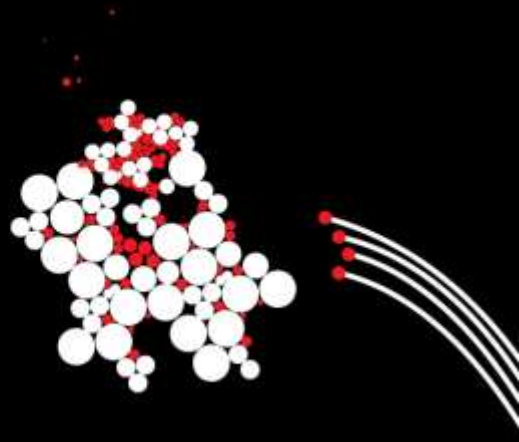
UNIVERSITY OF TWENTE.

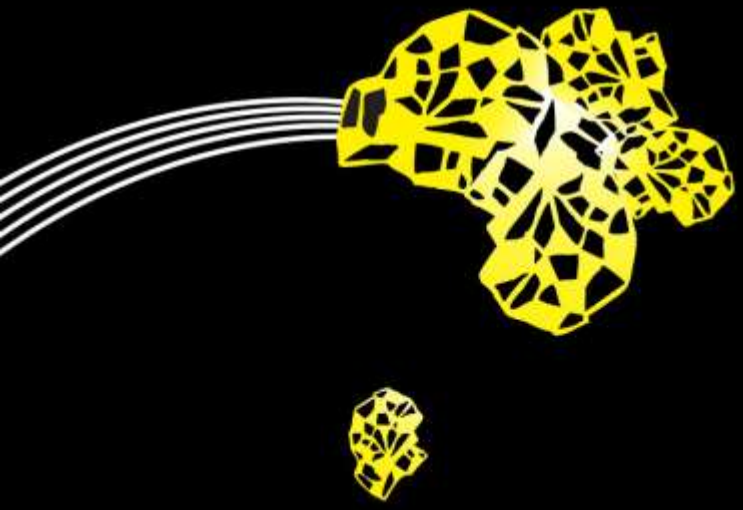


COMPILER CONSTRUCTION:

CC 3-3: SYMBOL TABLES

MODULE 8: PROGRAMMING PARADIGMS

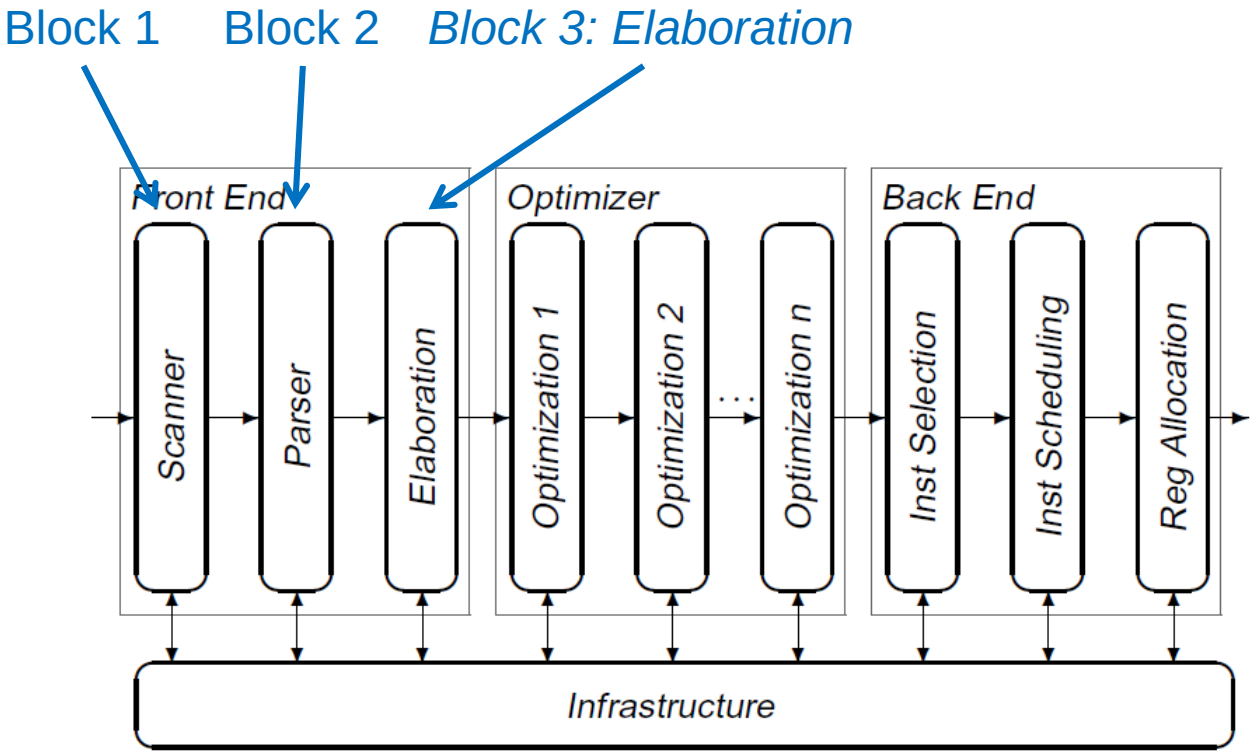
9 MAY 2019



CC TAKE-HOME 1 INCOMING

- Expected release: tonight
- Deadline: Friday, May 24, 23:59 CEST
  - two weeks from today
- Remember: this is an individual test!
  - Plagiarists hunted down and removed from course…

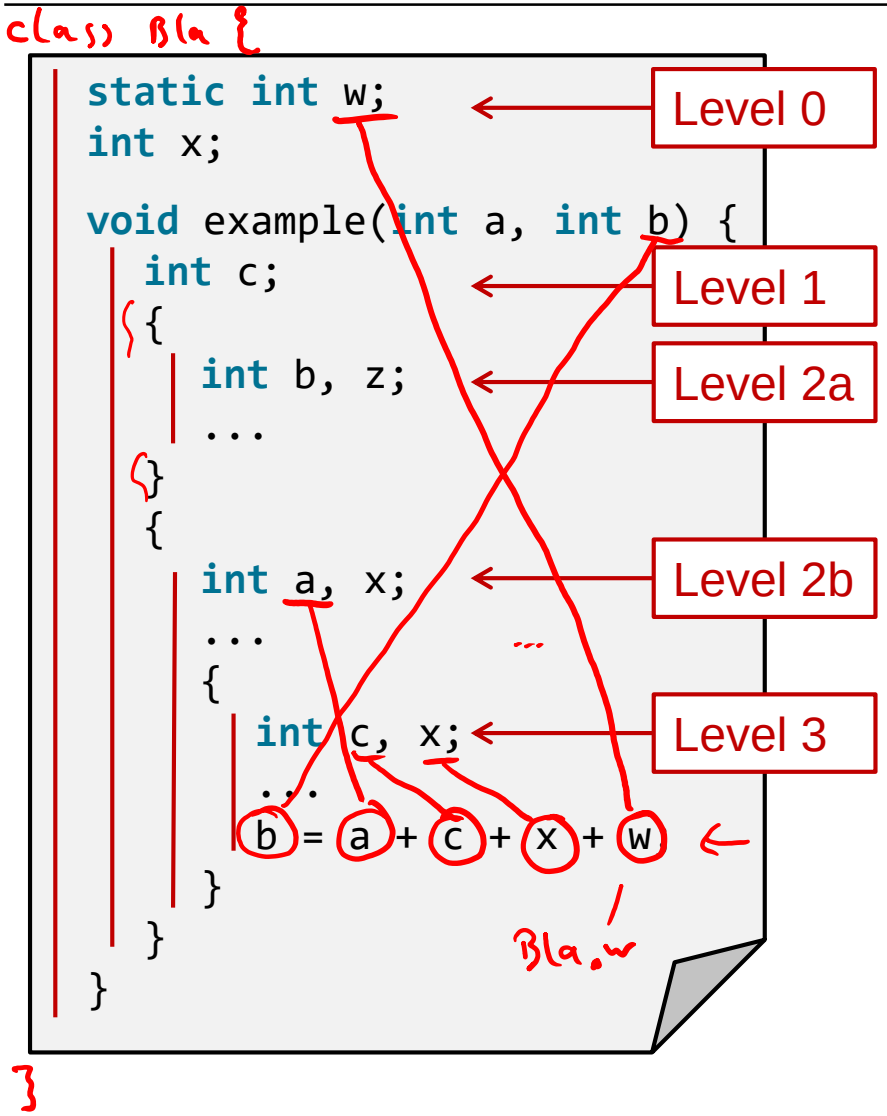
WHERE ARE WE?


