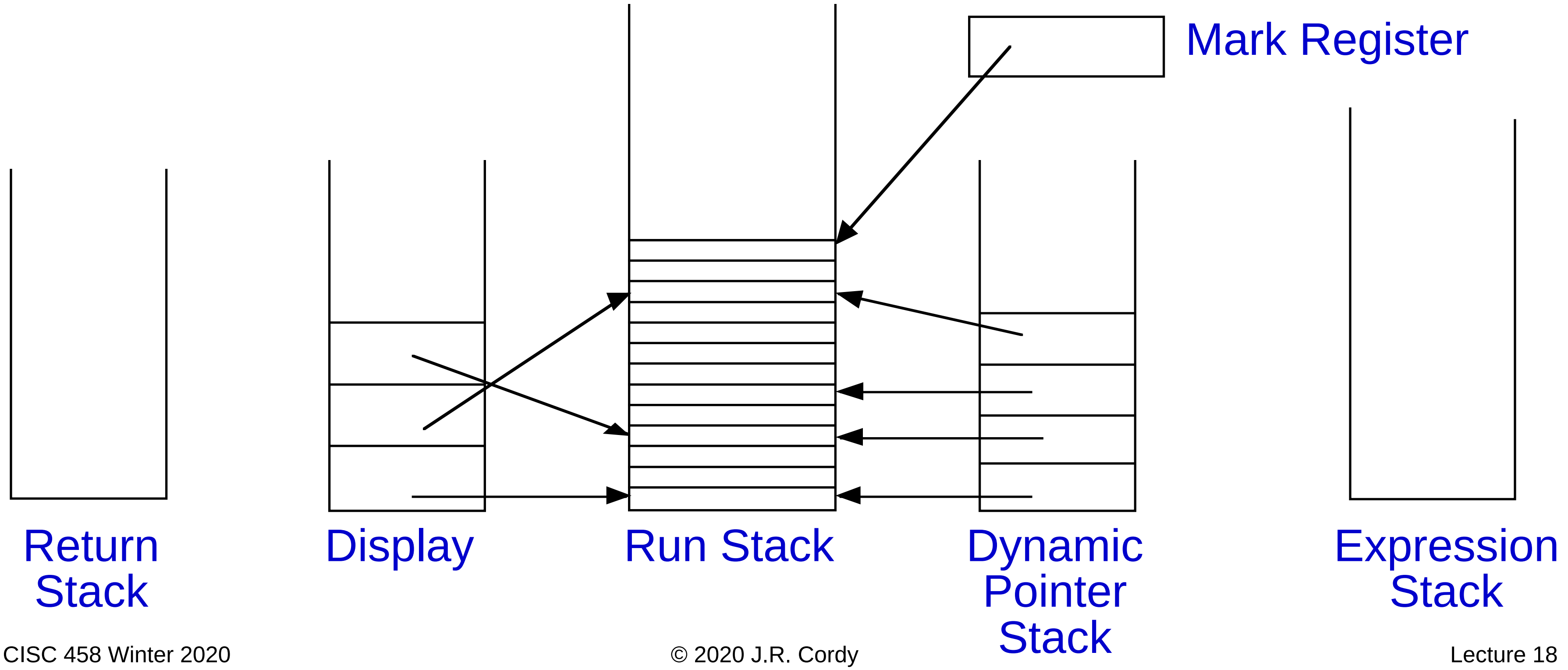
Previously ...

- Runtime Model - 5 Stacks
  - *Expression Stack* - Expression Evaluation
  - *Run Stack* - Storage Allocation
  - *Display* - Scope Management (not really a stack ...)
  - *Dynamic Pointer Stack* - Restoring Run Stack and Display
  - *Return Stack* - Remembers Program Counter for Call/Return



Previously ...

- Variables accessed by *Lexical Level, Order Number* (LL,ON)
  - *Lexical Level* is used as an index into the Display, to give an address in the Run Stack
  - *Order Number* is an offset that is added to the address from the Display
- *Pass by Value* and *Pass by Reference* parameters
- *Arrays* and *records* are modeled as contiguous variables, standard formulas used to calculate the address of a subscripted element from the address of the array as a whole
- Variable *addressing* operations were examined and are used for parameters, arrays, records and assignment
- *Return values* from functions

Semantic Analysis

- Two main problems solved by Semantic Analysis

Validation - Is the program legal and meaningful?

- is the program legal?
- variable declaration and use
- type checking
- scope analysis

Annotation - What does the program mean?

- preparation for code generation
- information about declaration and type of variables gets distributed to all operations on the variables
- all information needed to generate code is located in the places where code will be generated

Semantic Analysis

- Two main techniques used in Semantic Analysis

  Tabulation - Collection of information into look-up tables

  - Compilation of declaration and contextual information into tables - hence the name *compiler*
  - One such table is the *Symbol Table*
  - Collects Name, Type, Size, other attributes of program symbols

  Simulation - Simulate ideal execution to determine meaning

  - Simulate execution state of the Expression Stack of our ideal stack machine
  - Used to type check expressions, and to infer types of expression results and other attribute information

Semantic Analysis Problem 1 - Validation

- Is the program legal?

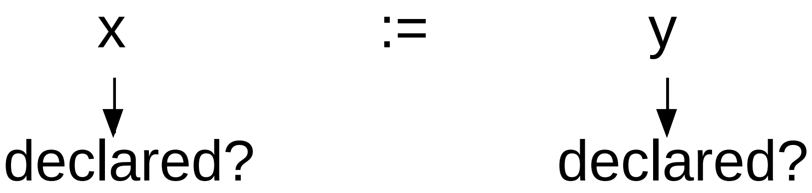
Validation

- Is the program legal?   Four parts to the answer

```
          x          :=          y
```

Validation

- Is the program legal?   Four parts to the answer



x := y

declared?   declared?

