COMP 520 - Compilers

Lecture 09 – Type Checking Theory and Contextual Analysis in PA3

Midterm 1 soon!

- Make sure you finish PA2 on time.
- Open notes, open lecture slides, open course website.
- No other internet access.

- Can use your IDE to read code, but cannot write any new code. Cannot run anything in IDE either.
  - Syntax highlighting is the main goal.

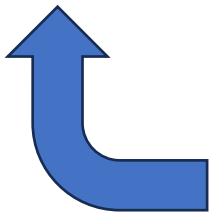
Midterm 1 soon! (2)

- The reason midterm 1 is so early in the semester is to gauge how you are doing.

- By looking at your midterm 1 results, you can determine how the rest of the semester will go. PA3 is difficult, PA4 is even more so.

More than one LL

```
this(.id)*(= Expr ;| [Expr] = Expr ; | ( ArgList? ) ; )
| boolean id = Expr ;
| id ( ((.id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ) )
    | [] id = Expr ; )
```



How problematic is this really?
Especially given that Lexers need to be simple state machines.

More than one LL(2)

```
id ( ( (. id)*( = Expr ; | [ Expr ] = Expr ; | ( ArgList? ) ; ) )
       | [] id = Expr ; )
```

```
id ( . id(. id)*(=Expr;|[Expr]=Expr;|(ArgList?);)
       | =Expr; | (ArgList?);
       | [(] id = Expr;|Expr] = Expr; )
```

More than one LL(3)

```
id ( .id(.id)*(=Expr;|[Expr]=Expr;|(ArgList?);)
     |=Expr; | (ArgList?);
     |[(] id =Expr;|Expr] =Expr; )
```

Drawback: A lot more symbols and repeated sequences. (Shown is only one of the 3 options in the first choice)

More than one LL(3)

```
this(.id)*(= Expr ;| [Expr] = Expr ; | ( ArgList? ) ; )
| boolean id = Expr ;
| id ( .id(.id)*(=Expr;|[Expr]=Expr;|(ArgList?);)
      |=Expr; | (ArgList?);
      | [(] id = Expr;|Expr] = Expr;    )
```

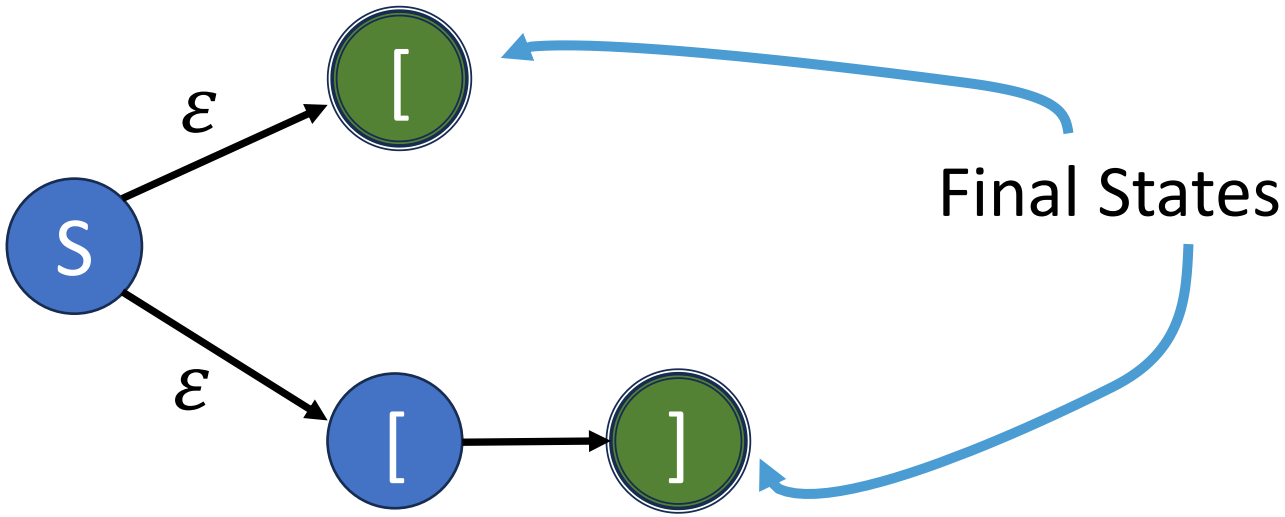
Drawback: A lot more symbols and repeated sequences.

Parsing, Lexing, and []

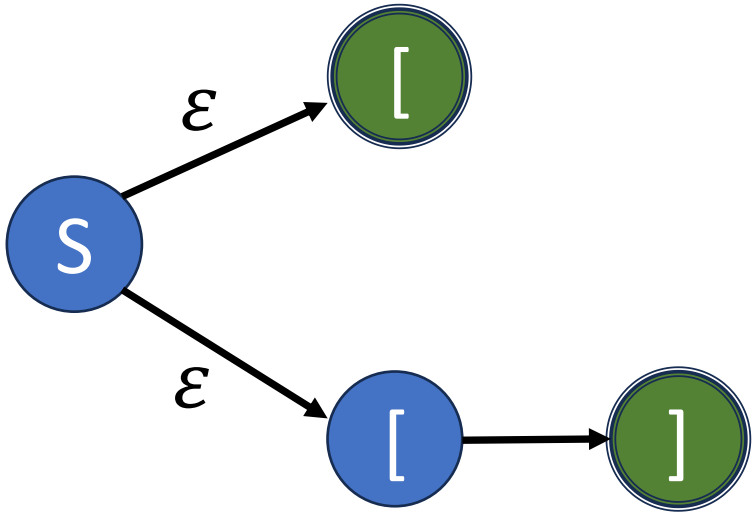
- Two ways to represent those rules as LL(1)
- If going for the simpler one, then need to scan [] as a token by itself.

- Question: how simple will the Lexer DFA be?

ε-closure and $\hat{\delta}$, Start with NFA



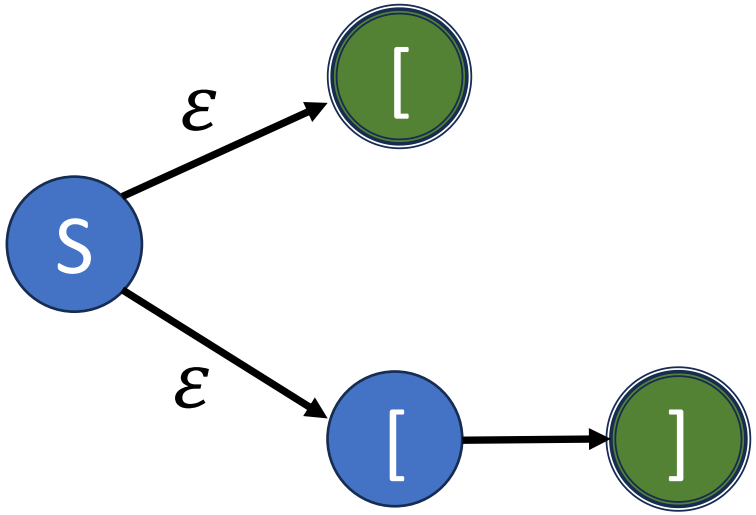
ε-closure and $\hat{\delta}$, Good enough for DFA?



Final State

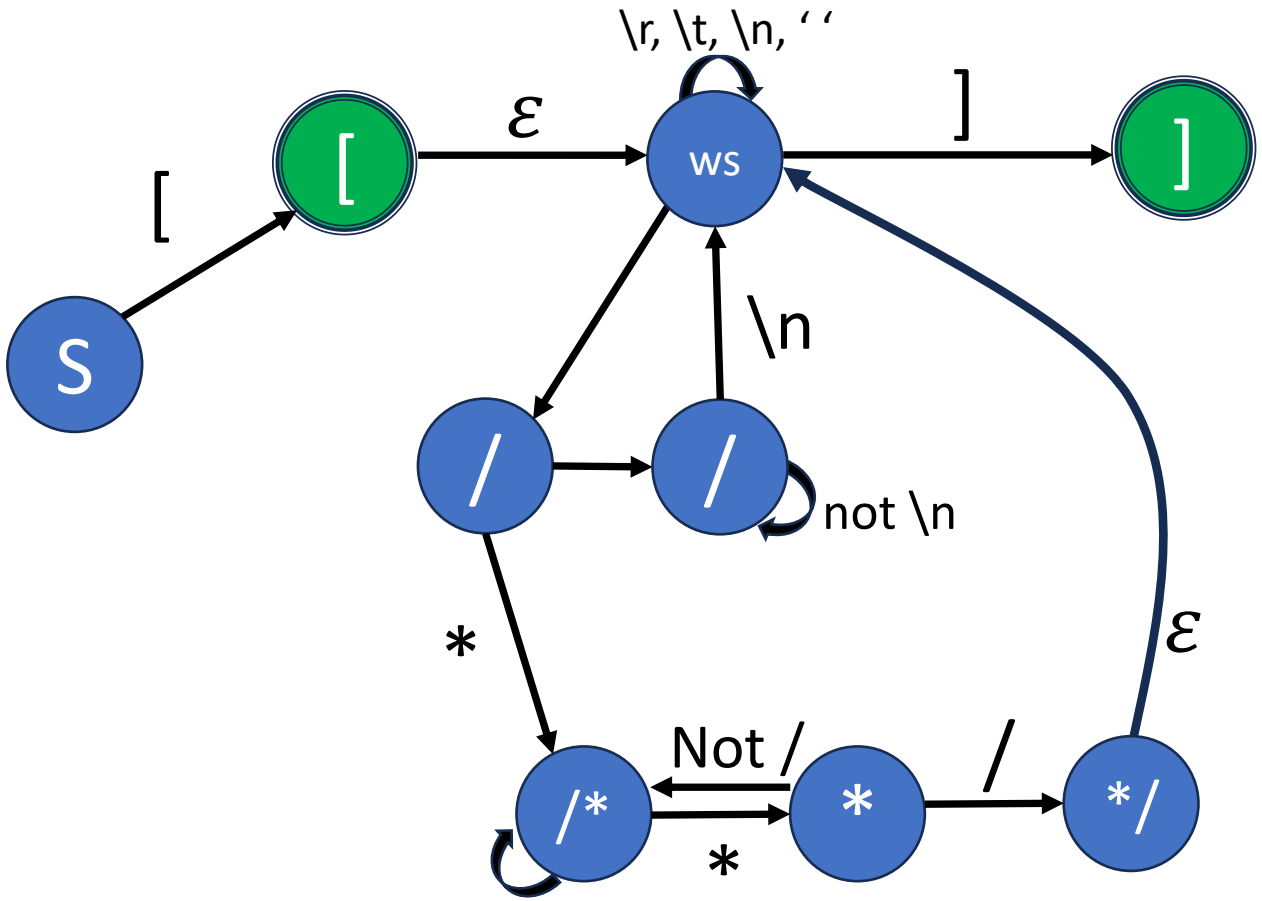
Any problems?