Block 1
Block 2
Block 3: Elaboration
Front End
Optimizer
Back End
Scanner
Parser
Elaboration
Optimization 1
Optimization 2
Optimization n
Inst Selection
Inst Scheduling
Reg Allocation
Infrastructure

WHAT'S IN A NAME?

- … that which we call a rose by any other name would smell as sweet
  (Shakespeare, Romeo and Juliet)
  - But only if that (other) name has been properly declared!
  - This is now (better?) known as the principle of *alphaconversion*
- *Scope* of a name (variable, method, class, …)
  - That portion of the program/parse tree where the name is known
  - Explicitly (syntactically) delimited, typically by block structure
  - All occurrences of a name in the same scope refer to the same item
    - In many languages, names are explicitly declared
  - Uses are bound together (and to declaration) during elaboration phase
- Alphaconversion: within scope, names can be replaced by others
  - Without affecting the meaning of the program
  - If the new names do not already exist in that scope
  - This is a well-known refactoring step
- Mechanism for keeping track of scopes: *Symbol table*

EXAMPLE PROGRAM (EC FIG 5.10)



Scope Level	Declared Names
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x

(this redeclaration of x not allowed in Java)

- Levels 0, 1, 2b, 3 are *nested*
  - And 2a is nested inside 1
- Levels 2a, 2b are *independent*

Connecting *uses* to *declarations*

- E.g., for type checking

OTHER CASES OF SCOPING

```
int example1() {
  int j = 5;
  for(int i = 0; i <= 5; ++i) {
    j += i;
  }

  return j + i;
}
```

