Lecture 09

Semantic Analyzer

Part 1: Scope checking

Hyosu Kim

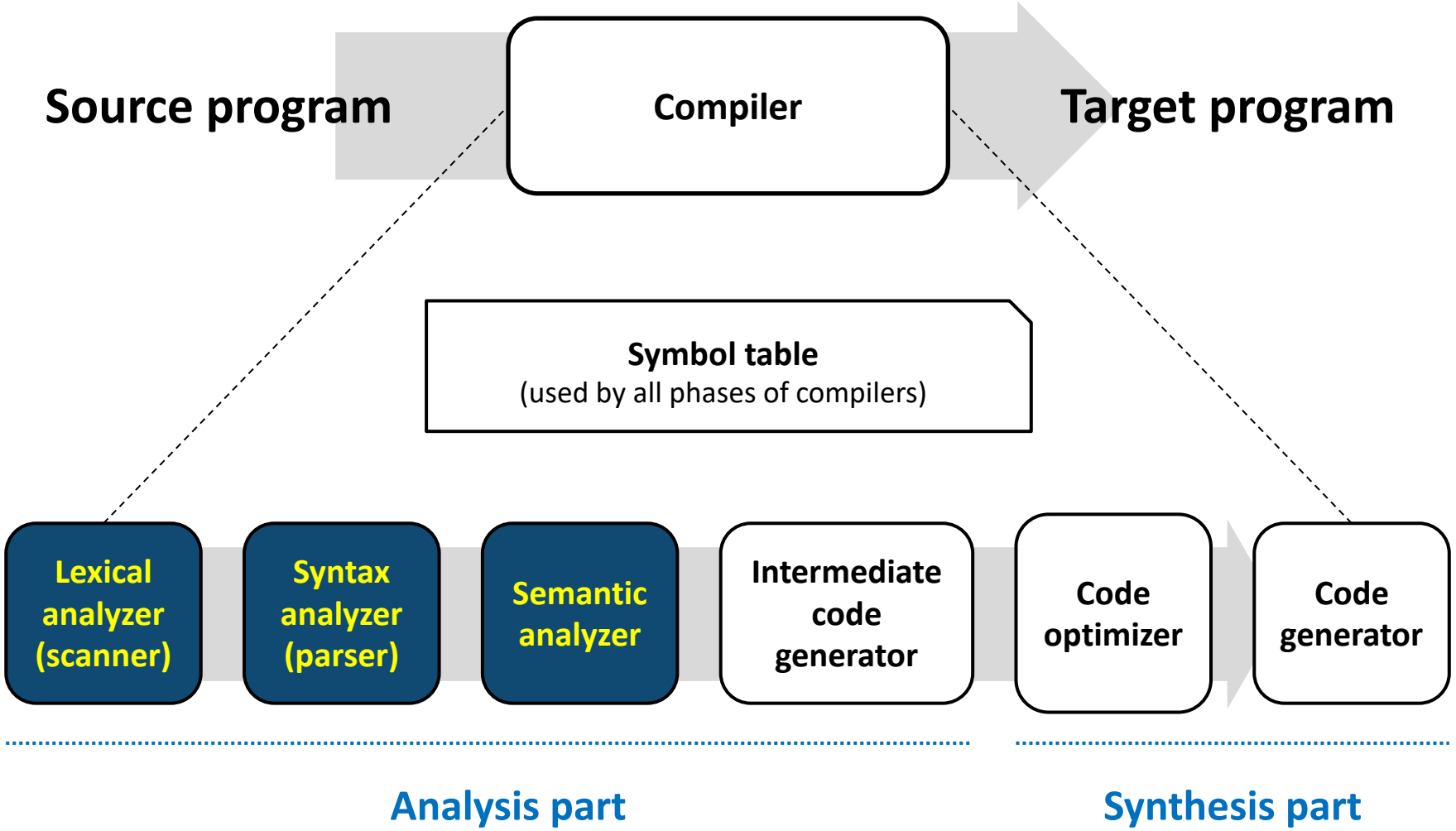
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://hcslab.cau.ac.kr

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Overview


Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

Overview

What does a semantic analyzer do?

My / daughter / is / a / boy

<possessive> <noun> <verb> <article> <noun>

The sentences are syntactically valid

But, semantically invalid

Overview

What does a semantic analyzer do?

int / main / (/ void /) / { / return / a / ; / }

<type> <id> <lparen> <type> <rparen> <lbrace> <return> <id> <semi> <rbrace>

The source code is syntactically valid

But, semantically invalid

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)

```
int a;
void foo () {
   a = 3;
}
```

```
void foo (int a) {
   a = 3;
}
```

```
void foo() {
   int a;
   a = 3;
}
```

```
void foo() {
   a = 3;
   int a;
}
```

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)

```
int a;
void foo () {
   a = 3;
}
```

```
int a;
void foo () {
   int a = 3;
}
```

```
void foo() {
   int a = 3;
   int a = 2;
}
```

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)

```
void foo () { … }
```

```
void foo () { … }
void foo () { … }
```

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables

```
int a = 3;
```

```
int a, b;
a = 0;
b = a;
```

```
int a = 3.5;
```

```
char a = 3.5;
```

```
a[3.5] = 0;
```

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables
5. All functions must be used with the right number and type of arguments

```
void foo(int a) {…}
foo(3);
```

```
void foo(int a) {…}
foo(“compiler”);
```

```
void foo(int a) {…}
foo(1, 2);
```

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables
5. All functions must be used with the right number and type of arguments

How to check them?????

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depe

Through scope checking!!

Common semantic grammars

1. All variables must be declared before their us globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables
5. All functions must be used with the right number and type of arguments

How to check them?????

Semantic analyzer

Checks many kinds of semantic grammars

Semantic grammars can be different depending on the programming language

Common semantic grammars

1. All variables must be declared before their use (globally or locally)
2. All variables must be declared only once (locally)
3. All functions must be declared only once (globally)
4. All variables must be used with the right type of constant or variables
5. All functions must be used with the right number and type of arguments

How to check them?????

Through type checking!!

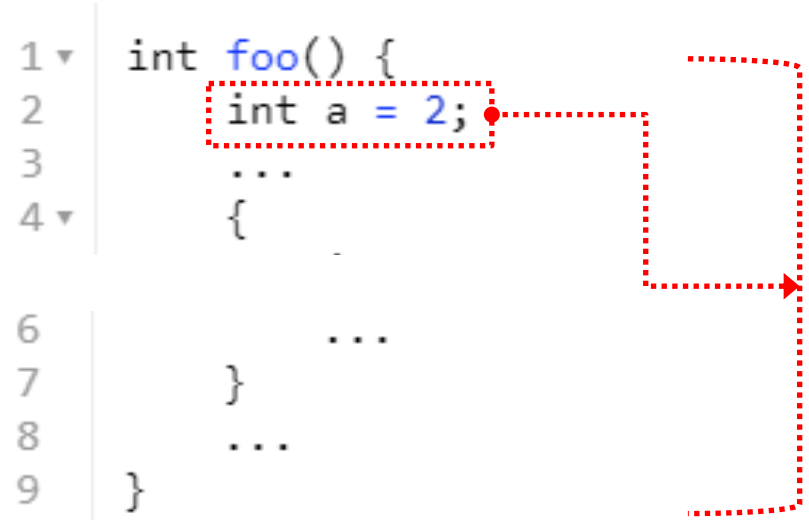
Scope checking

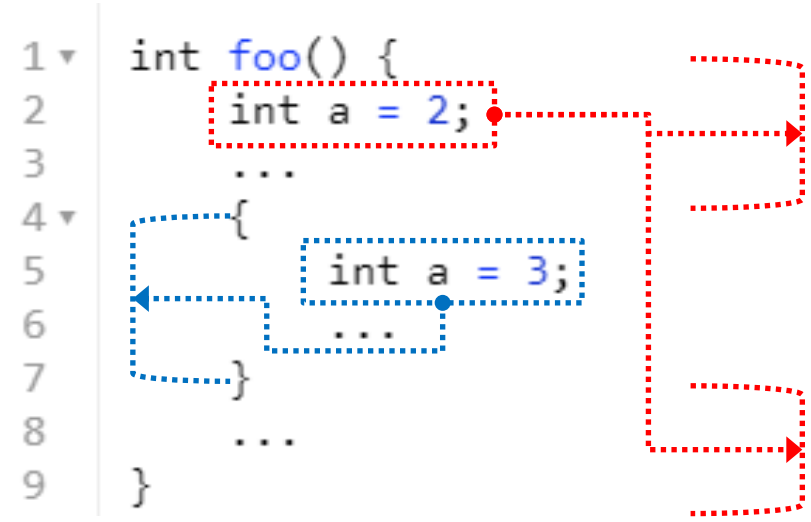
The scope of an identifier is the portion of a program

in which the identifier can be accessed

- Scope matches identifier declarations with uses
- The same identifier may refer to different things in different scopes

Examples of scope





Scope checking

Two types of scope

- **Static** scope (used in most programming languages)

  Scope depends on the physical structure of program text (e.g., {}, (), …)

- **Dynamic** scope (used in Lisp, SNOBOL)

  Scope depends on execution of the program

  (e.g., the most currently declared identifier is used)

- **Examples**

```
void foo() { int a = 3; }
…
int a = 2;    foo();   print(a);
…
```

Scope checking

In most programming languages, the scope of identifiers are determined with

- Function declarations

  int funcName(…), the identifier funcName is declared

- Class declarations

  class className {…}, the identifier className is declared

- Variable declarations

  int varName, the identifier varName is declared

- Formal parameters

  int func(int formal1, int formal2), two identifiers formal1 and formal2 are declared

The most-closely nested scope rule

An identifier is matched with the identifier declared in the most-closely nested scope

The identifier should be declared before it is used

Examples

```
1   int a = 2;
2
3   int foo() {
4       printf("%d\n", a);
5
6       int a = 3;
7       printf("%d\n", a);
8
9       return 0;
10  }
```

Global declaration

Local (nested) declaration

Stdout: 2

Stdout: 3

The most-closely nested scope rule

An identifier is matched with the identifier declared in the most-closely nested scope

The identifier should be declared before it is used

Exceptions

- Function

```
1  class Test {
2  public:
3      void bar() {
4          foo();
5          return;
6      }
7
8      void foo() {
9          printf("foo");
10         return;
11     }
12
13 };
```

Function call before declaration

Function declaration

Stdout: foo

The most-closely nested scope rule

An identifier is matched with the identifier declared in the most-closely nested scope

The identifier should be declared before it is used

Exceptions

- Object-oriented languages

```
1  class Parent {
2  public:
3      void foo() {
4          printf("foo");
5          return;
6      }
7  };
8
9
10 class Test : public Parent{
11 public:
12     void bar() {
13         foo();
14         return;
15     }
16 };
```

Function declaration

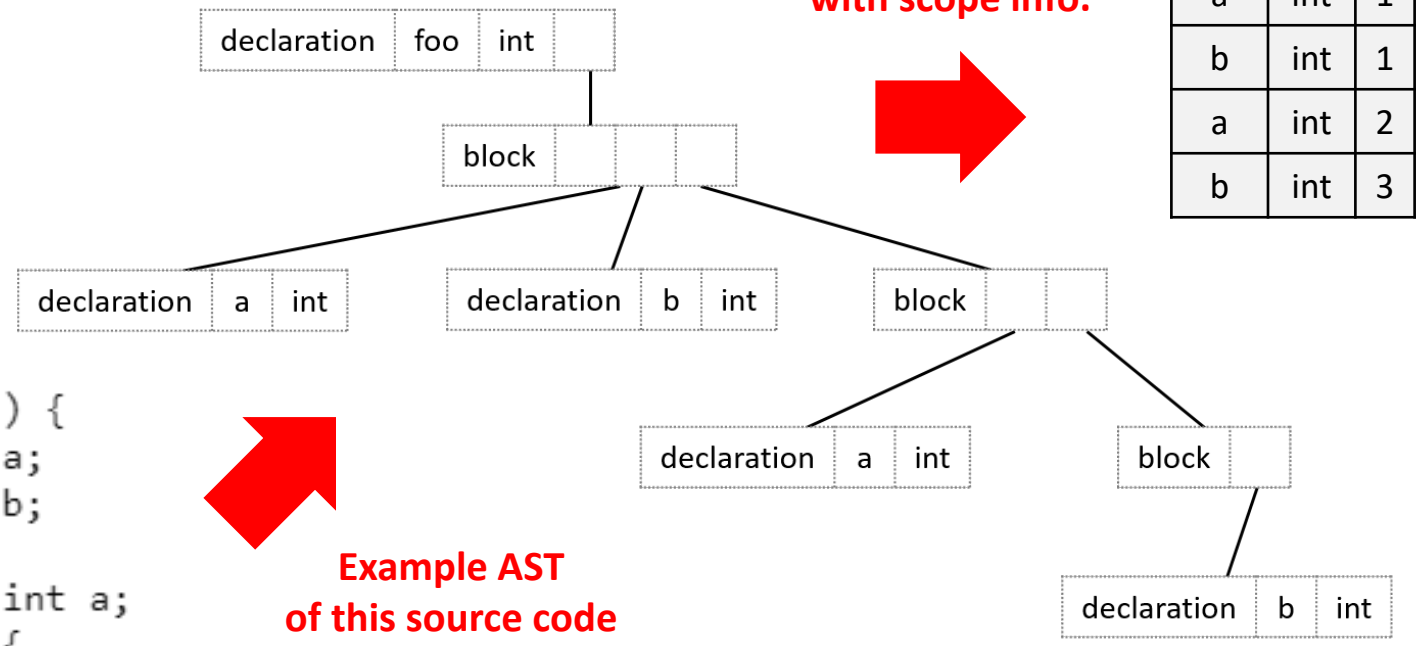
Stdout: foo

Inherited function call

Implementation of scope checking

While traveling AST

Update the symbol table with the scope information



Symbol table with scope info.

foo	int	0
a	int	1
b	int	1
a	int	2
b	int	3

```
1  int foo() {
2      int a;
3      int b;
4      {
5          int a;
6          {
7              int b;
8          }
9      }
```