ε-closure and $\hat{\delta}$- Cannot use NFA

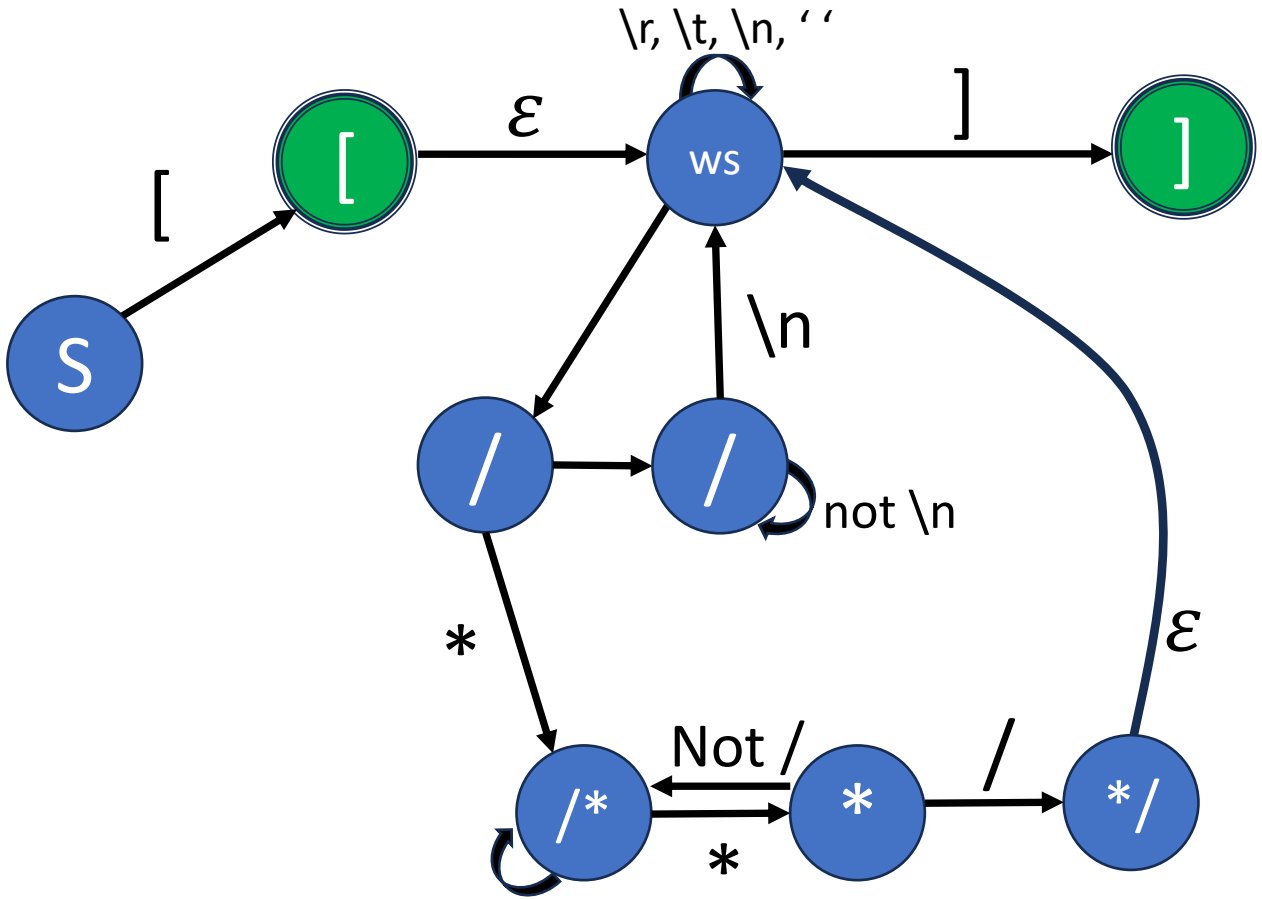


Non-deterministic, therefore not scannable. (And whitespace issues)

ε-closure and $\hat{\delta}$ (2)

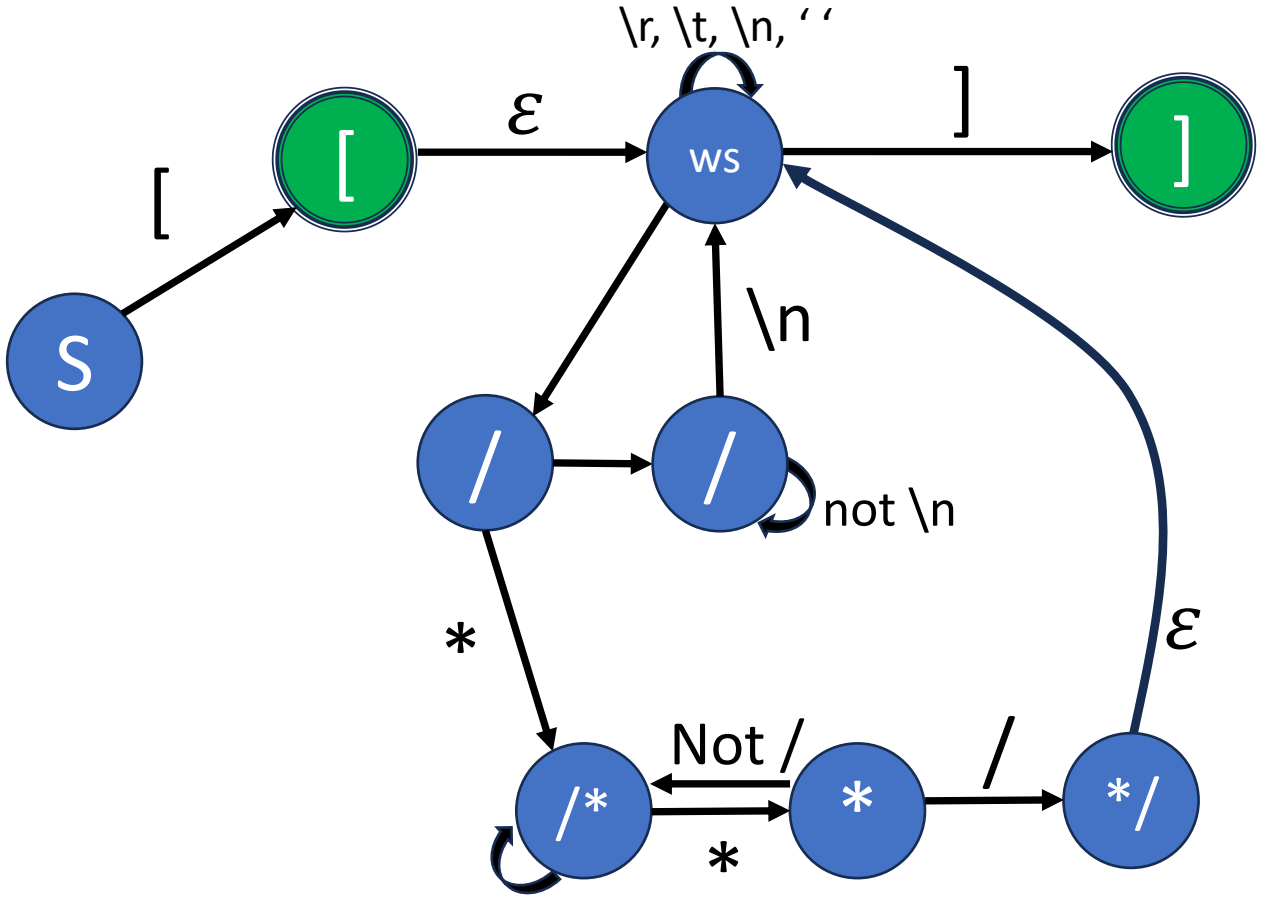


ε-closure and $\hat{\delta}$ (3)