Validation

- Is the program legal? Four parts to the answer

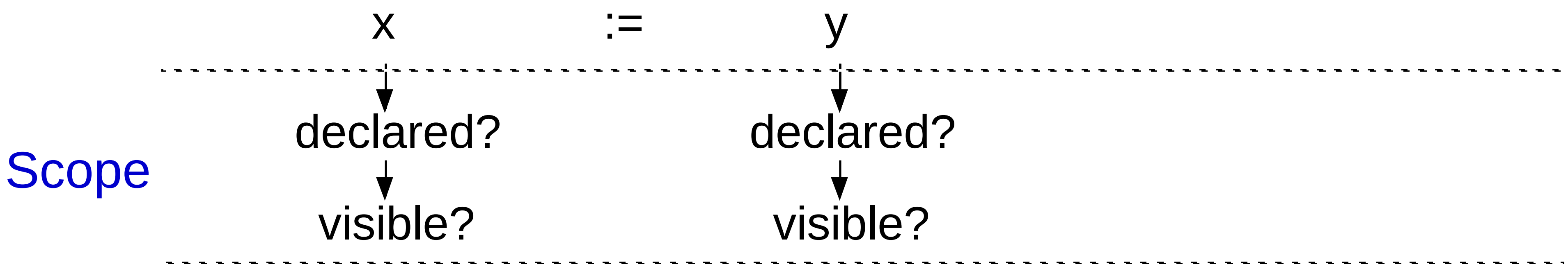
x := y

declared? declared?

visible? visible?

Validation

- Is the program legal? Four parts to the answer



x := y

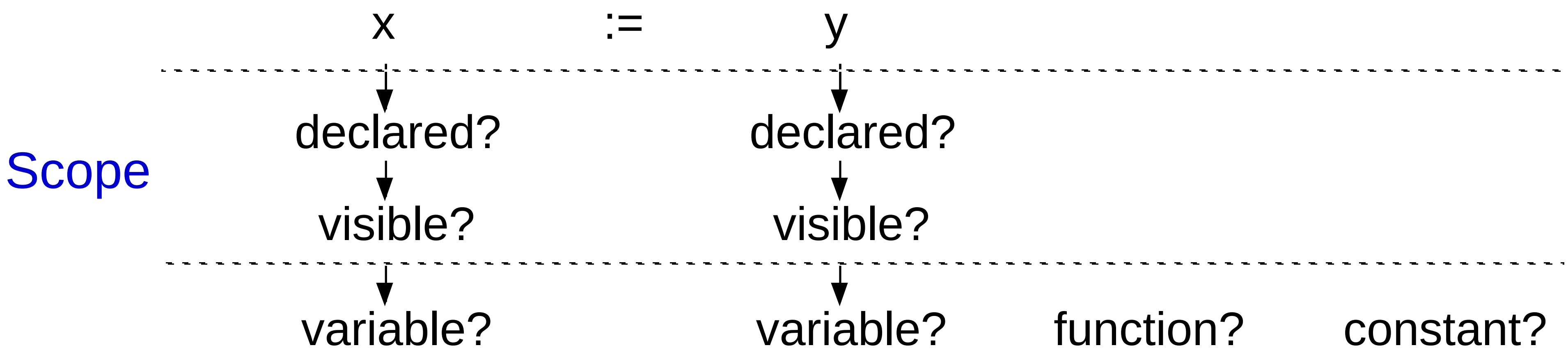
Scope

x → declared? → visible?

y → declared? → visible?

Validation

- Is the program legal?   Four parts to the answer



x := y

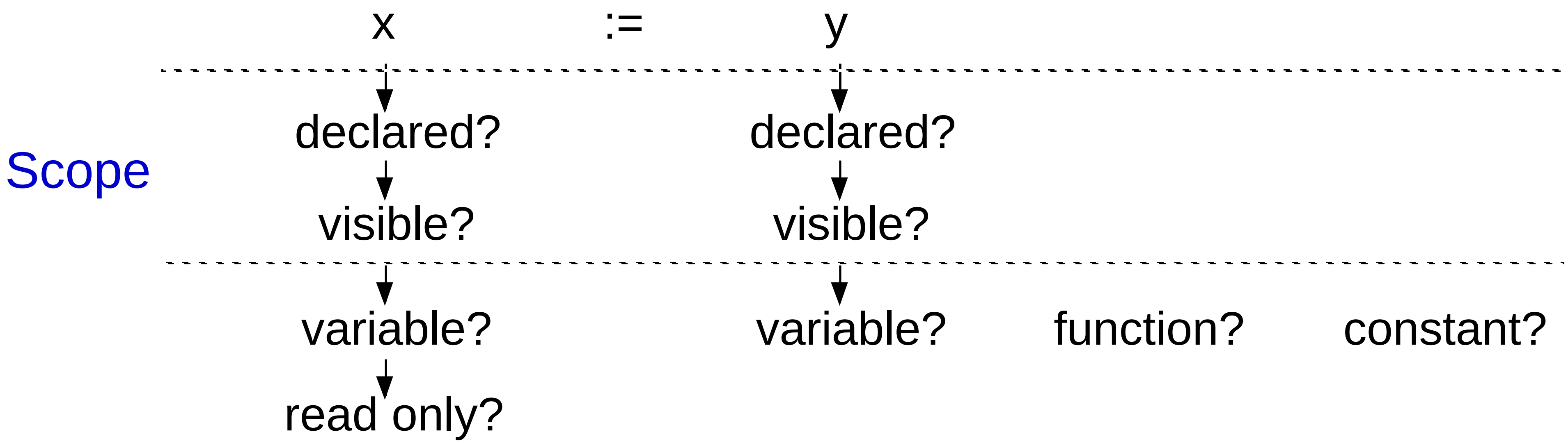
Scope

declared? → visible? → variable?

declared? → visible? → variable? function? constant?

