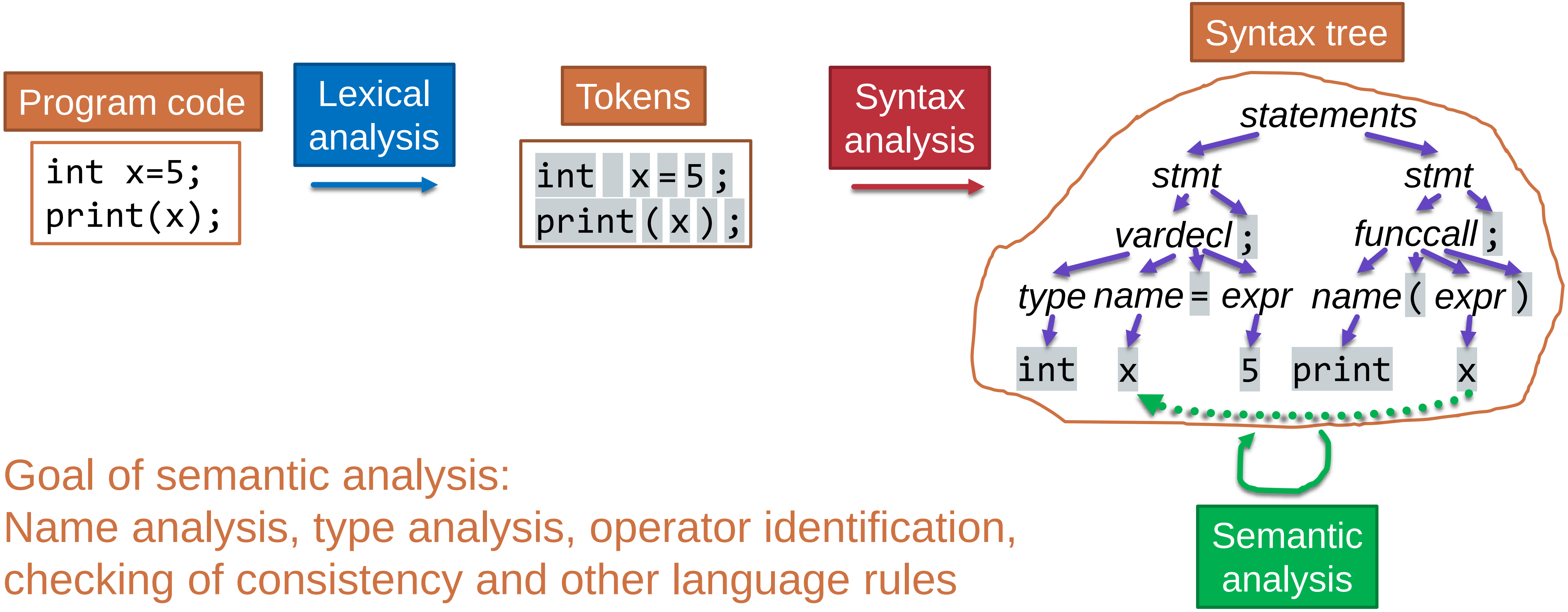
Model-based Software Development

Lecture 4

Semantic Analysis

Dr. Balázs Simon

Compiler front-end



Goal of semantic analysis:
Name analysis, type analysis, operator identification, checking of consistency and other language rules

This lecture: Semantic analysis

Semantic analysis example: construction of a symbol table

```
public abstract class Example
{
    private int x;

    public void Foo()
    {
        int y;
        if (!flag)
        {
            x = 3 + y;
            var ex = new Example();
        }
    }
    public int Bar(string x)
    {
        return "hello" + x;
    }
}
```

Symbol table:

- Example: class
  - x: field
  - Foo: method
    - y: local
    - ex: local
  - Bar: method
    - x: param

Semantic analysis example: name analysis

```
public abstract class Example
{
    private int x;

    public void Foo()
    {
        int y;
        if (!flag)
        {
            x = 3 + y;
            var ex = new Example();
        }
    }
    public int Bar(string x)
    {
        return "hello" + x;
    }
}
```

flag is not declared

Symbol table:

- Example: class
  - x: field
  - Foo: method
    - y: local
    - ex: local
  - Bar: method
    - x: param

Semantic analysis example: type analysis

```
public abstract class Example
{
    private int x;

    public void Foo()
    {
        int y;
        if (!flag)
        {
            x = 3 + y;
            var ex = new Example();
        }
    }
    public int Bar(string x)
    {
        return "hello" + x;
    }
}
```

type of ex is Example

type of return value is wrong

Symbol table:

- Example: class
  - x: field (int)
  - Foo: method
    - y: local (int)
    - ex: local (Example)
  - Bar: method
    - x: param (string)

Semantic analysis example: operator identification

```
public abstract class Example
{
    private int x;

    public void Foo()
    {
        int y;
        if (!flag)
        {
            x = 3 + y;
            var ex = new Example();
        }
    }
    public int Bar(string x)
    {
        return "hello" + x;
    }
}
```

int operator+(int,int)

string operator+(string,string)

Symbol table:

- Example: class
  - x: field (int)
  - Foo: method
    - y: local (int)
    - ex: local (Example)
  - Bar: method
    - x: param (string)

Semantic analysis example: other language rules

```
public abstract class Example
{
    private int x;

    public void Foo()
    {
        int y;
        if (!flag)
        {
            x = 3 + y;
            var ex = new Example();
        }
    }
    public int Bar(string x)
    {
        return "hello" + x;
    }
}
```

y is not initialized

Example cannot be instantiated

Symbol table:

- Example: class
  - x: field (int)
  - Foo: method
    - y: local (int)
    - ex: local (Example)
  - Bar: method
    - x: param (string)

Semantic analysis

- Semantic analysis
  - construction of the symbol table
  - name analysis
  - type analysis
  - operator identification
  - consistency checking
  - other language rules
- Why not as part of the syntax analysis?
  - programming languages are context sensitive (CF grammars are insufficient for them)
  - definitions later in code, cross references, interdependencies

Symbol table

- The database of the compiler
- Contains the definitions of all symbols
  - name
  - meaning
  - context
- Goals:
  - symbol lookup
  - symbol equality

This lecture: Semantic analysis

- I. Semantic analysis
- II. Attribute grammars
- III. Name analysis
- IV. Type analysis, operator identification
- V. Other language rules

Attribute grammar

- Semantic analysis needs extra information in the syntax tree
- Idea: define attributes for the syntax nodes
  - attribute definition: name and expression
- Two kinds of attributes:
  - inherited: top-down evaluation
    - defined for non-terminals on the right side of the rules
  - synthesized: bottom-up evaluation
    - defined for non-terminals on the left side of the rules

Attribute grammar example

CF grammar:

```
A → T x = E
E → E+C | C
C → 1 | "a"
T → int | string
```

Attribute grammar:

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType
```

synthesized (bottom-up)

inherited (top-down)

```
E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

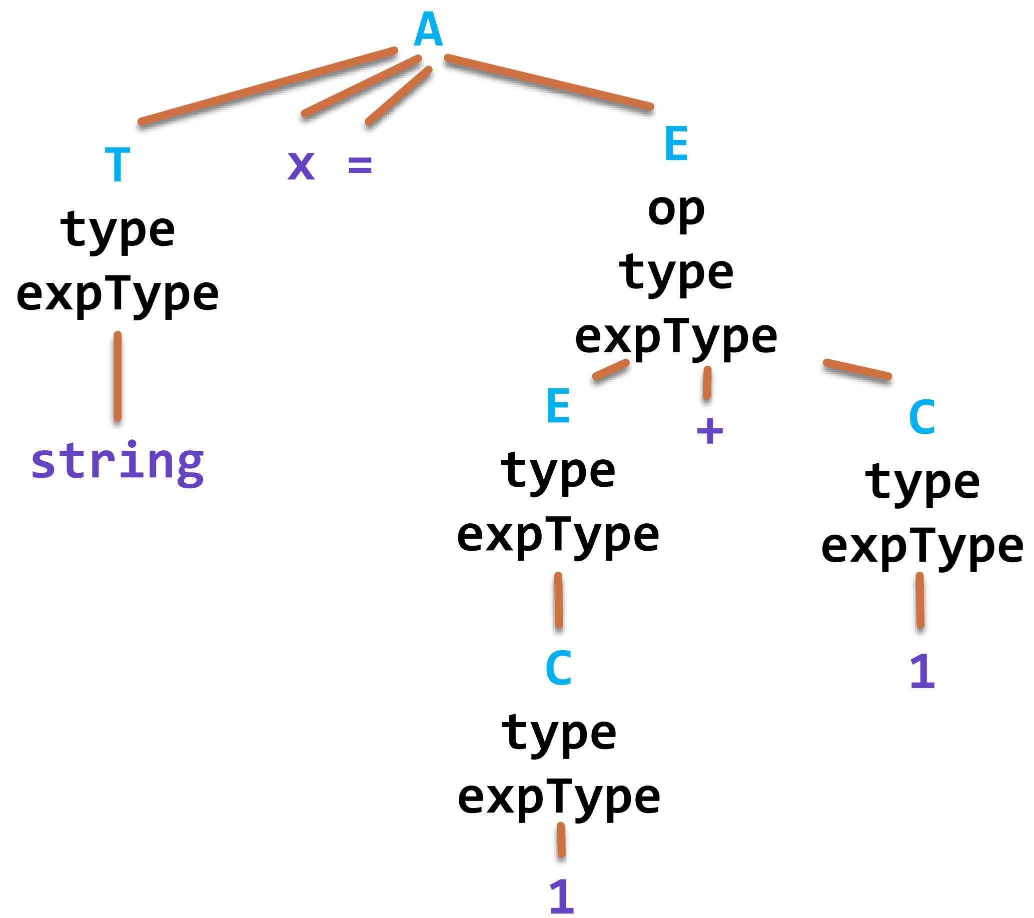
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



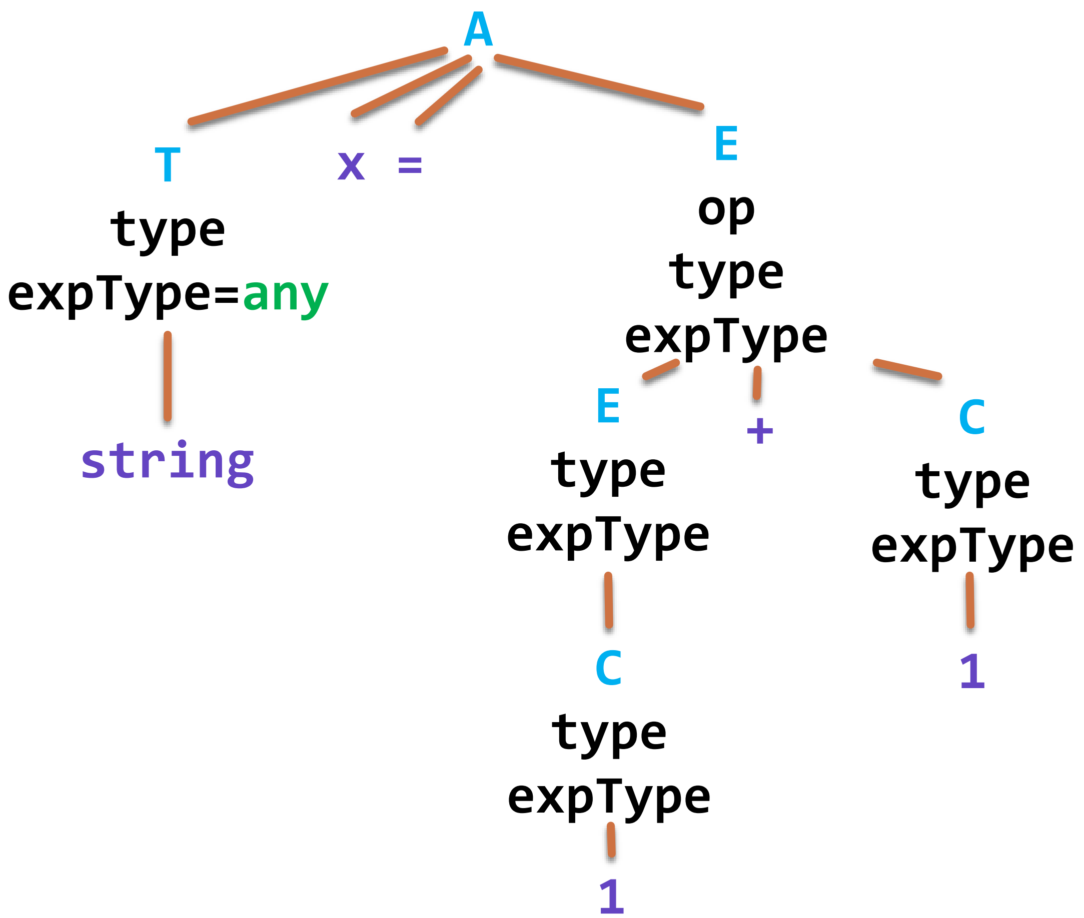
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



