COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia

`http://www.cse.unsw.edu.au/~cs3131`

`http://www.cse.unsw.edu.au/~cs9102`

Lectures (Schedule)

1. Week 1: Intro, lexical analysis, DFAs and NFAs √
2. Week 2: CFGs + parsing √
3. Week 3: Abstract syntax trees (ASTs) √
4. Week 4: Attribute grammars √
5. Week 5: Static semantics
6. Week 6: Half-term break
7. Week 7: Jasmin
8. Week 8: Code generation
9. Week 9: DFAs + NFAs + Parsing
10. Week 10: Revision

Week 5: Static Semantics

The semantic analyser enforces a language's semantic constraints

1. Two types of semantic constraints:
   - Scope rules
   - Type rules
2. Two subphases in semantic analysis:
   - Identification (symbol table)
   - Type checking
3. Standard environment
4. Assignment 4:
   - The visitor design pattern
   - The two subphases combined in one pass only

This week's lectures + Assn 4 spec $\Rightarrow$ Type Checker

- 1st Lecture: Identification (done for you)
- 2nd Lecture: Type checking (Assignment 4)

Static Semantics

- Is x a variable, method, array, class or package?
- Is x declared before used?
- Which declaration of x does this reference?
- Is an expression type-consistent?
- Does the dimension of an array match with the declaration?
- Is an array reference in bounds?
- Is a method called with the right number and types of args?
- Is break or continue enclosed in a loop construct?
- etc.

These cannot be specified using a CFG

Blocks

- Block: a language construct that can contain declarations:
  - the compilation units (i.e., the code files)
  - procedures/functions (or methods)
  - compound statements
- The two kinds of blocks in VC:
  - The entire program as one block (i.e., the outermost block)
  - compound statements { . . . }
- Block-structured language: permits the nesting of blocks
  - Examples: Ada, Pascal and Modula-2
  - C exhibits nested block structure (because { . . . } can be nested) but are not strictly block-structured languages (because functions cannot be nested inside other functions in the standard C)
- Basic and COBOL: the only block is the entire program
- Fortran: the entire program plus blocks contained in the program

Scope

- The scope of a declaration is the portion of the program over which the declaration takes effect
- A declaration is in scope at a particular point in a program if and only if the declaration's scope includes that point
- Scope rules: the rules governing declarations (called defined occurrences of identifiers) and references to declarations (called applied occurrences of identifiers)

The scope rules provide the answer to: what is the declaration for this variable referenced in the program?

Scope Rules in VC

1. The scope of a function: from the point at which it is declared to the end of the file
2. The scope of a variable in a block: from the point at which it is declared to the end of the block
3. The scope of a formal parameter: the same as the local variables in the function body

```
void f(int i, int j) {        void f( ) {
int k;                  ===>    int i;
                                int j;
...                             int k;
```

   (True in almost all languages such as C, C++ and Java.)

4. The scope of a built-in function: the entire program

Scope Rules in VC (Cont'd)

5. No identifier can be declared more than once in a block
6. Most closed nested rule: For every applied occurrence (i.e., use) of an identifier I in a block B, there must be a corresponding declaration, which is in the smallest enclosing block that contains any declaration of I.
7. Due to Rule 6, the scope of a declaration defined in each of the first four rules excludes the scope of another declaration using the same name (inside an inner block).
   - Such a gap is known as a scope hole.
   - The inner declaration is said to hide the outer declaration
   - The outer declaration is not visible in the inner declaration

Implication of Rule 1 in VC

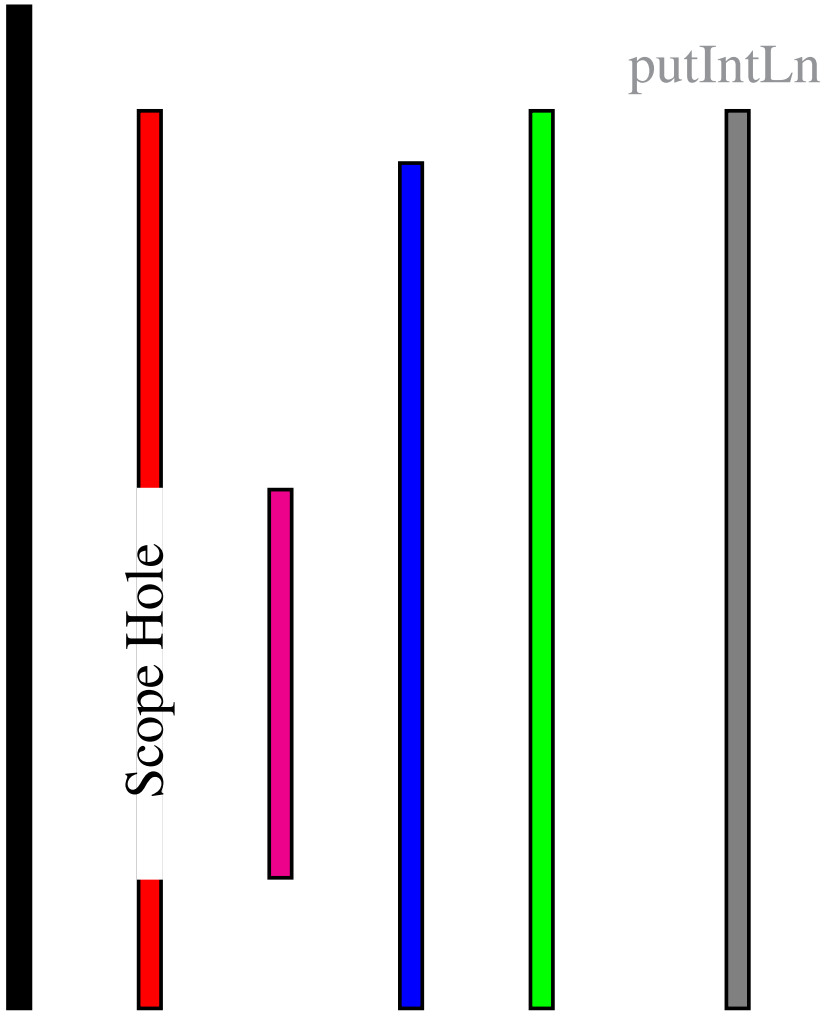
- A semantically illegal VC program:

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

- This allows identification and checking to be done in one-pass
- Pascal solves the problem by allowing forward references
- ANSI C and C++ solve the problem by allowing function prototypes

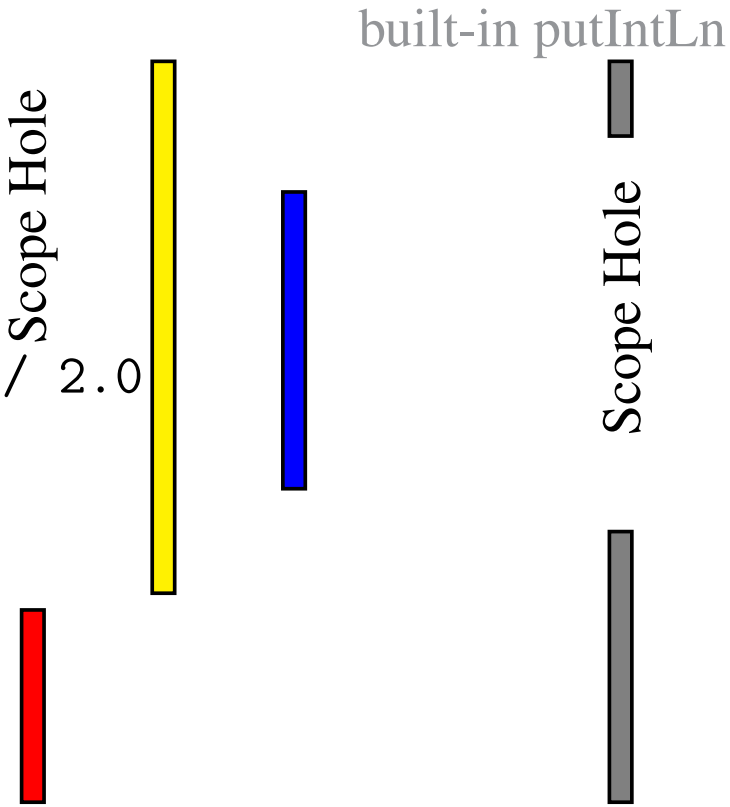
Example 1: Scope Rules

```
int k;
void foo() {
  int i;
  int j;
  i = 1;
  j = 7;
  putIntLn(i);    // 1
  putIntLn(j);    // 7
  {
    int i;
    i = 2;
    putIntLn(i); // 2
    putIntLn(j); // 7
  }
  putIntLn(i);    // 1
  putIntLn(j);    // 7
}
```



Example 2: Scope Rules

```
int foo(int foo) {
  putIntLn(foo);    // 1
  {
    int putIntLn;
    putIntLn = 2;
    putFloatLn(putIntLn); // 2.0
  }
  putIntLn(foo);    // 1
}
void main() {
  foo(1);
}
```



Bad programming style but compiles!