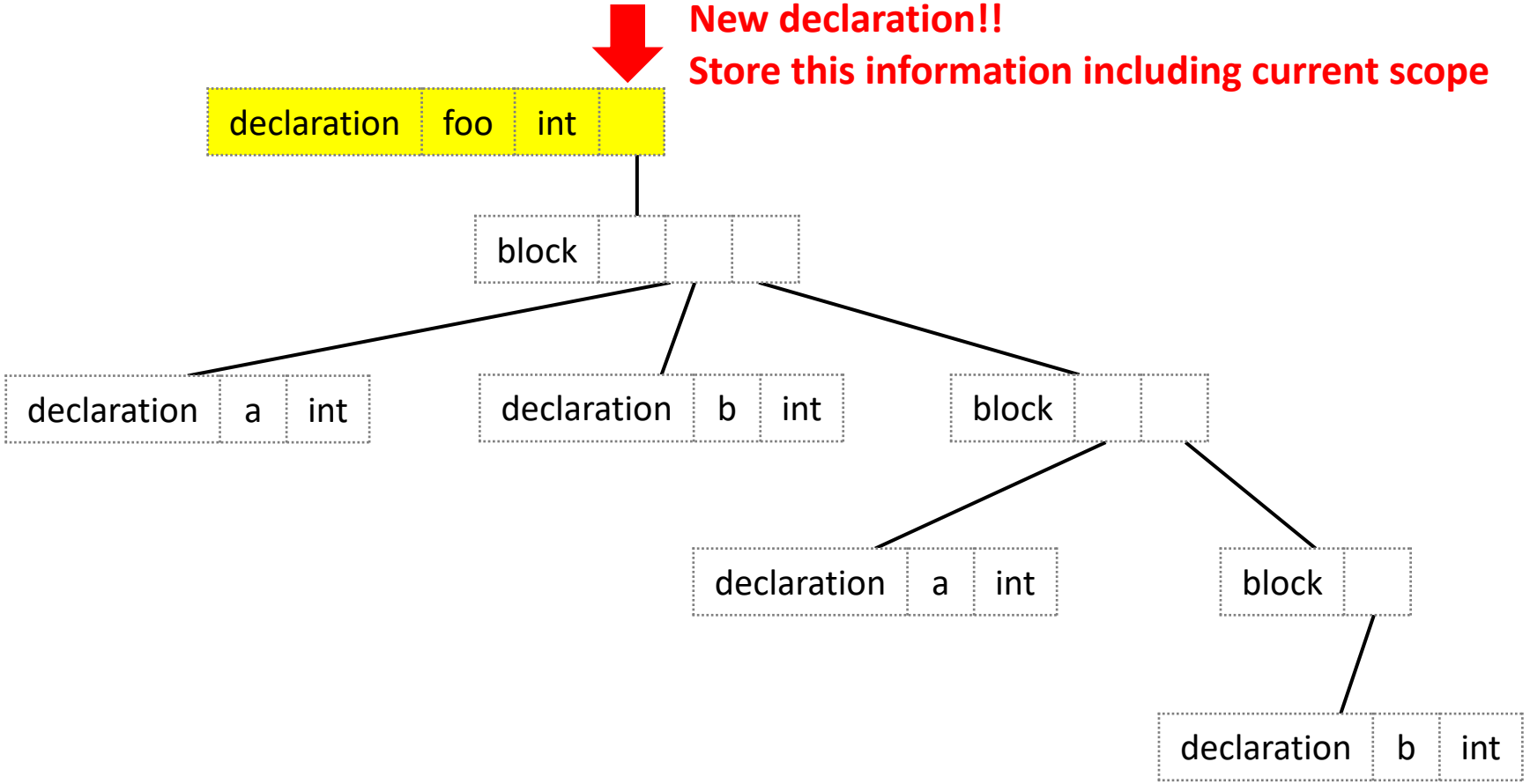
Example AST of this source code

Implementation of scope checking

Example

Current scope = 0

foo	int	0

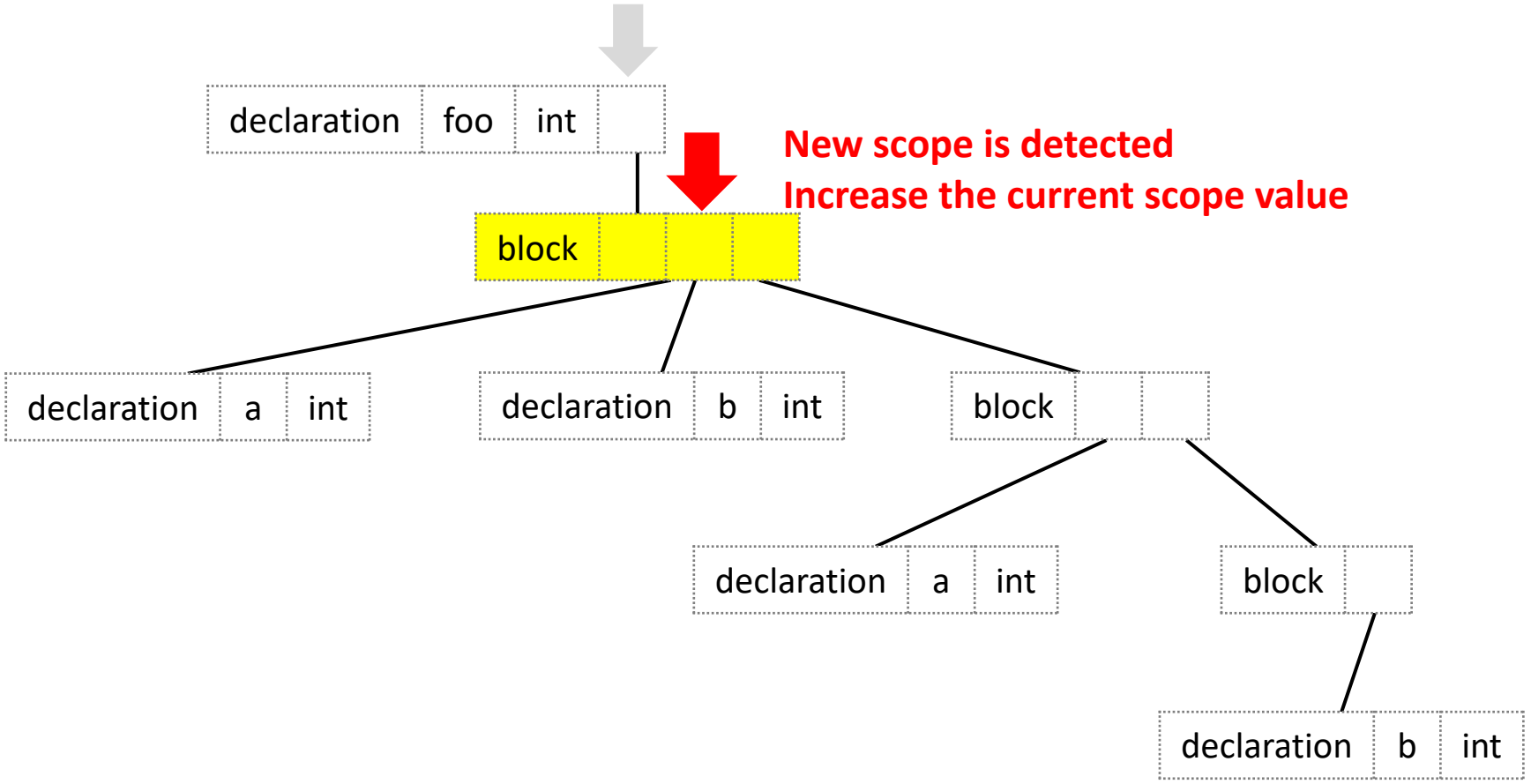


Implementation of scope checking

Example

Current scope = 0 ⇒ 1

foo	int	0

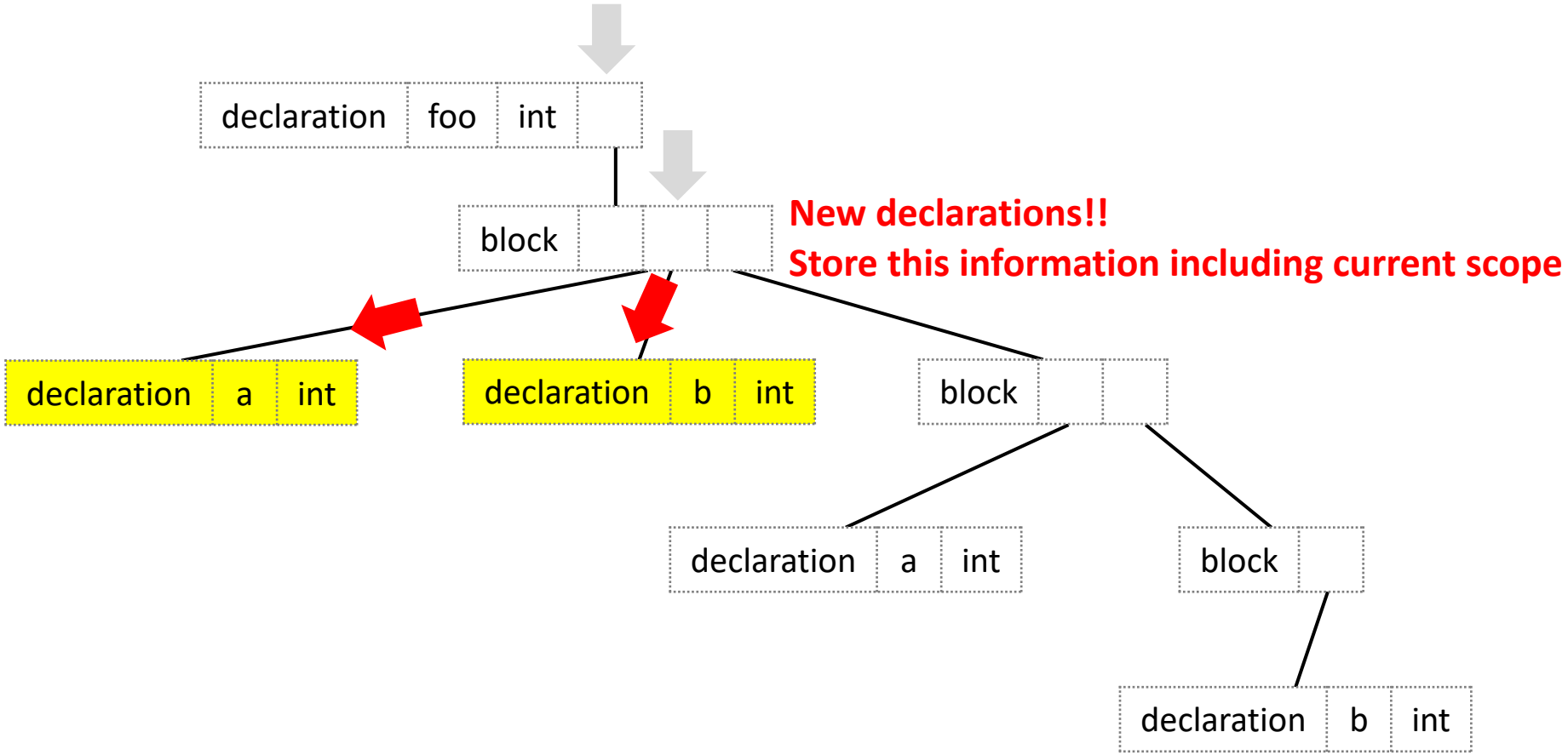


Implementation of scope checking

Example

Current scope = 1

foo	int	0
a	int	1
b	int	1

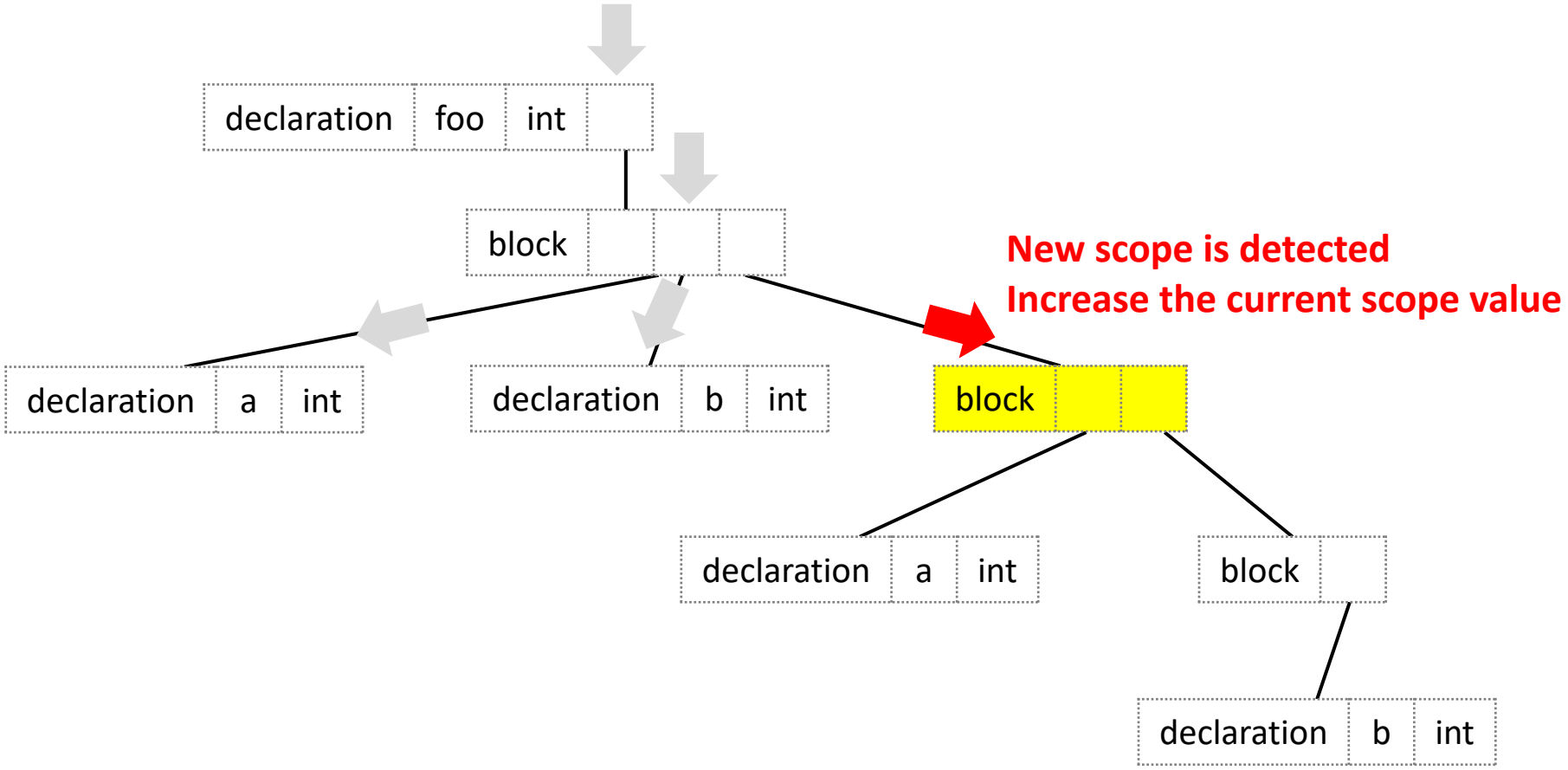


Implementation of scope checking

Example

Current scope = 1 ⇒ 2

foo	int	0
a	int	1
b	int	1

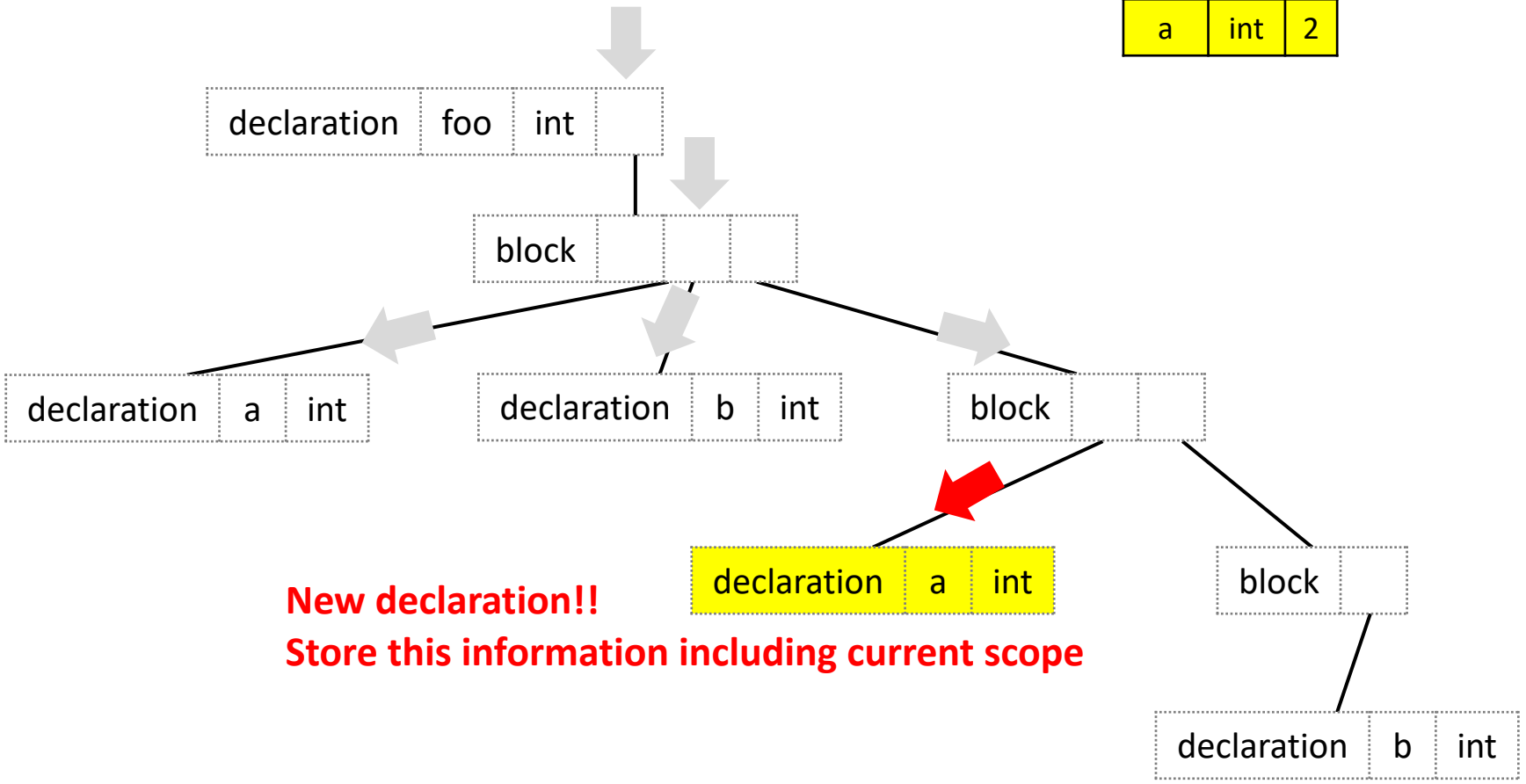