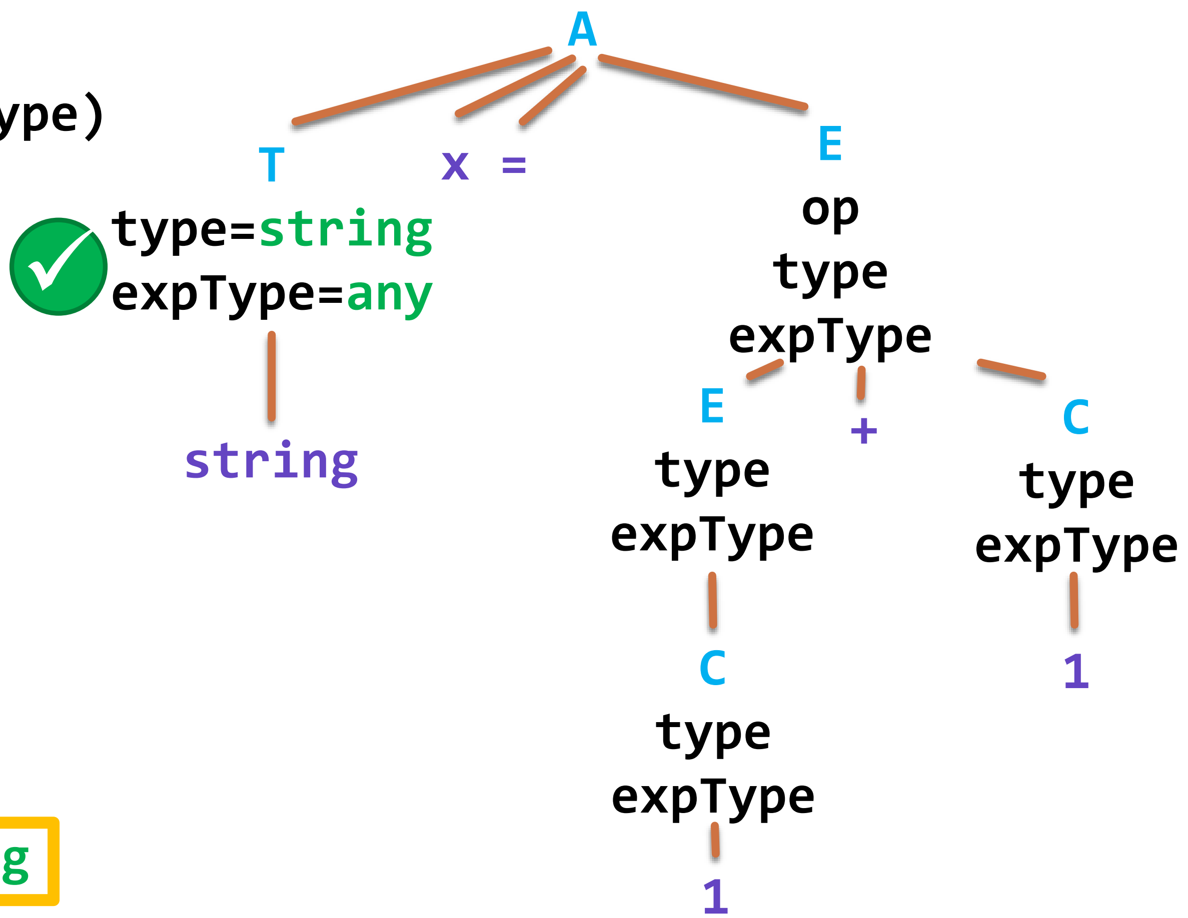
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



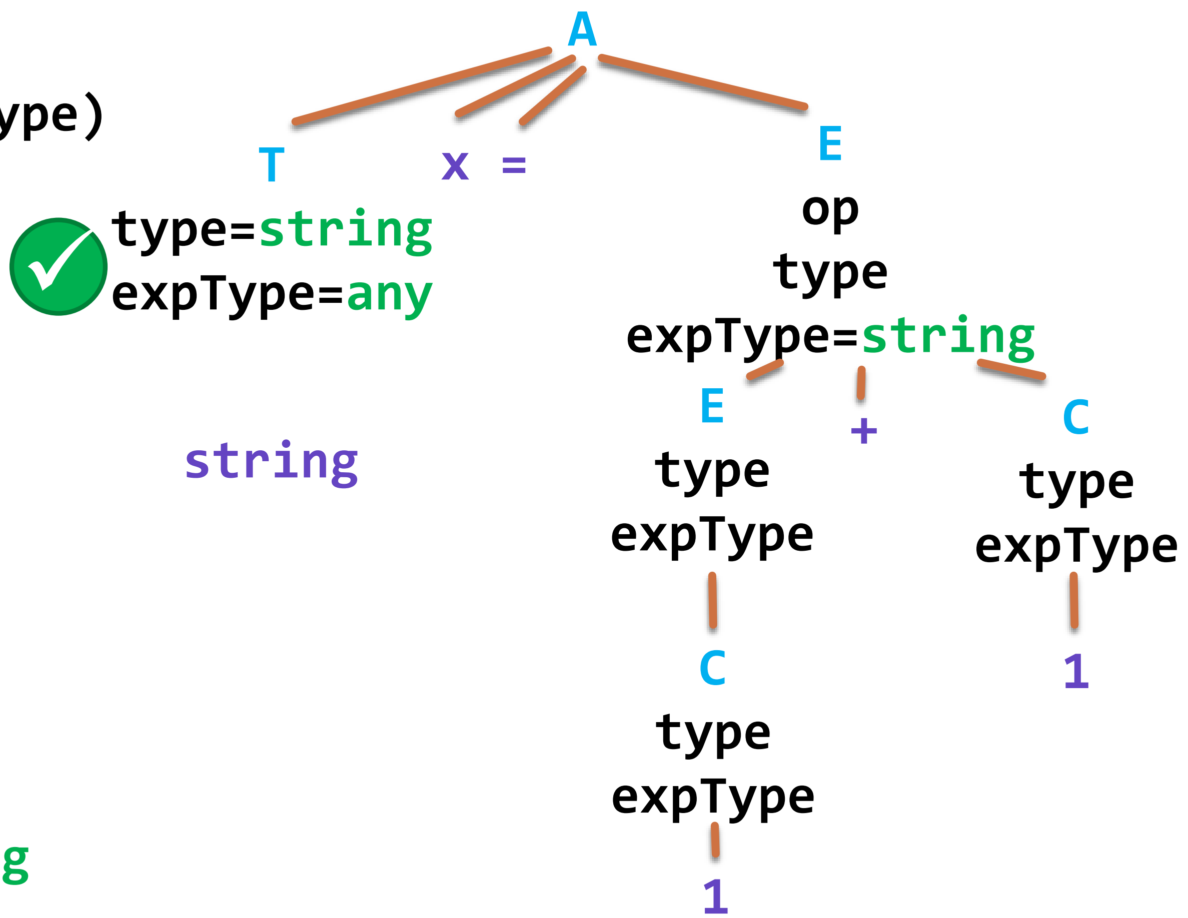
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