What happens if the input is: [/8

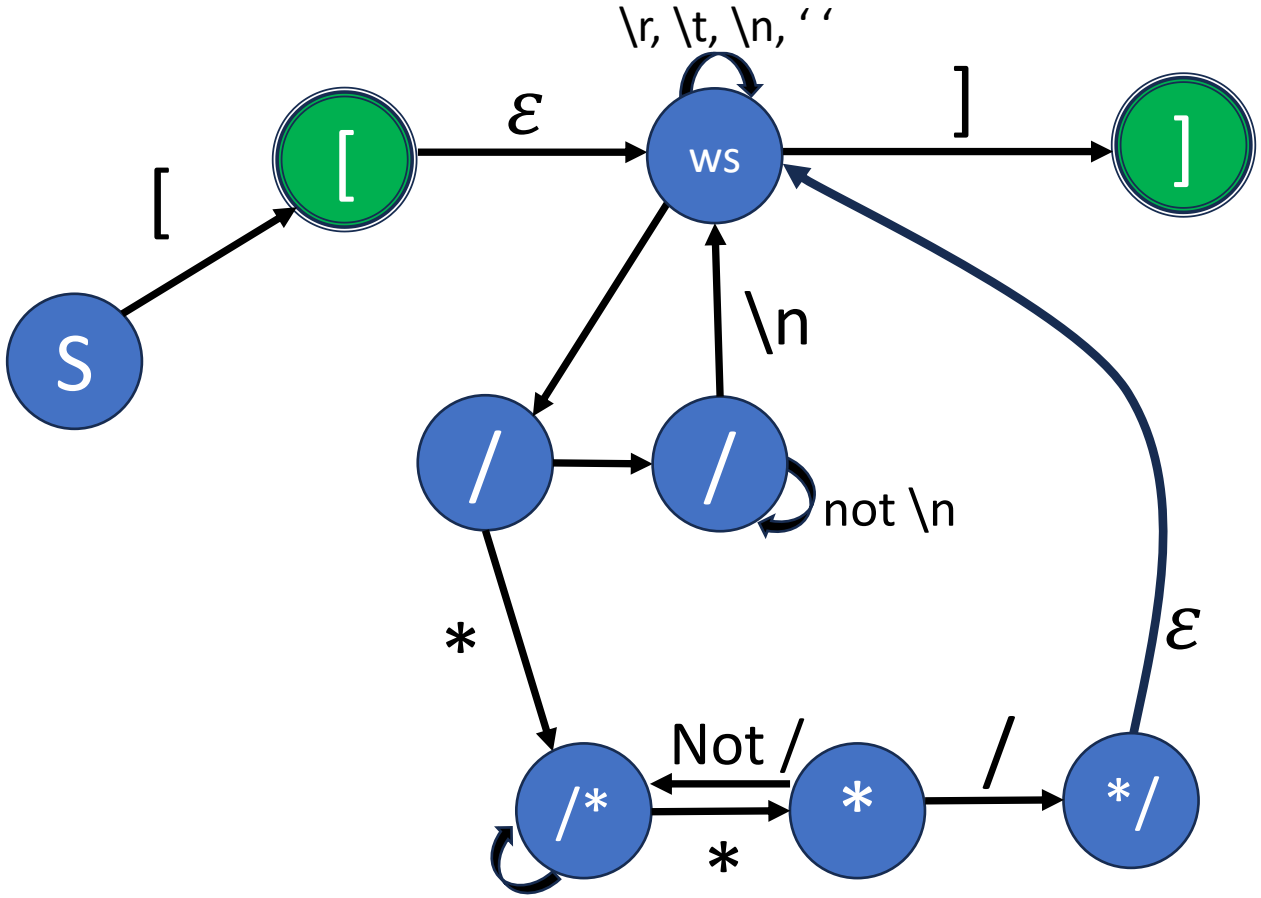
ε-closure and $\hat{\delta}$ (4)



ε-closure method of NFA will "**rewind**" back to the latest known final state.

How practical is rewinding?

ε-closure and $\hat{\delta}$ (5)



ε-closure method of NFA will "**rewind**" back to the latest known final state.

How practical is rewinding? So long as it isn't more than one Token, not a problem (keep the next token in your accumulated text).

Type Checking

Type-Checking Table

- For every type, should there be an entry where…

$$\alpha \times \beta \rightarrow \gamma$$

E.g., int × int → int

- Is this enough?
- Will every int op int yield an int?

Type-Checking Table (2)

- For every type, should there be an entry where…

$$\alpha \times \beta \rightarrow \gamma$$

E.g., int × int → int

- What about 3 == 2?
  In that case, int × int → boolean

Type-Checking Table (3)

Therefore, Type-Checking is a 3 input, 1 output table:

$$(\beta \times \alpha \times \text{op}) \rightarrow \gamma$$

E.g., int × int × {+,−,*,/} → int

int × int × {==, !=, >, >=, <, <=} → boolean

Operator Overloading

- Operator Overloading: give custom definitions to ops

- Example, consider a "MyData" object that can accumulate integers and sorts them.
- In C++, can create **a method** that defines:

$$\text{MyData} \times \text{int} \times \{+\} \rightarrow \text{MyData}$$

Where adding integers to MyData types will call **that method**.

Bad Types

- When an entry is missing, then the resultant type should be a specially designated type that is compatible with EVERY type.

- This type, "**ErrorType**", prevents type-checking errors from filling up your compiler's error log.

Bad Types (2)

- This type, "**ErrorType**", prevents type-checking errors from filling up your compiler's error log.
- Consider miniJava: (myObj + 3) + 5;

ClassType × int × {+} → ErrorType (**report error here**)

ErrorType × int × {+} → ErrorType (no need to report)

Second type check was valid.

Bad Types (3)

- Generalization:

$(\{\text{ErrorType}\} \times \text{AnyType} \times \text{AnyOp}) \rightarrow \text{ErrorType}$

$(\text{AnyType} \times \{\text{ErrorType}\} \times \text{AnyOp}) \rightarrow \text{ErrorType}$

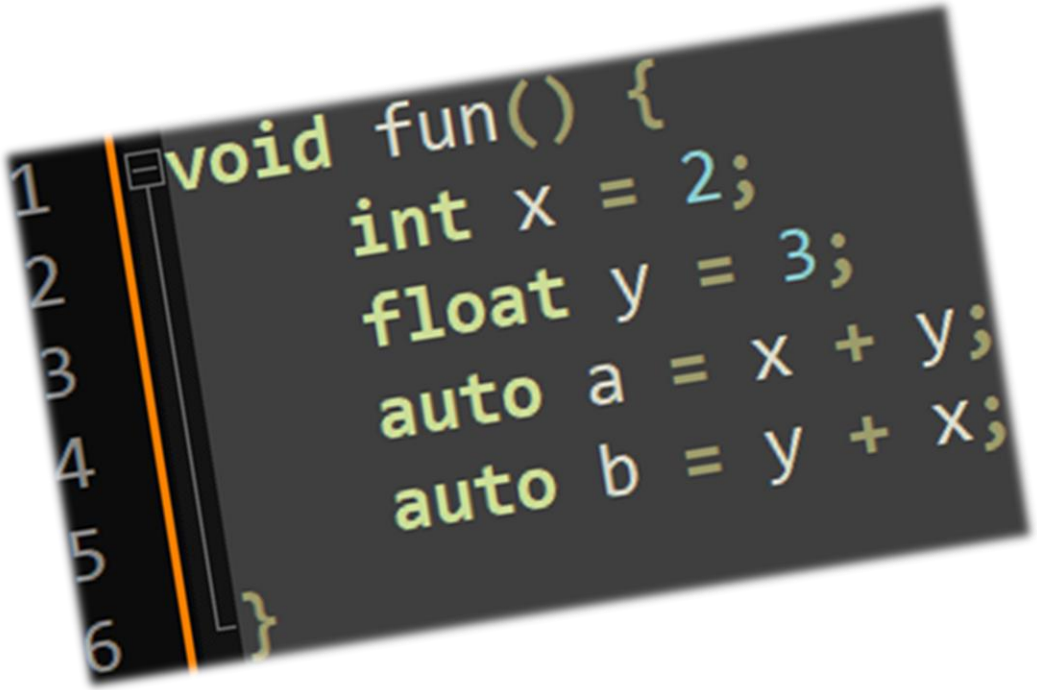
$((\text{TypeA},\text{TypeB},\text{Op}) \notin \text{TypeTable}) \rightarrow (\text{Result}=\text{ErrorType})$

$((\text{TypeA},\text{TypeB},\text{Op}) \in \text{TypeTable}) \rightarrow$

$(\text{Result} = \text{GetEntry}(\text{TypeA},\text{TypeB},\text{Op}))$

Does Order Matter?

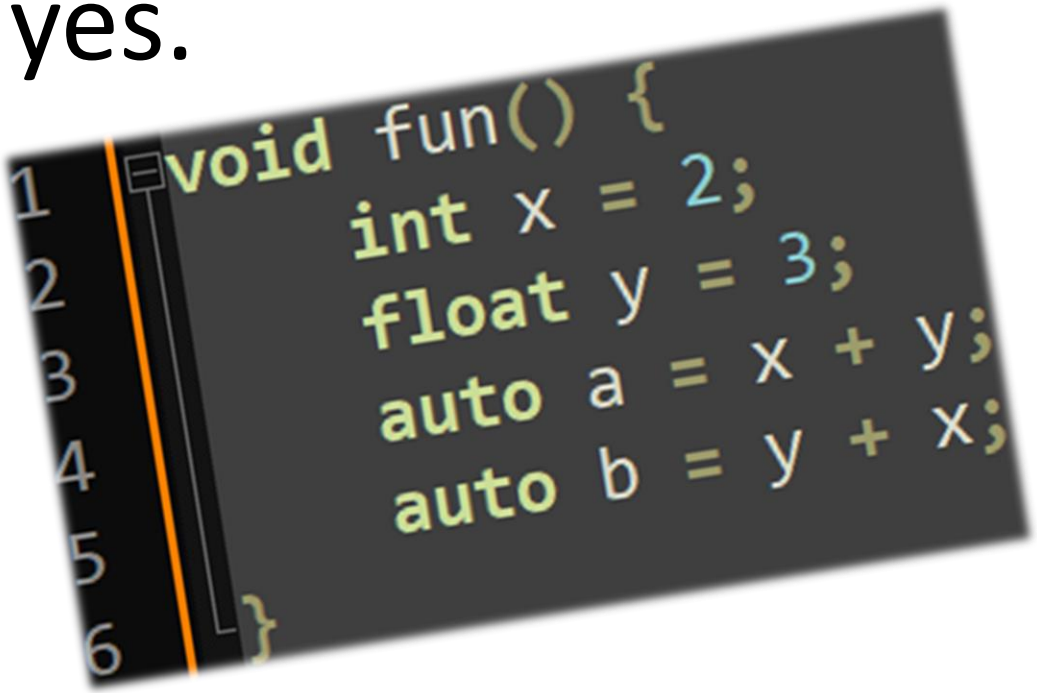
- Is A × B the same as B × A?



Does Order Matter?

- Is A × B the same as B × A?
- Previous definitions state yes.

- Your thoughts?



Does Order Matter?

- Is $A \times B$ the same as $B \times A$?
- Previous definitions state yes.

- Not for miniJava, but yes for some languages.
- Why?