Implementation of scope checking

Example

Current scope = 2

foo	int	0
a	int	1
b	int	1
a	int	2

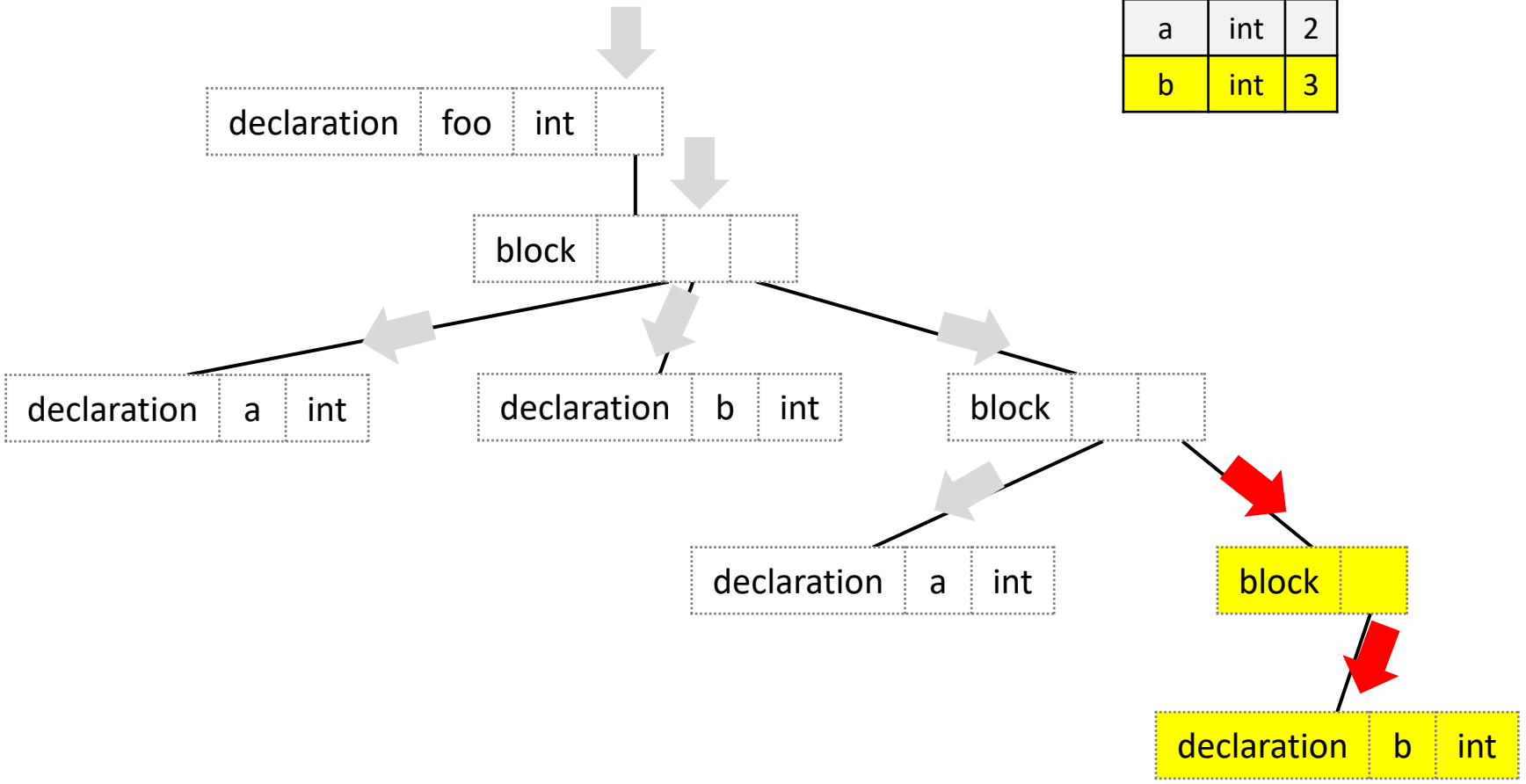


Implementation of scope checking

Example

Current scope = 2 ⇒ 3

foo	int	0
a	int	1
b	int	1
a	int	2
b	int	3

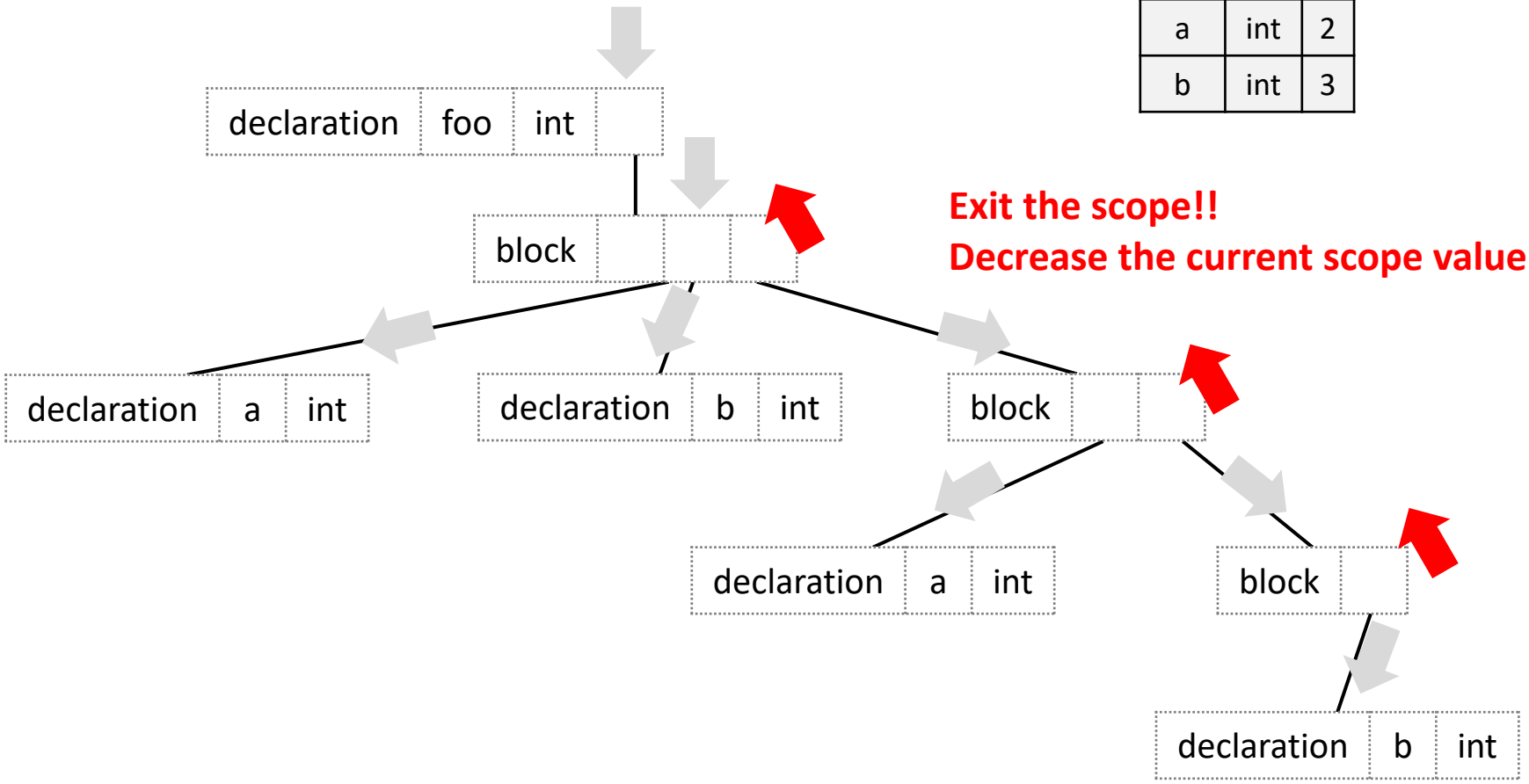


Implementation of scope checking

Example

Current scope = 3 ⇒ 2 ⇒ 1 ⇒ 0

foo	int	0
a	int	1
b	int	1
a	int	2
b	int	3

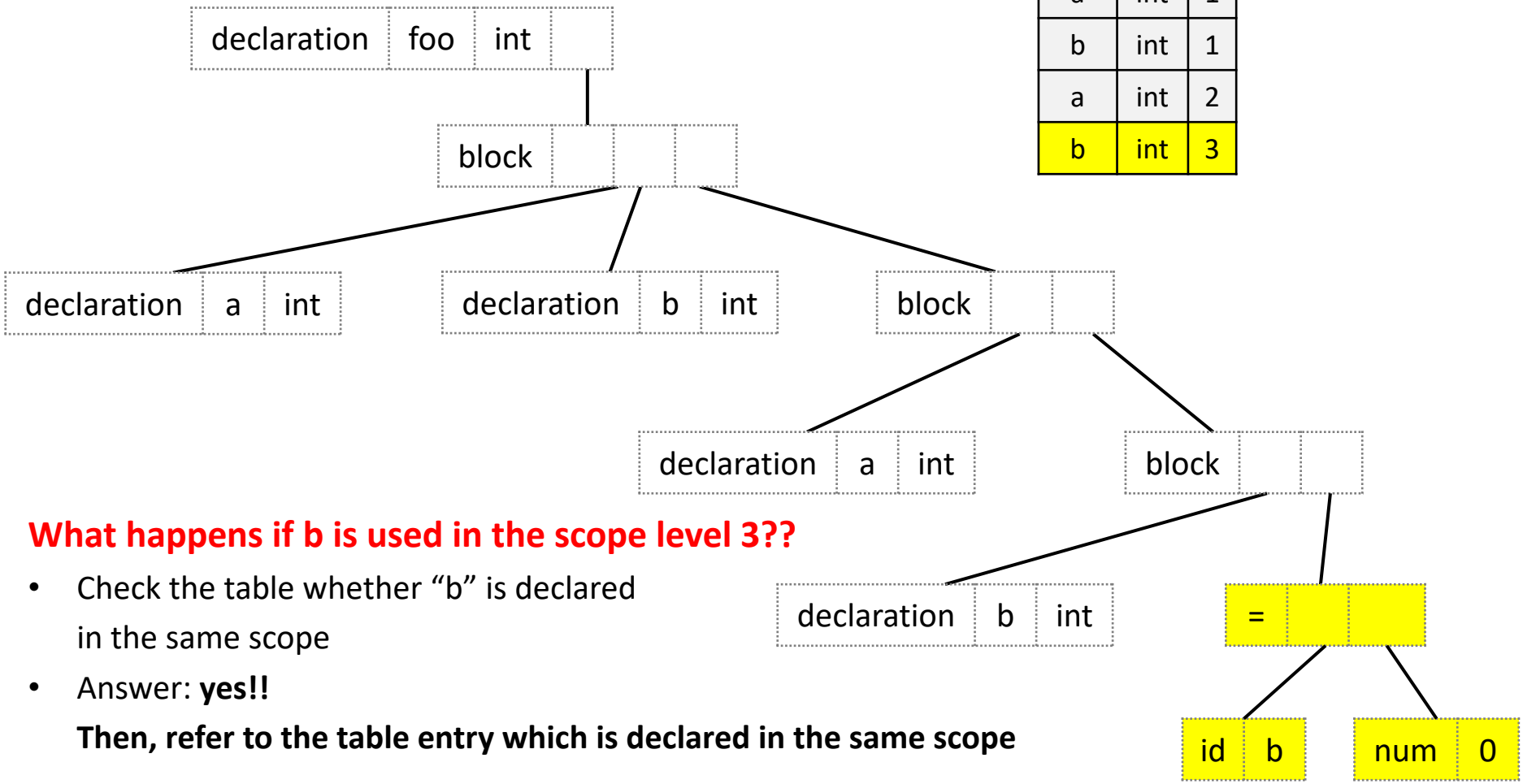


Implementation of scope checking

Example

Current scope = 3

foo	int	0
a	int	1
b	int	1
a	int	2
b	int	3



What happens if b is used in the scope level 3??

