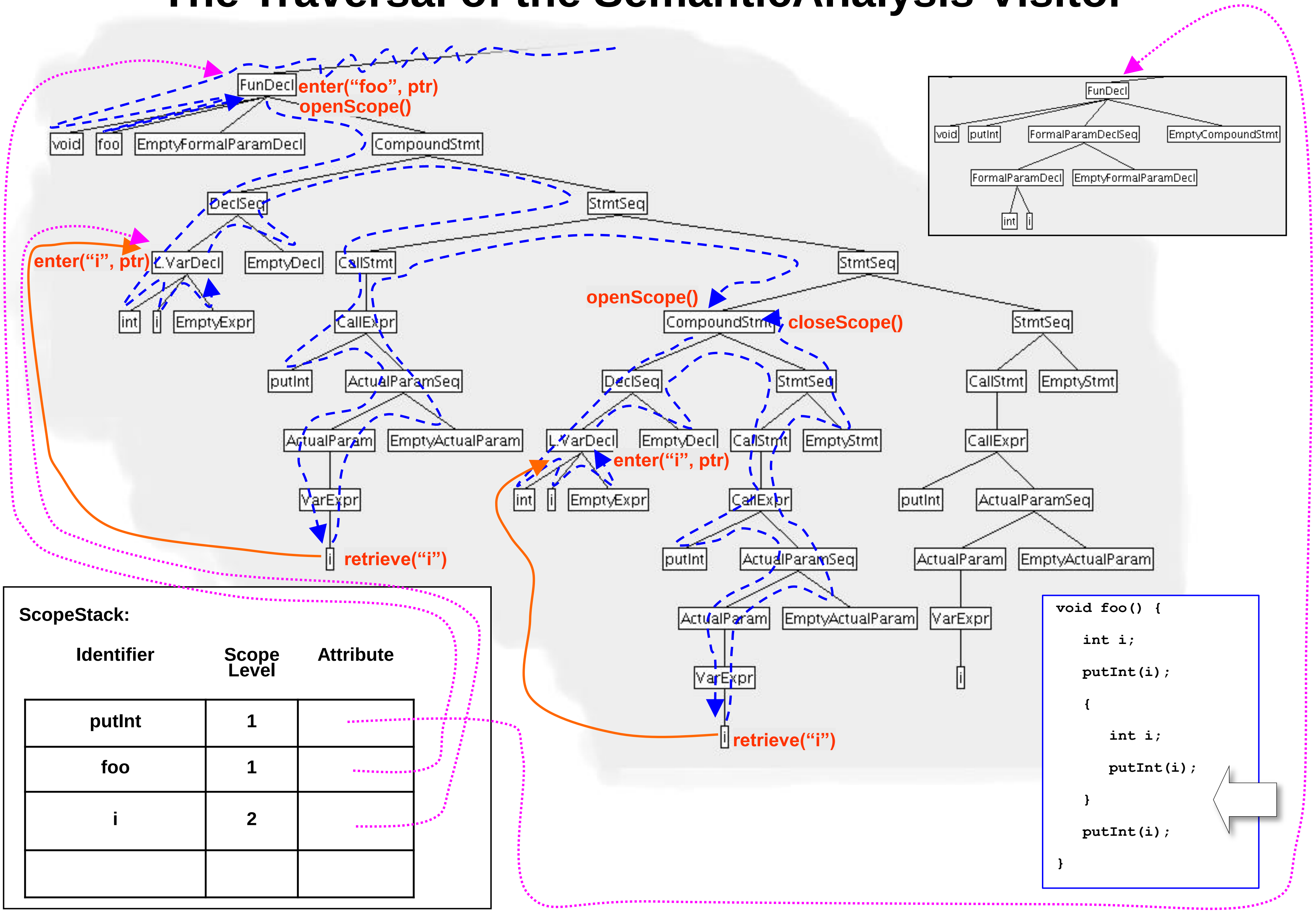
enter(“i”, ptr)

retrieve(“i”)