Scope Levels in Block-Structured Languages

- Scope levels in general:
  1. The declarations in the outermost block are in level 1
  2. Increment the scope level by 1 every time when we move from an enclosing to an enclosed block
  3. Typically, the pre-defined functions, variables and constants are in level 0 or 1 or the innermost level (the last is uncommon)
- Scope levels in VC:
  1. All function and global variable declarations are in level 1
  2. Rule 2 as above
  3. All built-in functions are in level 1 $\Rightarrow$ They cannot be redeclared as user-functions or global variables (Rule 6)

Example 1: Scope Levels

```
int i;
void foo() {
  int i;
  int j;
  i = 1;
  j = 7;
  putIntLn(i);
  putIntLn(j);
  {
    int i;
    i = 2;
    putIntLn(i);
    putIntLn(j);
  }
  putIntLn(i);
  putIntLn(j);
}
```

Identifier	Level
Built-in putIntLn	1
foo	1
i	1
i	2
j	2
i	3

Example 2: Scope Rules

```
int foo(int foo) {
  putIntLn(foo);
  {
    int putIntLn;
    putIntLn = 2;
    putFloatLn(putIntLn);
  }
  putIntLn(foo);
}
void main() {
  foo(1);
}
```

Identifier	Level
Built-in putIntLn	1
Built-in putFloatLn	1
foo	1
foo	2
User-defined putIntLn	3

Lecture 8: Static Semantics

The semantic analyser enforces a language's semantic constraints

1. Two types of semantic constraints:
   - Scope rules $\surd$
   - Type rules
2. Two subphases in semantic analysis:
   - Identification (symbol table)
   - Type checking
3. Standard environment
4. Assignment 4:
   - The visitor design pattern
   - The two subphases combined in one pass only

This week's lectures + Week 8 Tutorial + Ass 4 spec $\Rightarrow$ Type Checker

Static Semantics: Identification

- Identification: Relate each applied occurrence of an identifier to its declaration and report an error if no such a declaration exists
- Symbol Table: Associates identifiers with their attributes
- The attributes of an identifier: a variable or a function; the type in the former and the function's result type and the types of a function's formal parameters in the latter
- Two approaches to representing the attributes in the table:
  - The information distilled from the declaration and typically stored in the symbol table or
  - A pointer to the declaration itself (used in Assignment 4)

The Inherited Attributed Decl Used in VC.ASTs.Ident.java for Decorating ASTs in Identification

```
package VC.ASTs;
import VC.Scanner.SourcePosition;

public class Ident extends Terminal {
  public AST decl;
  public Ident(String value , SourcePosition position) {
    super (value, position);
    decl = null;
  }
  public Object visit(Visitor v, Object o) {
    return v.visitIdent(this, o);
  }
}
```

Symbol Table Implementation in VC (Two Classes)

- IdEntry: each IdEntry object has three instance fields:

```
id:    the lexeme
level: the scope level of id
attr:  ptr to the corresponding declaration in the AST
```

- SymbolTable – one table for all scopes!

```
constructor: creates a new table; set scope level to 1
             called at the start of semantic analysis
insert:      insert a new id entry into the table
             called at each declaration
retrieve:    retrieves the entry for an id
             called at each applied occurrence of an id

retrieveOneLevel: retrieve the entry for an id from the current scope

openScope:   increment the scope level by 1
             called at the start of a block
closeScope:  delete all entries in the current level
             called at the end of a block
```

- See VC.Checker.IdEntry.java and VC.Checker.SymbolTable.java

Two Tasks in Identification

- Processing declarations:
  - Call openScope at the start of a block
  - Call closeScope at the end of a block
  - Call insert to enter an id along its scope level and a pointer to the corresponding declaration into the symbol table
- Processing applied occurrences – decorating Ident nodes
  - Call retrieve to link the field Decl in an Ident node (an inherited attribute) to its corresponding declaration
  - Decl = null if no corresponding declaration found – The fact to be used by you to report errors

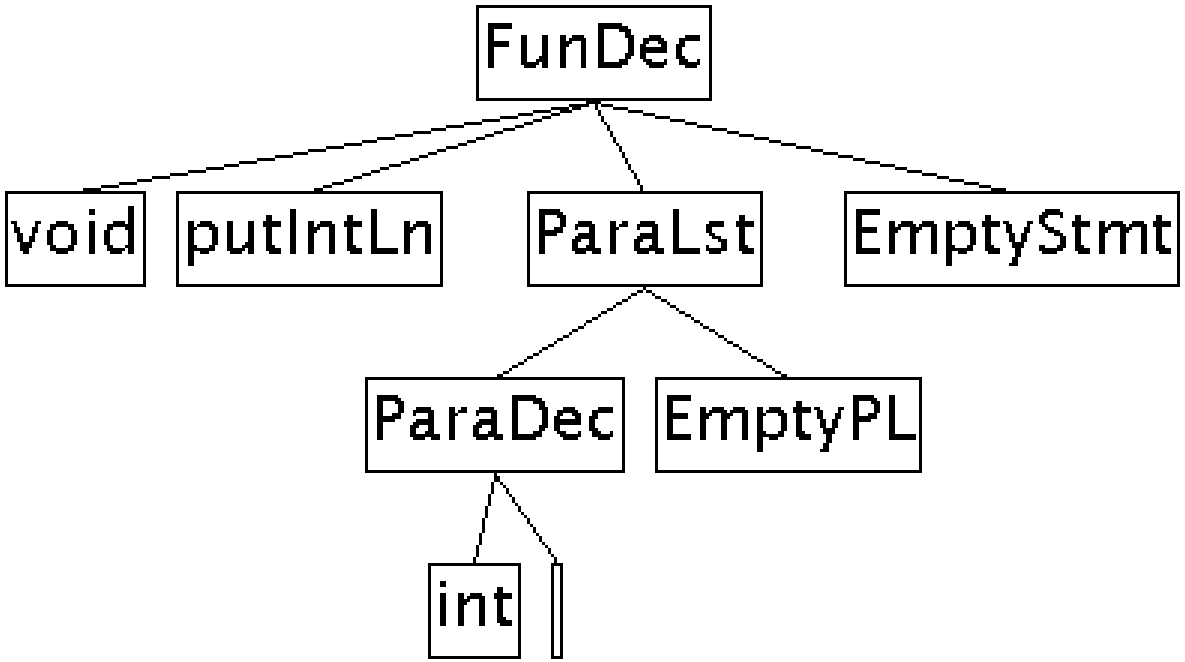
Standard Environment

- Most languages contain a collection of predefined variables, types, functions and constants
  - Java: java.lang
  - Haskell: the standard prelude
  - VC: 11 built-in functions and a few primitive types
- At the start of identification, the symbol table contains 11 small ASTs for the nine built-in functions

Identifier	Level	Attr
getInt	1	ptr to the getInt AST
putInt	1	ptr to the putInt AST
putIntLn	1	ptr to the putIntLn AST
the entries for the other 5 built-in function		
putLn	1	ptr to the putLn AST

Standard Environment (ASTs for Built-in Functions)