- Check the table whether “b” is declared in the same scope
- Answer: **yes!!**

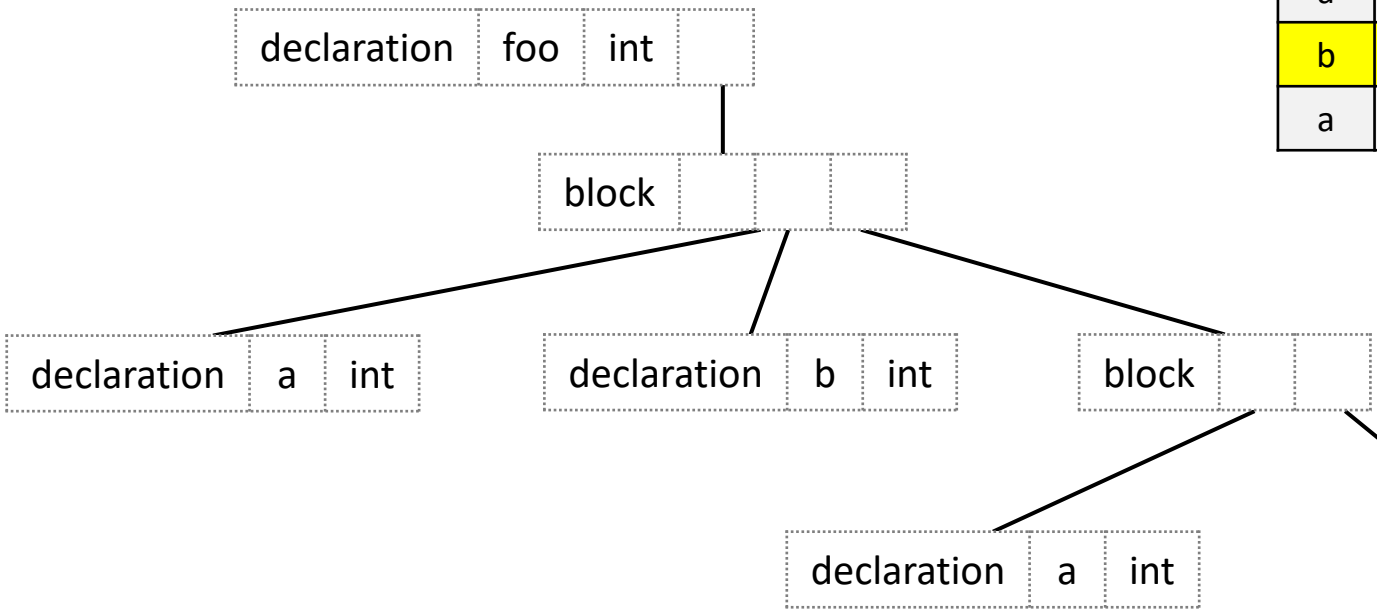
  Then, refer to the table entry which is declared in the same scope

Implementation of scope checking

Example

Current scope = 3

foo	int	0
a	int	1
b	int	1
a	int	2



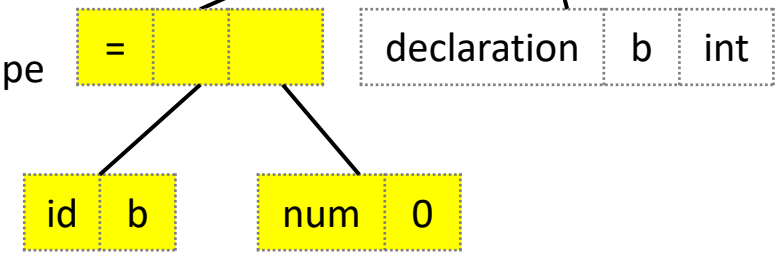
What happens if b is used in the scope level 3,

but before b is declared??

- Check the table whether each id is declared in the same scope
- Answer: **no!!**

  Then, check the parent scopes

  and refer the table entry declared in the closest scope

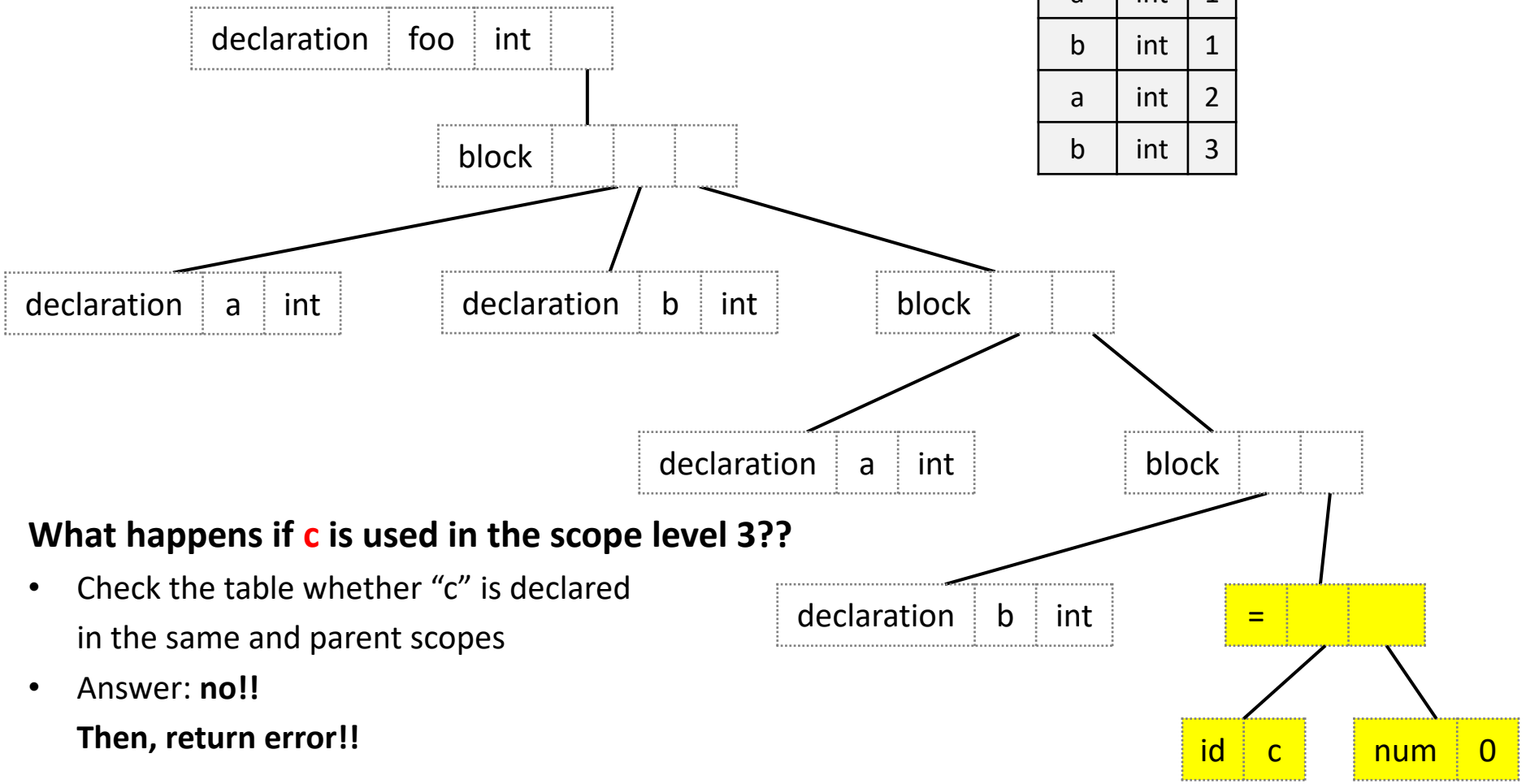


Implementation of scope checking

Example

Current scope = 3

foo	int	0
a	int	1
b	int	1
a	int	2
b	int	3



What happens if c is used in the scope level 3??

- Check the table whether "c" is declared in the same and parent scopes
- Answer: **no!!**
  Then, return error!!

Implementation of scope checking

While traveling AST

Update the symbol table with the scope information

But, such simple solution can incur problems

```
1  int foo() {
2      int a;
3      int b;
4      {
5          int a;
6          {
7              int b;
8          }
9      }
10     {
11         int c;
12     }
13 }
```

name	type	scope
foo	int	0
a	int	1
b	int	1
a	int	2
b	int	3
c	int	2

Problem #1: ambiguity

a and c are in the same level of scope.

But, they are in different scopes

Implementation of scope checking

While traveling AST

Update the symbol table with the scope information

But, such simple solution can incur problems

```
1  int foo() {
2      int a;
3      int b;
4      {
5          int a;
6          {
7              int b;
8          }
9      }
10     {
11         int c;
12     }
13 }
```

name	type	scope
foo	int	0
a	int	1
b	int	1
a	int	2
b	int	3
c	int	2

Problem #2: inefficiency

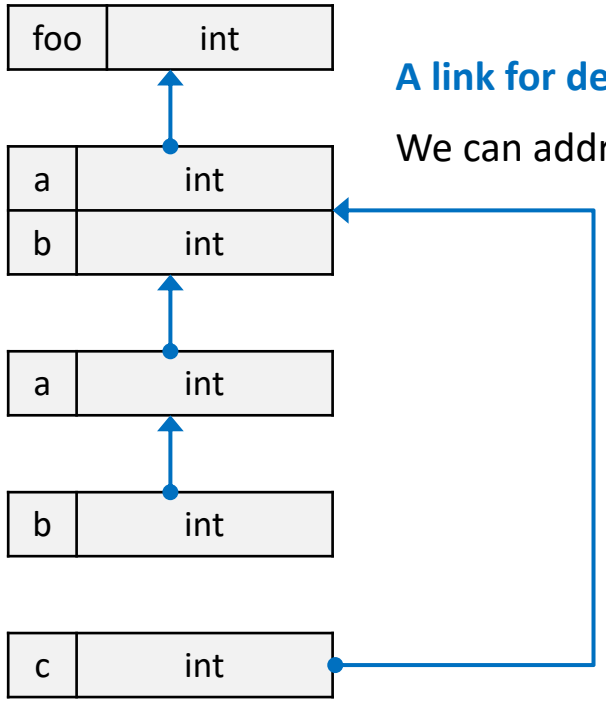
We should search the entire table entries every time

Implementation of scope checking

While traveling AST

1. **Construct a symbol table for each scope, describing a nesting structure**
2. Update the symbol table with information about what identifiers are in the scope

```
1  int foo() {
2      int a;
3      int b;
4      {
5          int a;
6          {
7              int b;
8          }
9      }
10     {
11         int c;
12     }
13 }
```

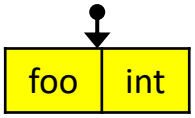


A link for describing the nesting structure

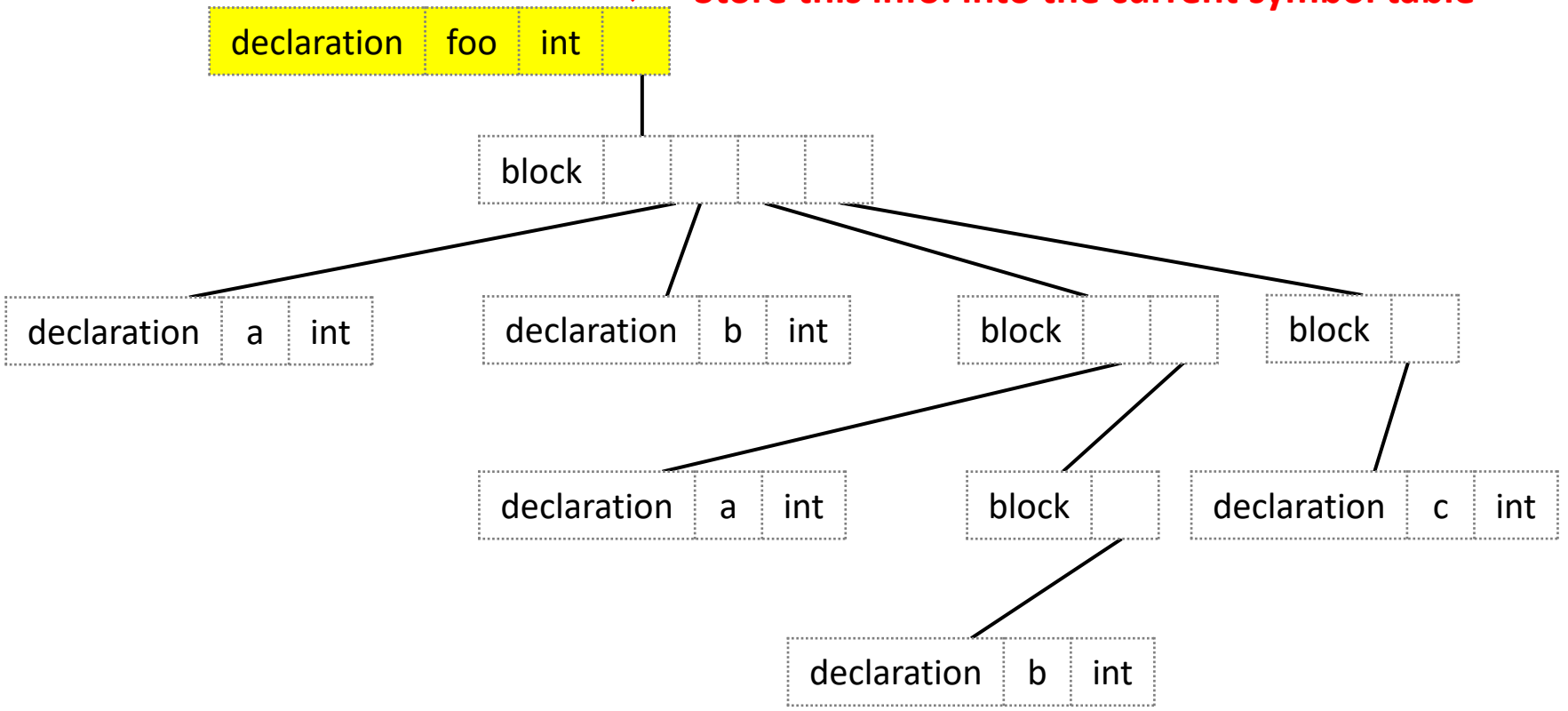
We can address the ambiguity problem!!