Example: computing attributes

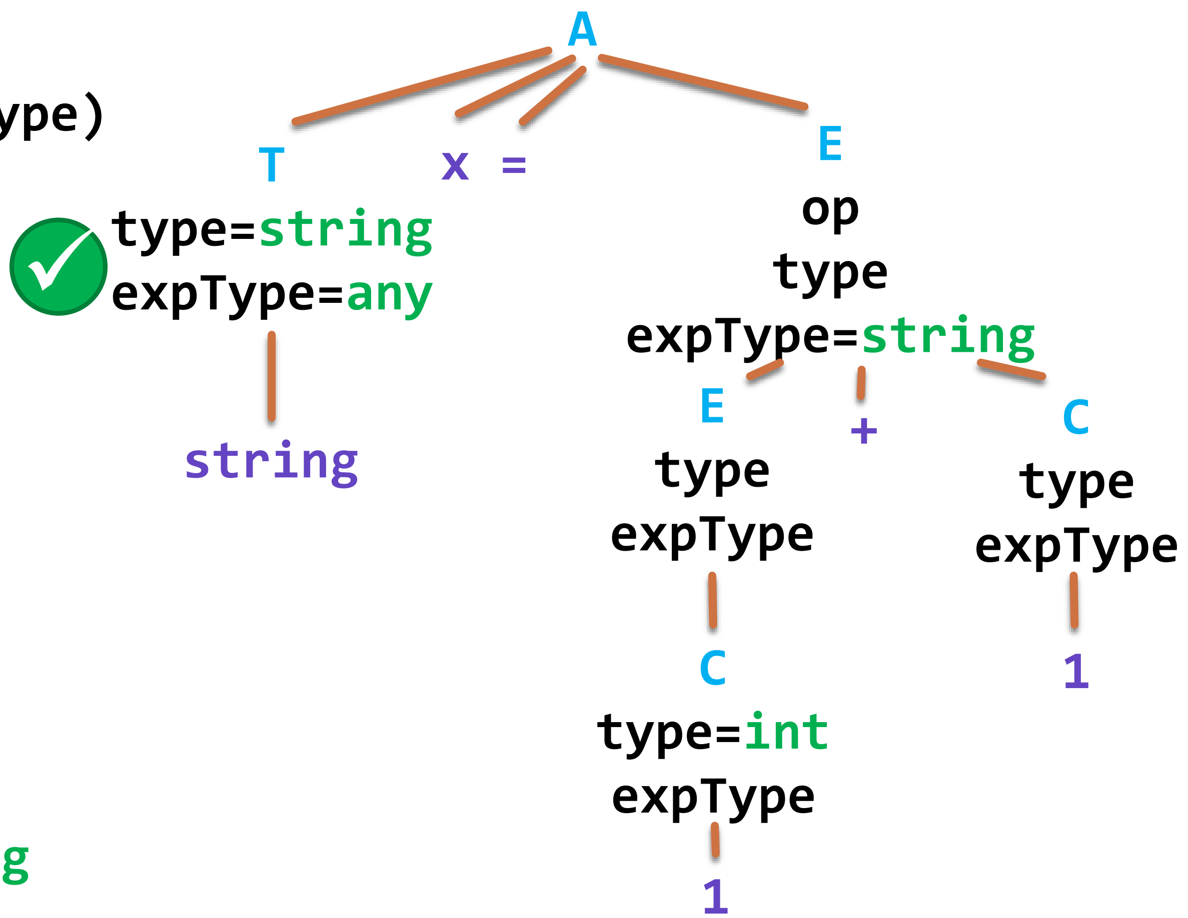
```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string
```

```
T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



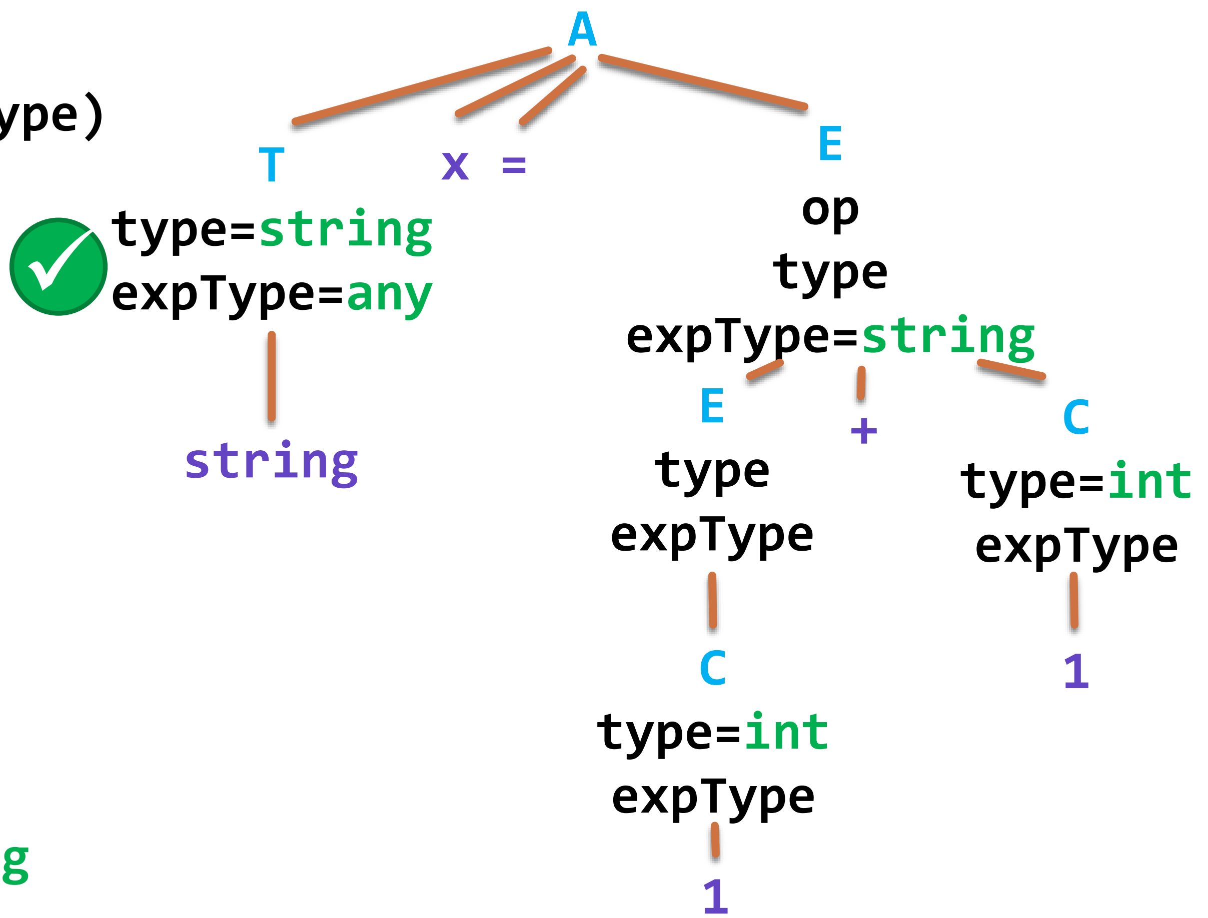
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



Example: computing attributes

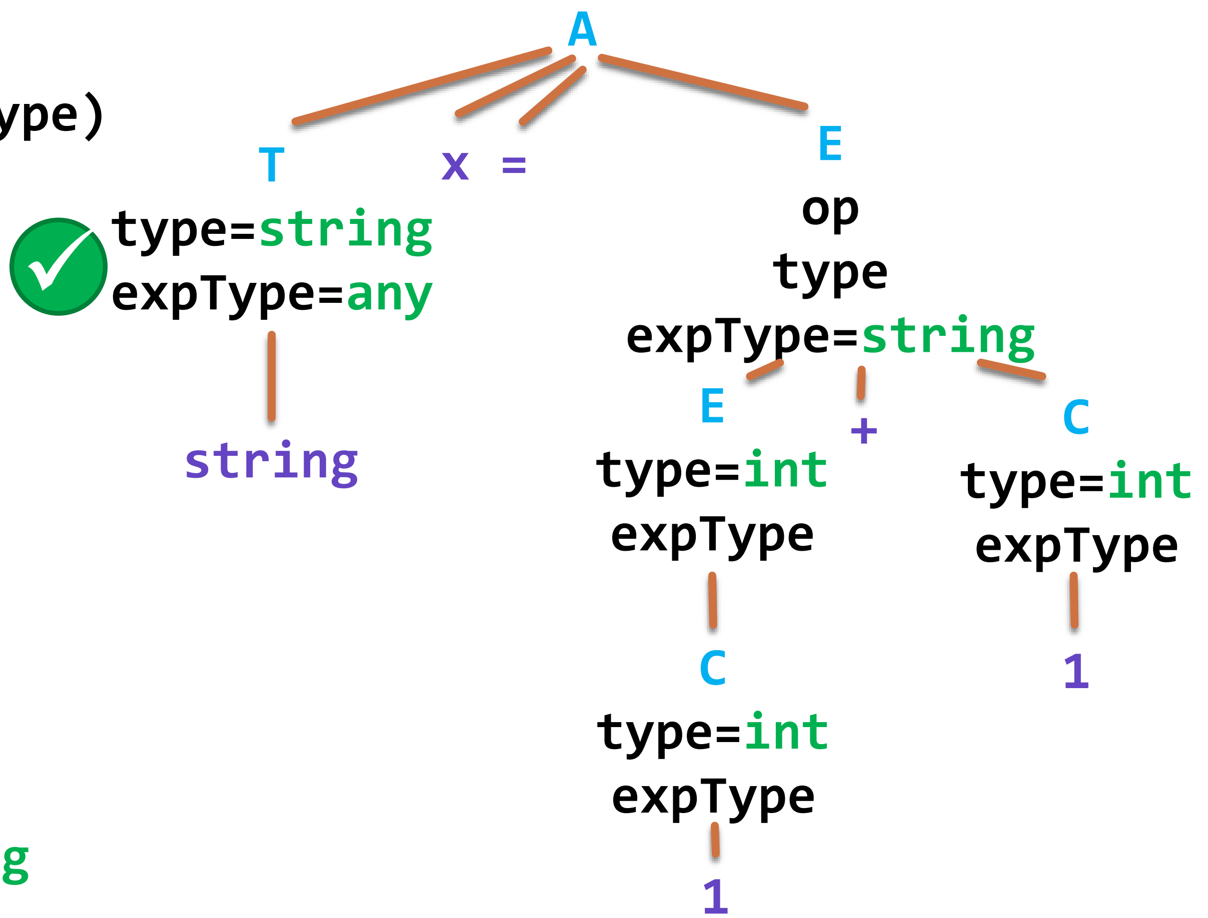
```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string
```