- The AST for putIntLn



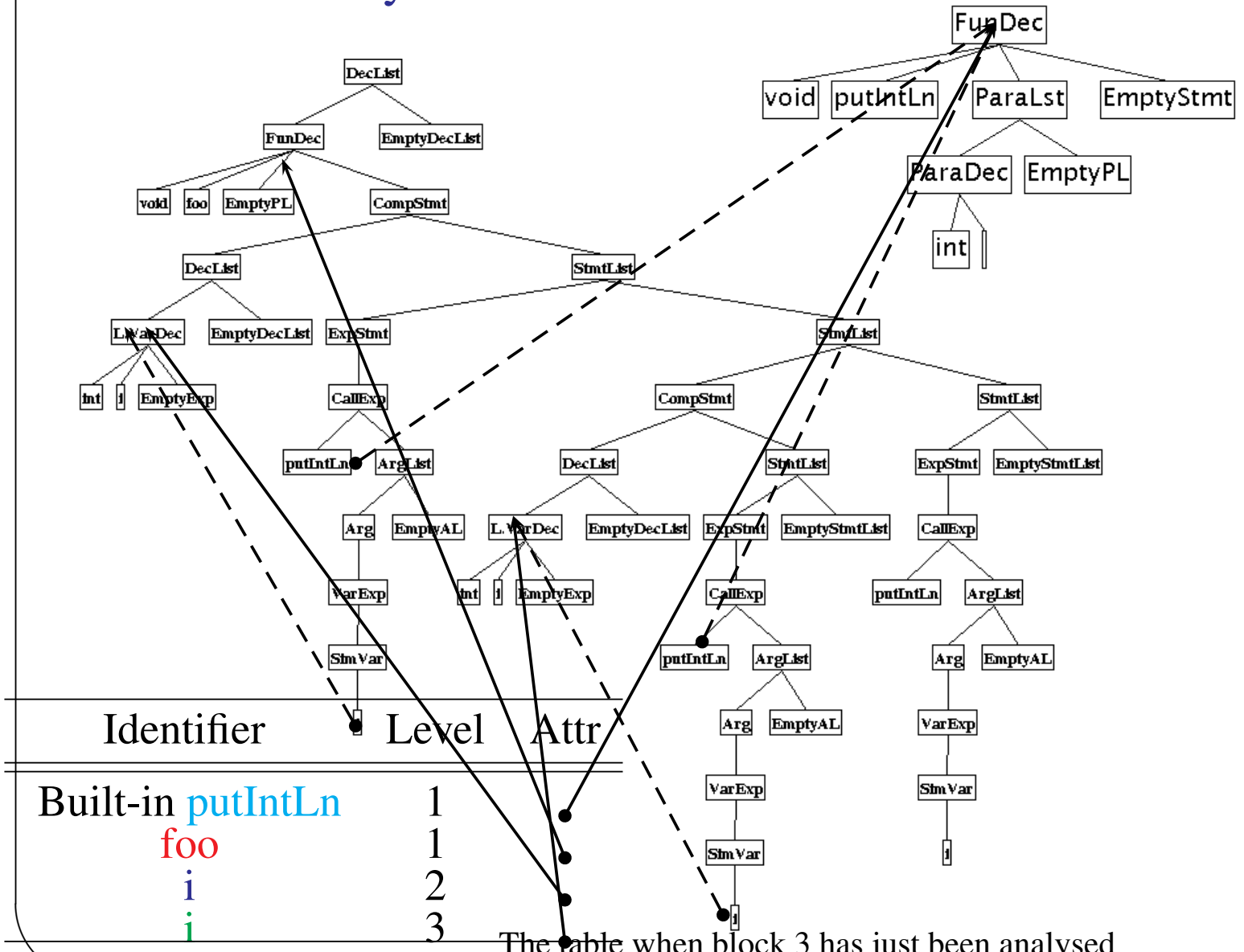
- The name of the formal parameter is set to ""
- Initialised by `establishStdEnvironment()` in `Checker`

Example 3

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

- The ASTs for Examples 1 and 2 are too big to be used.
- Exercises: Print and decorate the ASTs for Examples 1 and 2

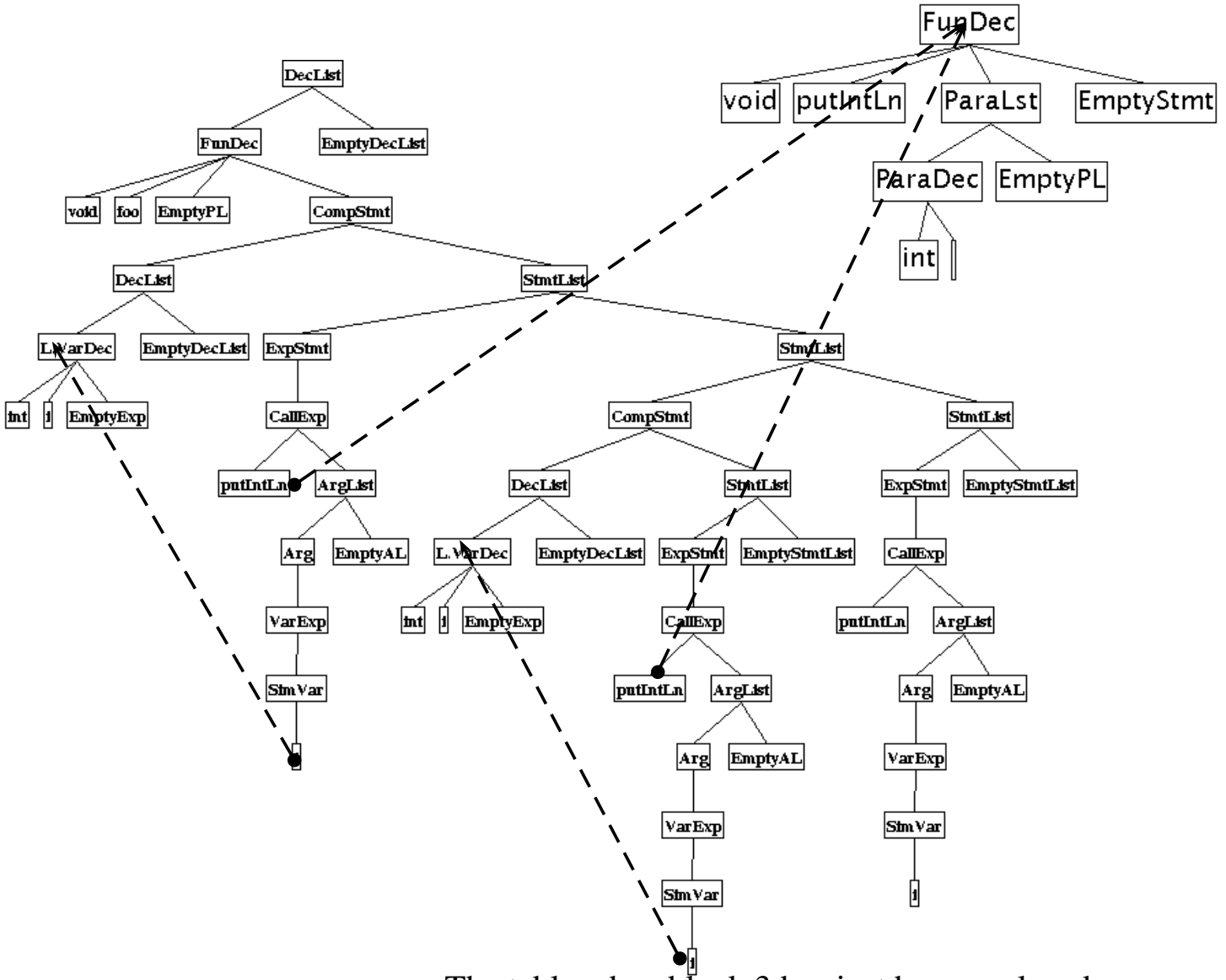
Symbol Table and Decorated AST



Identifier	Level	Attr
Built-in putIntLn	1	
foo	1	
i	2	
i	3	

The table when block 3 has just been analysed

The Decorated AST (After the Symbol Table Dumped)



The table when block 3 has just been analysed

Symbol Table Implementations

- In VC: one symbol table as a stack for all scopes
- In industry-strength compilers:
  - A new symbol table for each scope
  - Link the tables from inner to outer scopes
  - More efficient data structures for tables are used:
    - Hash tables
    - Binary search trees
- Need to handle languages that import and export scopes

Lecture 8: Static Semantics

The semantic analyser enforces a language's semantic constraints

1. Two types of semantic constraints:
   - Scope rules $\surd$
   - Type rules
2. Two subphases in semantic analysis:
   - Identification (symbol table) $\surd$
   - Type checking
3. Standard environment $\surd$
4. Assignment 4:
   - The visitor design pattern
   - The two subphases combined in one pass only