Validation

- Is the program legal? Four parts to the answer



x := y

Scope

declared? → visible? (x)

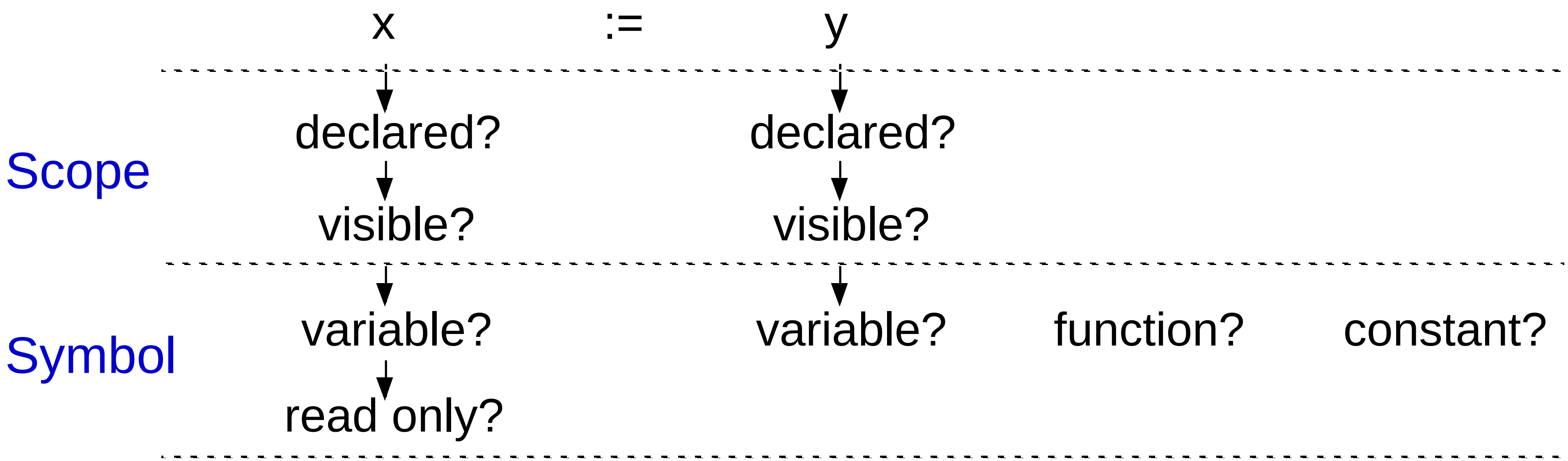
declared? → visible? (y)

variable? → read only? (x)

variable? function? constant? (y)

Validation

- Is the program legal?   Four parts to the answer



x := y

Scope

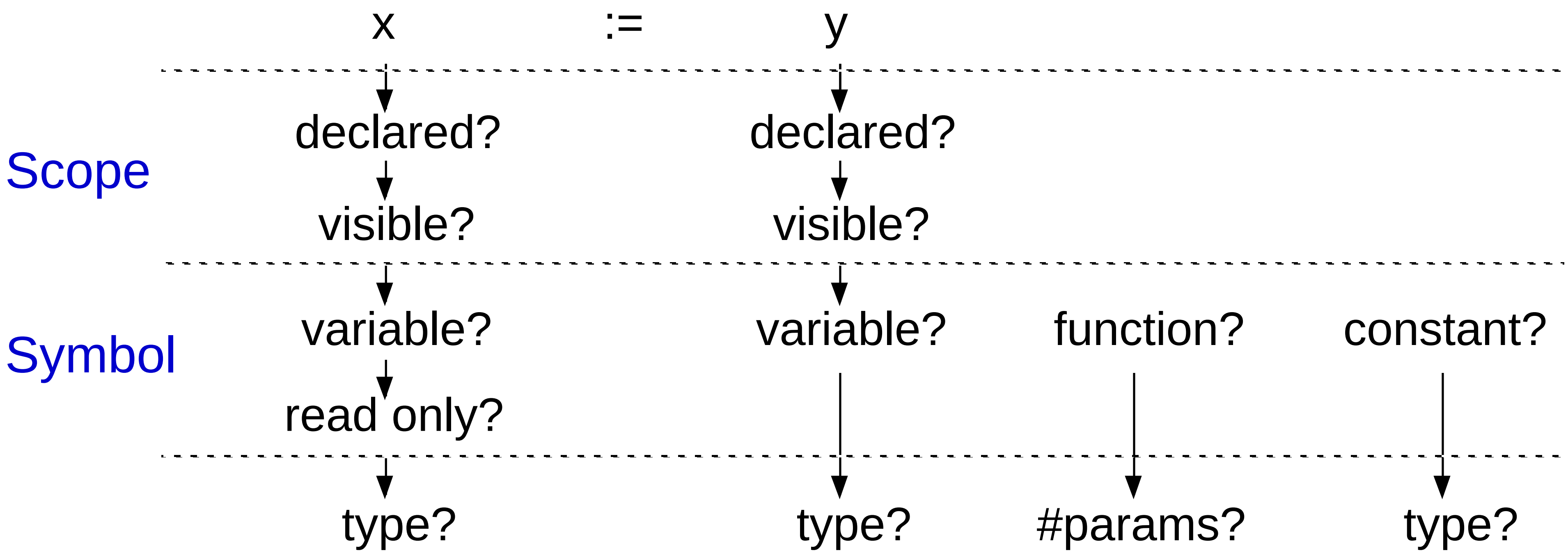
- x: declared? → visible?
- y: declared? → visible?

Symbol

- x: variable? → read only?
- y: variable? function? constant?

Validation

- Is the program legal? Four parts to the answer



x := y

Scope

declared? declared?

visible? visible?