Order – Language Specifications

- Language may specify that order can matter
- C++ can specify order in the program itself:

More Totally Sane Code

- Why?

```
class A {
public:
    int x;
    A(int x) : x(x) {}
};

A operator+(const A& a, int b) {
    return A(a.x + b);
}

A operator+(int a, const A& b) {
    return A(b.x * a);
}

void main() {
    A a(5);
    A b = a + 3;
    A c = 3 + a;
    printf("%d %d %d\n", a.x, b.x, c.x);
}
```

More Totally Sane Code

- Why?
- You could make your compiler “keep” the first type if you wanted.
- For example: 3 + 3.2 * 4.4;
- Specify: int × float … → int
- Specify: float × int … → float

```
class A {
public:
    int x;
    A(int x) : x(x) {}
};

A operator+(const A& a, int b) {
    return A(a.x + b);
}

A operator+(int a, const A& b) {
    return A(b.x * a);
}

void main() {
    A a(5);
    A b = a + 3;
    A c = 3 + a;
    printf("%d %d %d\n", a.x, b.x, c.x);
}
```

Type-Checking Table

- Thus, how you construct your type checking table depends on the language.

TypeCasting Tables

TypeCasting / Type Conversions

- Typecasting (or the conversion of a type) does not have an operator.
- Formally:

$$\alpha \times \beta \rightarrow \alpha$$

- Thus, the table for typecasting is:
  (TypeA,TypeB) → TypeA, and some skip the → entirely
- E.g. ((int)3.4f) → 3 (an int), which is (int,float)→int

TypeCasting / Type Conversions (2)

$$\alpha \times \beta \rightarrow \alpha$$

- The table for typecasting takes an input of (TypeA,TypeB), result is just TypeA.

What is the exception?

- If (x,y) ∉ Table, then this is a type checking error.
- The table usually has a result just to make the code easier to write up.

Automatic Conversions

- boolean is just a 0 or 1, so we could:

int × boolean → int

boolean × int → boolean

- This is also described as:

int → {int, boolean}

(depends on how you construct the table)

Automatic Conversions (2)

- Java has more automatic conversions:

String × Enum → String

- The Enum will automatically become a string that represents the text of that enum entry.

Automatic Conversions (3)

- Automatic conversions are AUTOMATIC
- Note:

```
private int somefun( int a ) { … }
…
somefun( true );
```

… does not require explicitly typecasting `(int)true`

Automatic Conversions (4)

- Java has more automatic conversions:

String × Enum → String

- Combine this with: String × String × {+} → String (concatenation operation) and we can add enum to strings!
- E.g.: `myStr = myStr + someEnumVar;`

Automatic Type Casting Note

- From the previous example, we can see that if there does not exist a type-check entry for:

$$\alpha \times \beta \times \text{Op}$$

… then we have to check all possible automatic typecasting for both α and β.

If there exists conflicts, manual typecasting is enforced.

Manual Typecasting

- Consider the following:

(String,String,+)→String

(Integer,Integer,+)→Integer

TypeA a; TypeB b;

And both TypeA and TypeB can be typecasted with:

(String,TypeA), (String,TypeB), (Integer,TypeA), (Integer,TypeB)

What is the result type of a + b?

Manual Typecasting

- Consider the following:

(String,String,+)→String

(Integer,Integer,+)→Integer

TypeA a; TypeB b; both TypeA, TypeB → {String,Integer}

What is the result type of a + b?

Answer: Your compiler must require a or b to be manually typecasted to either String or Integer. (String)a + b;

Please note: miniJava

- miniJava does not have typecasting nor automatic conversions.
- miniJava does not allow boolean × int × Op

- This lecture is meant to strengthen your compiler theory, please see next section for PA3 specifics.

Programming Assignment 3

Identification and Type-Checking

You own the ASTs for PA3 and forward

- Can add, edit, remove any ASTs you want.

Strategy

- Two separate Visitor implementations
- First identification, then type-checking

Strategy (2)

- Two separate Visitor implementations
- First identification, then type-checking

- It is possible to do this in one AST traversal. Optional, is a PA5 extra credit item.

Identification Goal

- Every Identifier gets a "decl" field added, of type Declaration
- We want to locate where every identifier is declared.

- Could be a VarDecl, ParameterDecl, MemberDecl, ClassDecl

Identification Cache

- Identifier "x" only makes sense in context.
- Even if two identifiers' underlying text is the same, the declaration can be different when appearing in different parts of the code.

```
class A {
    int x;
}

class B {
    int x;
}
```

Both use "x" as the identifier,
but can only tell them apart in context.

Identification Cache (2)

- Identifier “x” only makes sense in context.
- Even if two identifiers’ underlying text is the same, the declaration can be different when appearing in different parts of the code.
- If we can resolve the Declaration, store it in that identifier’s decl field so we won’t have to traverse again.

What is: IDTable

- An object that internally stores a mapping from String to Declaration.

- `HashMap<String,Declaration> idTable;`

Scoped Identification (SI)

- Implement a `Stack<IDTable>`

(Where `IDTable` is `HashMap<String, Declaration>`)

- Level 0: ClassDecl
- Level 1: MemberDecl (FieldDecl, MethodDecl)
- Level 2+: LocalDecl (ParameterDecl, VarDecl)

SI methods

- openScope
- closeScope
- addDeclaration
- findDeclaration

openScope / closeScope

- openScope: Push a new IDTable onto the stack
- closeScope: Pop the top-most IDTable from the stack

- Question: When should you open/close a scope?

openScope / closeScope (2)

- openScope: Push a new IDTable onto the stack
- closeScope: Pop the top-most IDTable from the stack

- When should you open/close a scope?
  - Entering/Leaving a method's StatementList
  - Entering/Leaving a BlockStmt

addDeclaration / findDeclaration

- addDeclaration: Map a String (and optional context) to a Declaration. If this String exists on level 2+, then IDError should be thrown.

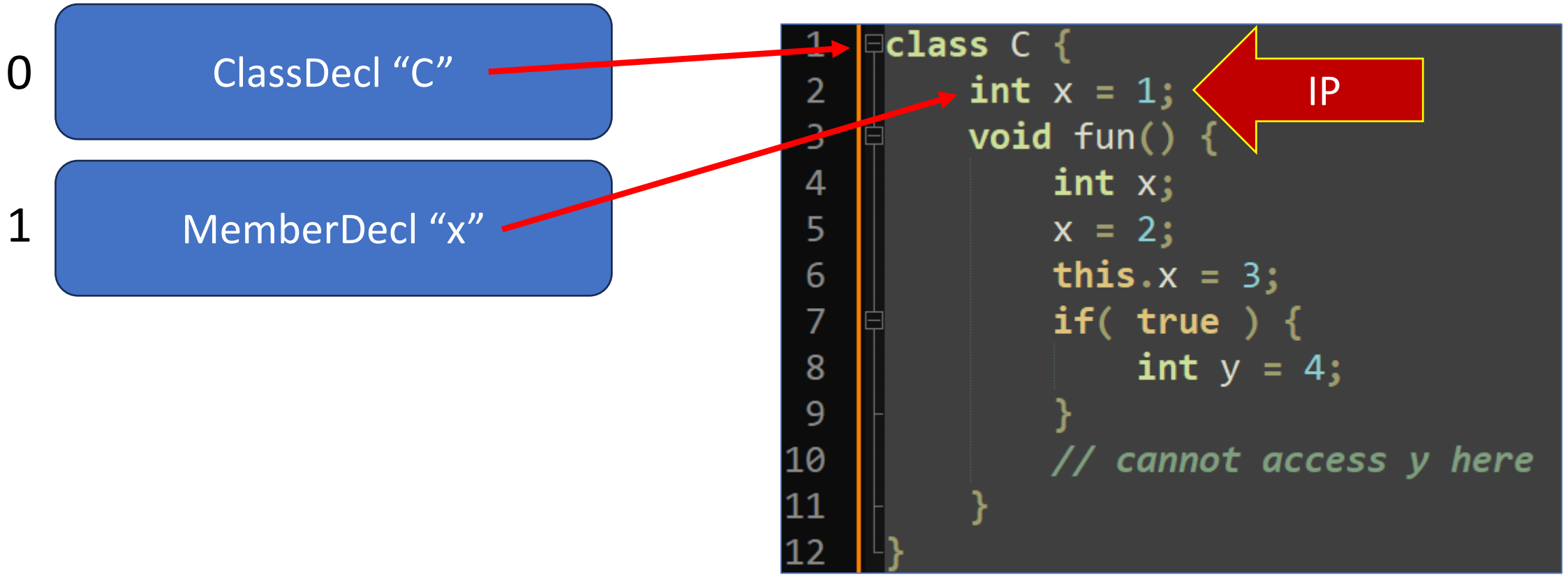
- findDeclaration: With a String (and optional context), find a Declaration starting from the largest indexed scope and then working down.

Identification Traversal – Method Body

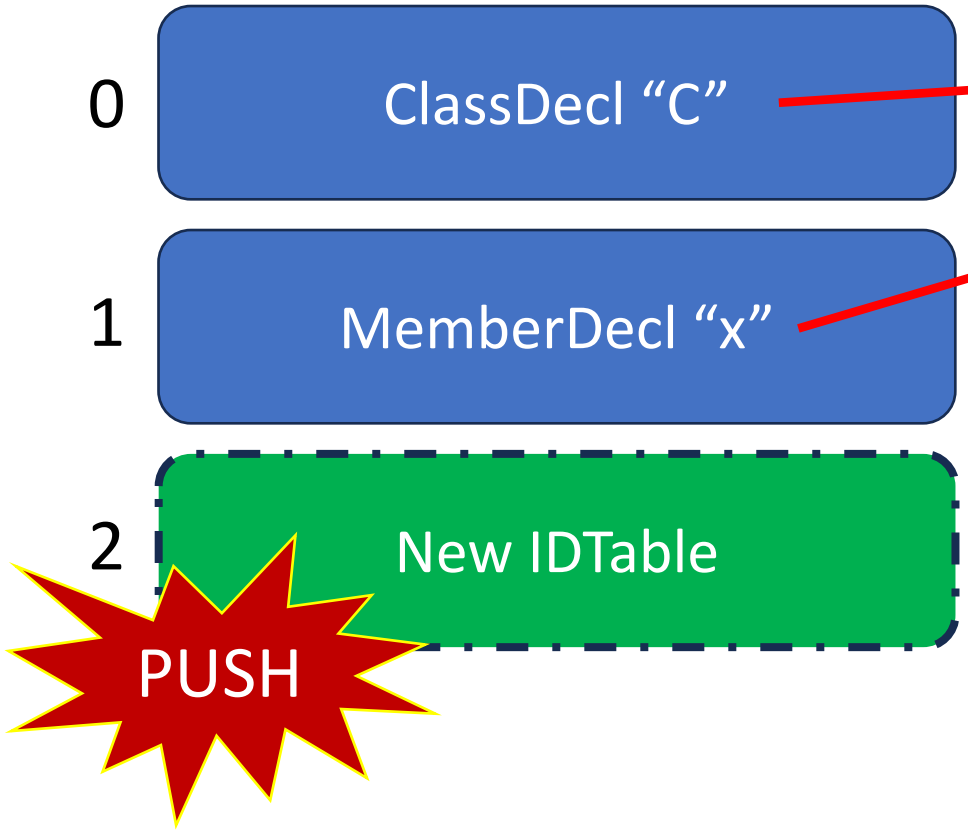
- In a method body, if you encounter a VarDeclStmt, visit the VarDecl and add that identifier to the **top-most IDTable** in your ScopedIdentification

- If that identifier already exists at scope level 2+, then that is an identification error!