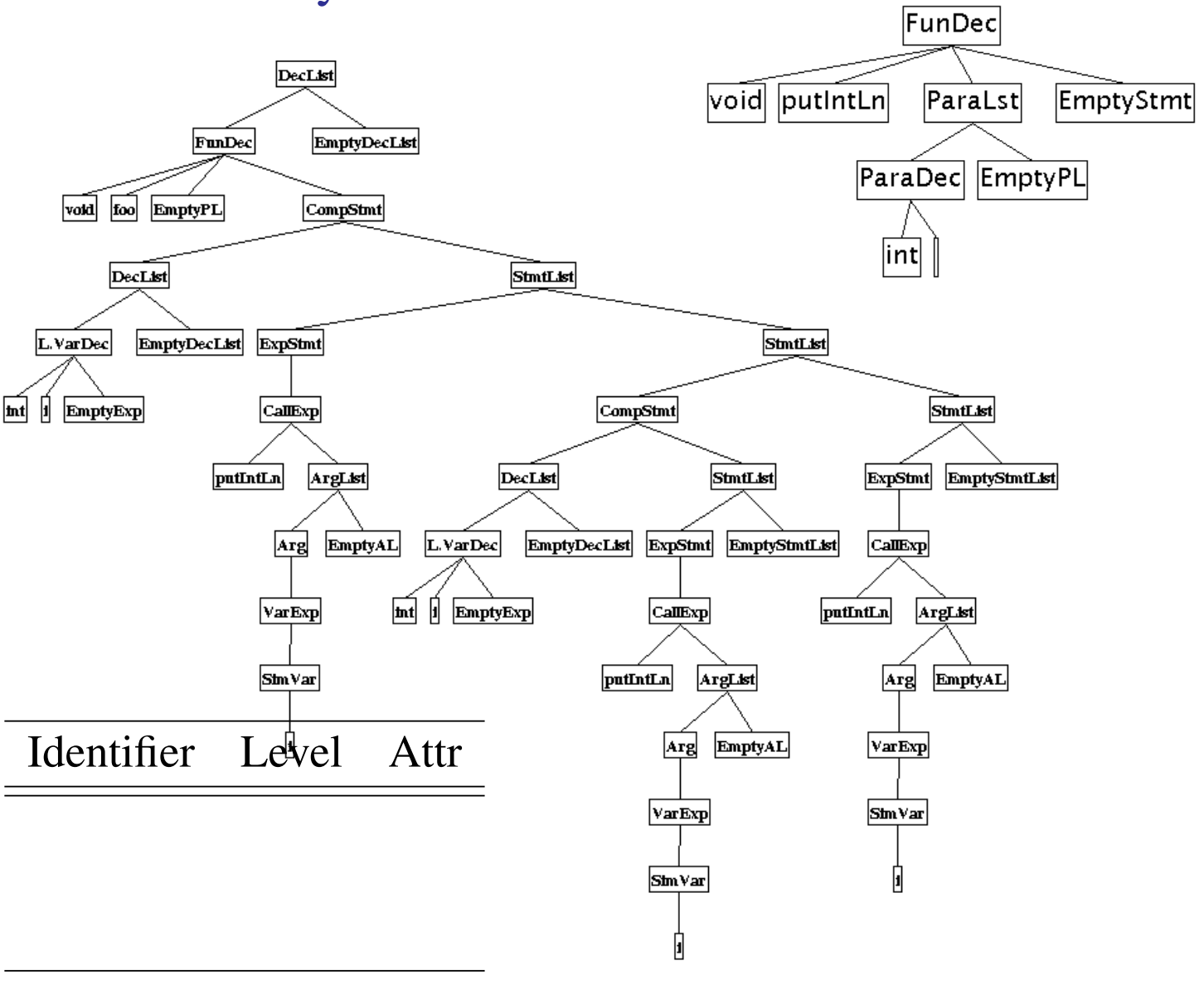
This week's lectures + Tutorial 9 + Ass 4 spec $\Rightarrow$ Type Checker

Reading

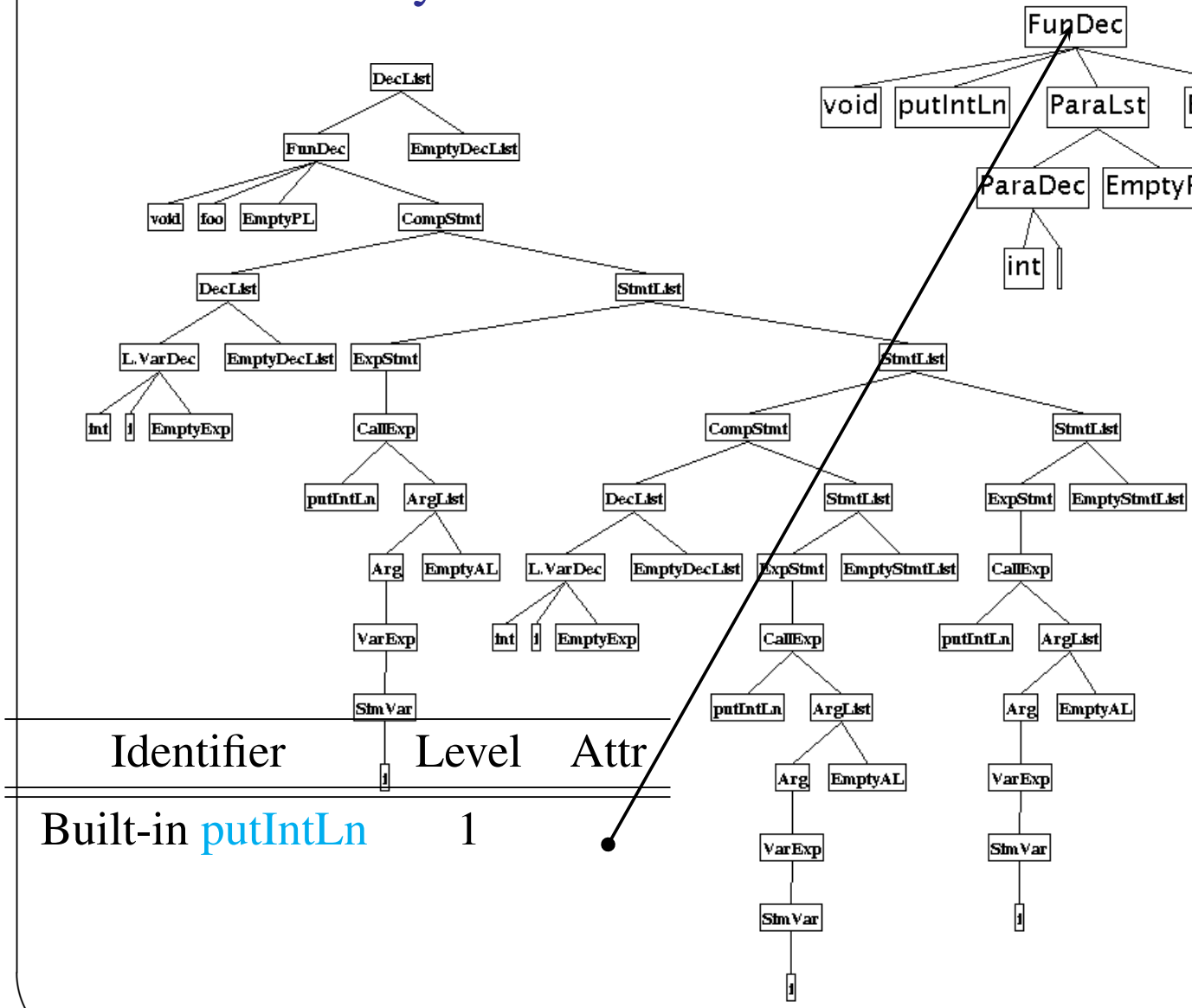
- Chapter 6 (Red Dragon) or Section 6.5 (Purple Dragon)
- TreeDrawer, TreePrinter and Unparser to understand the visitor design pattern
- The on-line VC language definition

Next class: Static Semantics (Type Checking)