Symbol

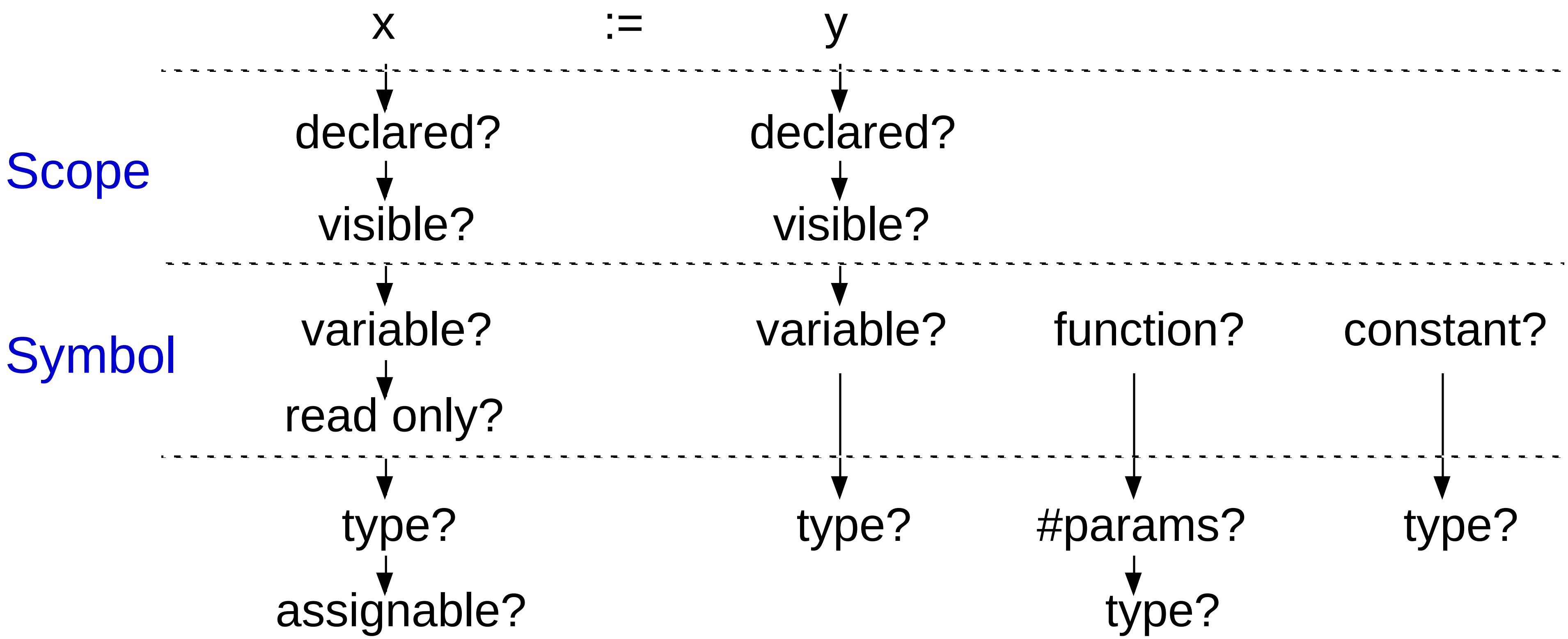
variable? variable? function? constant?

read only?

type? type? #params? type?

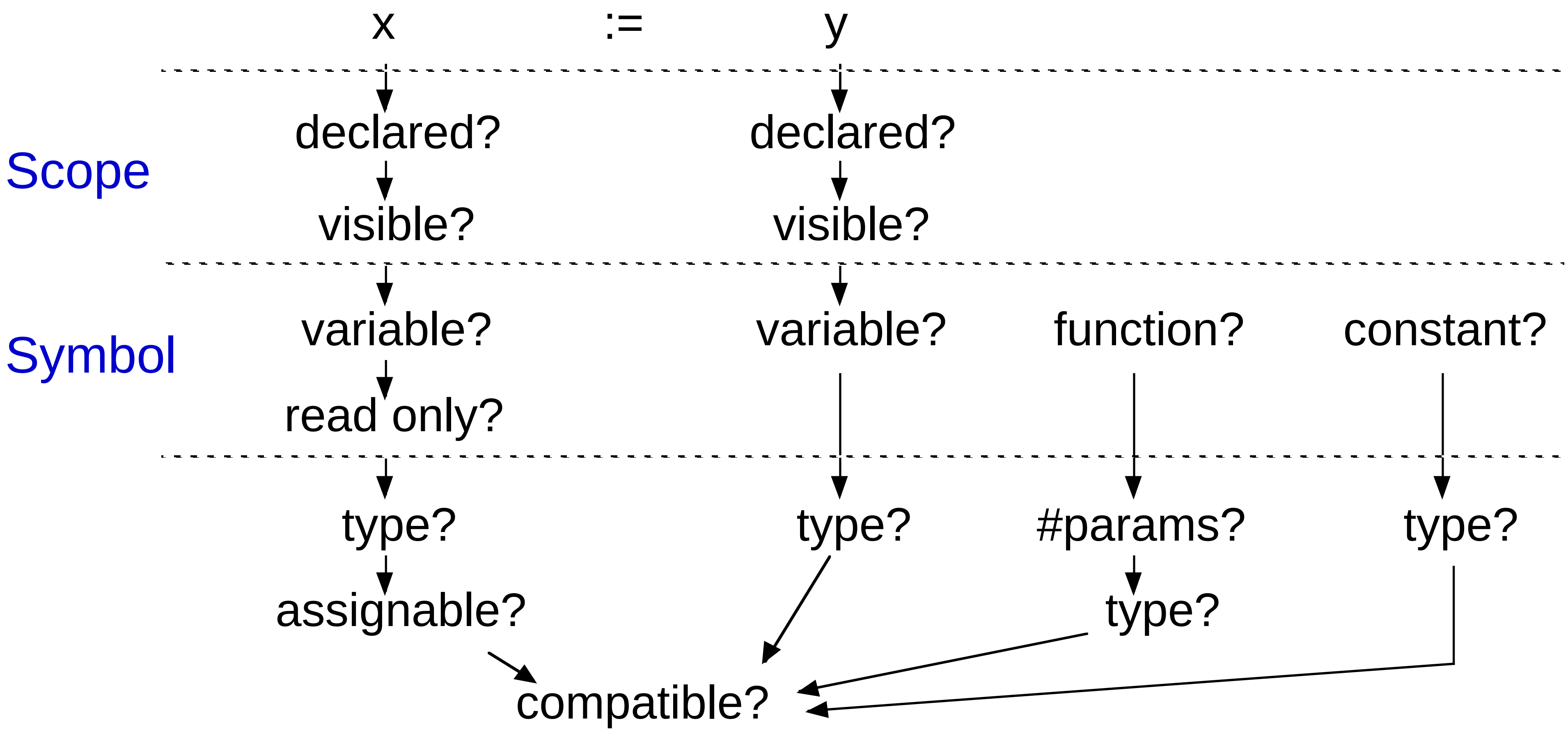
Validation

- Is the program legal?   Four parts to the answer



Validation

- Is the program legal?   Four parts to the answer



x := y

Scope

declared? → visible?

declared? → visible?