Scoped Identification – Stack Visualization



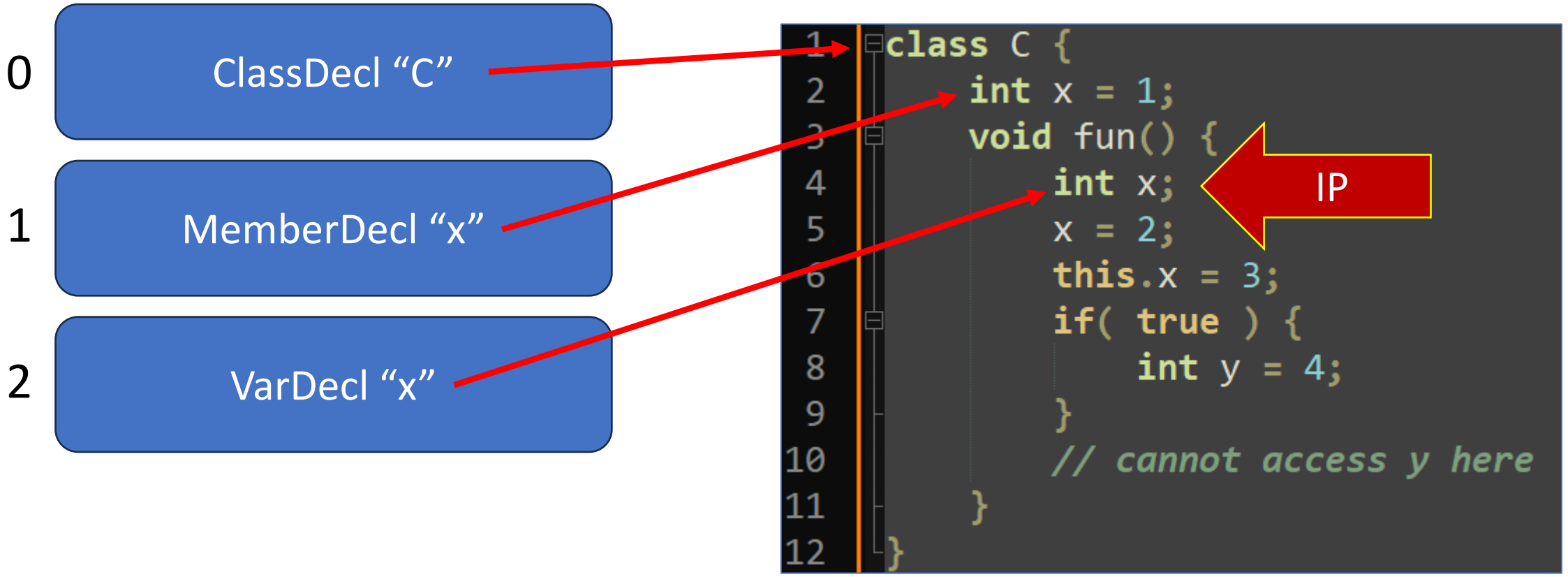
Scoped Identification – Stack Visualization



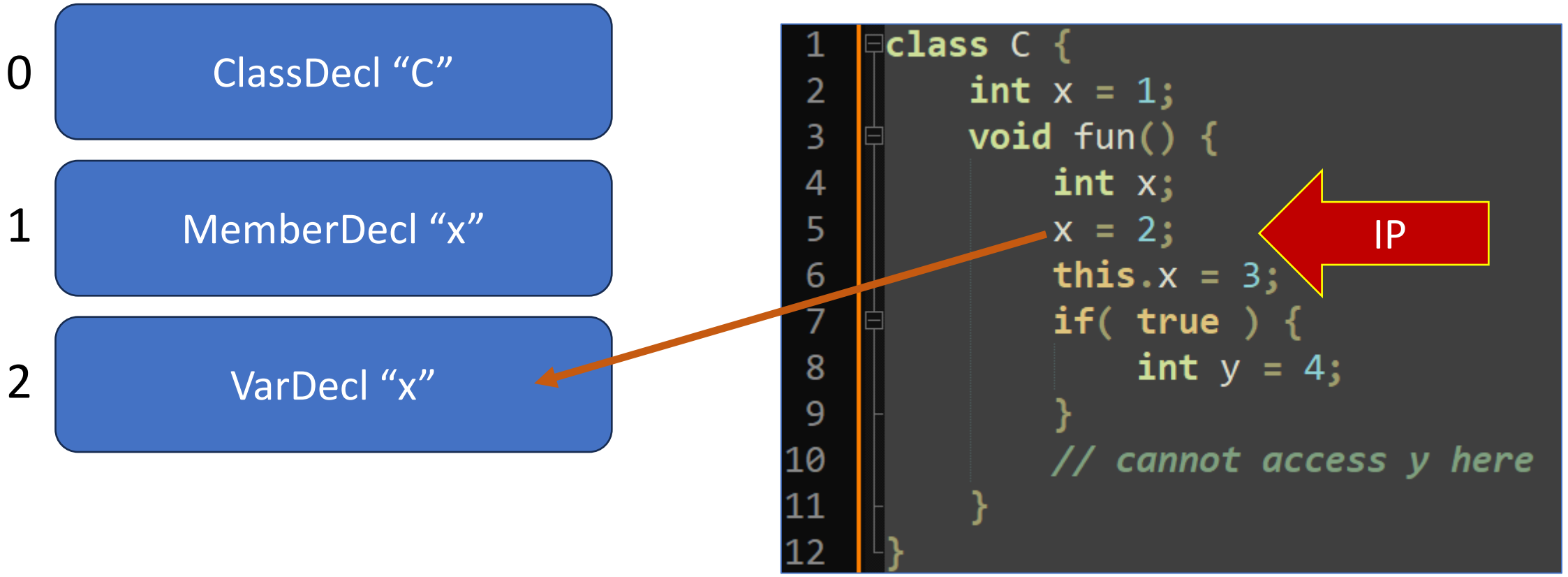

```
1  class C {
2      int x = 1;
3      void fun() {
4          int x;
5          x = 2;
6          this.x = 3;
7          if( true ) {
8              int y = 4;
9          }
10         // cannot access y here
11     }
12 }
```

IP

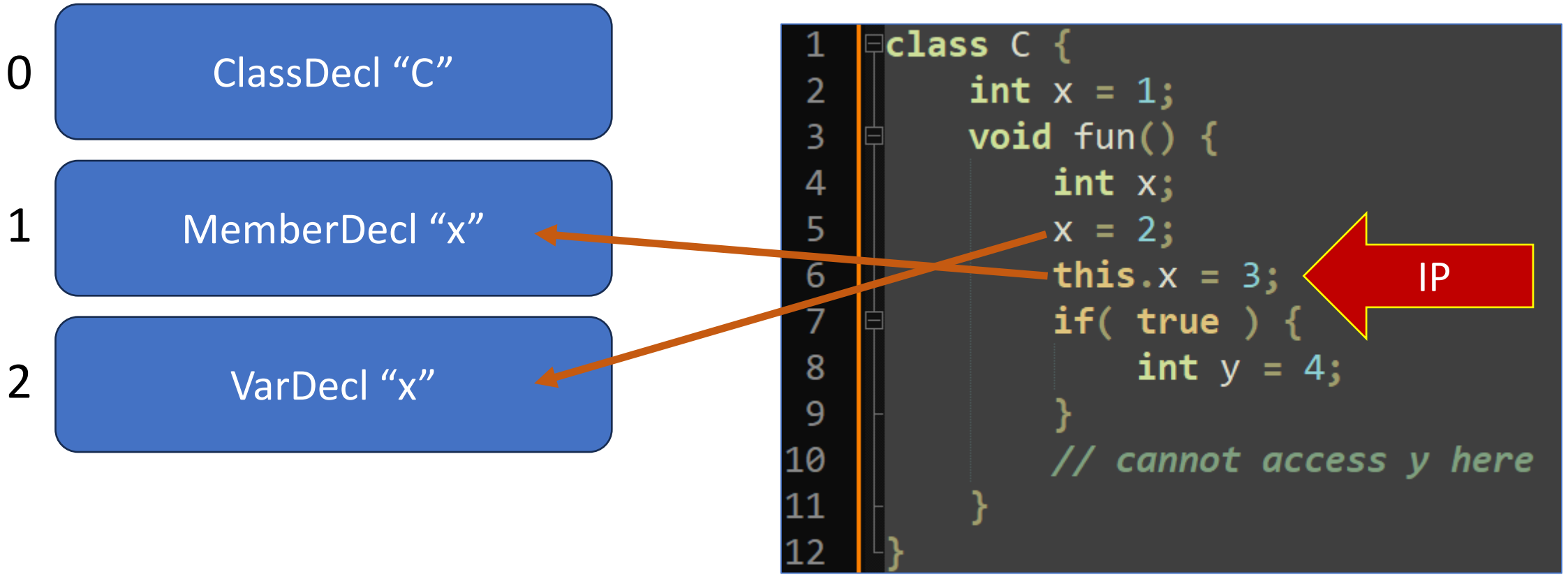
Scoped Identification – Stack Visualization