error

OTHER CASES OF SCOPING

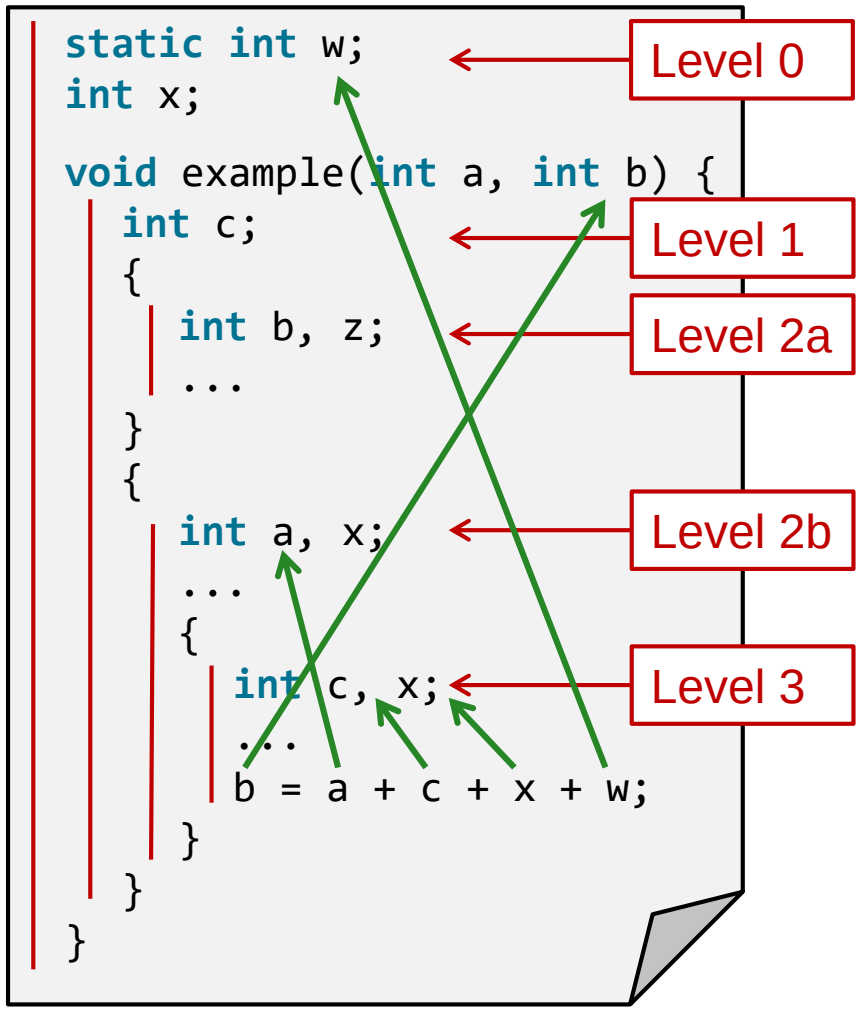
```
int example1() {
   int i = 5;
   for(int i = 0; i <= 5; ++i) {
      System.out.println(i);
   }

   return i;

}
```

```
class ClassA {
   ClassA ClassA() {
      return new ClassA();
   }
   void test() {
      var a = ClassA();
   }
}
```

```
void example2() {
   int a = 2;
   int b = a;

   int c = d;
   int d = 5;

}
```

```
int funA(int a) {
   if(a == 0) return 0;
   else return 1 + funB(a + 1);
}

int funB(int a) {
   return funA(a - 2);
}
```

EXAMPLE PROGRAM (EC FIG 5.10)



Scope Level	Declared Names
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x