Symbol

variable? → read only?

variable?

function?

constant?

type? → assignable?

type?

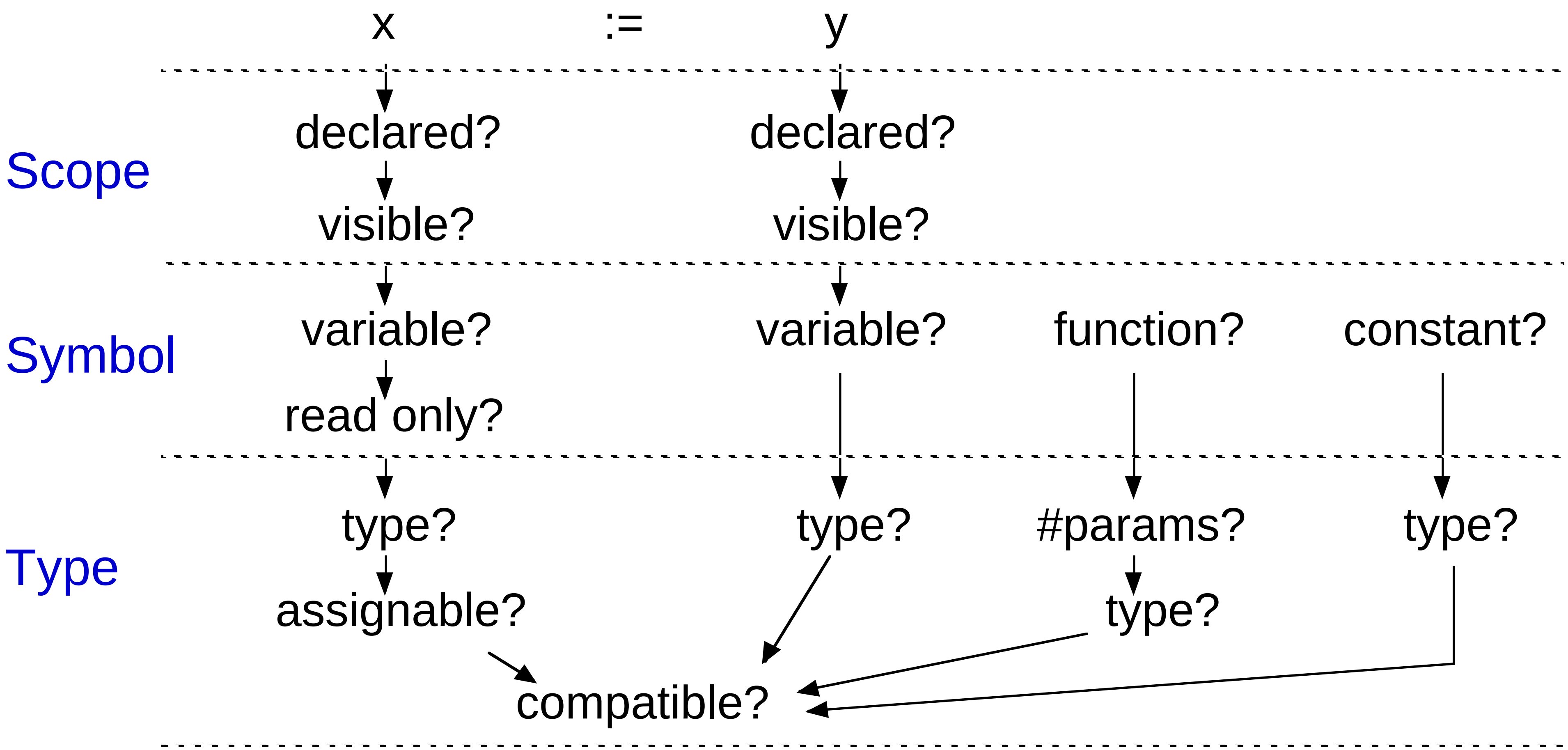
#params? → type?

type?

compatible?

Validation

- Is the program legal? Four parts to the answer



x := y

Scope

declared? → visible? (x)

declared? → visible? (y)

Symbol

variable? → read only? (x)

variable? function? constant? (y)