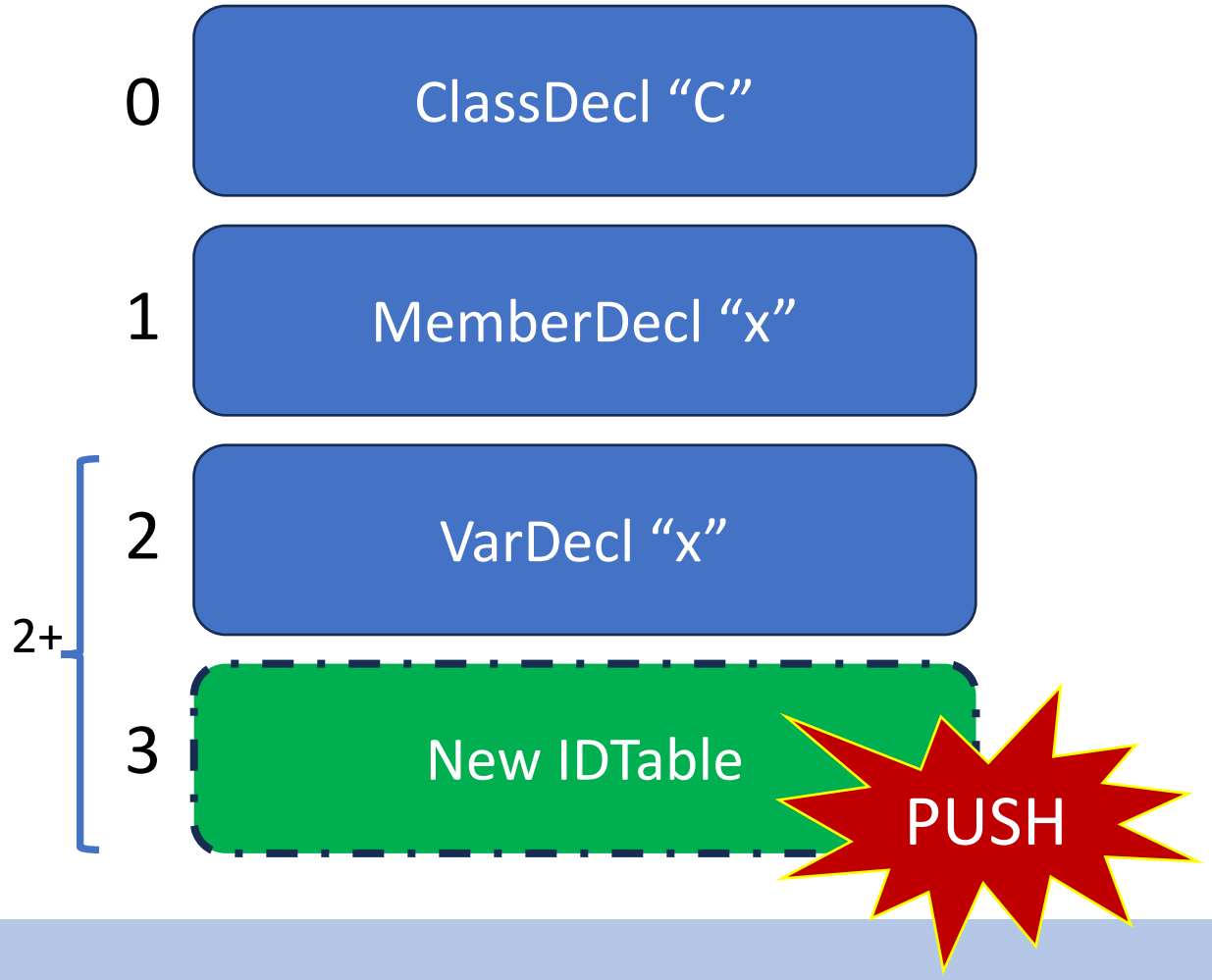
Scoped Identification – Stack Visualization



Scoped Identification – Stack Visualization



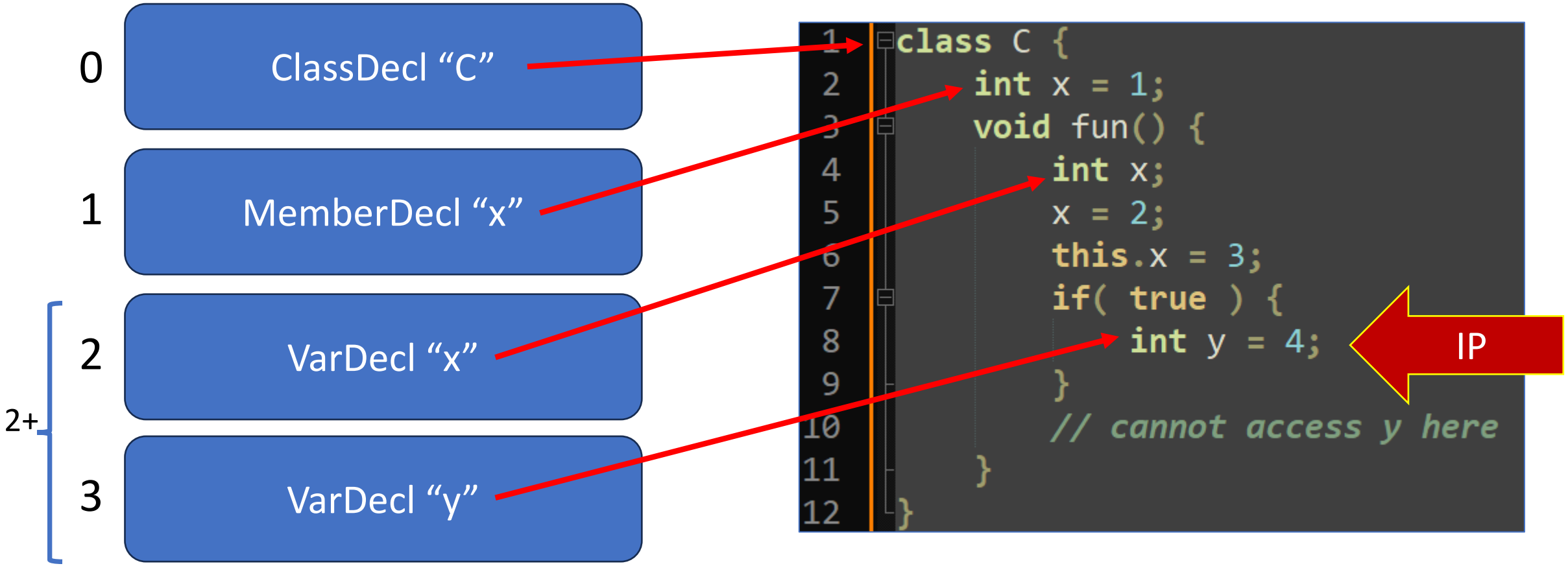
Scoped Identification – Stack Visualization



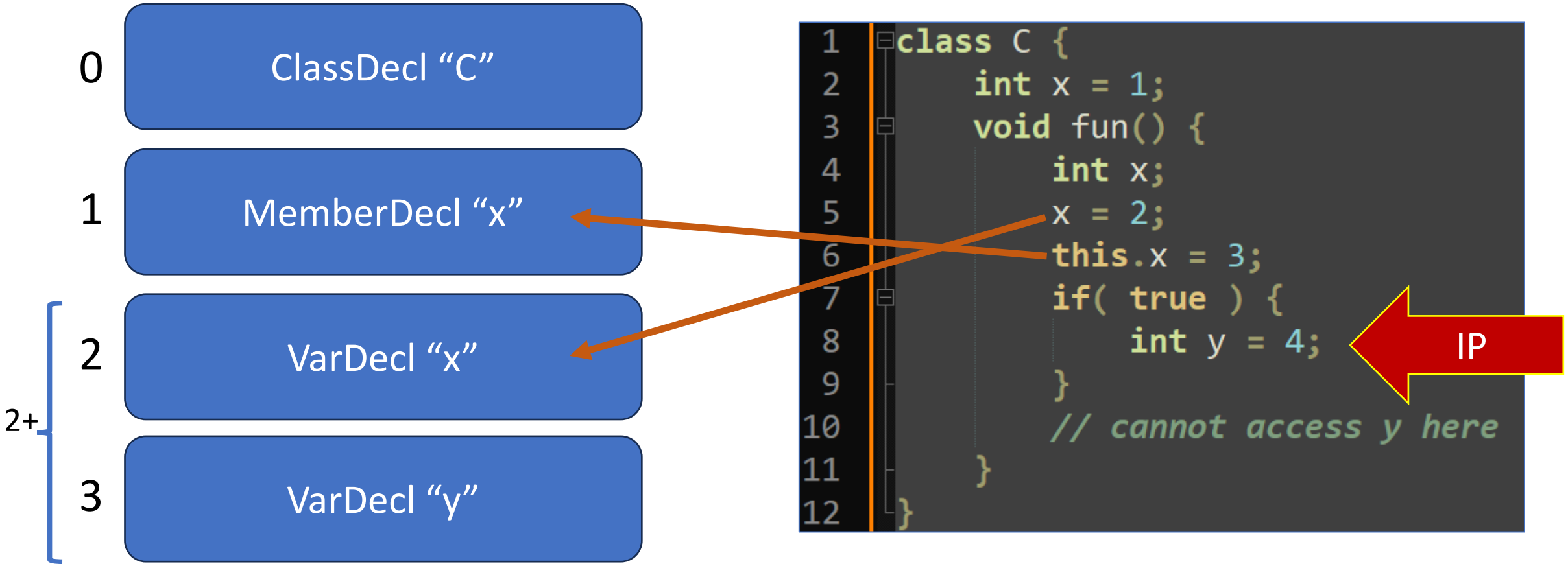

```
1  class C {
2      int x = 1;
3      void fun() {
4          int x;
5          x = 2;
6          this.x = 3;
7          if( true ) {
8              int y = 4;
9          }
10         // cannot access y here
11     }
12 }
```