```
T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



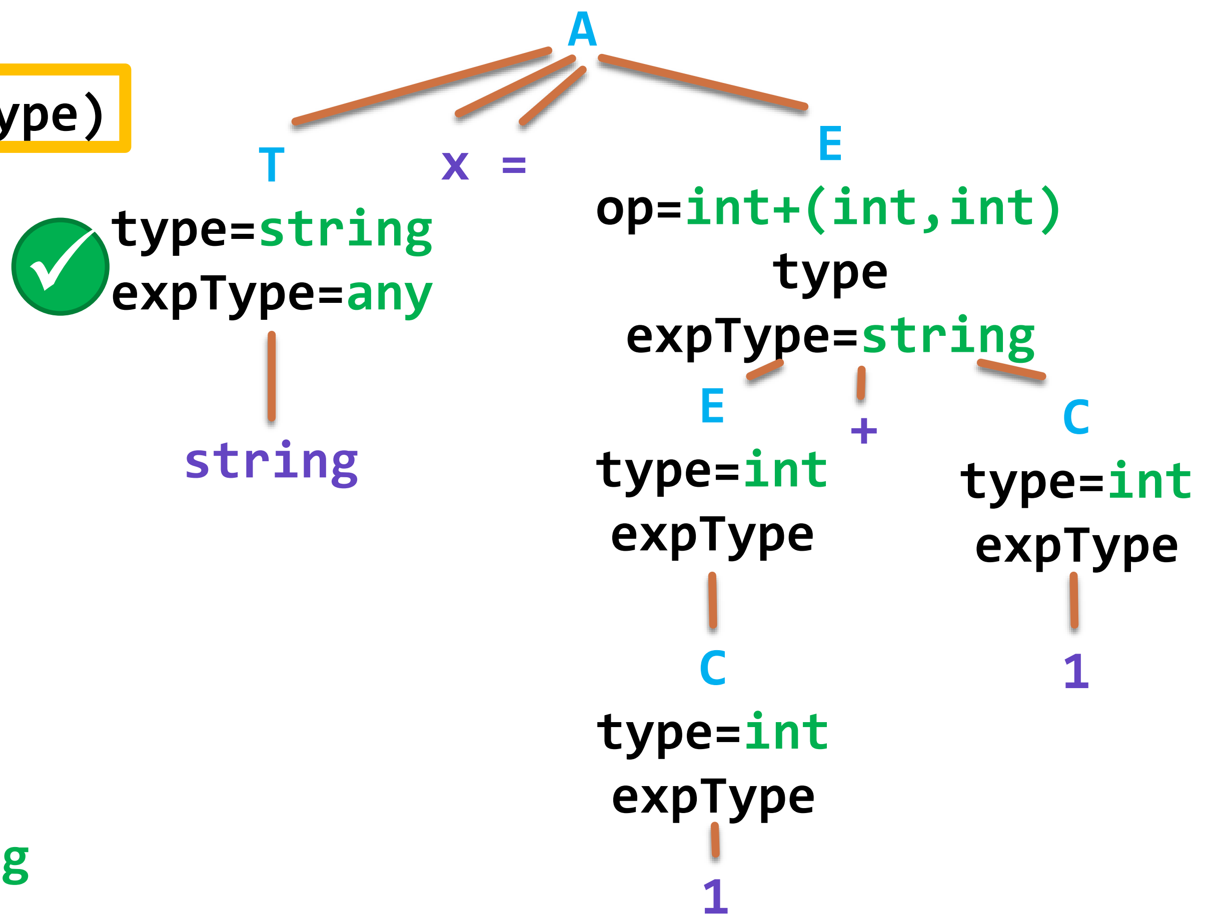
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