Implementation of scope checking

Example Current symbol table

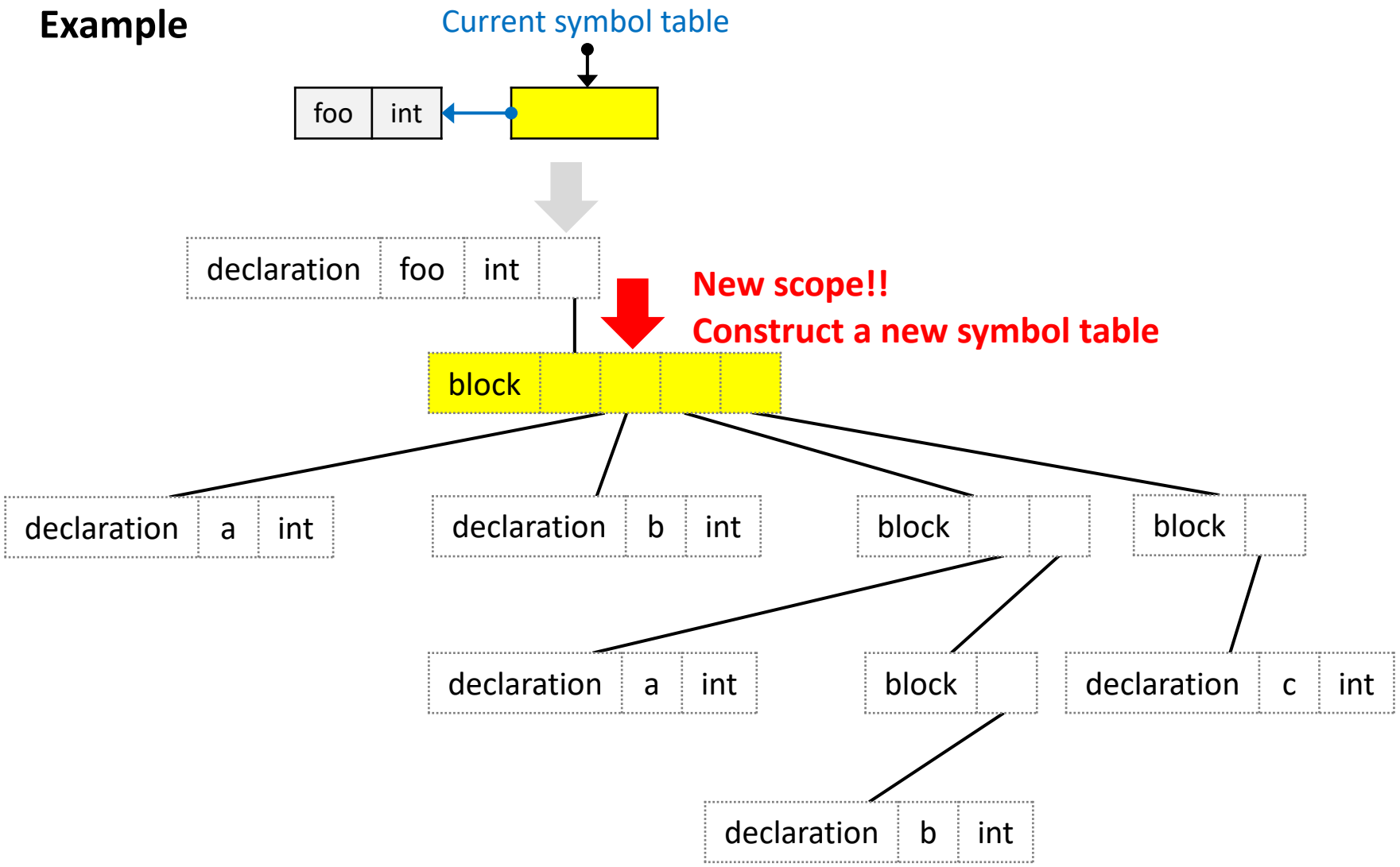


New declaration!!
Store this info. into the current symbol table



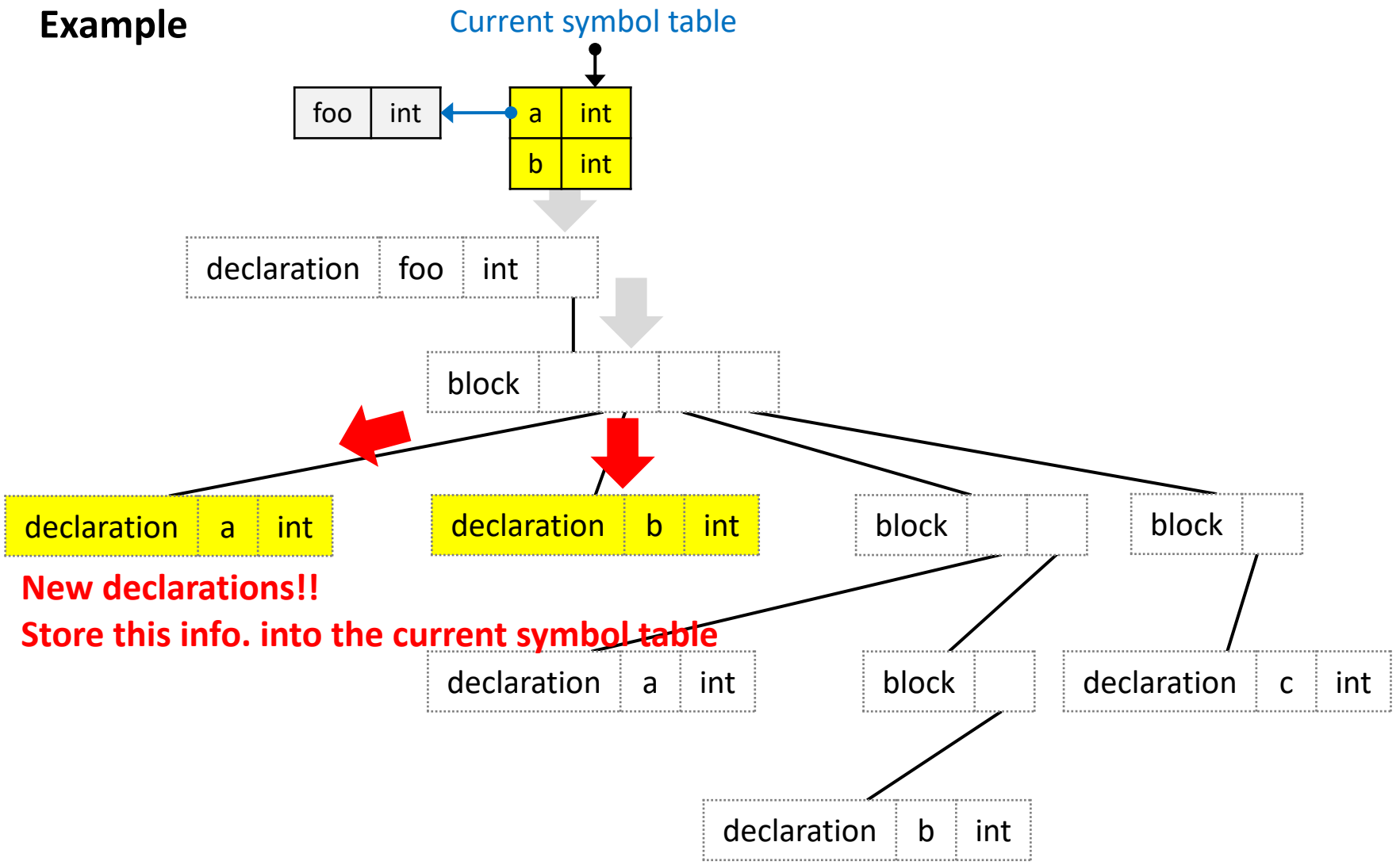
Implementation of scope checking

Example



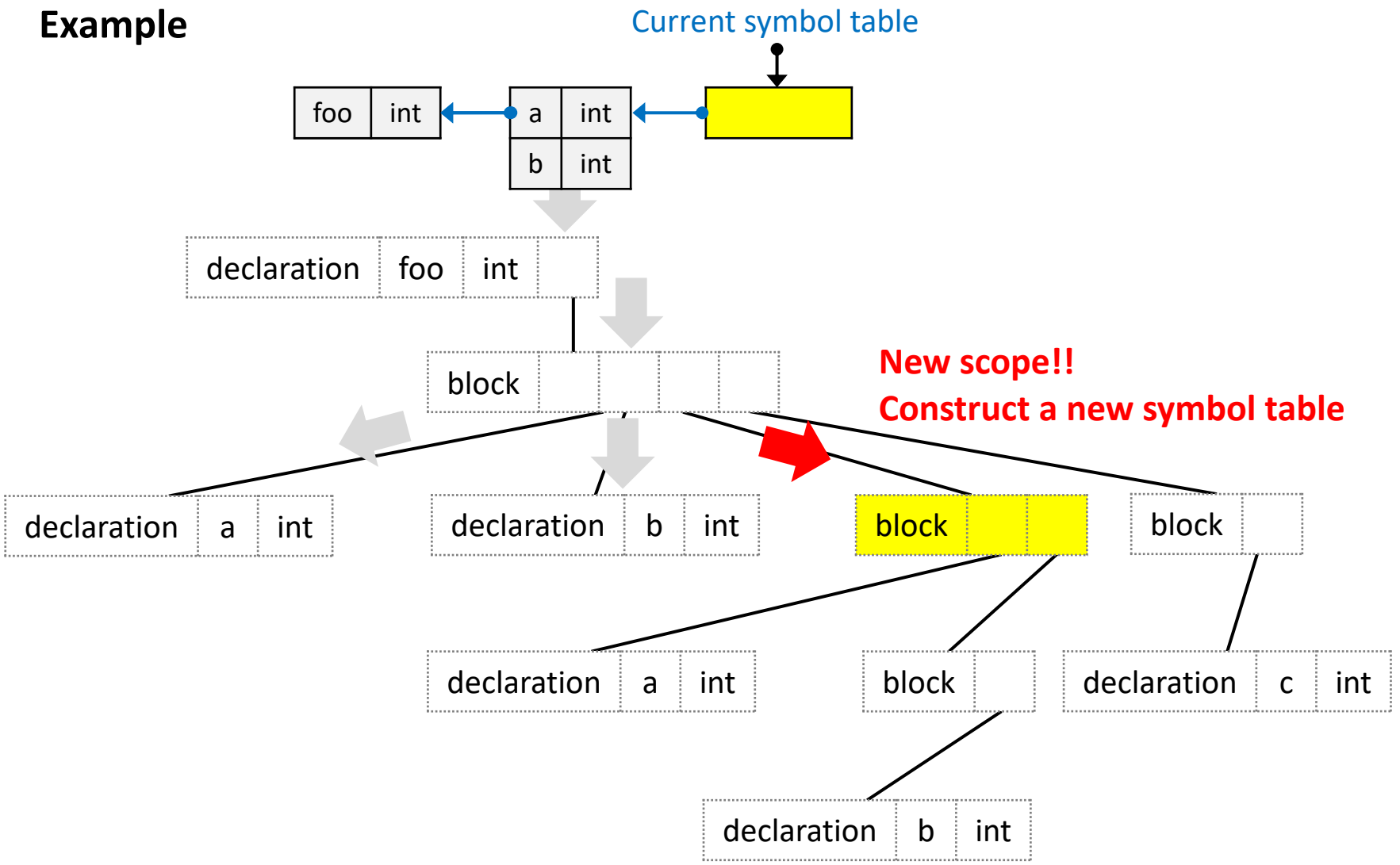
Implementation of scope checking

Example



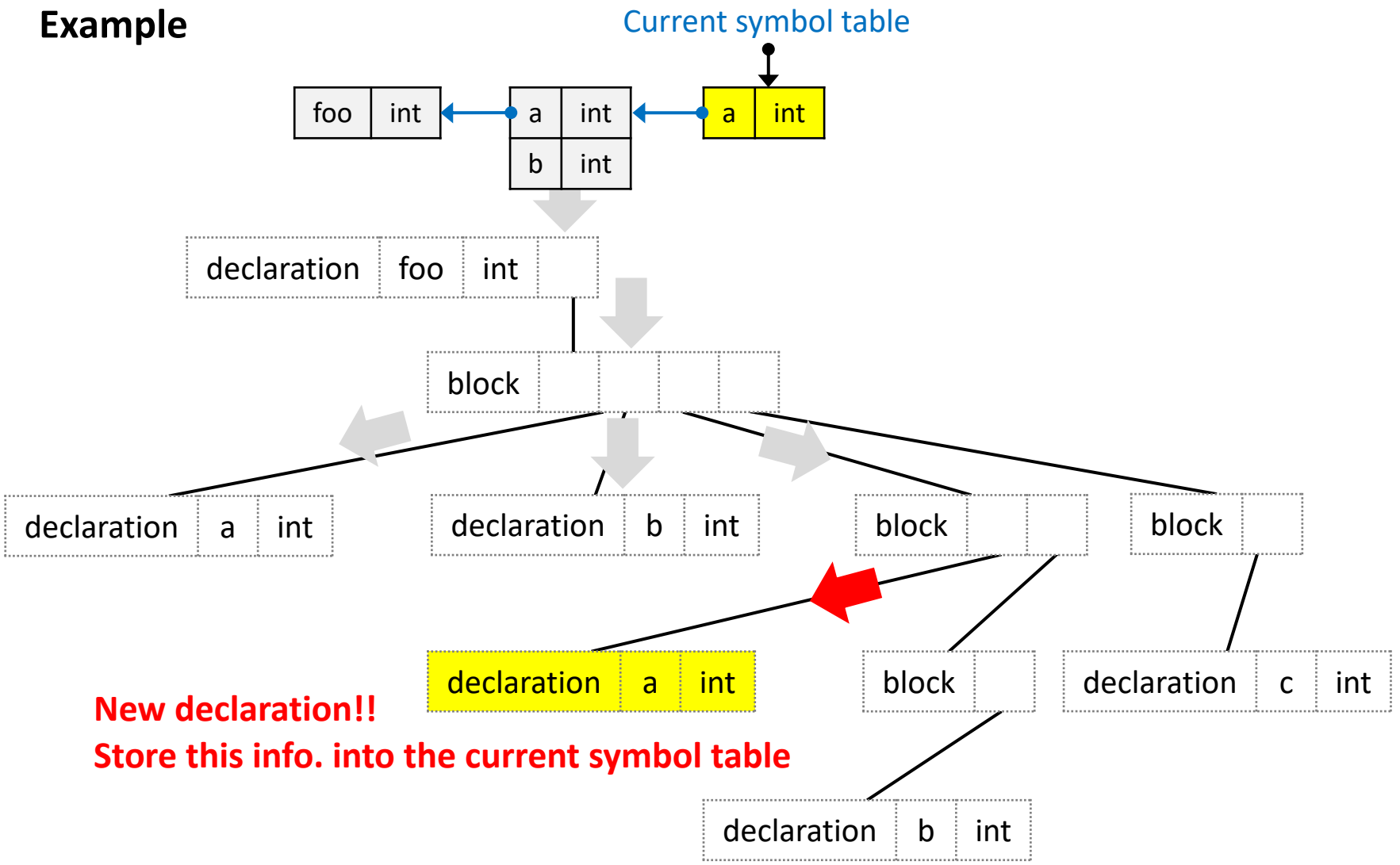
Implementation of scope checking

Example



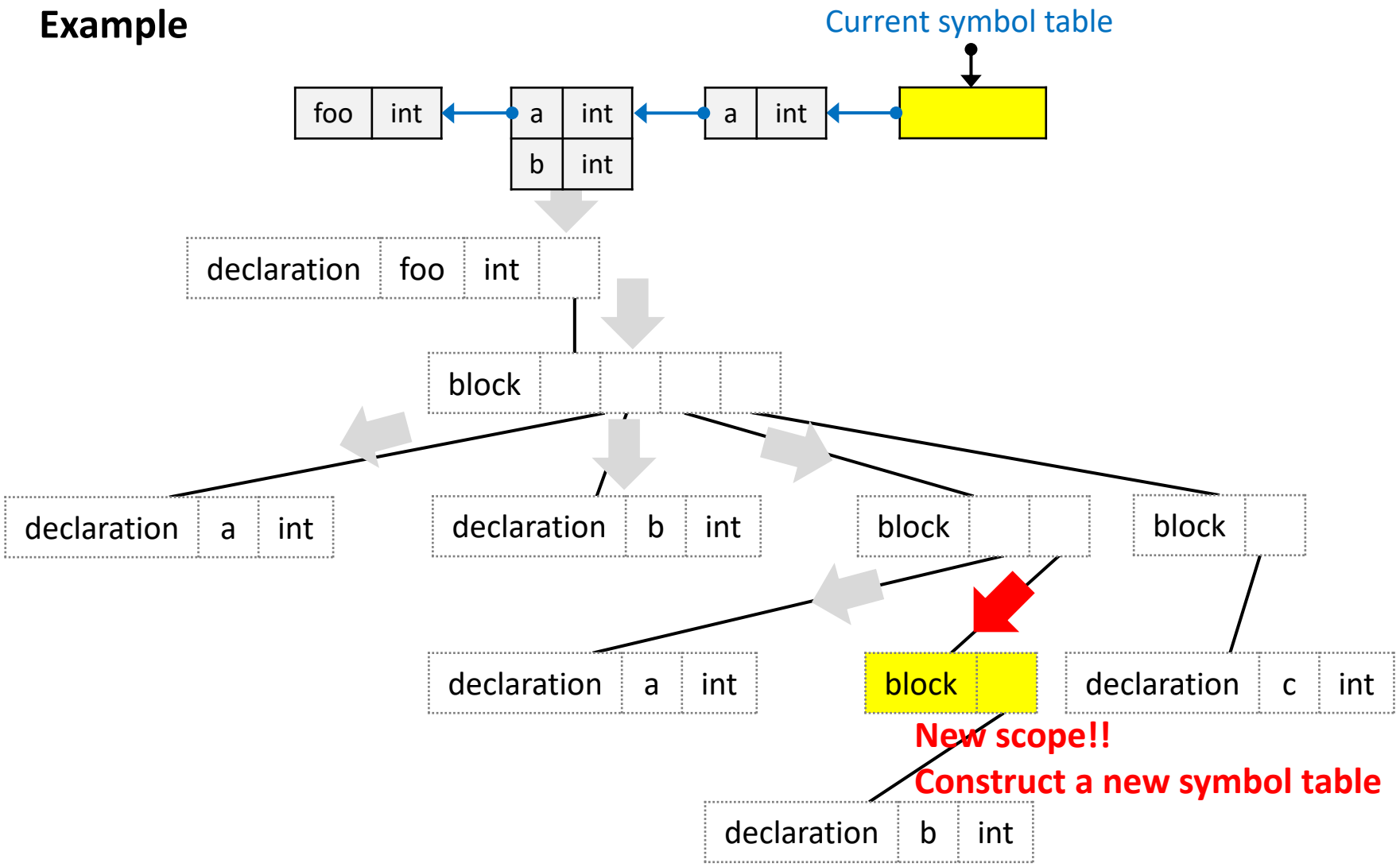
Implementation of scope checking

Example



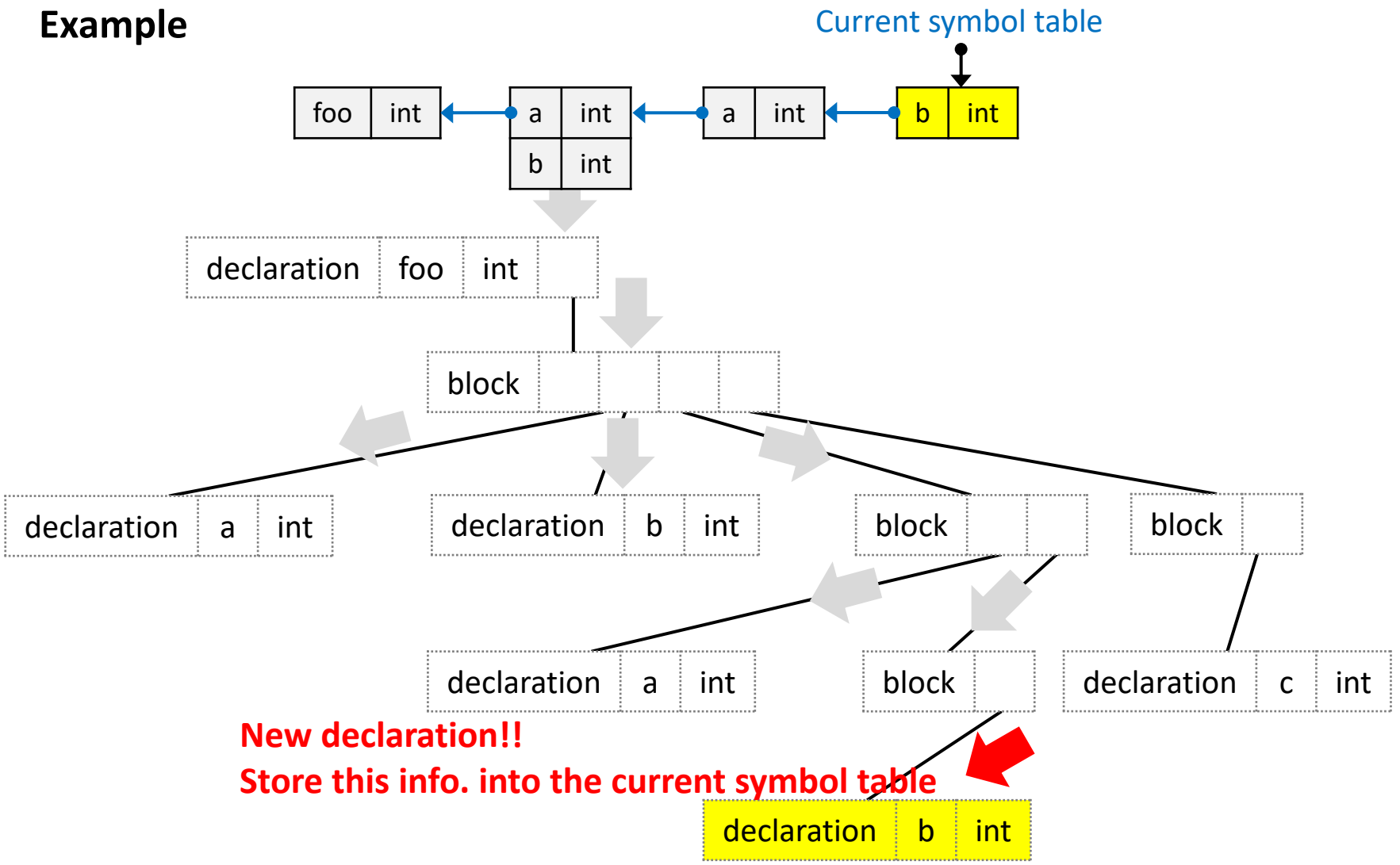
Implementation of scope checking

Example



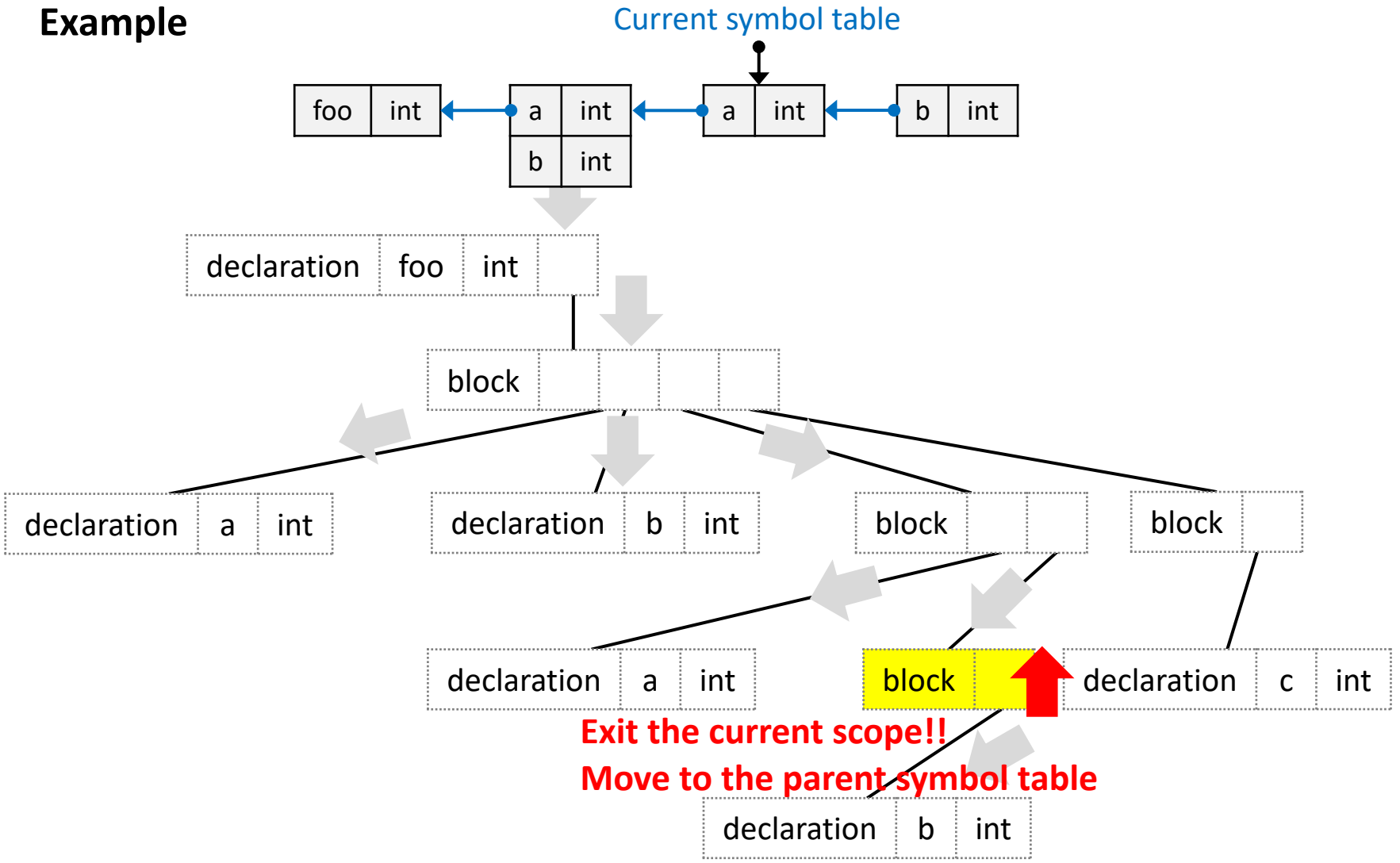