IP

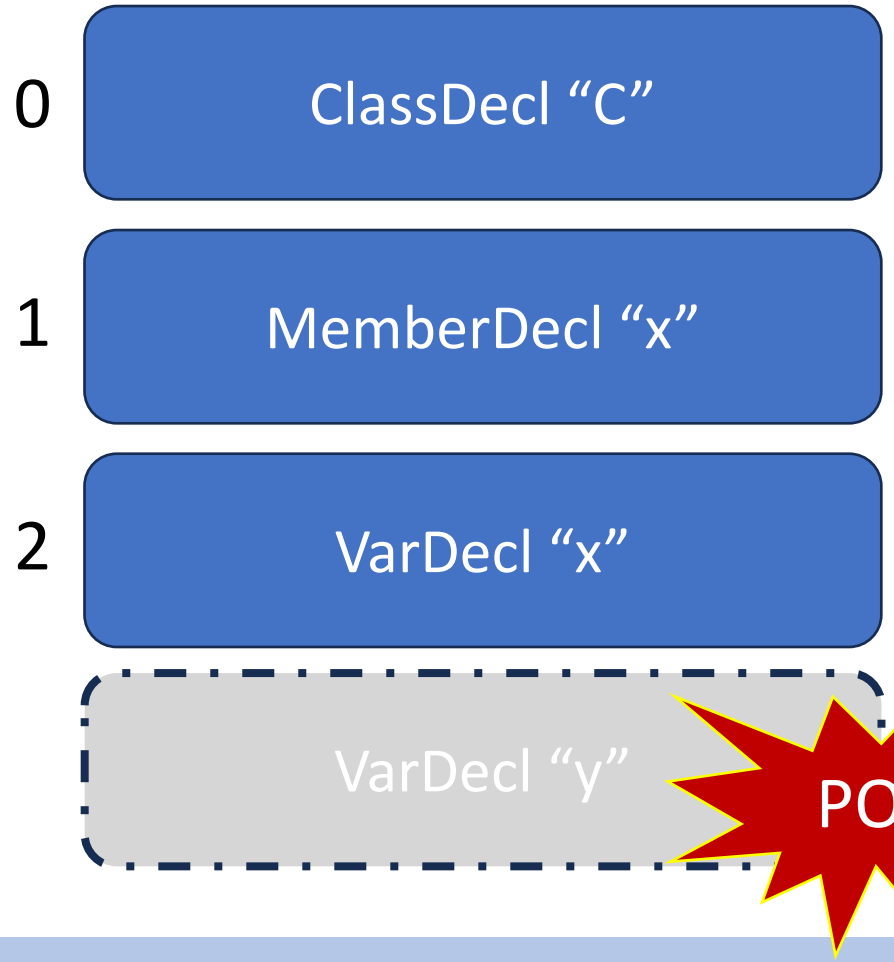
Scoped Identification – Stack Visualization



Scoped Identification – Stack Visualization



Scoped Identification – Stack Visualization




```
1  class C {
2      int x = 1;
3      void fun() {
4          int x;
5          x = 2;
6          this.x = 3;
7          if( true ) {
8              int y = 4;
9          }
           // cannot access y here
11     }
12 }
```

IP

More on Identification

Pre-defined names, out-of-order references

Pre-defined Names

- miniJava doesn't have imports, so we need to provide some basic functionality.
- You must MANUALLY add these entries into your IDTables for PA3, and we will use them in PA4.

```
class System { public static _PrintStream out; }
class _PrintStream { public void println( int n ) { } }
class String { }
```

- Note, println takes an int, not a String.

Out-of-order References

- Note: class B is used as a type when B hasn't yet been declared.

- Question: How can we put the classes in our IDTables ahead of time?

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Out-of-order References (2)

- Additionally, can refer to members of classes that have not yet been declared.

- Question: How can we put the members in our IDTables ahead of time?

```
1  class A {
2      void fun() {
3          B b = new B();
4          b.x = 3;
5      }
6  }
7
8  class B {
9      public int x;
10 }
```

Strategy: OOO References

- Before you visit a method's StatementList, add all ClassDecls in the Package AST to the level 0 IDTable.
- Then, add all MemberDecl (FieldDecl/MethodDecl) to the IDTable as well.
- Never drop below scope level 1.

Strategy: OOO References (2)

- Private members cannot be accessed externally.
- Possible strategy: **only add public members first**, then when processing a class, add that class's private members.
- When leaving a class, make sure to remove those entries! (How would you handle this?)

Alternatively: check private later when resolving declarations. (this is probably easier than changing your level 1)

Identification of QualRef

If one AST object has its own section, that means it should be the most fun part!

Left-most Reference

- Only the left-most reference should be resolved normally (start at the top of the SI stack, then work down).
- Once you know the Declaration of the left-most reference, you have a **context**.

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Left-most Reference

- QualRef(LHS,RHS): LHS is a Reference, and RHS is an Identifier.
- With the type of the LHS (the context), resolve the RHS.

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Left-most Reference

- QualRef(LHS,RHS): LHS is a Reference, and RHS is an Identifier.
- With the type of the LHS (the context), resolve the RHS.
- a.b means ".b" is resolved in the context of the type of "a", which is class "A".

```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

Left-most Reference

- With the type of the LHS (the context), resolve the RHS.
- **Note:** this means that you can bypass local variables.
- "a.b.c.x" but "b", "c", "x" were all locally defined.

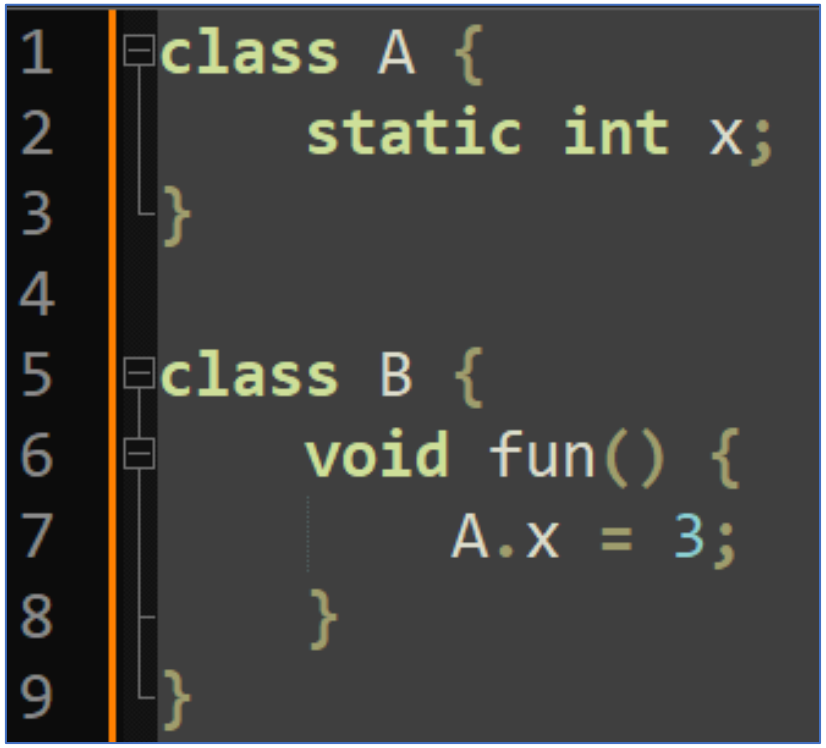
```
class A {
    B b;
    int x;
}

class B {
    C c;
    int x;
}

class C {
    int x = 2;
    void fun() {
        int b = 3;
        int c = 4;
        int x = 5;

        A a = new A();
        a.b.c.x = 6;
    }
}
```

A.b, where A is also a class name



```
1  class A {
2      static int x;
3  }
4
5  class B {
6      void fun() {
7          A.x = 3;
8      }
9  }
```

- Is it correct to say that if "A" is a class name, then "x" must be a static member?

A.b, where A is also a class name (2)

```
1  class A {
2      static int x;
3  }
4
5  class B {
6      void fun() {
7          A.x = 3;
8      }
9  }
```

```
1  class A {
2      static int x;
3  }
4
5  class B {
6      int A;
7      void fun() {
8          A.x = 3;
9      }
10 }
```

A.b, where A is also a class name

```
1  class A {
2      static int x;
3  }
4
5  class B {
6      void fun() {
7          A.x = 3;
8      }
9  }
```

```
1  class A {
2      static int x;
3  }
4
5  class B {
6      int A;
7      void fun() {
8          A.x = 3;
9      }
10 }
```

Static Reference

- In miniJava, if the **left-most reference** is a class name AND **is not declared at level 1+**, then the RHS must be a static member.
- E.g. “CLASSNAME.x”, where “x” must be a static member of “CLASSNAME”