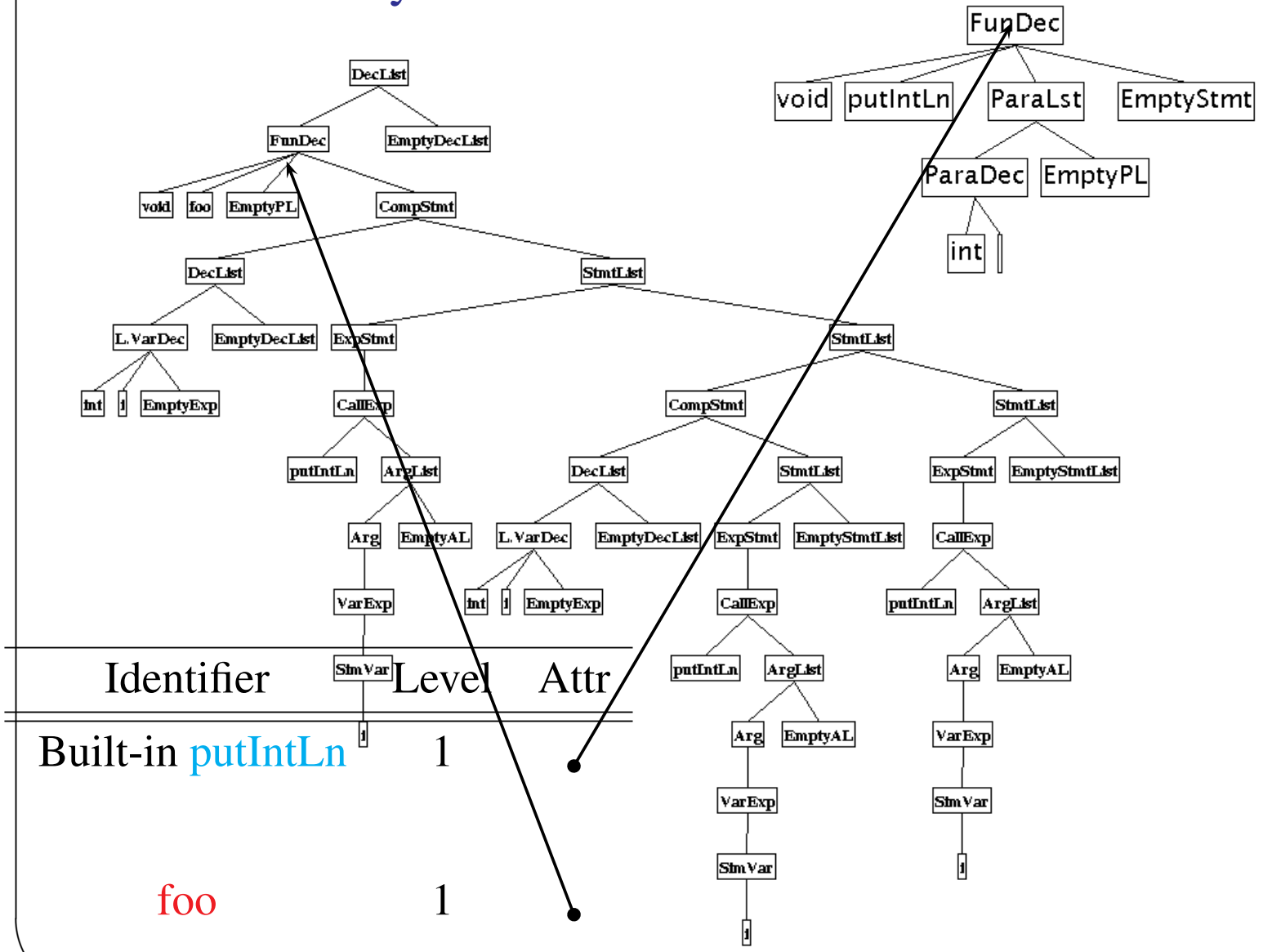
Symbol Table and Decorated AST



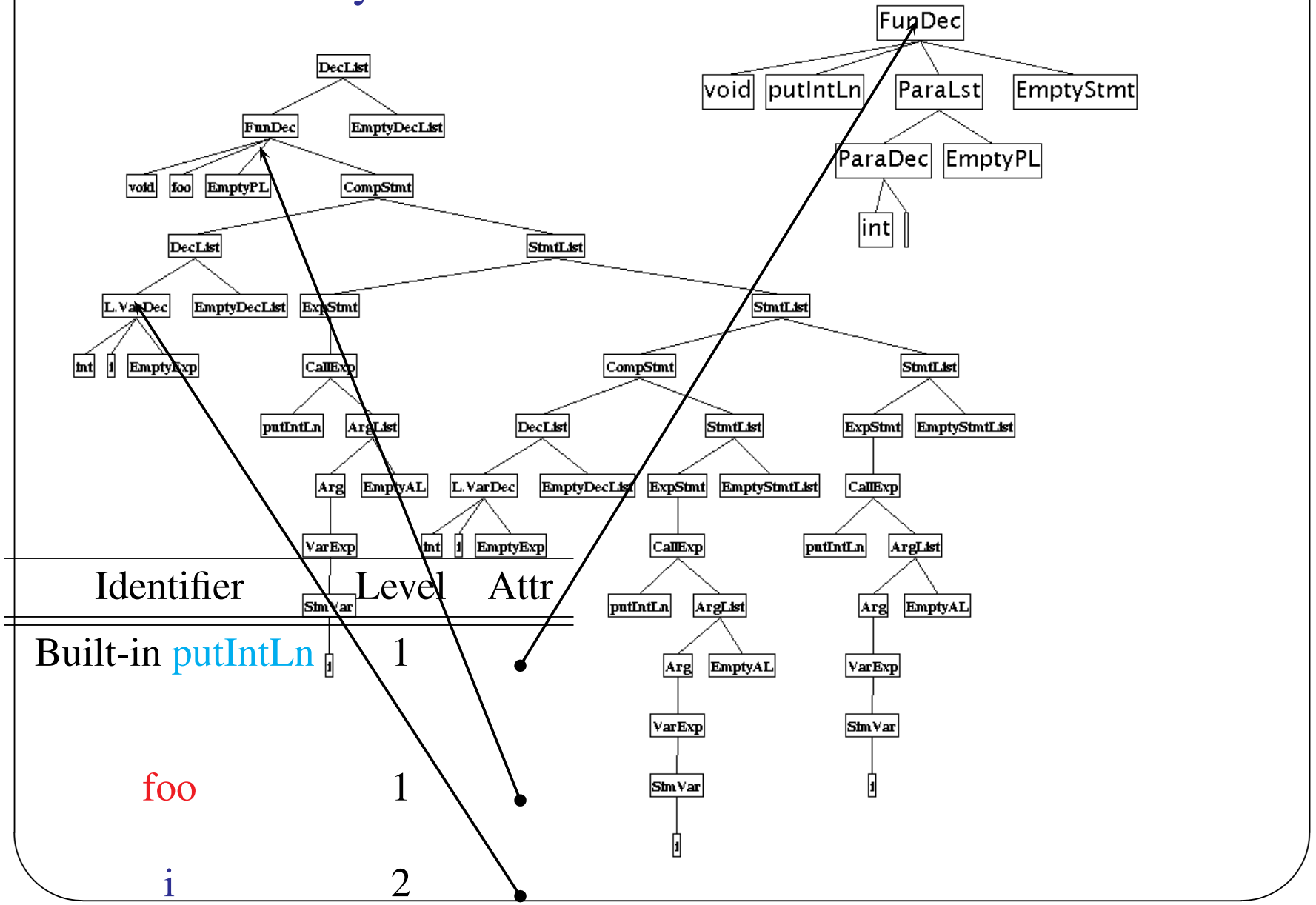
Symbol Table and Decorated AST



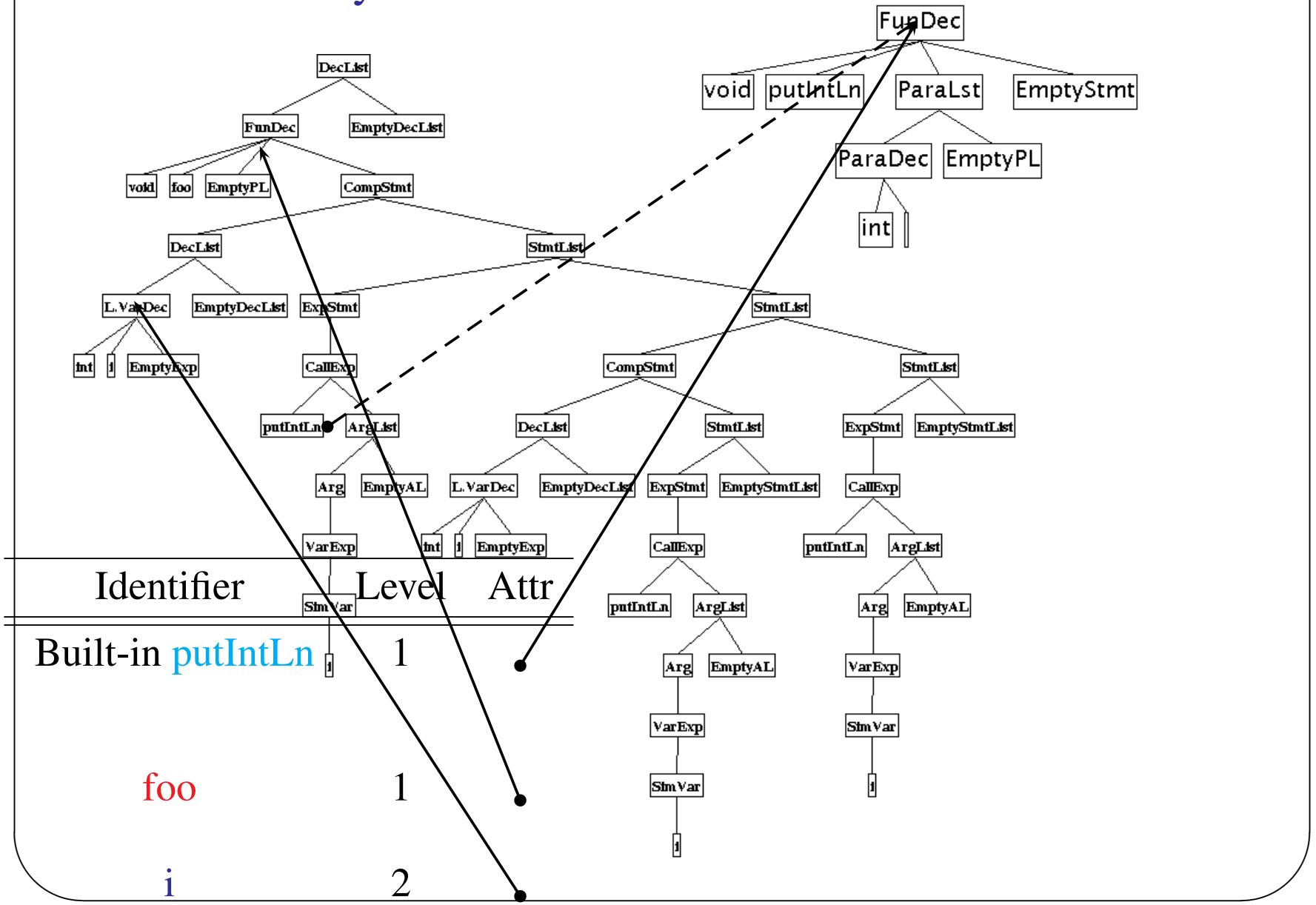