Type

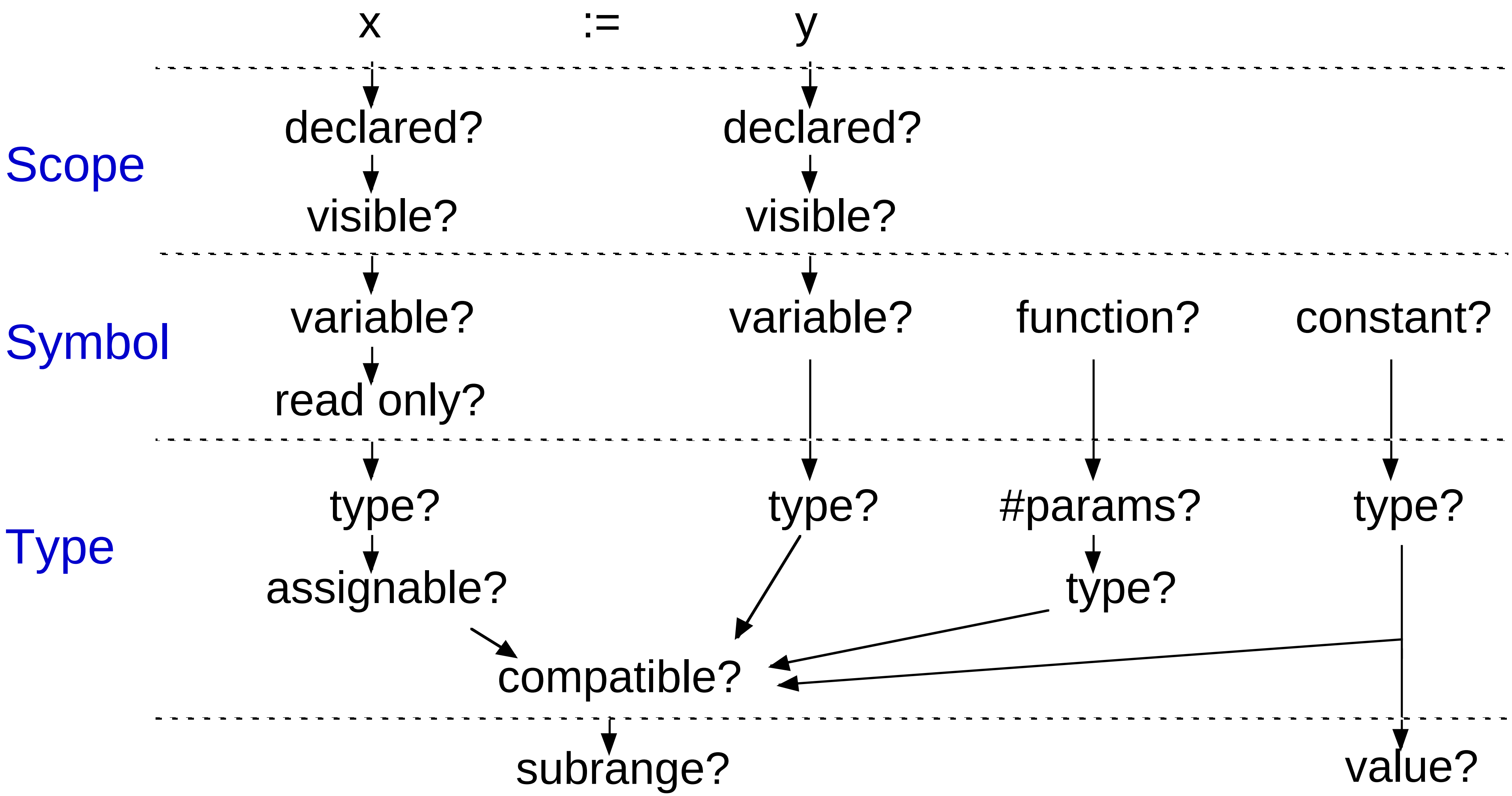
type? → assignable? (x)

type? #params? → type? type? (y)

compatible?

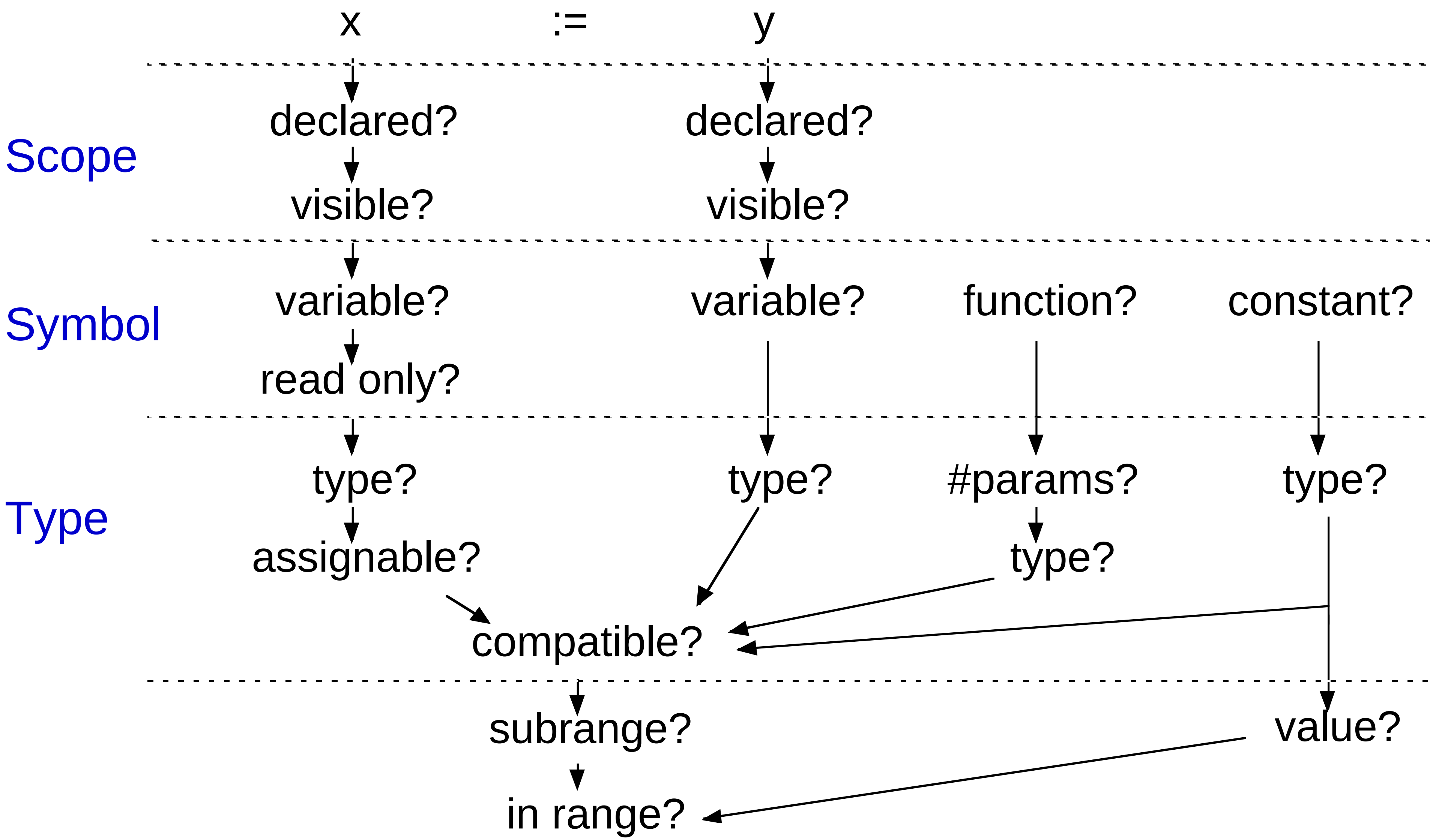
Validation

- Is the program legal? Four parts to the answer



Validation

- Is the program legal? Four parts to the answer



x := y

Scope

declared? → visible? (x)

declared? → visible? (y)

Symbol

variable? → read only? (x)

variable? function? constant?