Example: computing attributes

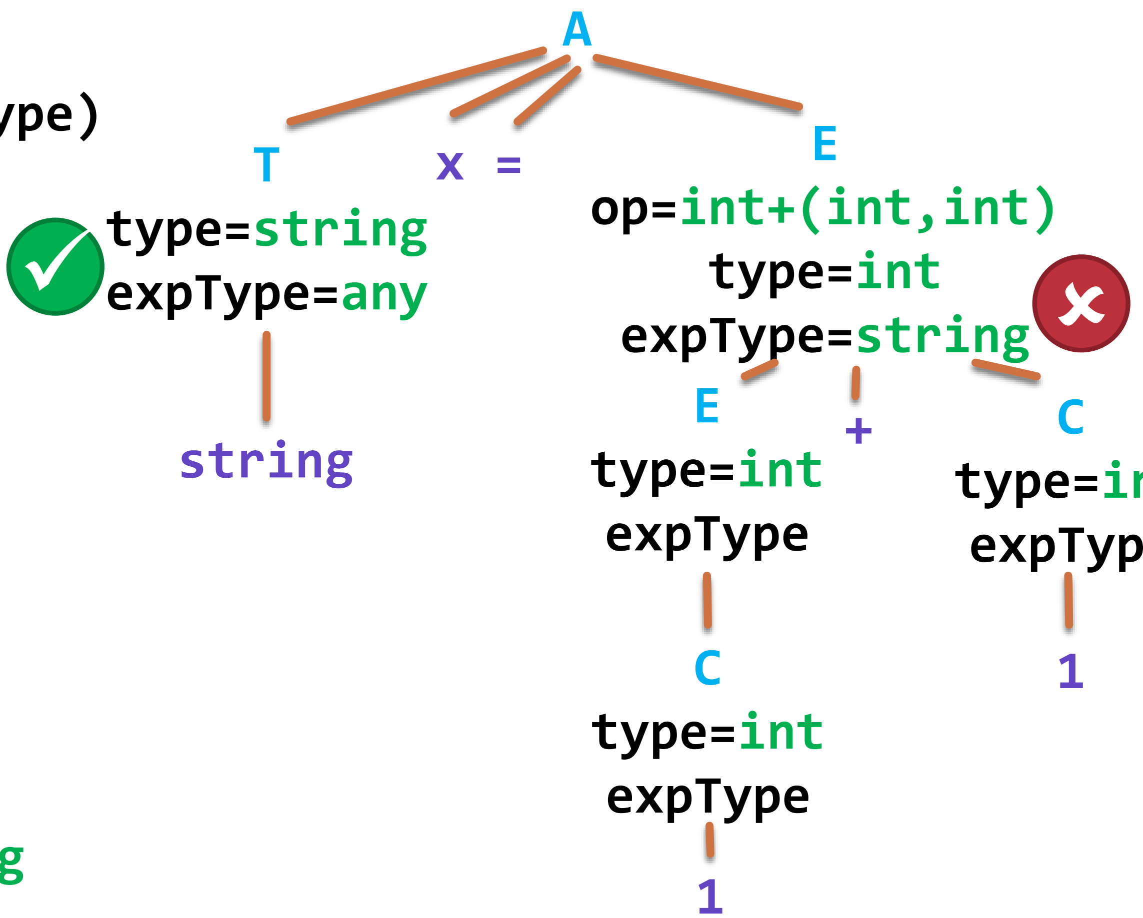
```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Error: string expression is expected!

Syntax tree with attributes:



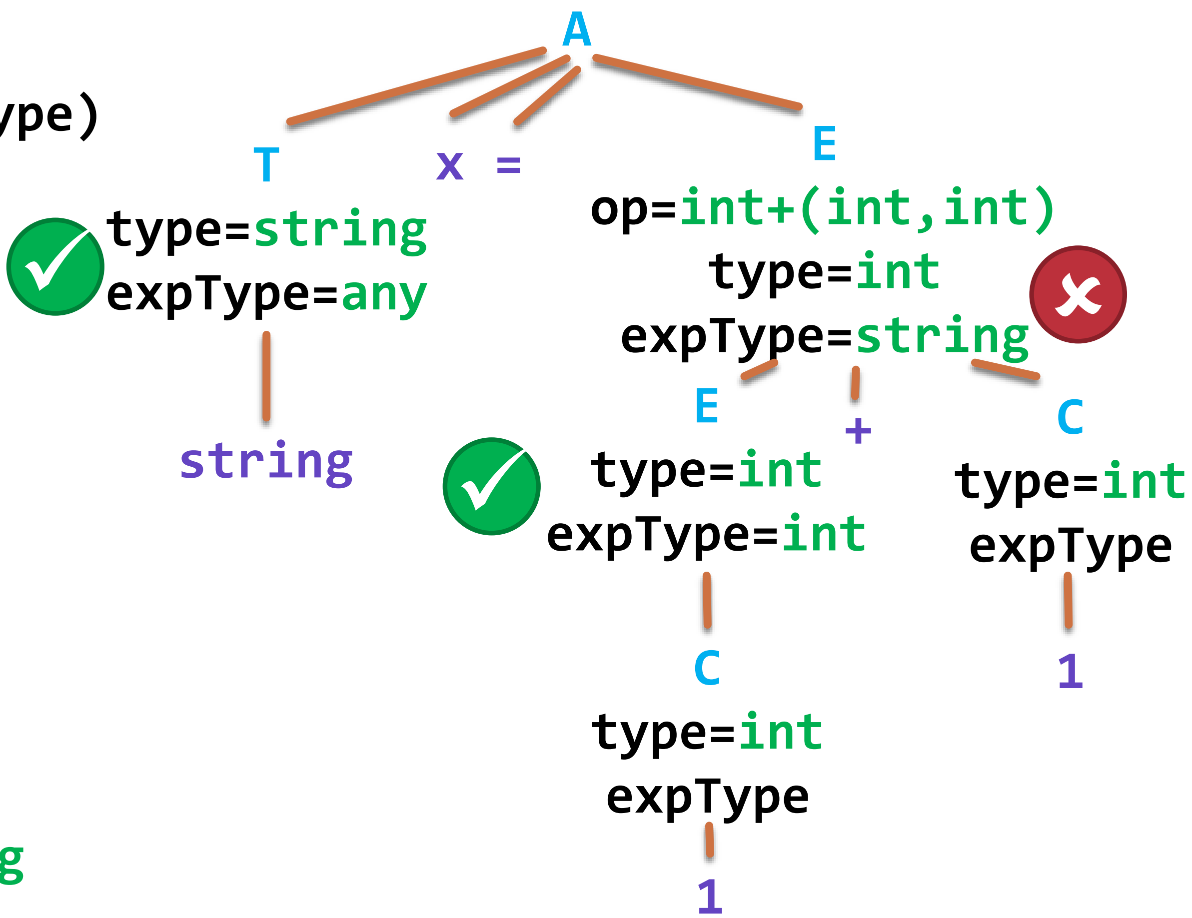
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



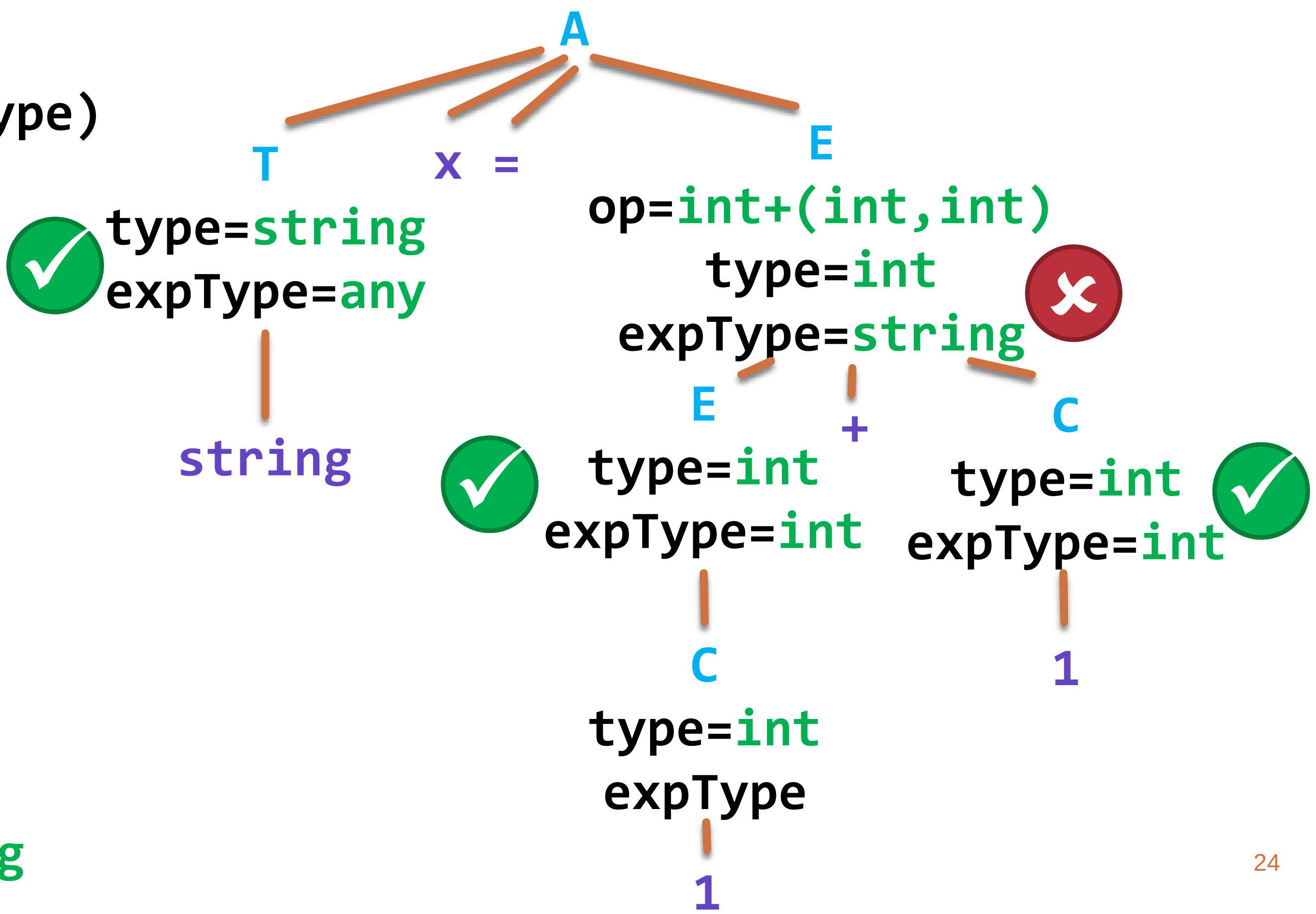
Example: computing attributes