Symbol Table and Decorated AST



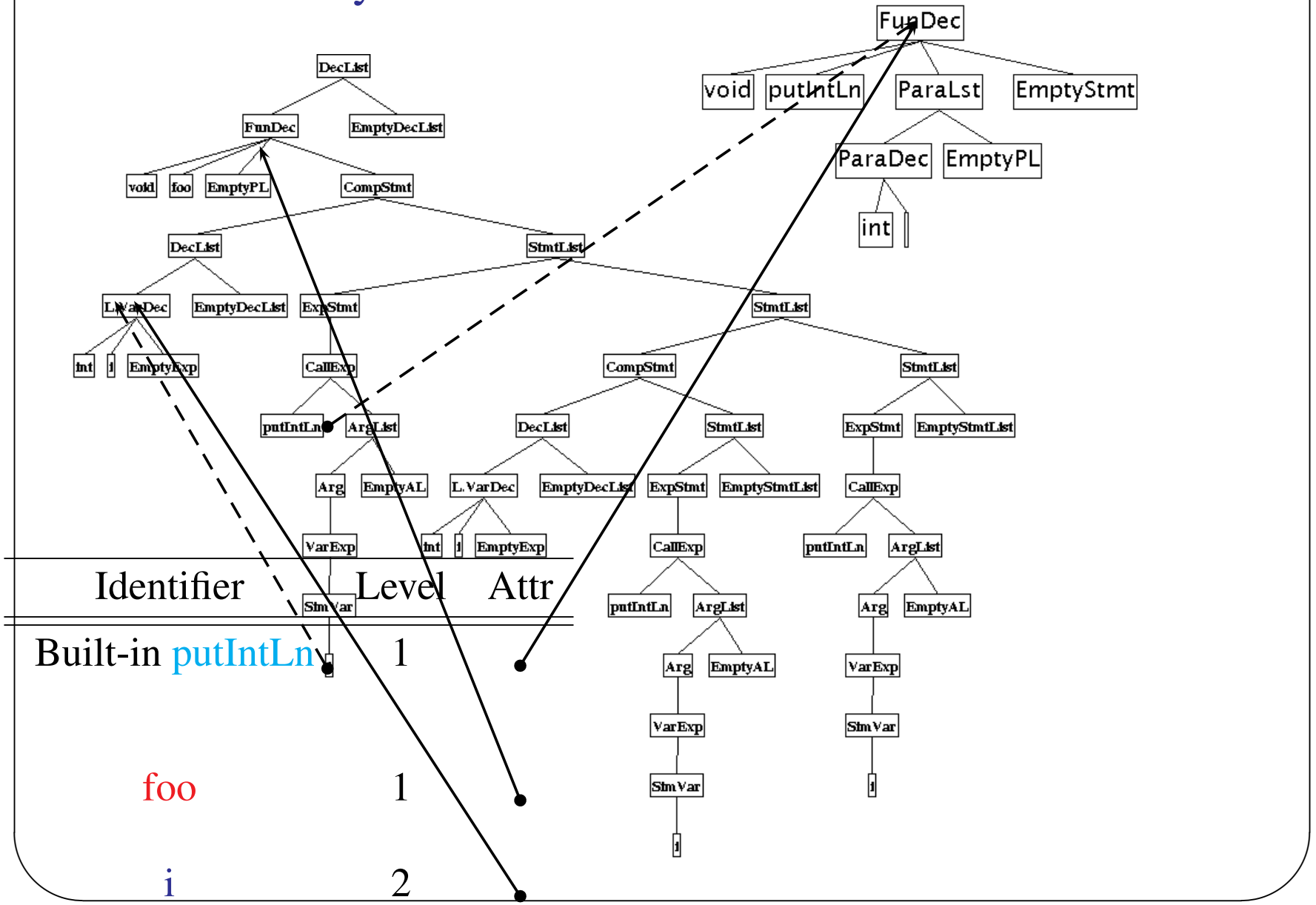
Symbol Table and Decorated AST



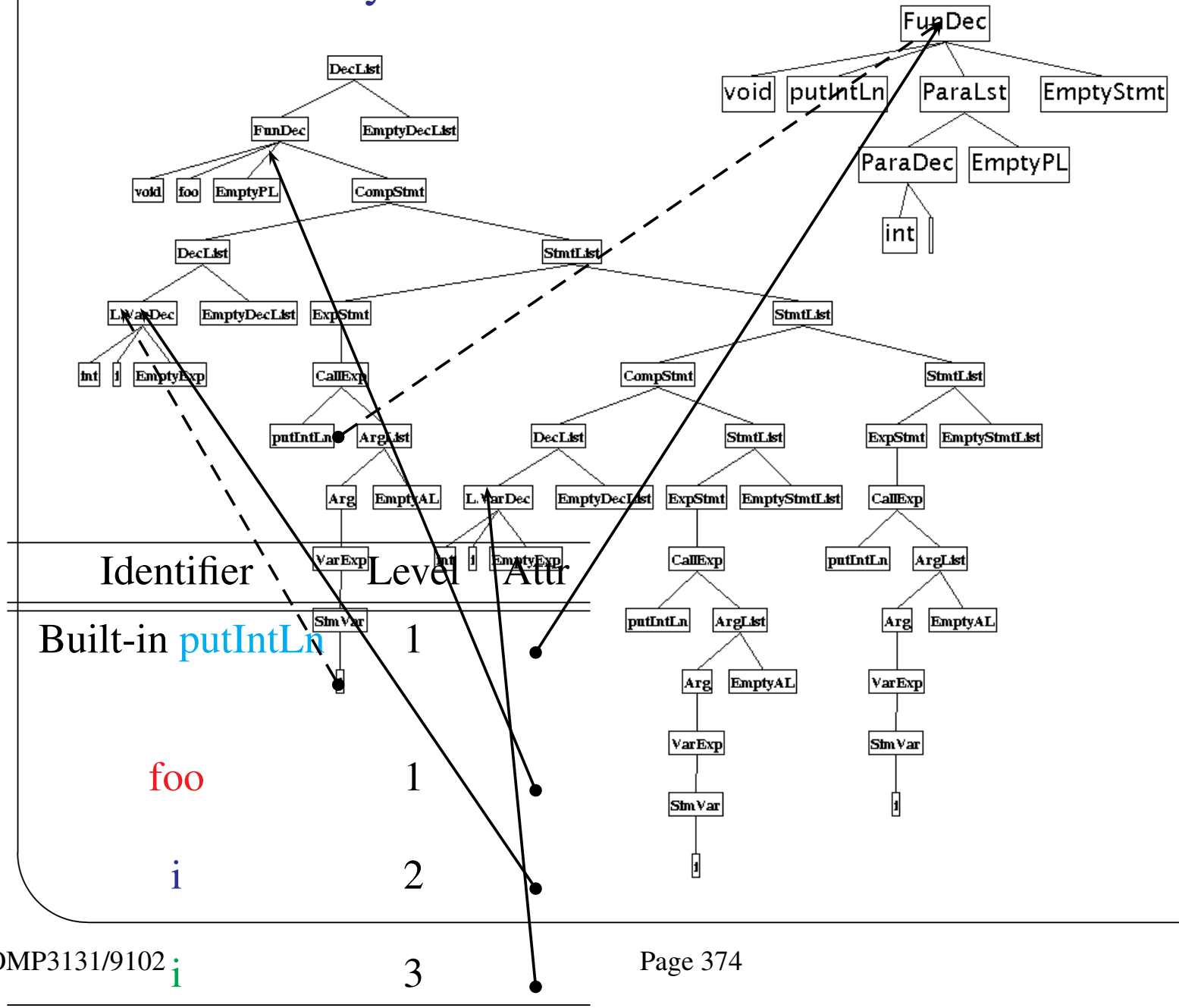
Symbol Table and Decorated AST