**ScopeStack:**

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| i | 2 | |
| i | 3 | |

# The Traversal of the SemanticAnalysis Visitor

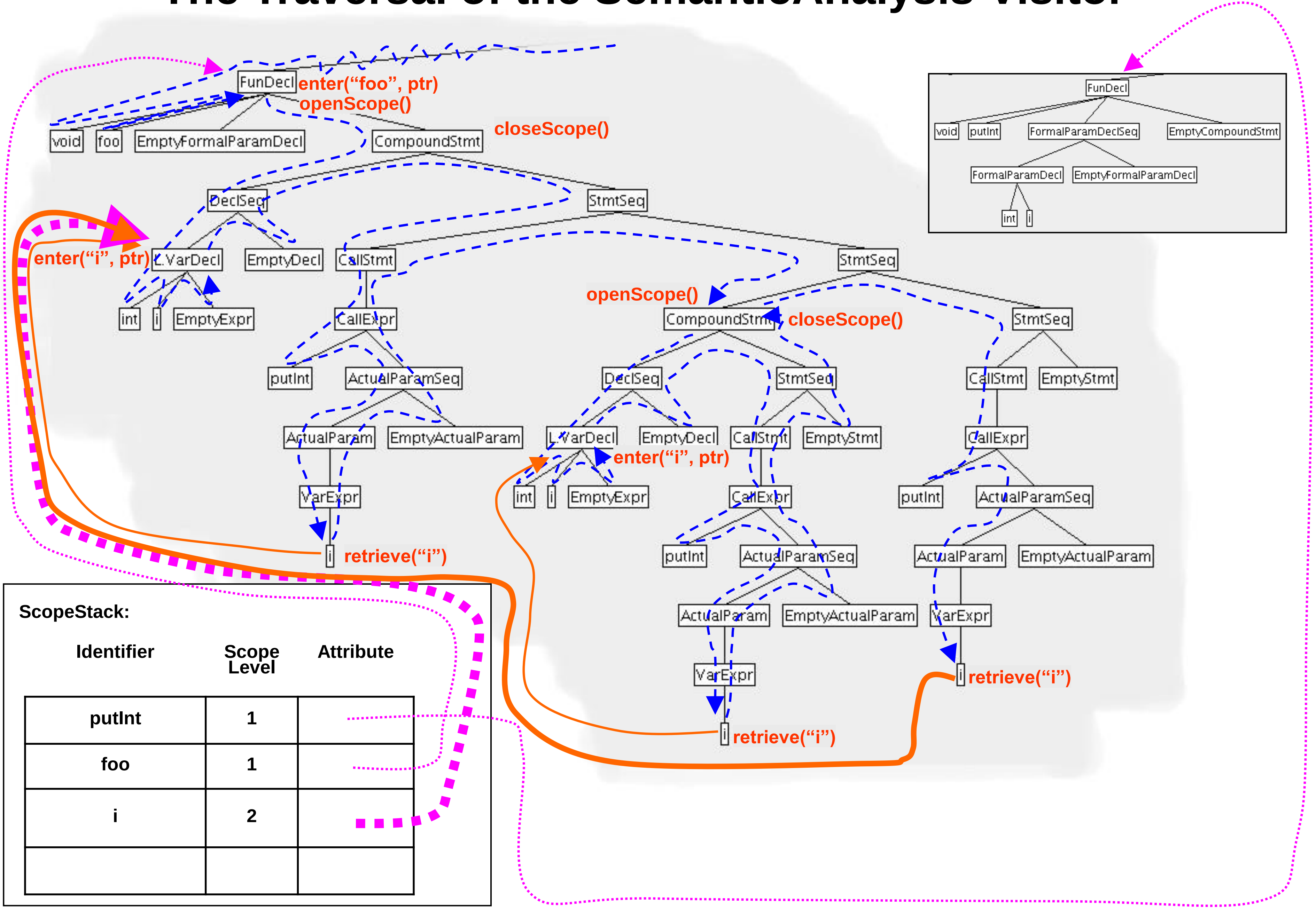


ScopeStack:

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| i | 2 | |
| | | |

```
void foo() {
   int i;
   putInt(i);
   {
      int i;
      putInt(i);
   }
   putInt(i);
}
```

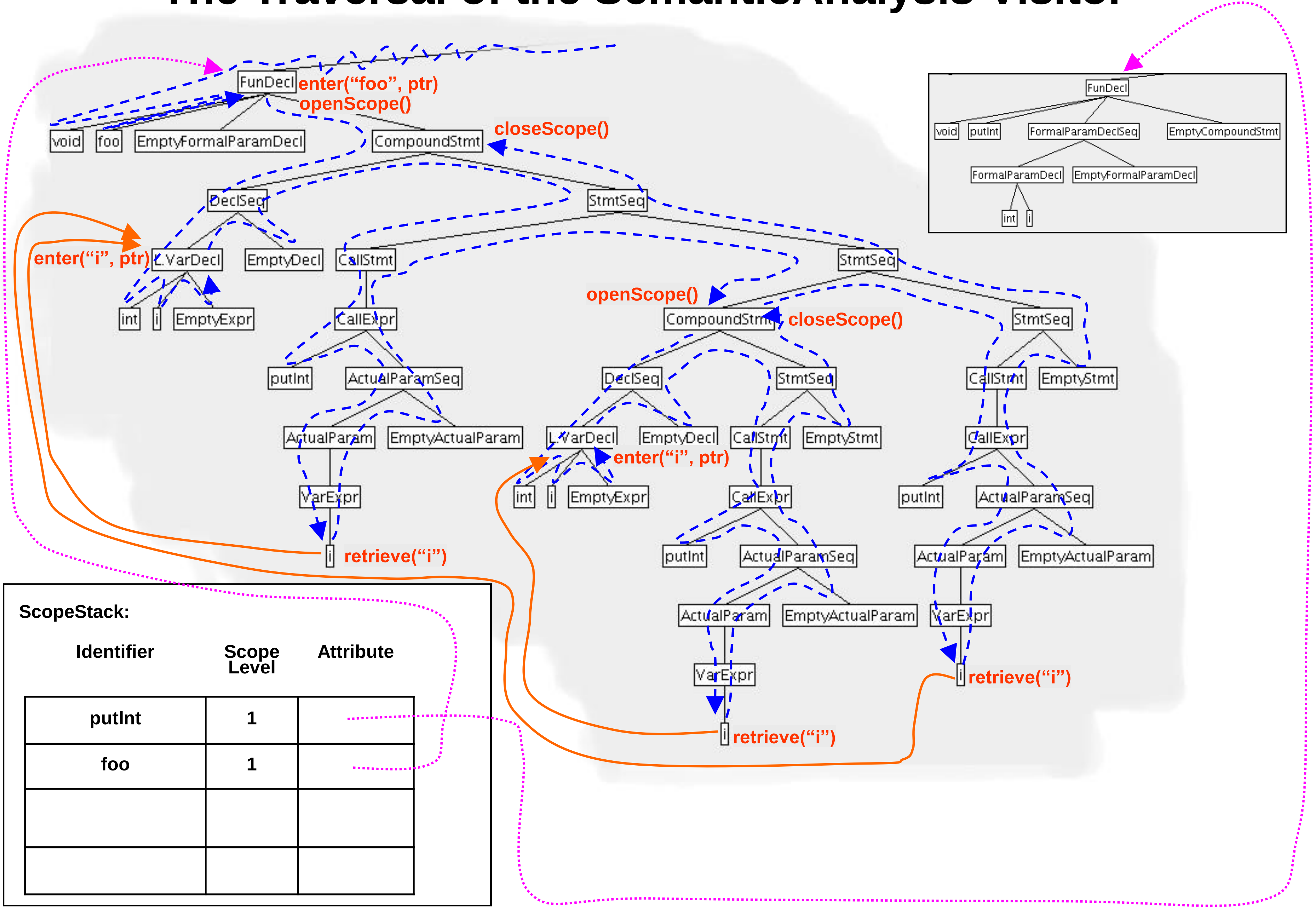
# The Traversal of the SemanticAnalysis Visitor



ScopeStack:

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| i | 2 | |
| | | |

# The Traversal of the SemanticAnalysis Visitor



**ScopeStack:**

| Identifier | Scope Level | Attribute |
|---|---|---|
| putInt | 1 | |
| foo | 1 | |
| | | |
| | | |

# Lecture 3: Semantic Analysis

1. Overview & Purpose ✓
   a) Static semantics ✓
   b) Dynamic semantics ✓

2. Static semantics
   a) Two types of semantic constraints:
      - Scope rules ✓
      - Type rules

        coming next

   b) Two subphases in static semantic analysis
      - Identification (to enforce scope rules) ✓
      - Type checking (to enforce type rules)

        coming next

   c) Standard environments ✓

   d) Assignment 2-(2)

3. Attribute Grammars

# Type Checking

Data type: a set of values plus a set of operations on those values.

Example: MiniC int: values range from $-2^{31}$ to $2^{31} - 1$

Operations +, -, *, / : *int X int → int*
+,- : *int → int*

Further operations on int, returning bool:
<, <=, >, >=, !=, == : *int X int → bool*

MiniC is statically type-checked:

- types of all entities are determined at compile-time.

Type-rules: the rules to determine the type of each language construct and decide whether the type is valid.

MiniC Example:
```
int a = 2;  // right-hand side must be assignment-
                                         // compatible to left-
  hand side!
```

Type Checking: applying the language’s type-rules.