Implementation of scope checking

Example



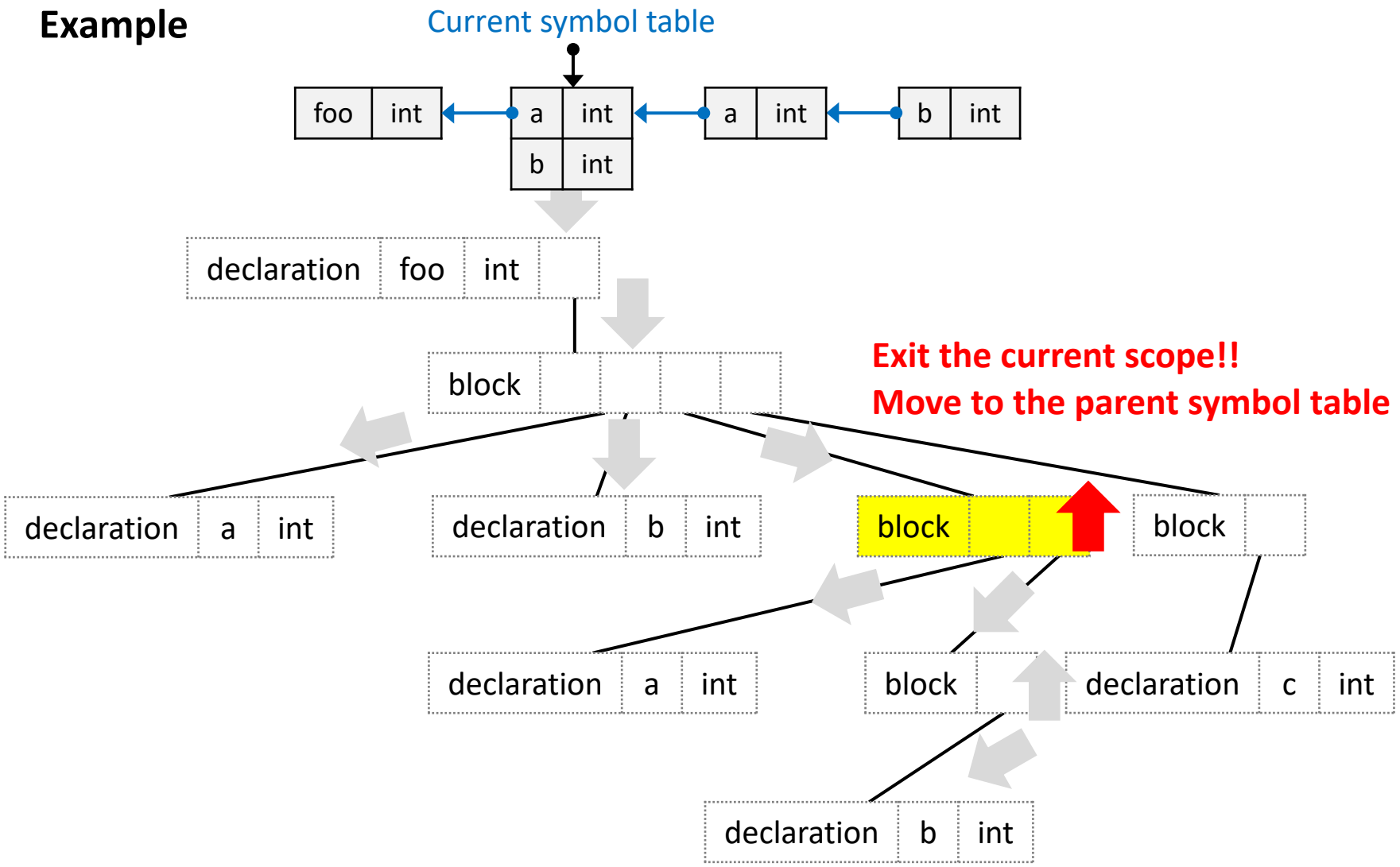
Implementation of scope checking

Example



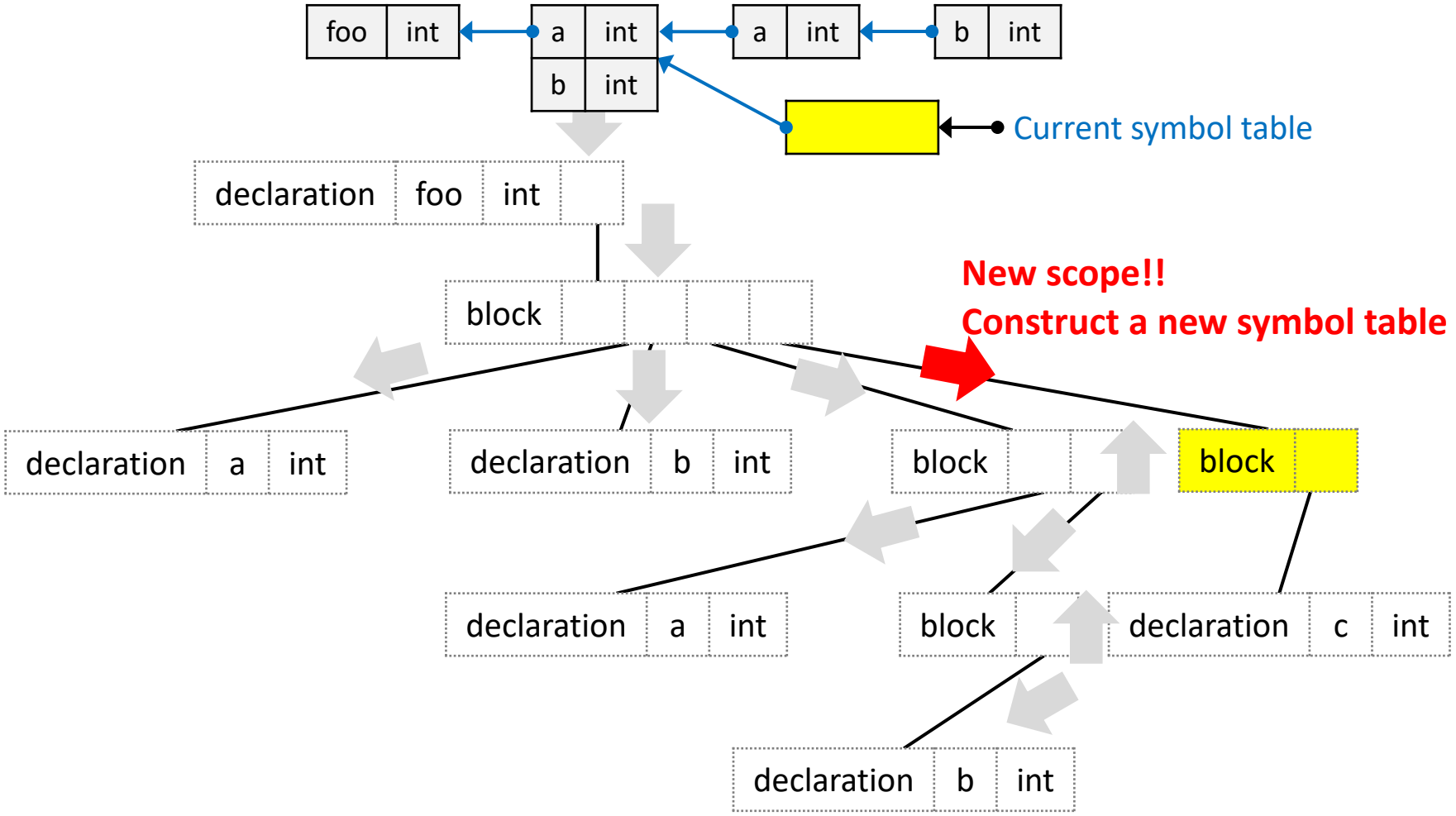
Implementation of scope checking

Example



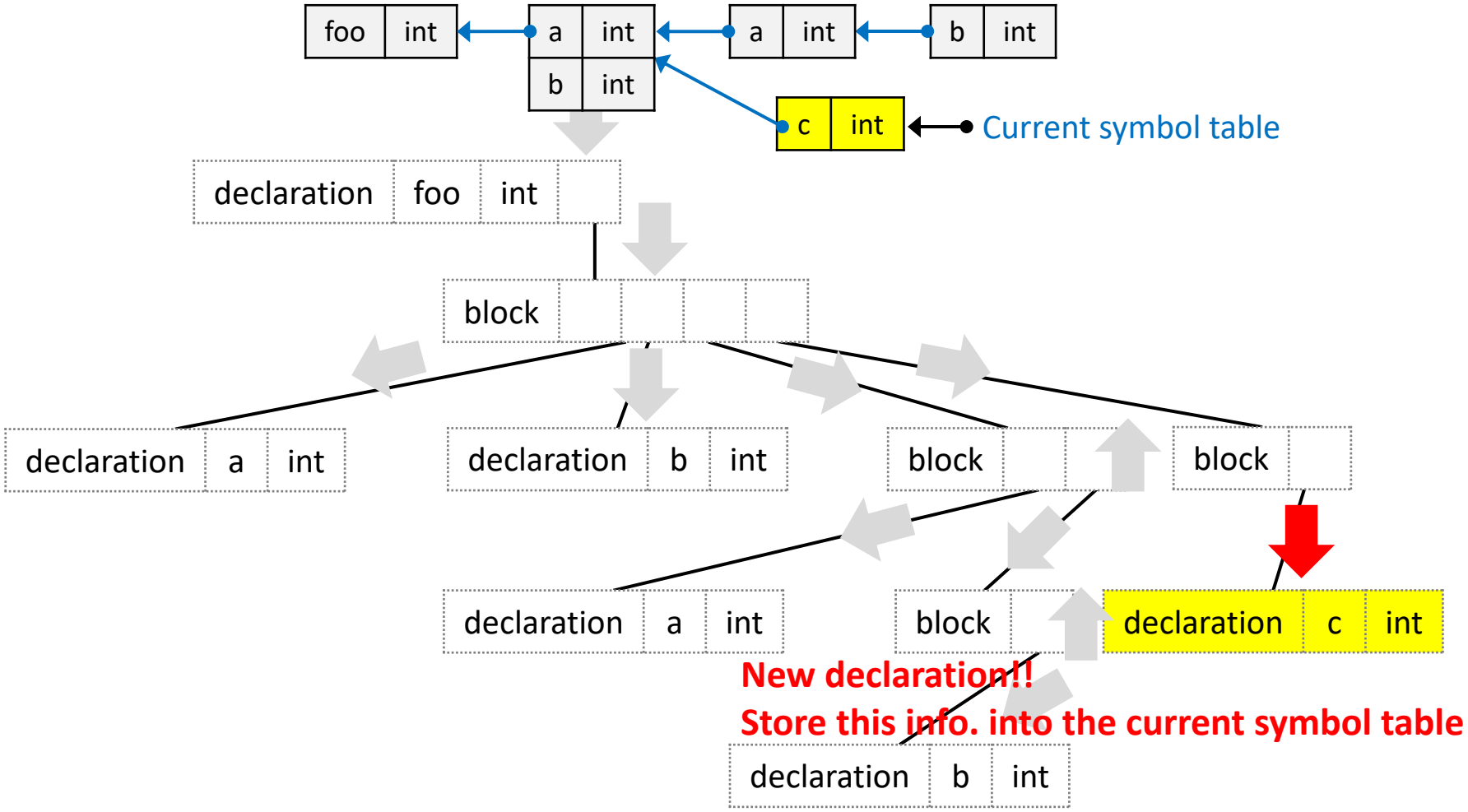
Implementation of scope checking

Example



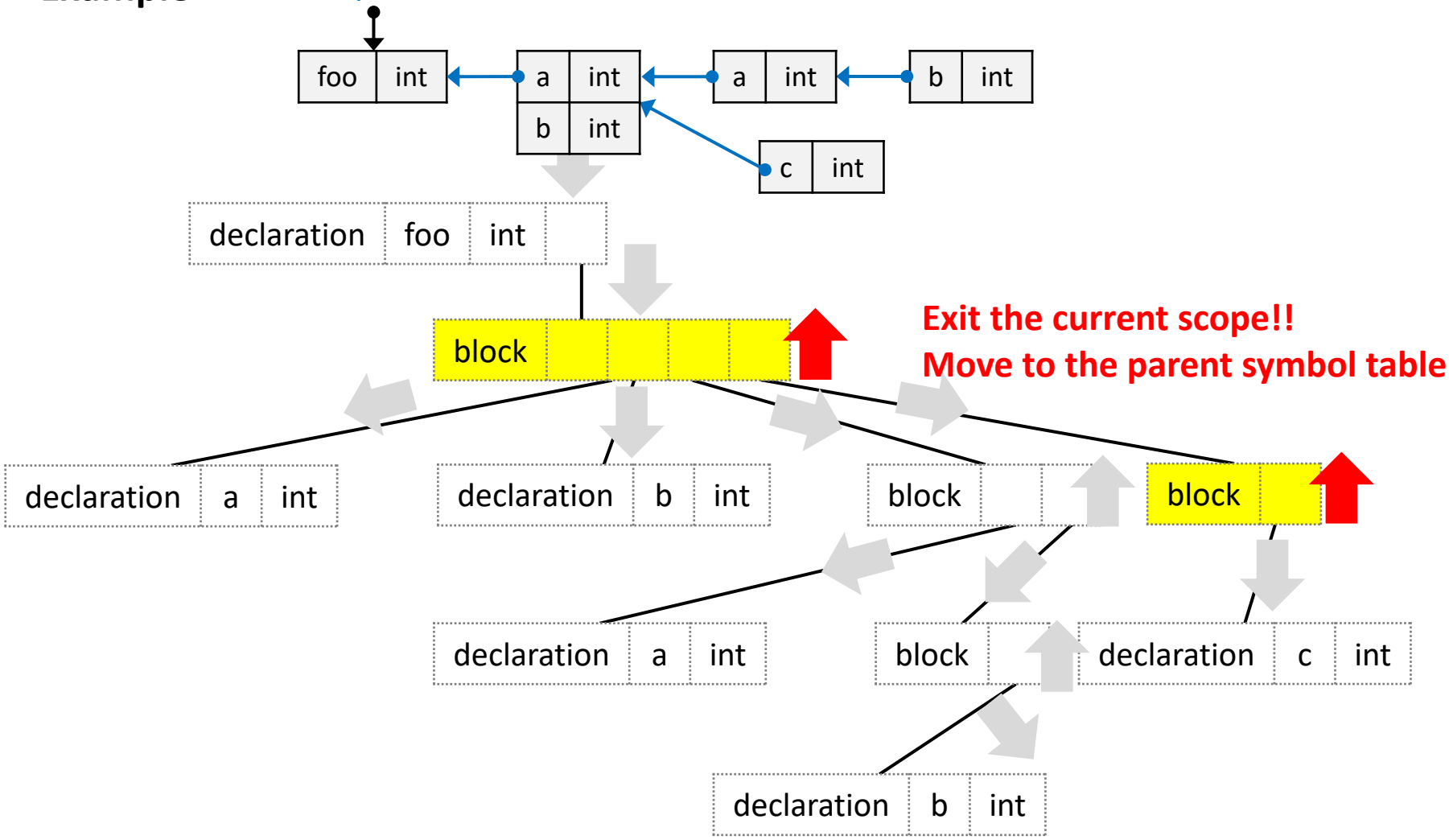
Implementation of scope checking

Example



Implementation of scope checking

Example Current symbol table



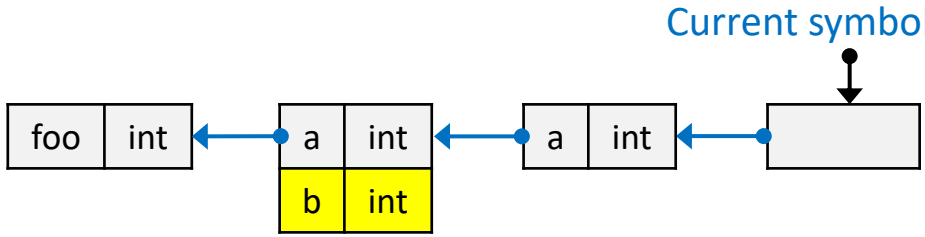
Implementation of scope checking

While traveling AST

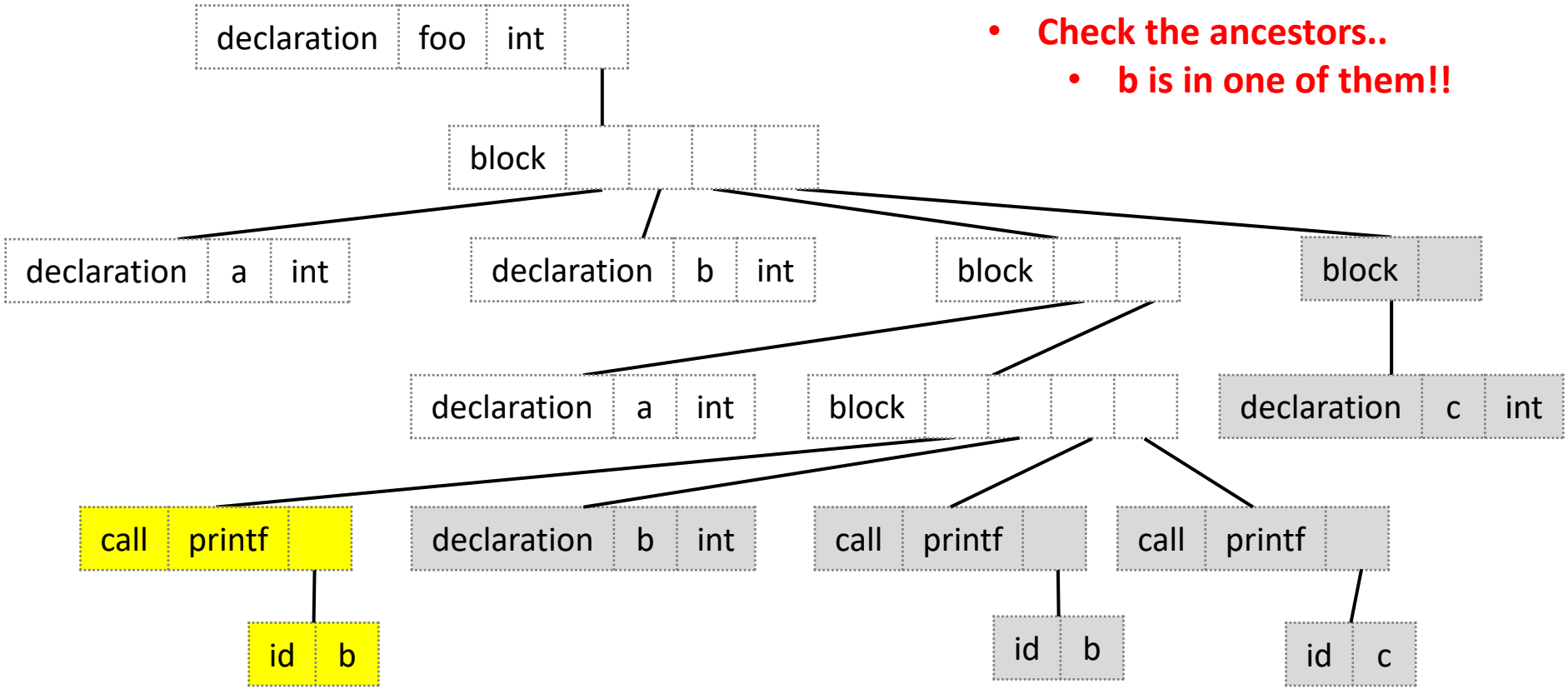
1. Construct a symbol table for each scope, describing a nesting structure
2. Update the symbol table with information about what identifiers are in the scope
3. **When an identifier is used, find the identifier in the current symbol table**
   - if not exist, moves to its parent table and find the identifier in the table
   - Repeat this process until the identifier is detected or there is no more parent table

```
int foo() {
    int a = 0;
    int b = 1;
    {
        int a = 2;
        {
            printf("%d\n", b);

            int b = 3;

            printf("%d\n", b);

            printf("%d\n", c);
        }
    }
    {
        int c = 4;
    }
}
```