Symbol Table and Decorated AST

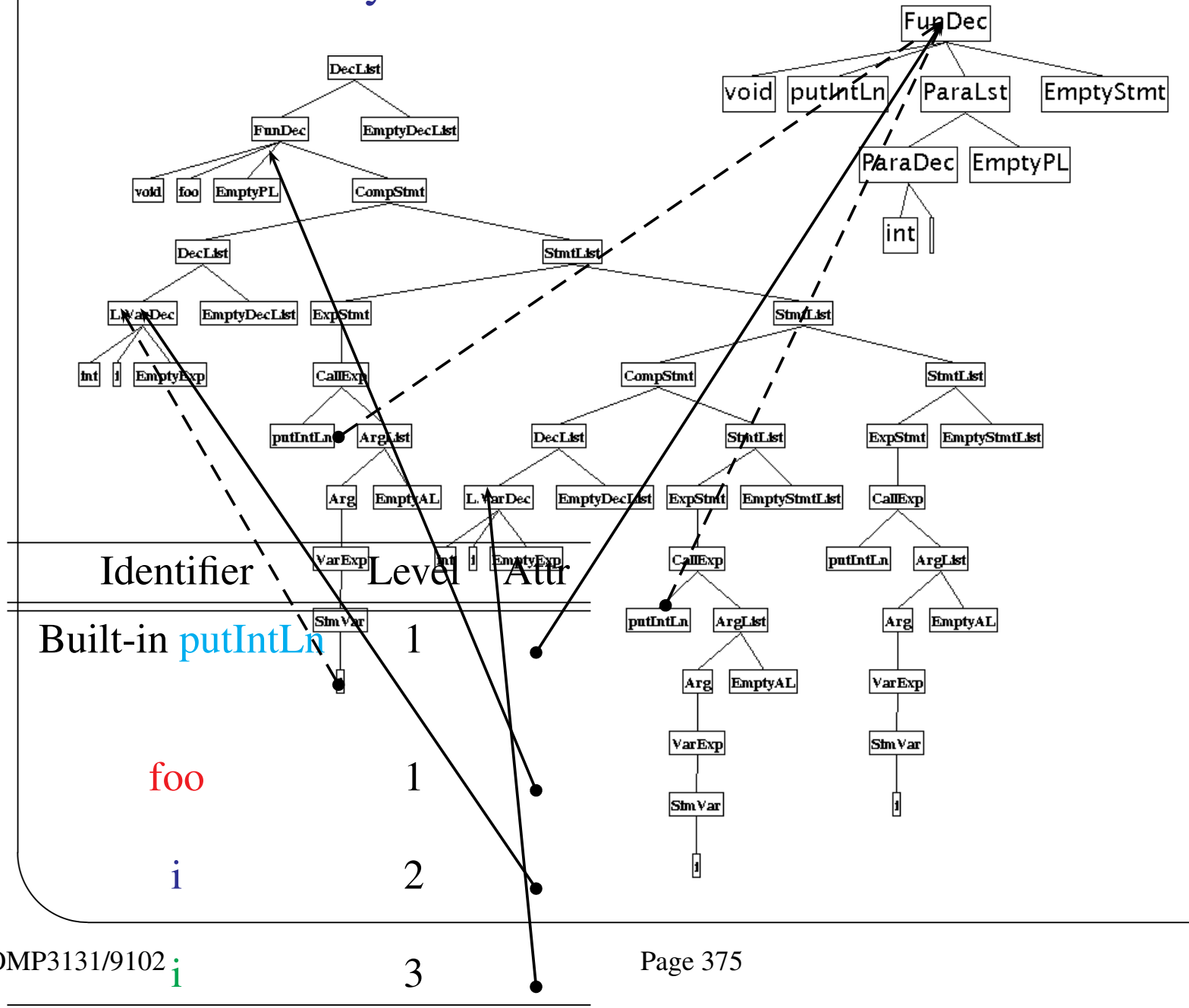


Symbol Table and Decorated AST



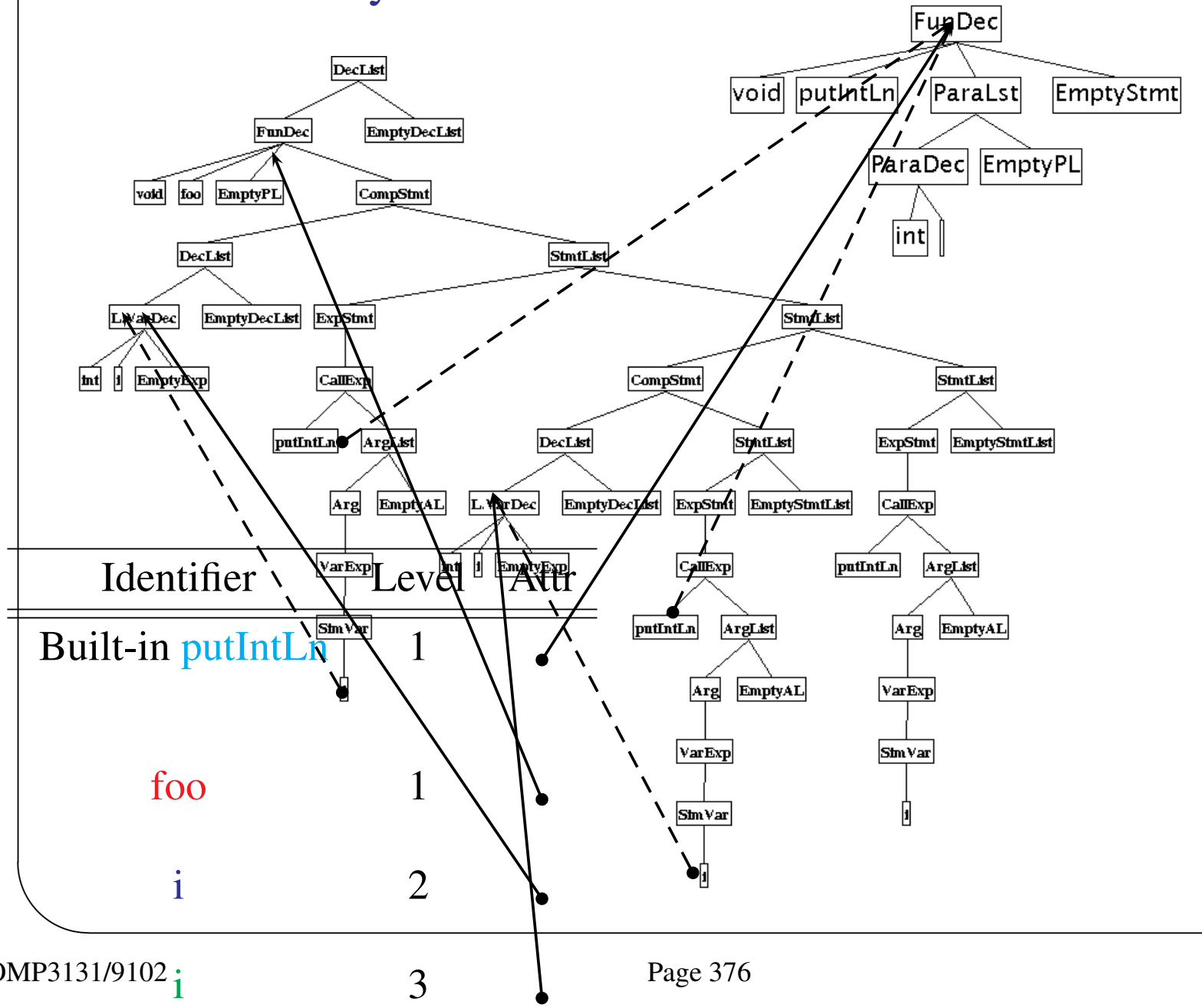
Identifier	Level	Attr
Built-in putIntLn	1	
foo	1	
i	2	
i	3	