# The Synthesized Attribute type in Expr.java

The abstract class Expr.java:

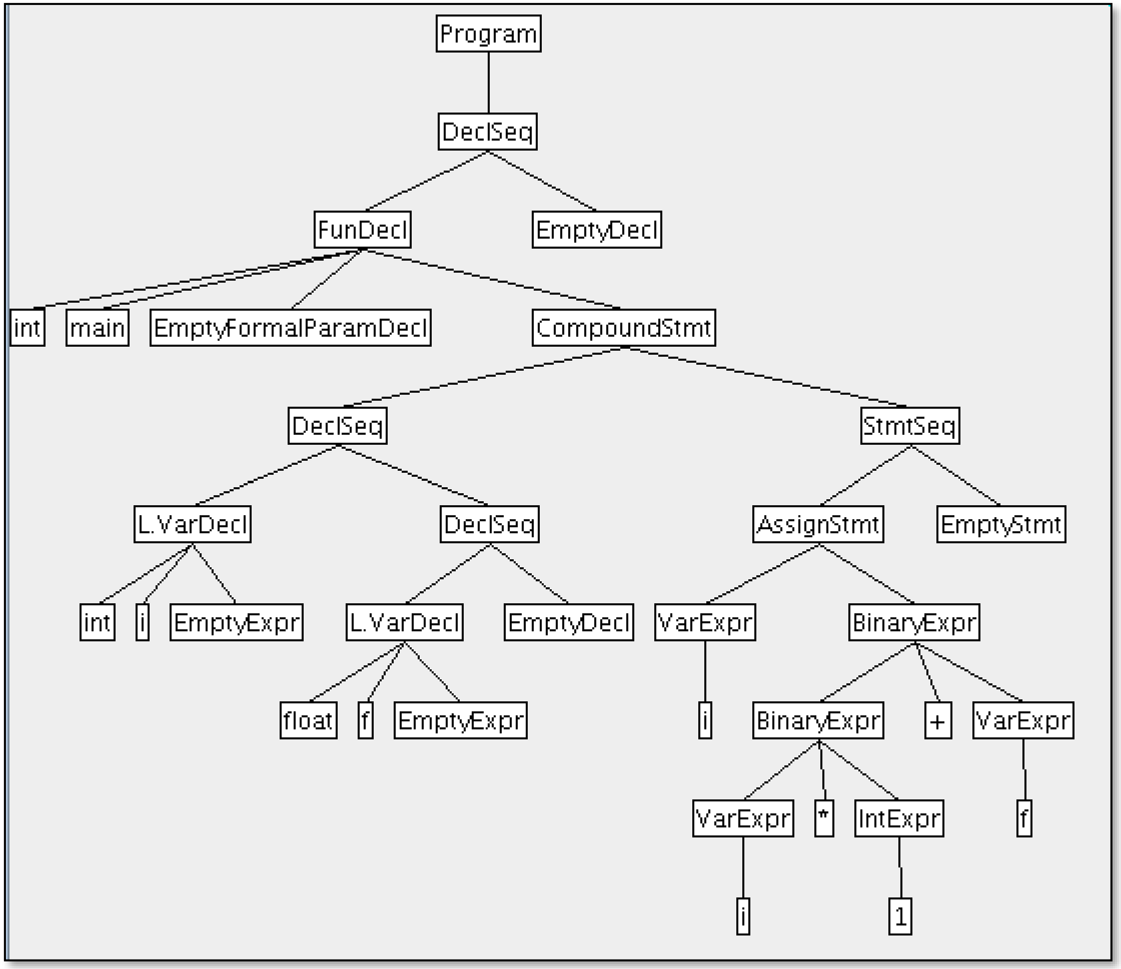
```
package MiniC.AstGen;

import MiniC.Scanner.SourcePos;

public abstract class Expr extends AST {

    public Type type;

    public Expr (SourcePos pos) {
          super (pos);
    }

}
```

All concrete Expr classes inherit the type instance variable.

The word ``synthesized” refer to attribute grammars (coming soon..). Synthesized information is passed bottom-up in the AST.

# Type-Checking Expressions

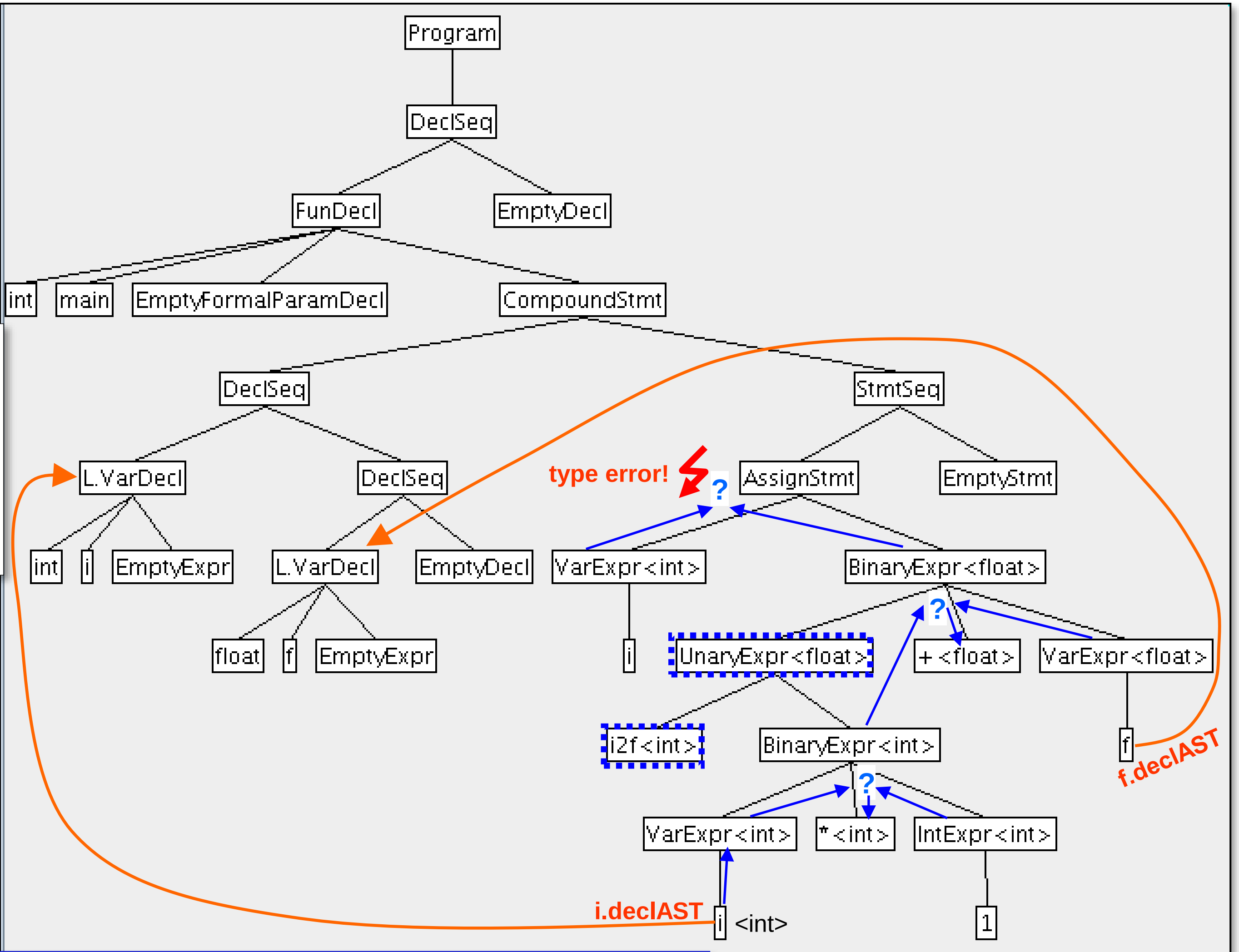
```
int main() {
  int i;
  float f;
  i = i * 1 + f;
}
```