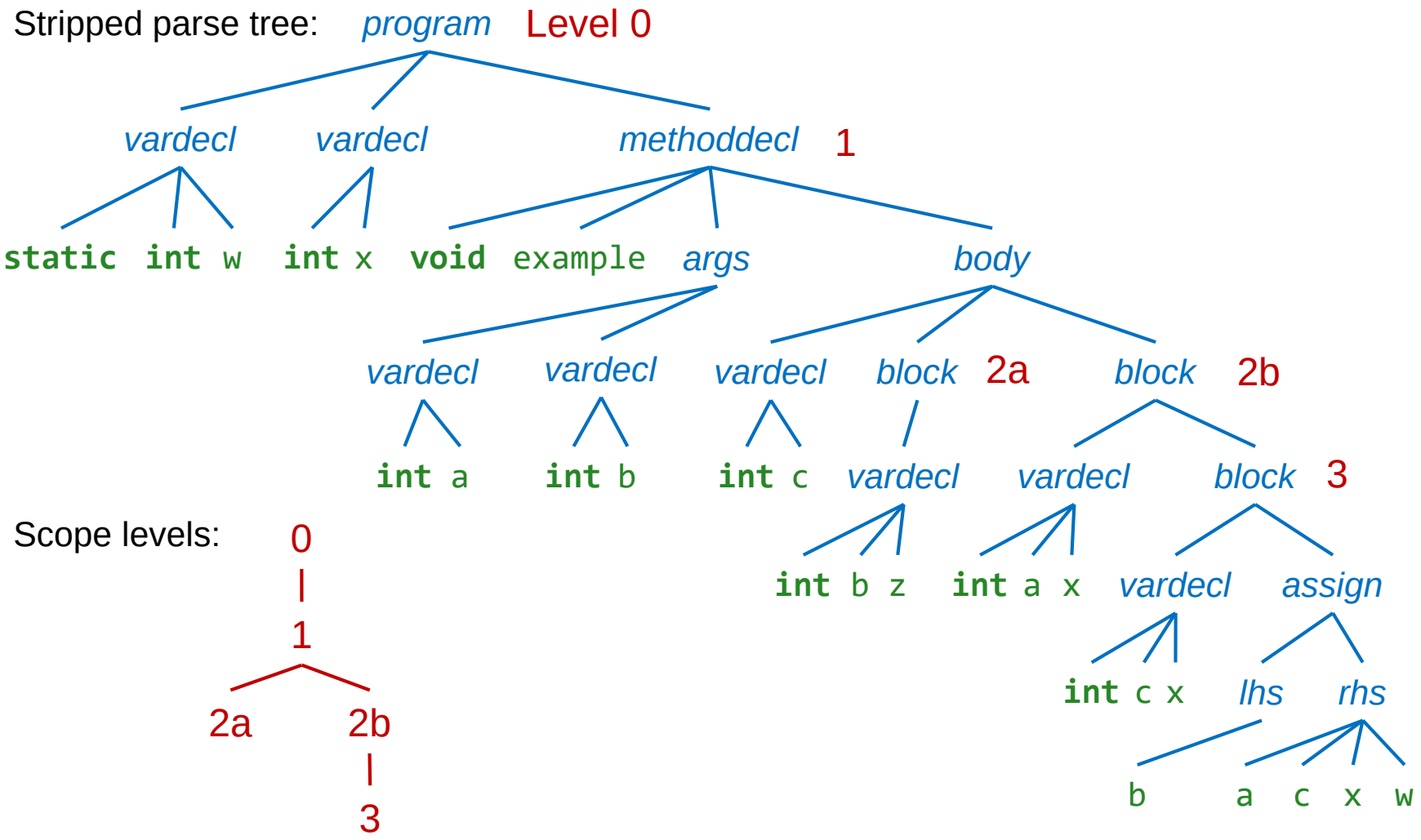
(this redeclaration of x not allowed in Java)

- Levels 0, 1, 2b, 3 are *nested*
  - And 2a is nested inside 1
- Levels 2a, 2b are *independent*