Type

type? → assignable?

type?

#params? → type?

type?

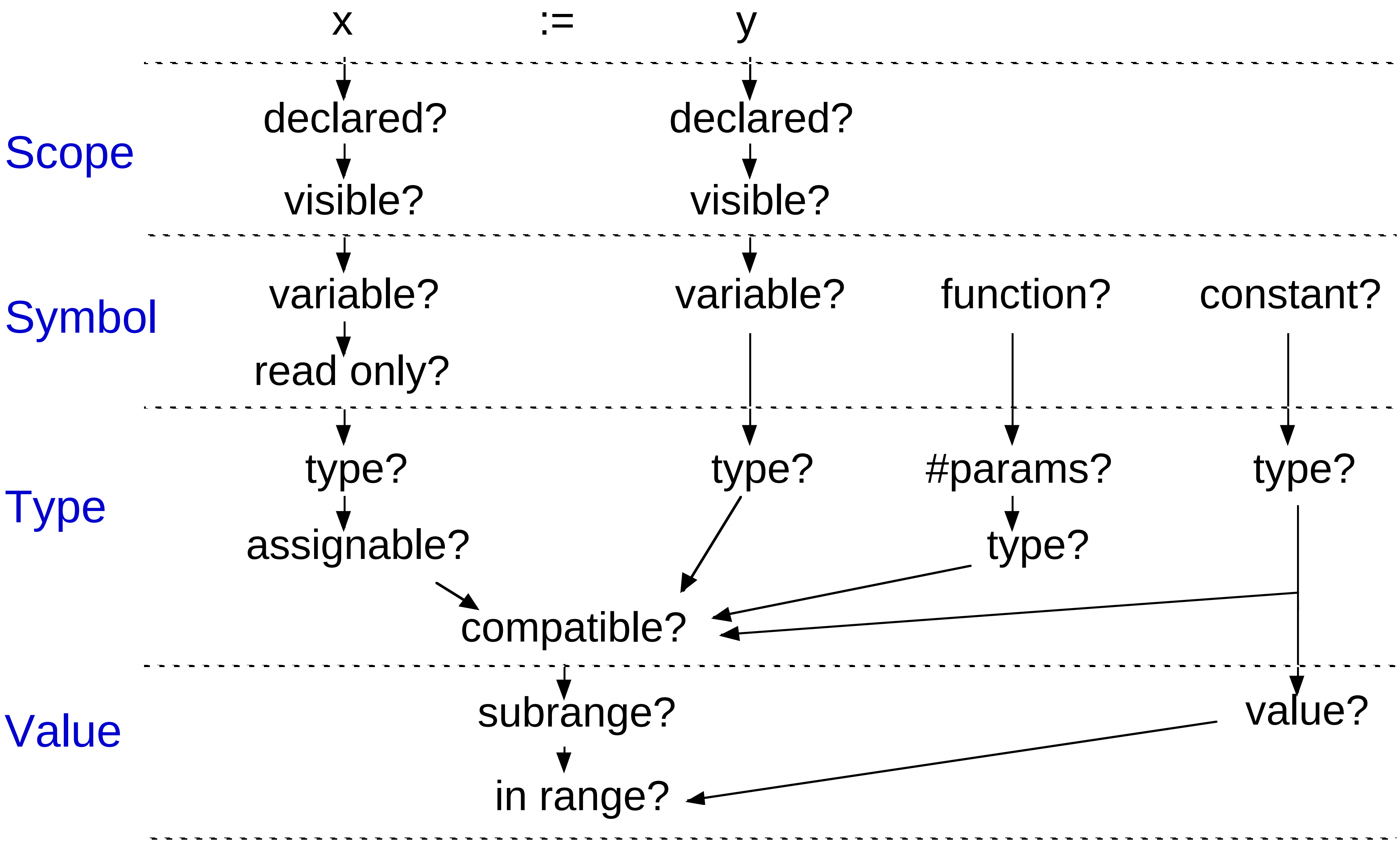
compatible?

subrange? → in range?

value?

Validation

- Is the program legal? Four parts to the answer



x := y

Scope

declared? → visible? (x)

declared? → visible? (y)

Symbol

variable? → read only?

variable?

function?

constant?

Type

type? → assignable?

type?

#params? → type?

type?

compatible?

Value

subrange? → in range?

value?

Validation

- **Scope Analysis**
  - Does a used symbol exist? Is it visible here?
  - Enforces language rules related to the scope, visibility and accessibility of declared symbols (variables, constants, etc.)
  - Inheritance (not in OO sense) of symbols into subscopes
  - Information hiding: import/export lists (modules), public/private symbols (classes)
  - Other languages issues such as OO inheritance and templates

Validation

- **Symbol Checking**
  - What kind of symbol is it?
  - A symbol might be declared to be a type, a procedure, a variable, a constant, ...
  - Checks that use of symbols is consistent with their kind
  - Example:

```
type point:
    record
        x: integer
        y: integer
    end
```

```
z := point
```

invalid since *point* is a type name, not a variable name

  - Enforces const, readonly, and anti-aliasing restrictions

Validation

- **Type Checking**
  - Now that we know which symbols are types and which are variables, etc., have to make sure that types match
  - Assignability (compatibility) of types
    e.g., in C, `long` <– `int` <– `char`
  - Automatic conversions
  - Type equivalence (for reference parameters)
  - Some languages require the types to be identical, others permit type equivalency rules
    e.g., records with equivalent fields are same type in Euclid,
    different types in C
  - Opaque types (e.g., structures/classes with only private fields)
  - Type rules can be extremely complex
    e.g., PL/I, 40 pages of reference manual describe type rules