The undecorated AST before semantic analysis.

# Type-Checking Expressions

```
int main() {
  int i;
  float f;
  i = i * 1 + f;
}
```



The AST after semantic analysis, decorated with type information. Nodes marked ⬚ have been inserted for type coercion (see next slides). Blue arrows show the flow of the *type* attribute, <...>.

# Type Coercions

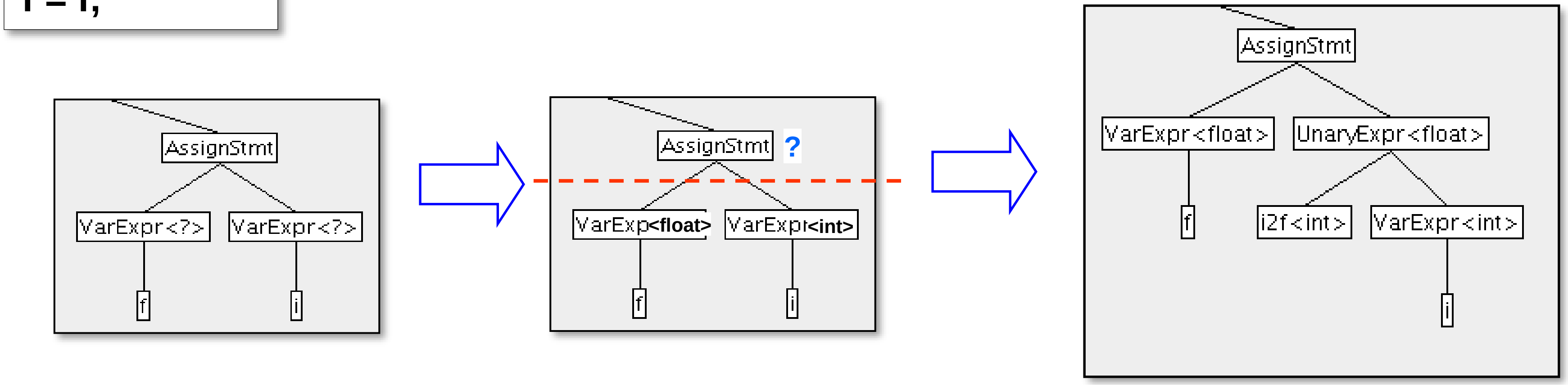
- MiniC: Two types of operations for symbol "+":
  "+": *int x int → int,* "+": *float x float → float*
  - "+" is overloaded with two operations
- Overload resolution: based on argument types, choose "+" operation:
  - integer addition when both operands are of type integer
  - floating point addition when both operands are floats
  - Example:

```
1 + 2       // "+": int x int → int
2.0 + 1.0   // "+": float x float → float
```

- Type coercion: programming languages tend to **relax** type rules a bit: if type T is expected in a given situation, then there might be other types T' that are accepted as well.
  Example: `int i; float f; f = i;`
- Float f can only be assigned an expression of type float.
  - However, MiniC allows int type instead (to be nice to the programmer).
  - This only works if the compiler converts the int to a float.
  - Such a type conversion is called implicit type conversion or type coercion.

# Type Coercion Example

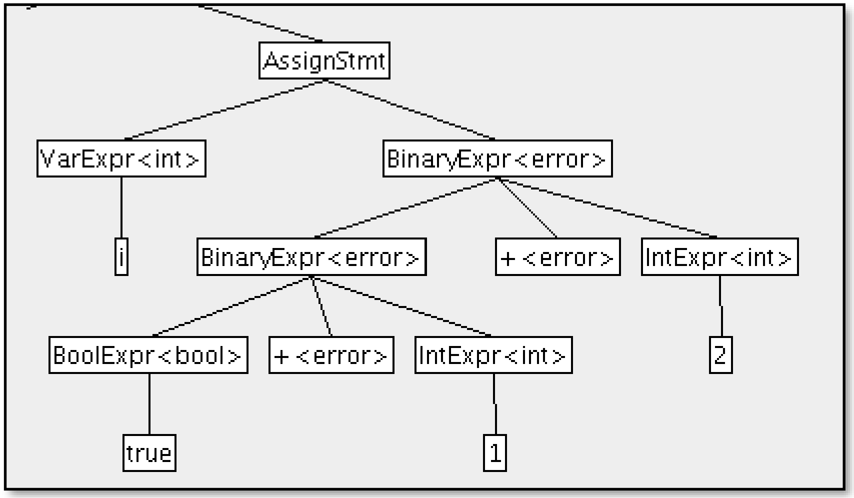
```
int i;
float f;
f = i;
```



- The AST subtrees below the red dashed line have already been semantically analyzed. The next step is to type-check the assignment statement.
- To assign the right-hand side (int) to the left-hand side (float), it must be coerced to float.
  → the UnaryExpr node with operator i2f is inserted.
- The effect of coercion, described in MiniC:

```
int i;
float f;
f = i2f(i);
```

# Error Detection, Reporting and Recovery



- Error detection: based on type rules
- Reporting: prints meaningful error messages
- Recovery: continue type checking in the presence of type errors.
- An ill-typed expression is given type `StdEnvironment.errorType` (shown as <error> in the AST images).
- To avoid **cascaded error messages**, compilers do not report an error if one operand of an expression is already of type `StdEnvironment.errorType`.

# Assignment 2-(2)

We implement a one-pass semantic analyzer using the visitor design pattern.

- ❑Identification
- ❑Type checking
  - – ensure MiniC type rules (see the Assignment 2-(2) spec)
  - – add i2f where needed
  - – perform overload resolution
- ❑Decorated ASTs:
  - – the synthesized `type` attribute in Expr nodes.
  - – the inherited `astDecl` attribute in ID nodes.