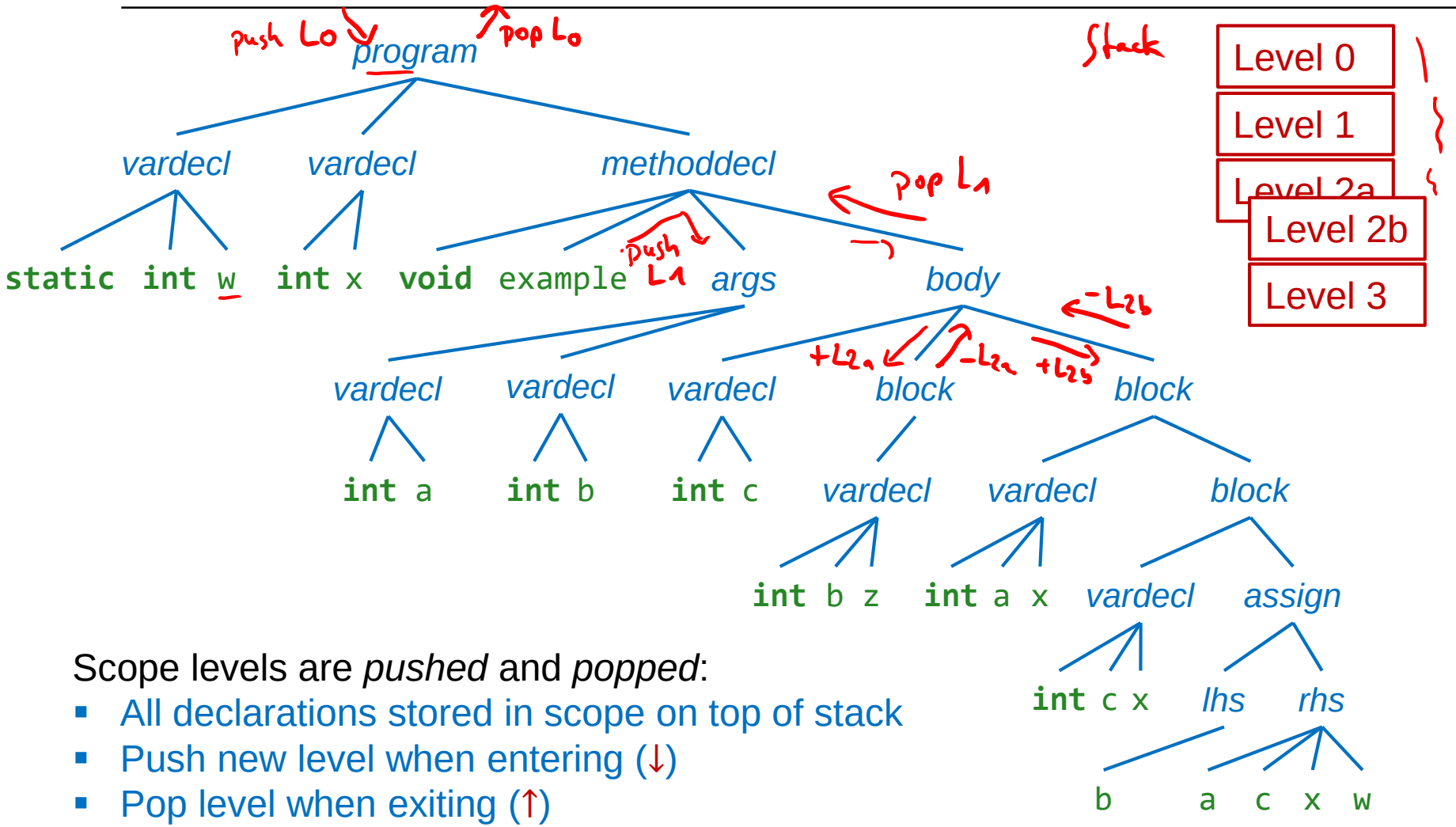
Connecting *uses* to *declarations*

- E.g., for type checking

SCOPE STRUCTURE FOLLOWS PARSE TREE STRUCTURE



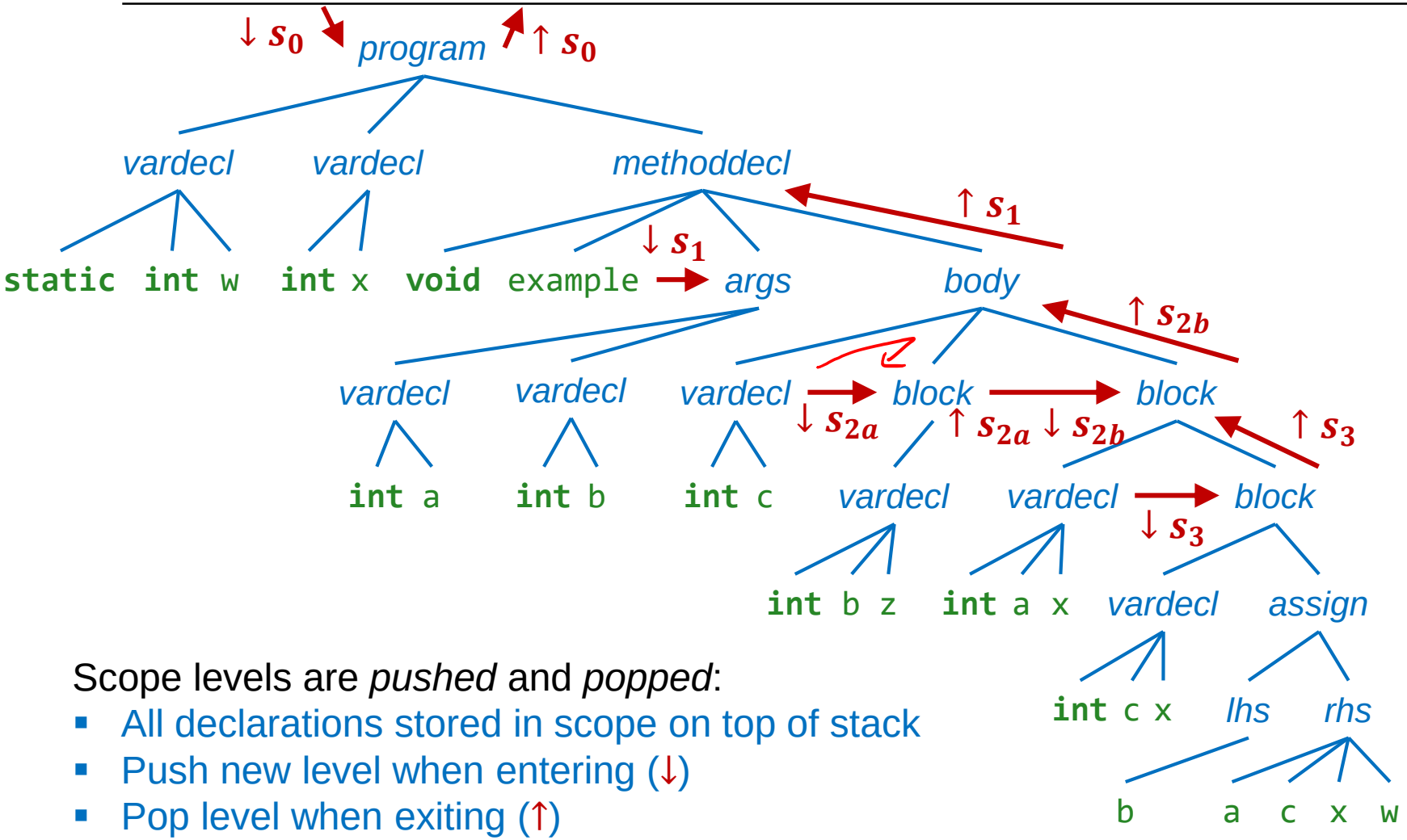
STACK OF SCOPES



Scope levels are *pushed* and *popped*:

- All declarations stored in scope on top of stack
- Push new level when entering (↓)
- Pop level when exiting (↑)

STACK OF SCOPES



Scope levels are *pushed* and *popped*:

- All declarations stored in scope on top of stack
- Push new level when entering (↓)
- Pop level when exiting (↑)
- Can be programmed in tree listener

SYMBOL TABLE

What to store for an ID?
- Depends on kind of ID
- E.g., type of variable or signature of method
- Possibility: *parse tree node that declares ID*

```
public interface SymbolTable<R> {
   /** Adds (pushes) a next scope level. */
   public void openScope();

   /** Removes (pops) the top scope level. */
   public void closeScope();

   /** Declares a given ID in the top scope. */
   public boolean put(String id, R rec);

   /** Looks up the declaration of a given ID. */
   public R get(String id);
}
```