```
1  class A {
2      static int x;
3  }
4
5  class B {
6      static int x;
7      int y;
8      void fun() {
9          A.x = 3;
10         B.x = 4;
11         B.y = 5; // BAD!
12     }
13 }
```

Static Context

- If you are in a static method, then you can only access static members in your Class, and public static members in other classes.

- Additionally, ThisRef is an identification error as "this" does not resolve to any declaration in a static method.
- Question: where is ThisRef in memory?

QualRef Strategy

- Try to get the "context" by visiting the LHS reference.
- With that context, resolve the RHS.

- E.g. "a.b" will return the context of class "B", thus allowing resolution of "a.b.c" where "c" is in the context of "B"

No one strategy dominates all others

- How you choose to identify "context" is up to you. It can be a String, ClassDecl, TypeDenoter, etc.

- Even more important to plan PA3 than other assignments before starting to code.
- If you change your Visitor's parameter or return type, you may have to redo the entire class declaration!

PA3 – Type Checking

Type-Checking Table

- In miniJava, type order does not matter, so A×B×op is the same as B×A×op

- This means we can simplify our TypeChecking table.

miniJava – Types must match

- For miniJava, the types must match. There is no automatic type conversions nor manual typecasting.

miniJava – Types must match

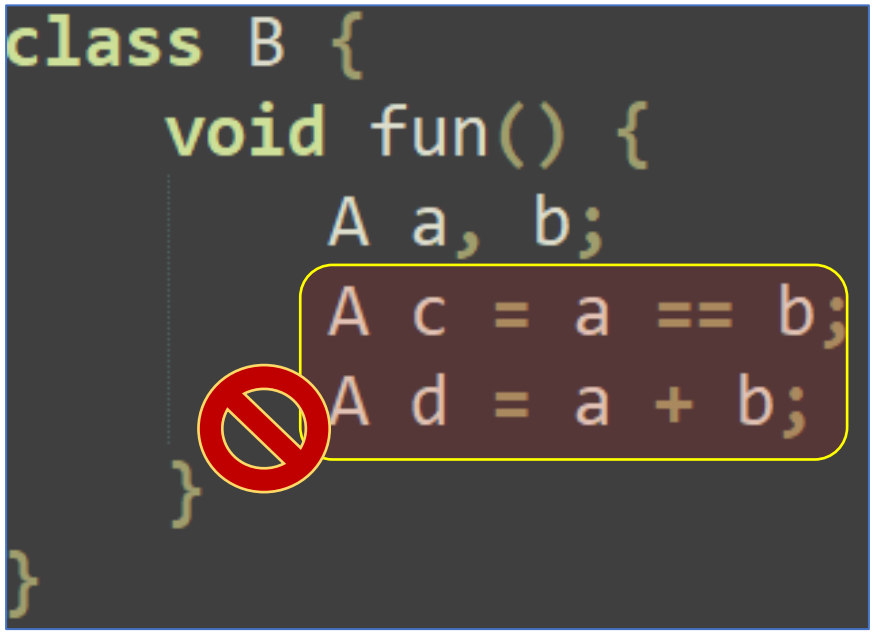
- For miniJava, the types must match. There is no automatic type conversions nor manual typecasting.

- Does this mean we can use a REALLY simple type-checking table where both types must match, **and the result type is that type**?

miniJava – Types must match

- Does this mean we can use a REALLY simple type-checking table where both types must match, **and the result type is that type**?

- Still need to formally clarify Type rules.



Type-Checking Table (2)

Type Checking Rules		
Operand Types	**Operand**	**Result**
boolean × boolean	&&, ||	boolean
int × int	>, >=, <, <=	boolean
int × int	+, -, *, /	int
α × α	==, !=	boolean
int	(Unary) -	int
boolean	(Unary) !	boolean

ClassType

- If two objects are both ClassType, are they comparable?

ClassType (No polymorphism in miniJava)

- If two objects are both ClassType, are they comparable?

- No, the underlying Identifier text must match.
- Why is this enough?

```
1  class A {
2  }
3
4  class B {
5      void fun() {
6          A a = this;
7      }
8  }
```

What type is ArrayType?

- **Recall:** `new int[4]`
- **This expression is of** `ArrayType(IntType)`
- **Thus, it can only be assigned to variables of type** `ArrayType(IntType)`

- `IntType` **is shorthand for:** `BaseType( TypeKind.INT )`

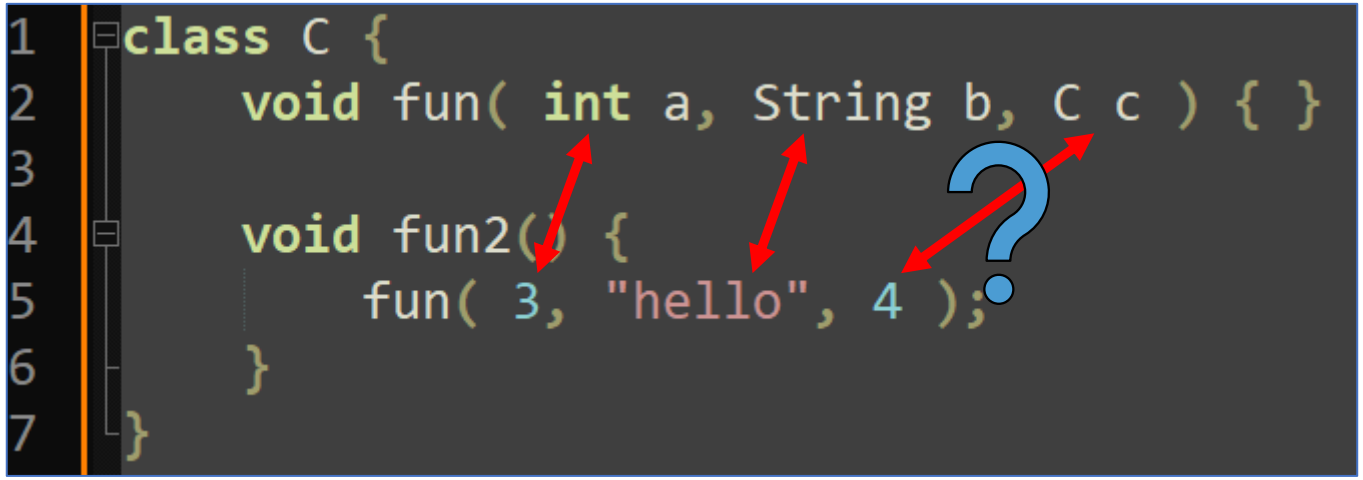
What type is ArrayType? (2)

- For array types:
  - First: Are both types ArrayType?
  - Second: Do the element types match? (Recursion)

- Recursion needed to match ArrayType of ArrayType of ArrayType of ClassType.

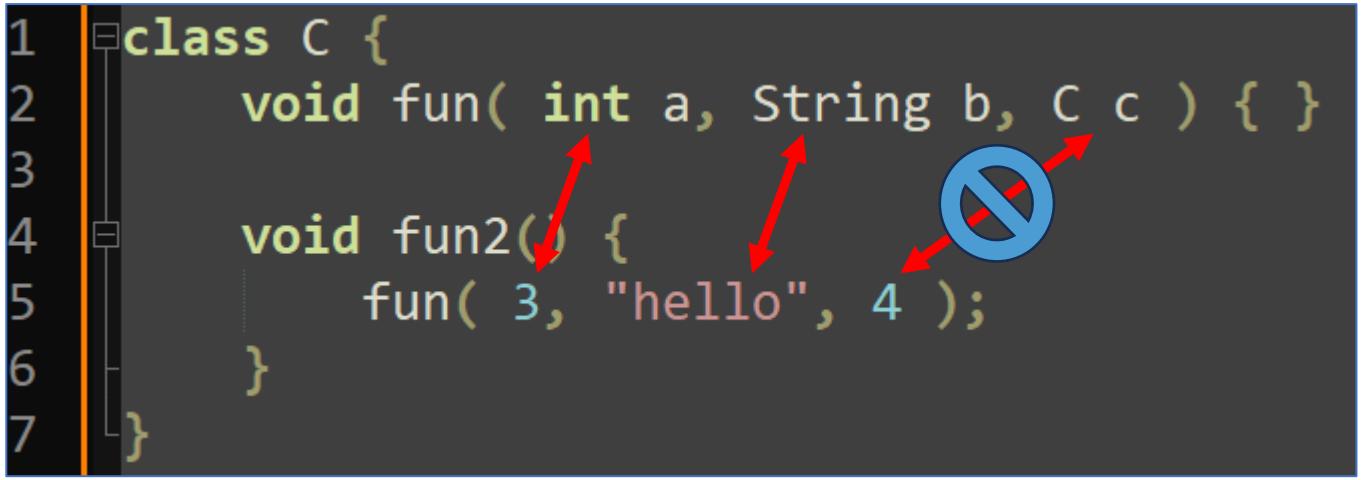
Type-Checking Methods

- Scoped Identification **only uses context and identifiers**. Therefore, overloading methods by parameter types/counts is not allowed in miniJava.



Type-Checking Methods (2)

- As such, make sure there is an expression for every parameter, and that the types match.



Type Errors

- The ErrorType is compatible with ALL other types, and the result type is always another ErrorType and this does not cause an error to be reported.

- If a type is not allowed in an operation with another type, then the result type is an ErrorType.

Unsupported Type

- The UNSUPPORTED type is not compatible with any type (including itself) and causes an error to be reported. The result type will be ErrorType.

- Make sure String's type is UNSUPPORTED, otherwise String can be initialized with new String(), which is not implemented in miniJava.

Unsupported Type (2)

- The String predefined class is an UNSUPPORTED type.
- String is not supported in miniJava, but available to be implemented as a part of PA5.

- We need String to be able to declare the main method.

Unsupported Type (3)

- UNSUPPORTED×ErrorType
- Question: should an error be reported?

Unsupported Type (4)

- UNSUPPORTED×ErrorType does not need to be reported (only way ErrorType exists is if an error was reported earlier anyway).

- But it can be reported if you want to report an extra error where String is utilized.

Type-Checking Strategy

- Implement a TypeChecking Visitor that uses a TypeDenoter return type.

- Visiting a node synthesizes a TypeDenoter for that type.
- Create a method, input is the two TypeDenoters, (or one for Unary), and output is the resultant TypeDenoter.
- Or create a table, but that would have a lot of null entries.

Type-Checking Strategy (2)

- Ensure index expressions are integers
- Ensure condition expressions in if/while are boolean
- Ensure operands are compatible, and return the appropriate type when visiting that BinExpr/UnaryExpr

Other Contextual Constraints

Contextual Analysis

- There are contextual parts of Java (and miniJava) that do not quite fit Identification or Type Checking.
- We can easily implement these as a part of either.

Contextual Analysis (2)

- If an identifier is being declared, then it cannot be used in the expression.
- Even if the expression can be evaluated first!

```
1  class C {
2      int x = 1;
3      void fun() {
4          int x = x + 1;
5      }
6  }
```


Not allowed!

Contextual Analysis (3)

- You cannot have a variable declaration only in a scope to itself.
- A BlockStmt (new scope) is necessary for VarDeclStmt.

```
1  class C {
2      int x = 1;
3      void fun() {
4          if( true )
5              int x = 3;
6      }
7  }
```

Not allowed!

Enjoy your weekend!

- PA3 released, but not due for over a month.
- PA2 due in less than a week!

- WA2 due Monday night!!!

- Midterm on Thursday 2024-02-22

End