# Lecture 3: Semantic Analysis

1. Overview & Purpose ✓
   a) Static semantics ✓
   b) Dynamic semantics ✓

2. Static semantics
   a) Two types of semantic constraints:
      - Scope rules ✓
      - Type rules ✓
   b) Two subphases in static semantic analysis
      - Identification (to enforce scope rules) ✓
      - Type checking (to enforce type rules) ✓
   c) Standard environments ✓
   d) Assignment 2-(2) ✓

3. Attribute Grammars

# Attribute Grammars

- Invented by Donald E. Knuth in 1968
- Attributes (synthesized and inherited)
  - S-attributed grammars
  - L-attributed grammars
- Semantic rules (also called semantic functions)
- Computation of Attributes
  - The Visitor Design Pattern

# Attribute Grammars and the Compiler Frontend

- Context-sensitive static semantics of a language cannot be specified with context-free grammars.
- Attribute grammars **extend CFGs** to complete the specification of what legal programs should look like.

After scanning and parsing:

- Semantic analysis enforces the static semantics of a language:
  - Identification (using a symbol table or the AST)
  - Type checking
- The compiler inserts run-time checks to enforce the dynamic semantics of a language.
  → this completes the semantic checks!

# Attribute Grammars

- An attribute grammar connects syntax and semantics.
- Each grammar production has a semantic rule with actions to modify values of attributes.
  - Each terminal/nonterminal may have any number of attributes
  - Attributes hold information related to the terminal/nonterminal.
- General form:

| production | semantic rule |
|---|---|
| A ::= B C | A.x :=…; B.x :=…; C.x :=…; |

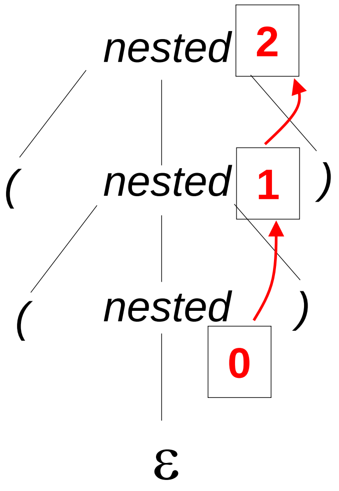
- Semantic rules are used by a compiler to enforce static semantics.

# Attribute Grammar Example 1

- The **val** attribute holds the subtotal value of the subexpression.
- Nonterminals are indexed in the attribute grammar to distinguish multiple occurrences of the nonterminal in a production.

| **productions** | **semantic rules** |
|---|---|
| $\text{nested}_1$ ::= ( $\text{nested}_2$ ) | $\text{nested}_1$.**val** := $\text{nested}_2$.**val** + 1 |
| nested ::= ε | nested.**val** := 0 |

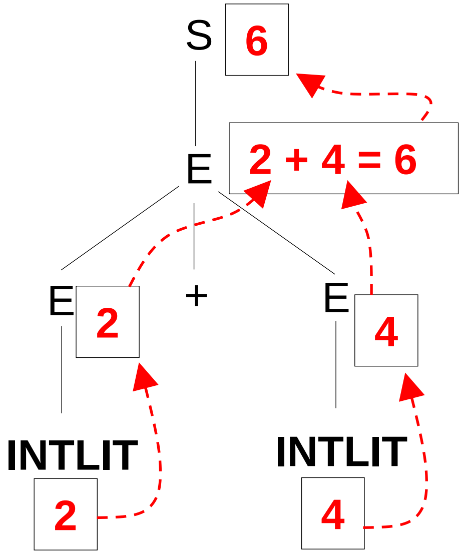
What does this attribute grammar compute?



# Attribute Grammar Example 2

- The **val** attribute holds the subtotal value of the subexpression.
- Nonterminals are indexed in the attribute grammar to distinguish multiple occurrences of the nonterminal in a production.

| **productions** | **semantic rules** |
|---|---|
| S ::= E | S**.val** := E**.val** |
| $E_1$ ::= $E_2$ + $E_3$ | $E_1$**.val** := $E_2$**.val** + $E_3$**.val** |
| $E_1$ ::= $E_2$ * $E_3$ | $E_1$**.val** := $E_2$**.val** * $E_3$**.val** |
| E ::= **INTLIT** | E**.val** := **INTLIT.val** |



Decorated tree for the sentence ``2 + 4''.

# Attribute Flow

- Synthesized attributes: flow from the bottom of the parse tree to the top (see the previous 2 examples).
  → computed from the children in the AST

- Inherited attributes: attributes can also flow into symbols from above or from the side (see the next example).
  → computed from the parent and from siblings

- An attribute flow algorithm propagates attribute values through the parse tree. Attributes must be set before they can be used.

- Attributes can be used to construct an AST from a parse tree!

# Attributes Associated with a Grammar Symbol

- An attribute can represent anything we choose:
  - a string
  - a number
  - a type
  - a memory location
  - a piece of source code
  - an AST
  - aso.
- Each attribute has a name and a type.

# We can use attribute grammars to determine the types and values of variables and expressions

The type of an expression is important

- To enforce type rules of a programming language
  **Example**: a floating-point number cannot be used as an array index.

```
int x[3];   // 3 ints @ x[0], x[1], x[2]
float y;

x[1.2] = 0; // wrong!
x[y] = 0;   // wrong!
```

- For code generation in the backend of the compiler.
  **Example (JVM)**: **`fadd`** vs. **`iadd`**
- Values of expressions can be used for compiler optimizations.

# Attribute Grammar Example 3

- A simplified expression grammar for integer and float division:

```
S  ::=  E
E  ::=  E / E
E  ::=  INTLIT
E  ::=  INTLIT . INTLIT
```

- Grammar is ambiguous
  - but we can use it to specify the static semantics if the AST has been built using an unambiguous grammar (see Lecture on Syntax Analysis).

- Assume that

  "/" is left-associative, and

  mixed expressions are promoted to floating point

  Example: 5 / 2 / 2.0 evaluated to 1.25, not 1.00

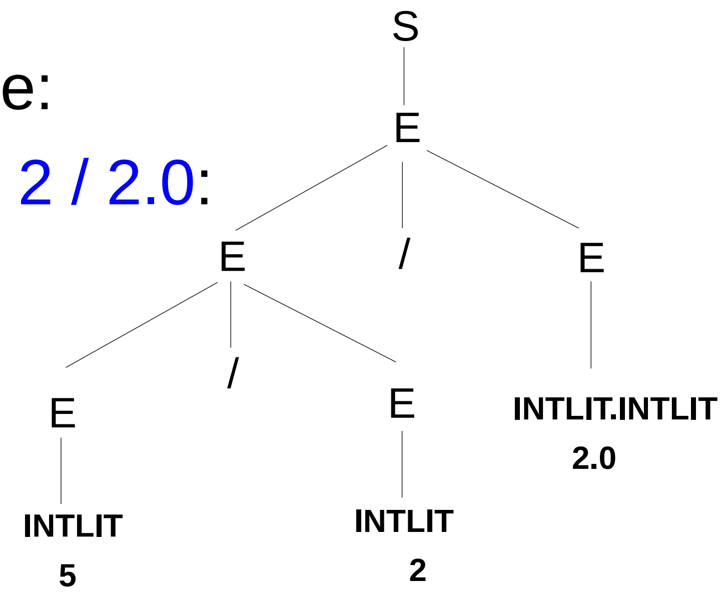
Left associative: a / b / c is interpreted as (a / b) / c.