Symbol Table and Decorated AST



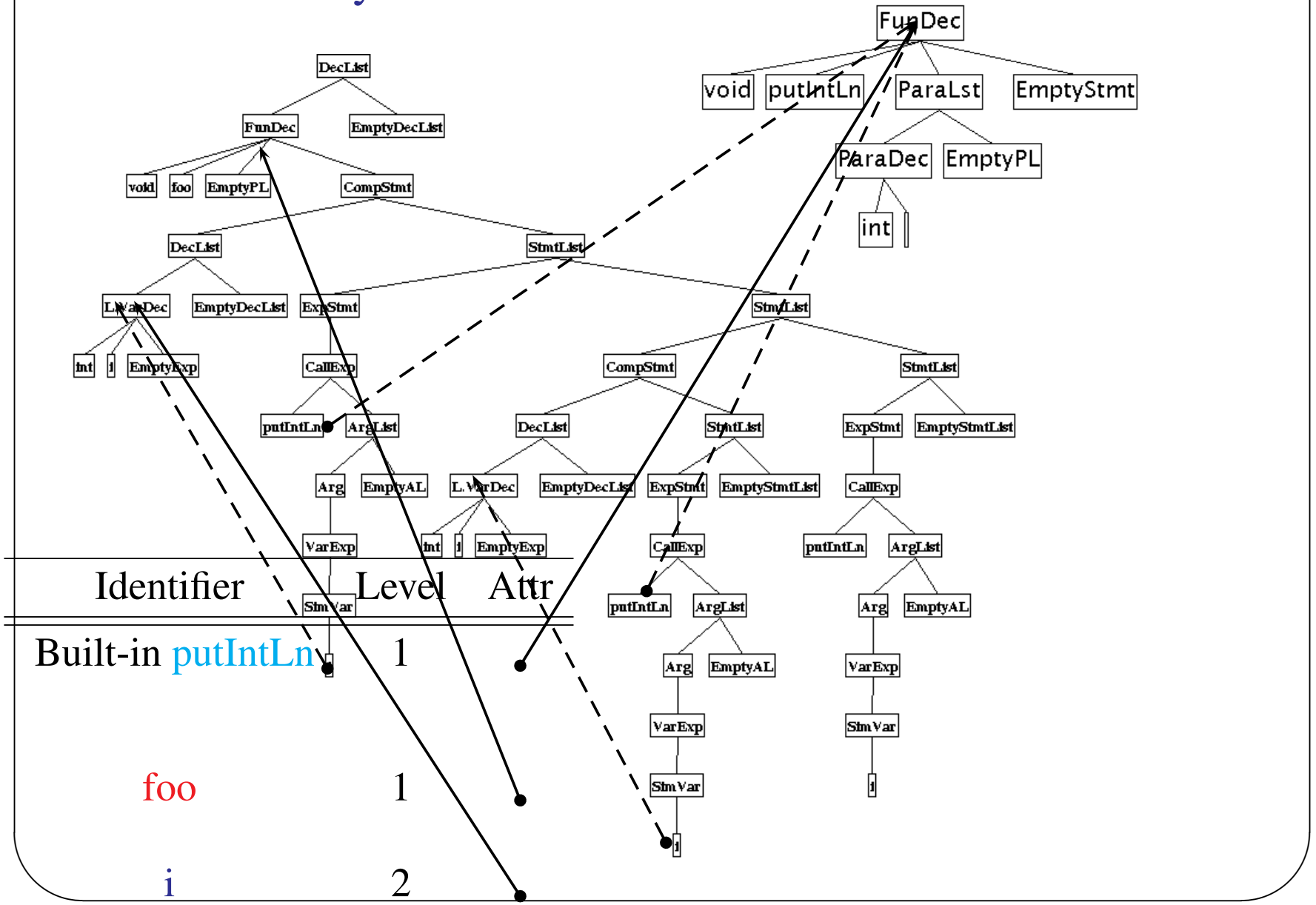
Identifier	Level	Attr
Built-in putIntLn	1	
foo	1	
i	2	
i	3	

Symbol Table and Decorated AST

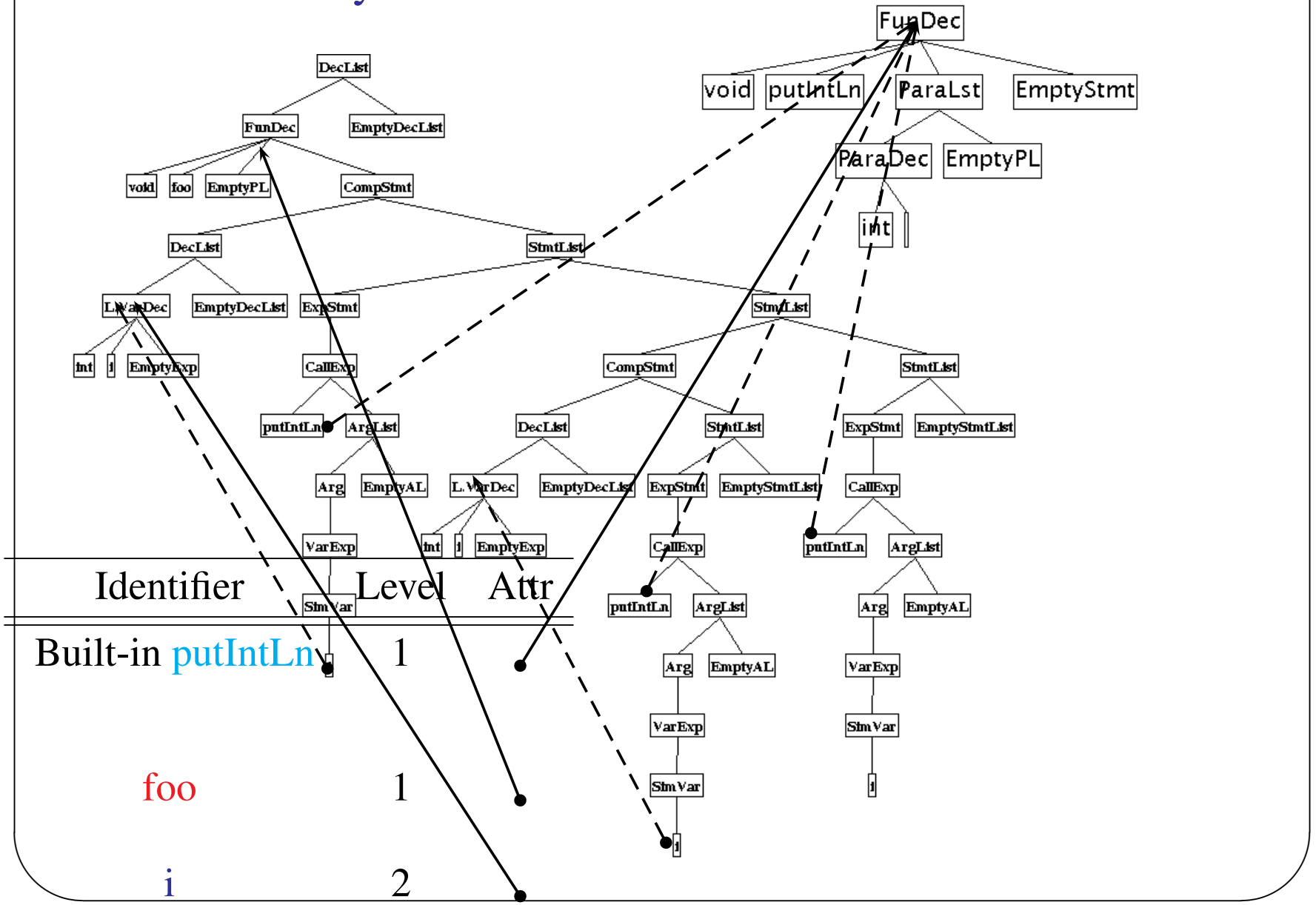


Identifier	Level	Attr
Built-in putIntLn	1	
foo	1	
i	2	
i	3	

Symbol Table and Decorated AST



Symbol Table and Decorated AST



Symbol Table and Decorated AST