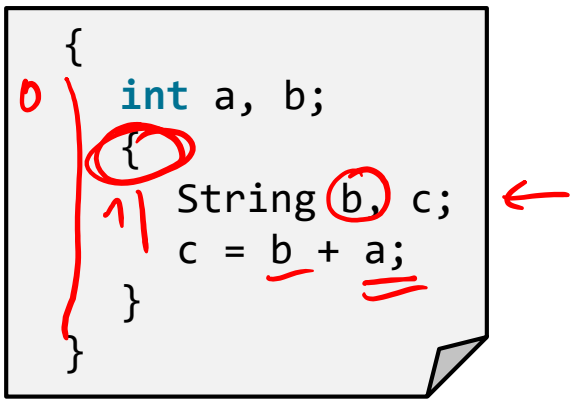
EC: *InitializeScope*

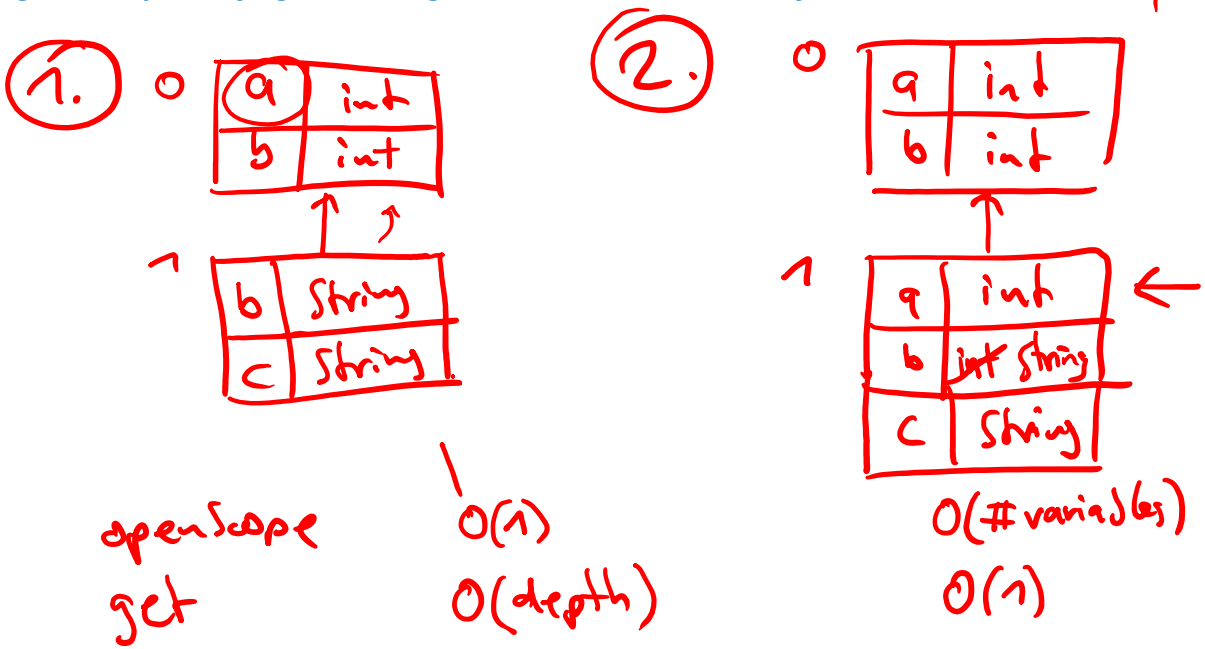
EC: *FinalizeScope*

What is the lookup strategy?
- Go through scopes from top to bottom
- Return record for first (= topmost) occurrence

DATA STRUCTURE FOR SYMBOL TABLE?

- `Stack` of `Map<String,R>`
  - Two possible strategies:
    1. Only new variables in stack top
       - Disadvantage: may have to go through entire stack on lookup
    2. Copy entire map to new stack top
       - Disadvantage: map may grow large, most names stay unused





DATA STRUCTURE FOR SYMBOL TABLE?

- `Stack` of `Map<String,R>`
  - Two possible strategies:
    1. Only new variables in stack top
       - Disadvantage: may have to go through entire stack on lookup
    2. Copy entire map to new stack top
       - Disadvantage: map may grow large, most names stay unused
  - Optimal strategy depends on language
    - E.g., no deeply nested scopes: lookup complexity low
- Information lost when scope is closed (i.e., stack is popped)
  - What if we need it again in later compilation phase?
  - Solution: store $R_{\texttt{name}}$ as attribute (`ParseTreeProperty`) for every use of `name`

STACK OF SCOPES