Validation

- **Value and Range Analysis**
  - Enforces rules about values and ranges
  - Subrange types e.g., `var a: 1 .. 5`
  - Array subscript ranges
    e.g., `var a: array [1 .. 5] of integer`
  - For constants, these can be checked at compile time,
    but for variable values must be done at run time
    => instruct code generator to generate checking code
  - Constant folding => replace expressions consisting of
    constant values only with the value of the expression

- **Context Dependent Syntax**
  - Syntax constraints that are not context free - e.g., matching of
    end labels, e.g.,

```
procedure a (int b, int c)
    ...
end a
```

Semantic Analysis Problem 2 - Annotation

- What does the program mean?

  Annotate each operation and operand with its meaning based on the declarations and use of the symbols involved

  Also referred to as *attribution*

- Euclid example:

  `x := y(z)`

  y is a array of real –> $x :=_{\text{real}}$ y (subscript z)

  y is an integer function –> $x :=_{\text{int}}$ y (parameter z)

  x is a set of type y –> $x :=_{\text{set}}$ y (elementlist z)

  x is a variable of type y –> $x :=_{y}$ y (convert z)

- All languages have similar issues (although Euclid is extreme)

Annotation

- As with validation, several components:

- **Reference Analysis**
  - symbol analysis and disambiguation
  - ( ) in previous example,
  - identification of which parameters are by reference, by value
    e.g., z in previous example

- **Operator Analysis, Operator Overloading**
  - which assignment operator (int, real, complex, array)
  - which arithmetic operator
    e.g., '+' can take many different types of operands, including mixed - the '+' in x + y might mean integer addition, real addition, set union, string concatenation, ...

- **Translation**
  - generation of abstract machine code or other intermediate representation for the code generator
    e.g., attributed parse tree, T-code, ...

The Semantic Phase

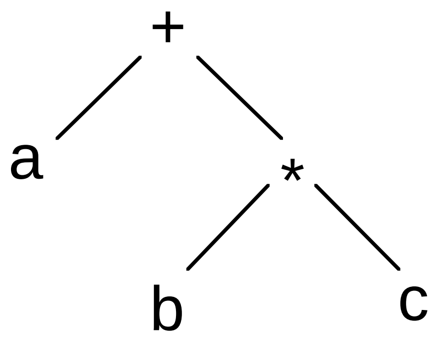
- Why separated from Code Generation and Parsing?
  - separated from parsing so that parsing can be context-free
  - separated from code-generation so that it can be machine independent (MC68000 vs Pentium vs SPARC vs ARM)
- Issues are restricted as much as possible to the semantics of the programming language
  - usually implements the semantics of the program language as one of:
    - a) attributed parse tree
    - b) annotated token stream
    - c) operation triples or quads (“register abstract machine”)
    - d) abstract machine code

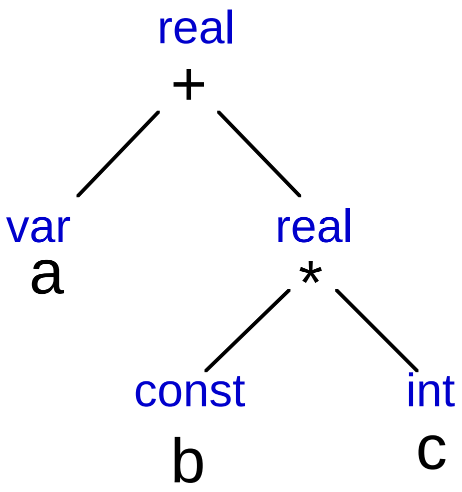
Annotation Example

a + b * c

- where *a* is a real variable, *b* is a real constant, *c* is an integer variable

<u>Attributed Parse Tree</u> (e.g. pcc, pqcc)





Annotation Example (cont'd)

Annotated Token Stream (e.g., Turing and Euclid compilers)

a b c * + –>
var a *const* b *var* c *convert_to_real* *real* * *real* +

Triples (code for an abstract register machine -
e.g., some PL/I and Pascal compilers)

(*convert_to_real*, var c, temp t)
(*real* *, const b, temp t)
(*real* +, var a, temp t)

- requires allocation of temporaries

Quadruples (e.g., original gcc - temps unique to each expression)

(*convert_to_real*, var c, null, temp t1)
(*real* *, const b, temp t1, temp t2)
(*real* +, var a, temp t2, temp t3)

Annotation Example (cont’d)

Abstract Stack Machine (e.g., PT)

a b c * +

```
tLiteralAddress  a          - push address
tFetchReal                  - evaluate
tLiteralAddress  b
tFetchReal
tLiteralAddress  c
tFetchInteger
tConvertReal
tMultiplyReal
tAddReal
```

(Note: PT does not actually have the *real* data type, so this is not pure PT T-code, just an example of the general approach)

Representation of Annotations

- Annotations are not always embedded directly in the streams as shown in the previous examples

- Annotations may be represented instead by *pointers* to a set of tables of information shared with the Code Generator
  - makes intermediate stream smaller, because annotation for each variable is represented only once (in the table)

- In compilers of 25 years ago, annotations were always done this way, with shared tables stored *on disk* because memories were small (64 Kb) and expensive - these compilers were often slow because they did so much disk access

- Modern compilers either do things like PT, with embedded inline annotations (e.g., gcc, Java), or use shared tables stored in virtual memory (because memory is now large and cheap, and VM is fast) (e.g., most IBM compilers)

Summary

- Semantic analysis consists of two main tasks :
    - Validation - *is the program valid and meaningful?*
    - Annotation - *what does the program mean?*
- We implement these tasks using two main techniques :
    - Tabulation - *collection of information into look-up tables*
    - Simulation - *simulate ideal execution to determine meaning*
- Next time :
    - Simulation