- On the next slides, we study expression evaluation using an attribute grammar.

# Attribute Grammar Example 3 (cont.)

Running Example:

parse tree for 5 / 2 / 2.0:



- The “/” operator is assumed to be left-associative.

- The tree for right-associative “/” is not considered.

# Attribute Grammar Example 3 (cont.)

| Productions | Semantic rules |
|---|---|
| S ::= E | E.**type** = if E.**isFloat** then float else int<br>S.**val** = E.**val** |
| $E_1$ ::= $E_2$ **/** $E_3$ | $E_1$.**isFloat** = $E_2$.**isFloat** or $E_3$.**isFloat**<br>E2.**type** = E1.**type**<br>E3.**type** = E1.**type**<br>E1.**val** = if (E1.**type** == int)<br>then $E_2$.**val** DIV$_{INT}$ $E_3$.**val**<br>else $E_2$.**val** DIV$_{FLOAT}$ $E_3$.**val** |
| E ::= **INTLIT** | E.**isFloat** = false<br>E.**val** = if (E.**type** == int)<br>then **INTLIT**.**val** else Float(**INTLIT**.**val**) |
| E ::= **INTLIT . INTLIT** | E.**isFloat** = true<br>E.**val** = **INTLIT.INTLIT**.**val** |

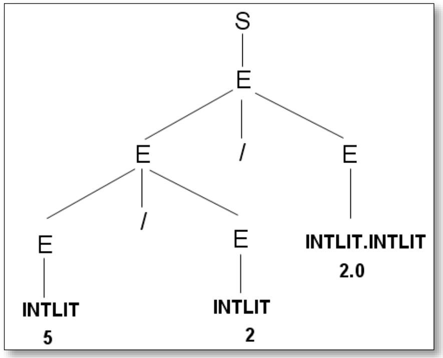
# Attribute Grammar Example 3 (cont.)

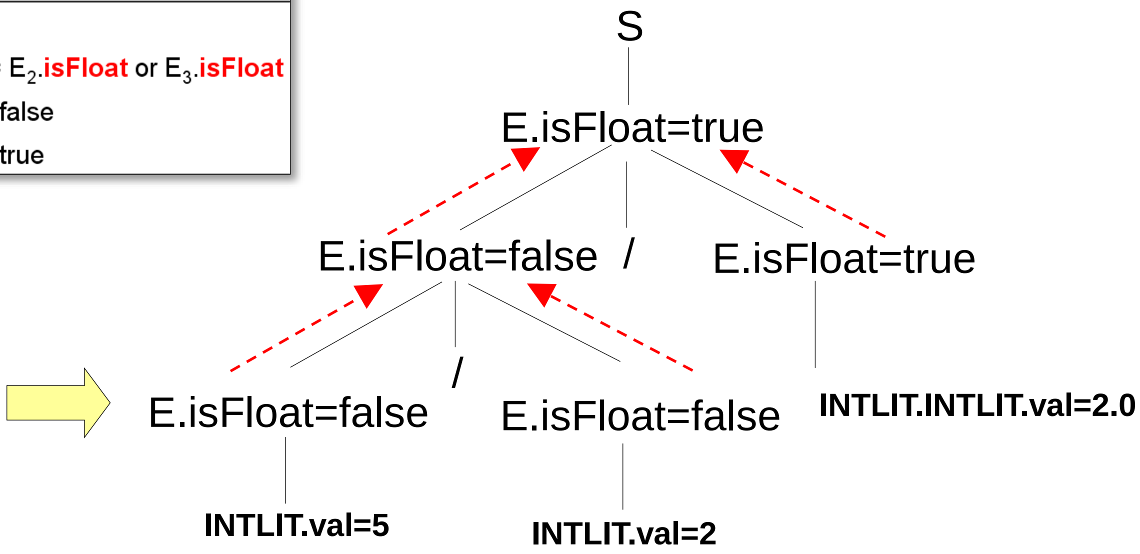
| Productions | Semantic rules |
|---|---|
| S ::= E | |
| $E_1$ ::= $E_2$ **/** $E_3$ | $E_1$.**isFloat** = $E_2$.**isFloat** or $E_3$.**isFloat** |
| E ::= **INTLIT** | E.**isFloat** = false |
| E ::= **INTLIT . INTLIT** | E.**isFloat** = true |

First pass: bottom-up computation of the isFloat attribute.

# Attribute Grammar Example 3 (cont.)

| nr | Productions | Semantic rules |
|---|---|---|
| 1 | S ::= E | |
| 2 | $E_1$ ::= $E_2$ / $E_3$ | $E_1$.isFloat = $E_2$.isFloat or $E_3$.isFloat |
| 3 | E ::= **INTLIT** | E.isFloat = false |
| 4 | E ::= **INTLIT . INTLIT** | E.isFloat = true |





The flow of synthesized attribute isFloat.

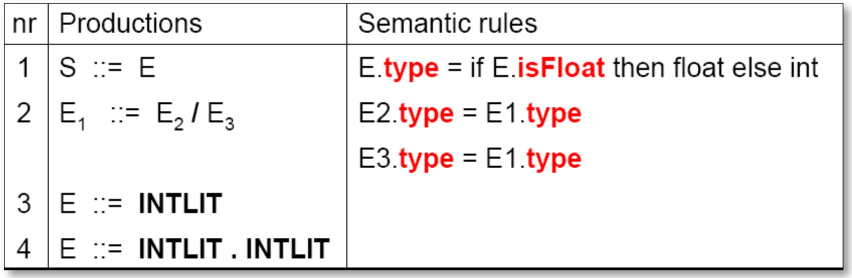
- Synthesized attributes flow bottom-up.