Implementation of scope checking



- **b is not in the current table**
- **Check the ancestors..**
  - **b is in one of them!!**



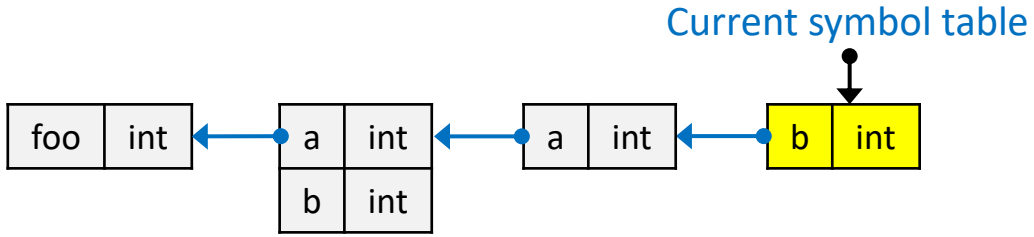
Implementation of scope checking

While traveling AST

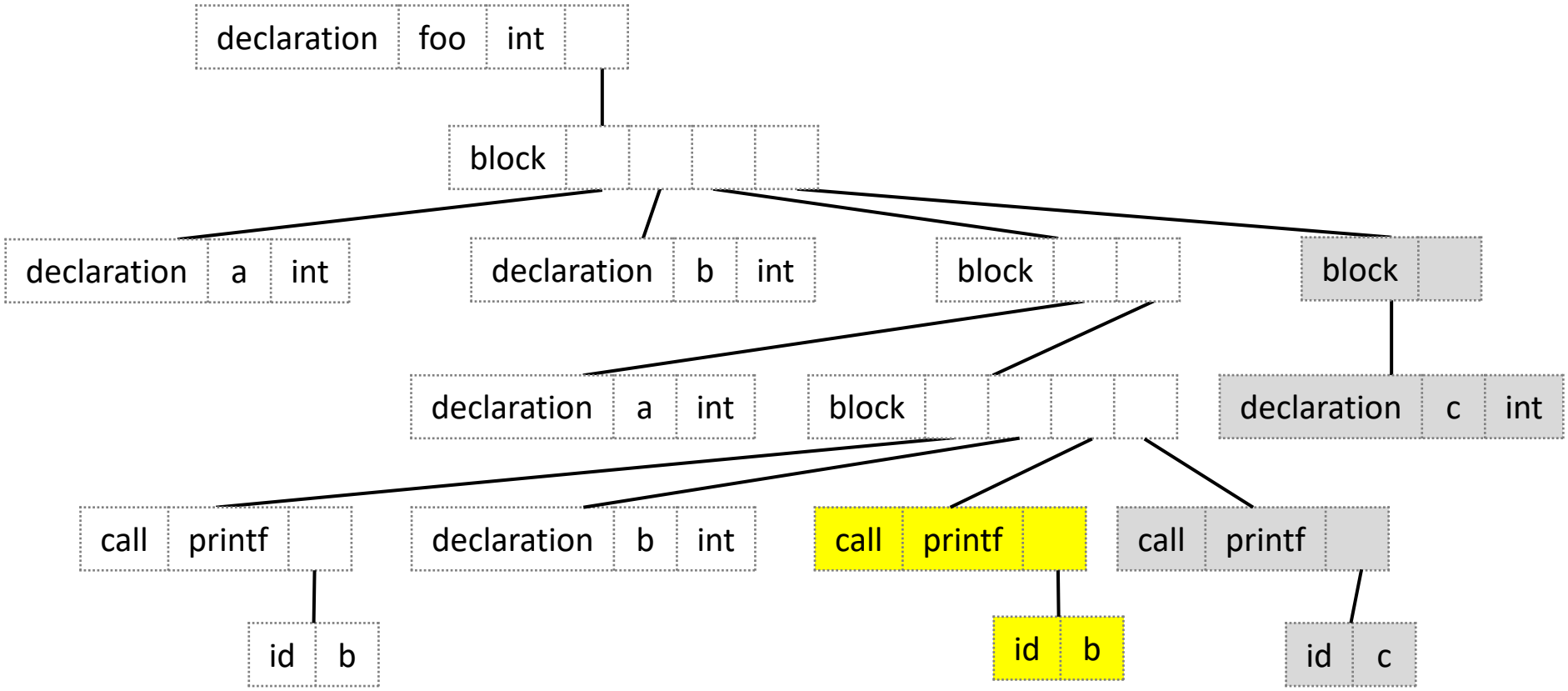
1. Construct a symbol table for each scope, describing a nesting structure
2. Update the symbol table with information about what identifiers are in the scope
3. **When an identifier is used, find the identifier in the current symbol table**
   - if not exist, moves to its parent table and find the identifier in the table
   - Repeat this process until the identifier is detected or there is no more parent table

```
int foo() {
    int a = 0;
    int b = 1;
    {
        int a = 2;
        {
            printf("%d\n", b);

            int b = 3;

            printf("%d\n", b);

            printf("%d\n", c);
        }
    }
    {
        int c = 4;
    }
}
```

Implementation of scope checking



- **b is in the current table!!**



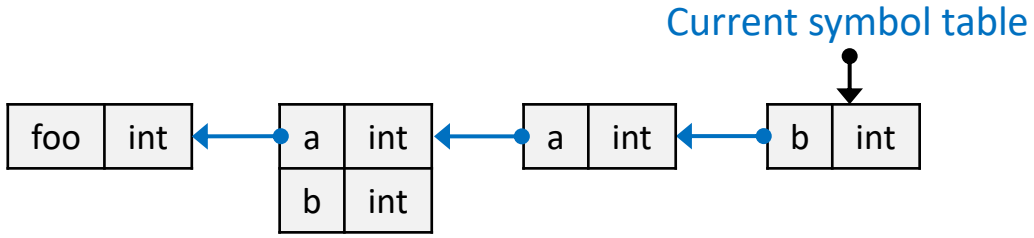
Implementation of scope checking

While traveling AST

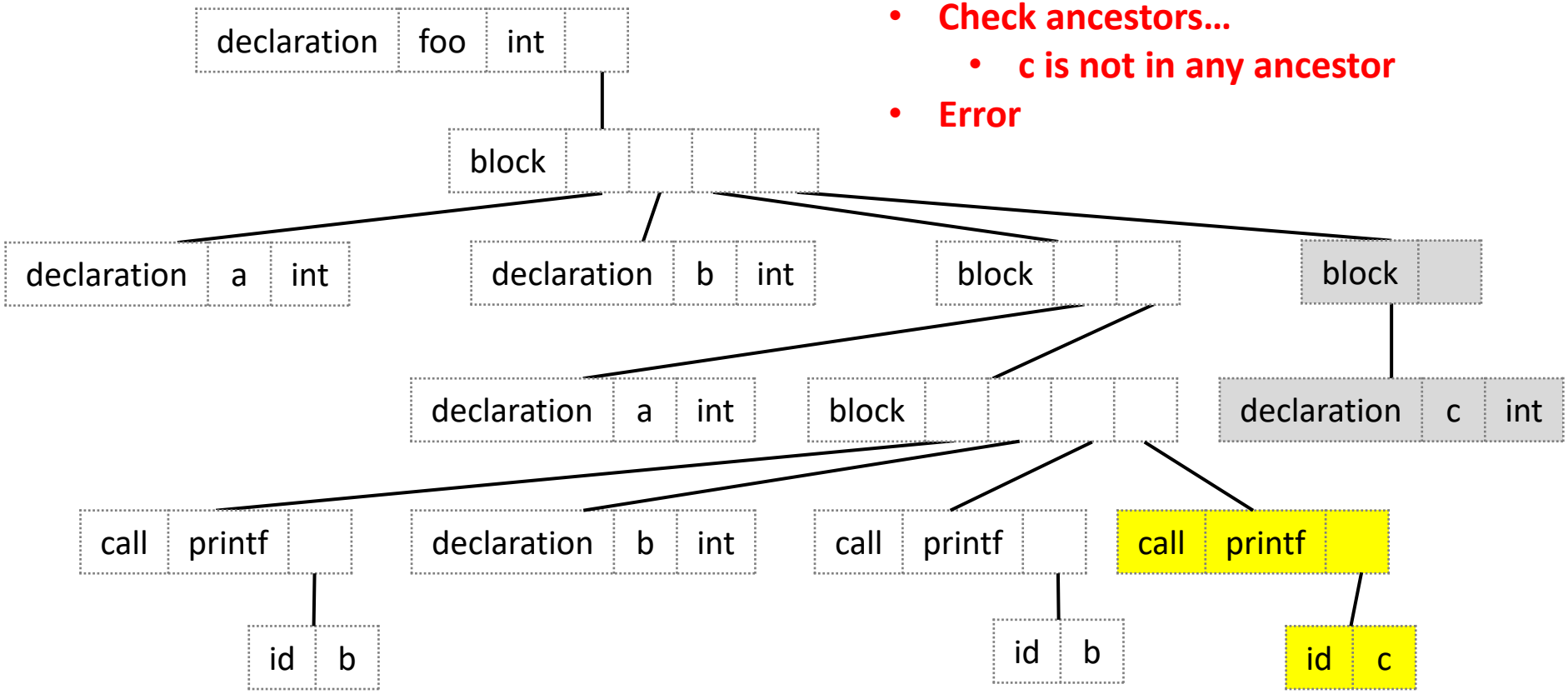
1. Construct a symbol table for each scope, describing a nesting structure
2. Update the symbol table with information about what identifiers are in the scope
3. **When an identifier is used, find the identifier in the current symbol table**
   - if not exist, moves to its parent table and find the identifier in the table
   - Repeat this process until the identifier is detected or there is no more parent table

```
int foo() {
    int a = 0;
    int b = 1;
    {
        int a = 2;
        {
            printf("%d\n", b);

            int b = 3;

            printf("%d\n", b);

            printf("%d\n", c);
        }
    }
    {
        int c = 4;
    }
}
```

Implementation of scope checking



- c is not in the current table
- Check ancestors…
  - c is not in any ancestor
- Error



Implementation of scope checking

While traveling AST

1. Construct a symbol table for each scope, describing a nesting structure
2. Update the symbol table with information about what identifiers are in the scope
3. When an identifier is used, find the identifier in the current symbol table
   - if not exist, moves to its parent table and find the identifier in the table
   - Repeat this process until the identifier is detected or there is no more parent table

```
1  class Test {
2  public:
3      void bar() {
4          foo();
5          return;
6      }
7
8      void foo() {
9          printf("foo");
10         return;
11     }
12
13 };
```

How to check function calls/class name used before declaration????

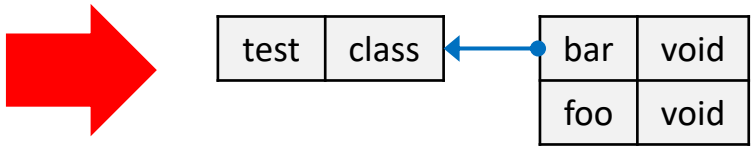
Implementation of scope checking

How to support function calls/class name used before declaration????

Solution: Multi-pass scope checking

- Pass 1: Gather information about all function / class names

```
1  class Test {
2  public:
3      void bar() {
4          foo();
5          return;
6      }
7
8      void foo() {
9          printf("foo");
10         return;
11     }
12
13 };
```



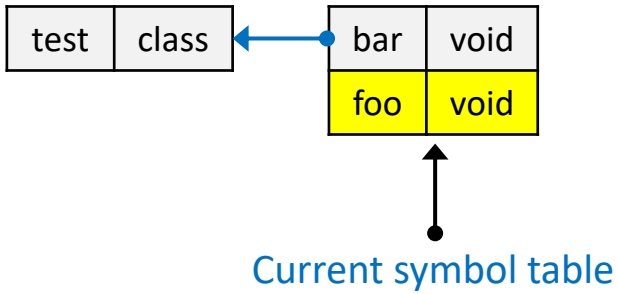
Implementation of scope checking

How to support function calls/class name used before declaration????

Solution: Multi-pass scope checking

- Pass 2: Do scope checking

```
1  class Test {
2  public:
3      void bar() {
4          foo();
5          return;
6      }
7
8      void foo() {
9          printf("foo");
10         return;
11     }
12
13 };
```



Implementation of scope checking

While traveling AST

1. Construct a symbol table for each scope, describing a nesting structure
2. Update the symbol table with information about what identifiers are in the scope
3. When an identifier is used, find the identifier in the current symbol table
   - if not exist, moves to its parent table and find the identifier in the table
   - Repeat this process until the identifier is detected or there is no more parent table

```
1  class Parent {
2  public:
3      void foo() {
4          printf("foo");
5          return;
6      }
7  };
8
9
10 class Test : public Parent{
11 public:
12     void bar() {
13         foo();
14         return;
15     }
16 };
```

Q. How to check inherited variables/functions???

Summary: scope checking

The scope of an identifier is the portion of a program

in which the identifier can be accessed

- Scope matches identifier declarations with uses
- The same identifier may refer to different things in different scopes

For scope check, we use the most-closely nested scope rule

1. Construct a symbol table for each scope, describing a nesting structure
2. Update the symbol table with information about what identifiers are in the scope
3. When an identifier is used, find the identifier in the current symbol table

Semantic analysis usually requires multiple (probably more than two) passes

To allow to use function calls / class names before declaration