# Attribute Grammar Example 3 (cont.)

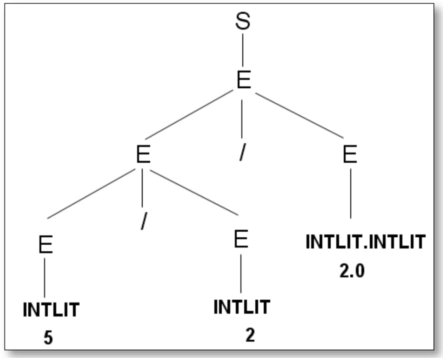
| Productions | Semantic rules |
|---|---|
| S ::= E | E.**type** = if E.**isFloat** then float else int |
| $E_1$ ::= $E_2$ **/** $E_3$ | E2.**type** = E1.**type**<br>E3.**type** = E1.**type** |
| E ::= **INTLIT** | |
| E ::= **INTLIT . INTLIT** | |

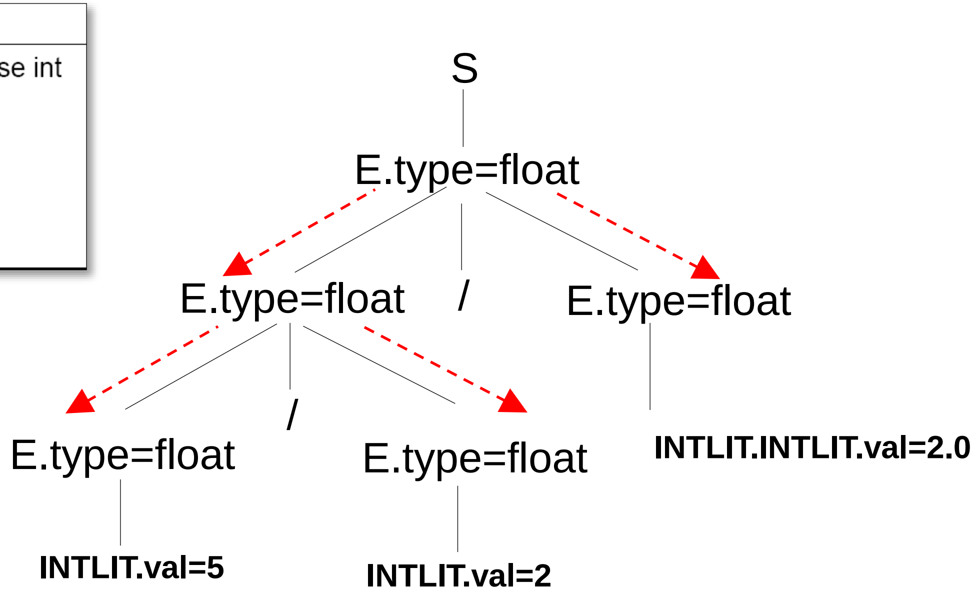
Second pass: top-down computation of the type attribute.

# Attribute Grammar Example 3 (cont.)



| nr | Productions | Semantic rules |
|---|---|---|
| 1 | S ::= E | E.**type** = if E.**isFloat** then float else int |
| 2 | $E_1$ ::= $E_2$ **/** $E_3$ | E2.**type** = E1.**type**<br>E3.**type** = E1.**type** |
| 3 | E ::= **INTLIT** | |
| 4 | E ::= **INTLIT . INTLIT** | |





The flow of inherited attribute type.

- Inherited attributes flow top-down (or left-to-right).

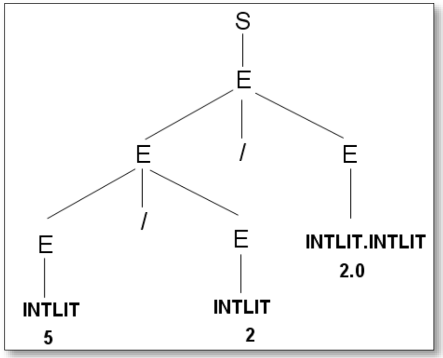
# Attribute Grammar Example 3 (cont.)

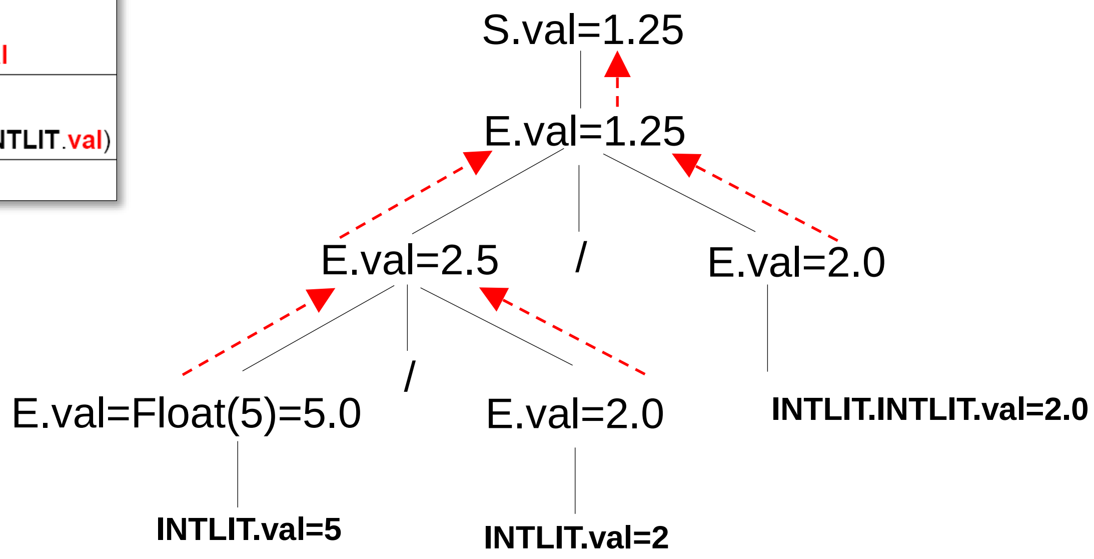
| Productions | Semantic rules |
|---|---|
| S ::= E | S.**val** = E.**val** |
| $E_1$ ::= $E_2$ **/** $E_3$ | E1.**val** = if (E1.**type** == int)<br>then $E_2$.**val** $\text{DIV}_{\textbf{INT}}$ $E_3$.**val**<br>else $E_2$.**val** $\text{DIV}_{\textbf{FLOAT}}$ $E_3$.**val** |
| E ::= **INTLIT** | E.**val** = if (E.**type** == int)<br>then **INTLIT**.**val** else Float(**INTLIT**.**val**) |
| E ::= **INTLIT . INTLIT** | E.**val** = **INTLIT.INTLIT**.**val** |

Third pass: bottom-up computation of the val attribute.

# Attribute Grammar Example 3 (cont.)

| nr | Productions | Semantic rules |
|---|---|---|
| 1 | S ::= E | S.val = E.val |
| 2 | $E_1$ ::= $E_2$ / $E_3$ | E1.val = if (E1.type == int)<br>then $E_2$.val $DIV_{INT}$ $E_3$.val<br>else $E_2$.val $DIV_{FLOAT}$ $E_3$.val |
| 3 | E ::= **INTLIT** | E.val = if (E.type == int)<br>then **INTLIT**.val else Float(**INTLIT**.val) |
| 4 | E ::= **INTLIT . INTLIT** | E.val = **INTLIT.INTLIT**.val |





The flow of synthesized attribute val.

- Synthesized attributes flow bottom-up.

# Attribute Grammar Example 3 (cont.)

- $\text{DIV}_{\text{INT}}$ denotes integer division
- $\text{DIV}_{\text{FLOAT}}$ denotes floating-point division
- Float(): converts an integer to a floating-point value
- INTLIT.val, INTLIT.INTLIT.val:
  - computed by the scanner (before semantic analysis)
  - called an intrinsic synthesized attribute

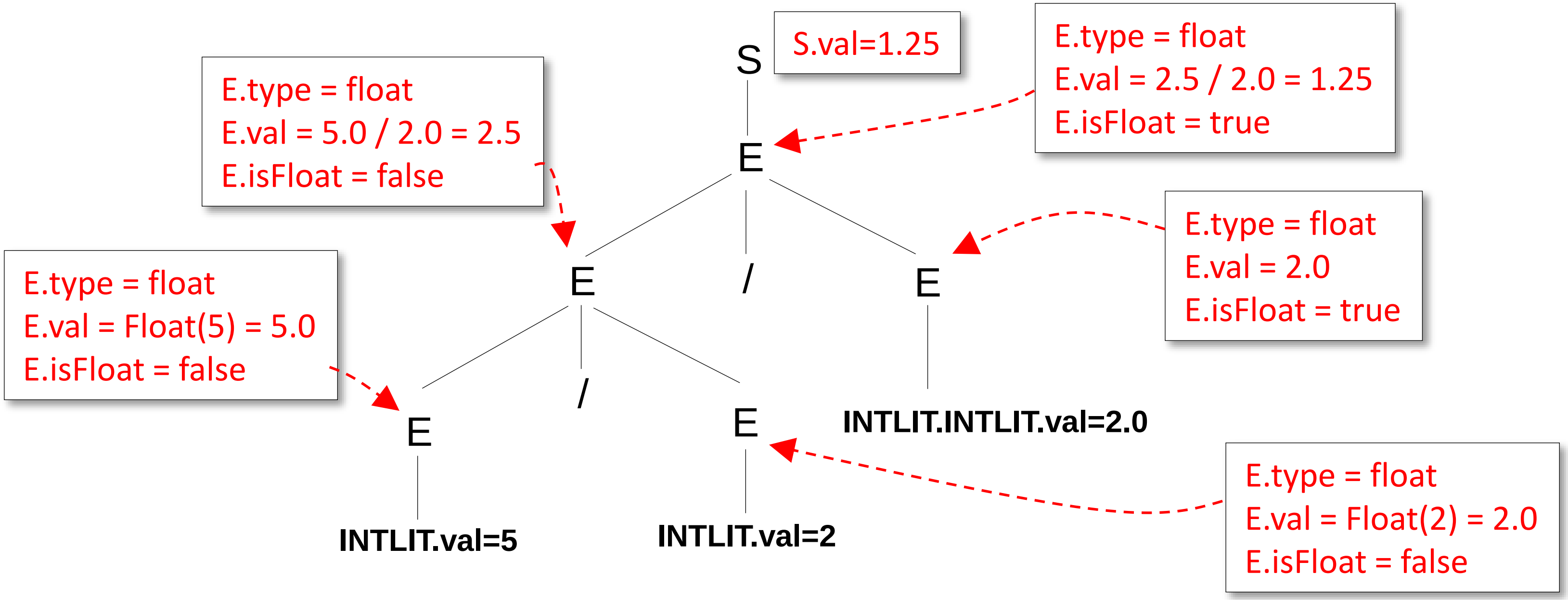
# Attribute Grammar Example 3 (cont.)

- Synthesized attribute `isFloat` over {true, false}
  - indicates if any part of a subexpression has a floating-point value
- Inherited attribute `type` over {int, float}
  - indicates the type of a subexpression
- Synthesized attribute `val`
  - gives the value of a subexpression
- Dependences between attributes:

isFloat → type → val

I.e., val depends on type, which depends on isFloat.

# Attribute Grammar Example 3 (cont.)



Fully decorated parse tree for expression 5 / 2 / 2.0

- All attribute values according to the attribute grammar for Example 3.

# A more "formal" Definition of Synthesized and Inherited Attributes

Synthesized Attributes

- computed from attributes of **children**

Propagated bottom up in the tree.

Inherited Attributes

- computed from attributes of **parent** or **siblings**

Propagated top-down and on same level of the tree.

Example: production X ::= A B C

- X.a is a synthesized attribute, if

  X.a = f (attributes of A, B, and/or C)

- B.a is an inherited attribute, if

  B.a = f (attributes of A, C, and/or X)

f is a semantic function.

An inherited attribute B.a may also depend on other attributes from B itself!

# Attribute Evaluators

Tree Walkers: traverse the parse tree in one or more passes at compile time.

- Capable of evaluating any **non-circular** attribute grammar
- An attribute grammar is **circular** if an attribute depends on itself.
- Circularity can be decided (in exponential time).
- Too complex to be used in practice.

Rule-based methods: the compiler writer analyses the grammar

and fixes an evaluation order at compiler-construction time.

- Still possible to use trees.
- Works for practically all grammars.
- Used with practically all compilers.
- Visitor design pattern.

# A Non-Circular Grammar Evaluator

```
1  while ( attributes remain to be evaluated )  {
2    visitNode (S)     // S is the start symbol of the grammar
3  }

3  void visitNode (AST X)  {
4     if ( X is a non-terminal ) {    // X ::= $X_1$ $X_2$, ..., $X_m$
5        for ( i=1 ; i <= m; i++) {
6           if ($X_i$ is a non-terminal) {
7               evaluate all possible inherited attributes of $X_i$
8               visitNode ($X_i$)
9           }
10       }
11    }
12    Evaluate all possible synthesized attributes of X
13 }
```

- Preorder part (line 7): propagate inherited attributes downwards in the tree.
- Postorder part (line 12): propagate synthesized attributes upwards.

# Rule-Based Methods

- See Lecture Slides on the Visitor design pattern.

# Lecture 3: Semantic Analysis

1. Overview & Purpose ✓
   a) Static semantics ✓
   b) Dynamic semantics ✓

2. Static semantics
   a) Two types of semantic constraints:
      - Scope rules ✓
      - Type rules ✓

   b) Two subphases in static semantic analysis
      - Identification (to enforce scope rules) ✓
      - Type checking (to enforce type rules) ✓

   c) Standard environments ✓

   d) Assignment 2-(2) ✓

3. Attribute Grammars ✓