```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string

T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



Example: computing attributes

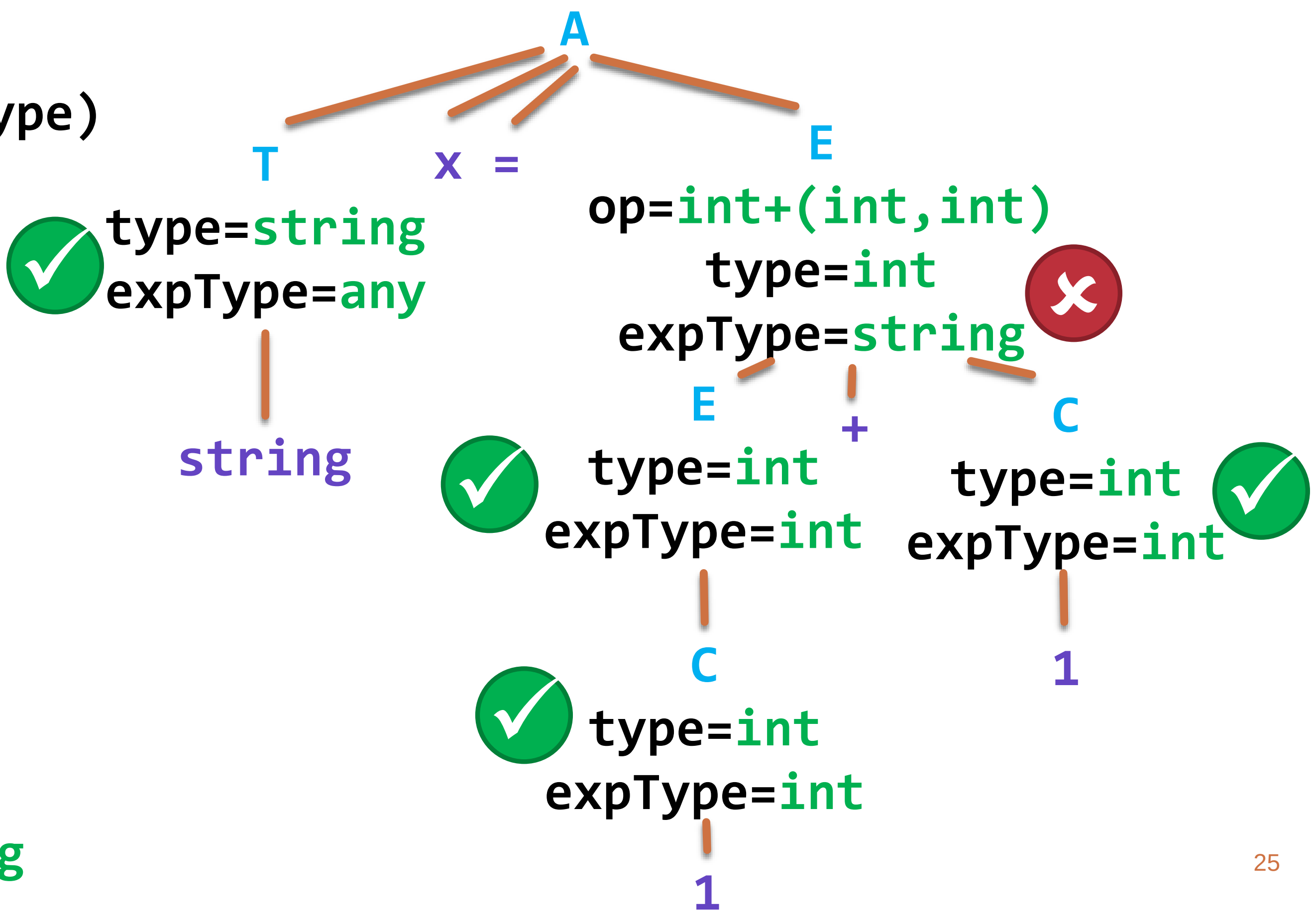
```
A → T x = E
E.expType = T.type
T.expType = any

E → E+C
E[1].op = GetOperator(+,E[2].type,C.type)
E[1].type = E[1].op.type
E[2].expType = E[1].op.expType
C.expType = E[1].op.expType

E → C
E.type = C.type
C.expType = E.expType

C → 1
C.type = int

C → "a"
C.type = string
```

```
T → int
T.type = int

T → string
T.type = string
```

Program code:

```
string x = 1+1
```

Syntax tree with attributes:



Attribute evaluation order

- Some left-to-right or right-to-left DFS runs
  - Left-, Right-, Alternating Attribute Grammar: LAG(k), RAG(k), AAG(k)
- Evaluation of sets of attributes globally (independent of non-terminals)
  - Ordered Attribute Grammar: OAG (-> checking this property has polynomial complexity)
  - topological order of attributes
- Evaluation of sets of attributes locally (dependent on the given non-terminal)
  - Partitionable Attribute Grammar: PAG (-> checking this property is NP-complete)
  - adding a few extra dependencies to PAG results in OAG
- Lazy evaluation of attributes
  - Well-defined Attribute Grammar: WAG

There must not be loops!

Compilation based on attribute grammars

- Attribute grammar: declarative description
- There are compilers based on attribute grammars:
  - Ox: https://sourceforge.net/projects/ox-attribute-grammar-compiler/
  - Silver: https://github.com/melt-umn/silver
- Can be programmed manually in an imperative way, too:
  - e.g., Roslyn has a similar imperative solution, however, attributes are not attached to the syntax tree: they are computed for the symbols

This lecture: Semantic analysis

- I. Semantic analysis
- II. Attribute grammars
- III. Name analysis
- IV. Type analysis, operator identification
- V. Other language rules

Name analysis

- Register symbol definitions into the symbol table
  - namespaces, types, fields, operations, parameters, variables, etc.
  - pre-defined symbols
  - detecting name collisions (can depend on the kind of the symbol)
  - merging symbols (e.g., namespaces, partial classes)
- Matching symbol references with the corresponding symbol definition
  - e.g., reading-writing the value of variables, constants, parameters, fields
  - e.g., reference to a namespace or type, method call, goto label
- The same name can mean different things in different places
  - the context (scope) of the name is important

Example: C-style blocks

```
Block → { Decls Stmts }
Stmts.scope = append(Decls.scope, Block.scope)
```

inherited

```
Decls → Decls Decl
Decls[1].scope = append(Decls[2].scope, Decl.scope)
```

synthesized

```
Decl → Type Var ;
Decl.scope = new Scope(Var.symbol)

Stmts → Stmts Stmt
Stmts[2].scope = Stmts[1].scope
Stmt.scope = Stmts[1].scope

Stmt → ... Var ... ;
Var.symbol = Stmt.scope.find(Var.name)
```

What is the problem with this solution?

Problems

- Problem: an attribute must be either inherited or synthesized, but cannot be both
- Solution: separating the attribute into two different attributes
- Alternatives for the scope of the definitions:
  - 1. can only be referenced after the definition (e.g., local variables)
    - scopeIn (inherited): symbols defined before this position
    - scopeOut (synthesized): symbols defined until this position
  - 2. can be referenced even before the definition (e.g., methods in the same class)
    - scopeAcc (synthesized): accumulation of symbol definitions
    - scopeLkp (inherited): symbols are looked up from this set

Example: can only be referenced after the definition

```
Block → { Decls Stmts }
Decls.scopeIn = Block.scopeIn
Stmts.scopeIn = Decls.scopeOut
Block.scopeOut = Block.scopeIn

Decls → Decls Decl
Decls[2].scopeIn = Decls[1].scopeIn
Decl.scopeIn = Decls[2].scopeOut
Decls[1].scopeOut = Decl.scopeOut

Decl → Type Var = Var ;
Type.symbol = Decl.scopeIn.find(Type.name)
Var[2].symbol = Decl.scopeIn.find(Var[2].name)
Decl.scopeOut = append(new Scope(Var[1].symbol), Decl.scopeIn)
```

Example: can be referenced even before the definition

```
Block → { Decls Stmts }
Decls.scopeLkp = append(Block.scopeLkp, Decls.scopeAcc)
Stmts.scopeLkp = Decls.scopeLkp
Block.scopeAcc = new Scope()

Decls → Decls Decl
Decls[2].scopeLkp = Decl[1].scopeLkp
Decl.scopeLkp = Decls[1].scopeLkp
Decls[1].scopeAcc = append(Decls[2].scopeAcc, Decl.scopeAcc)

Decl → Type Var = Var ;
Type.symbol = Decl.scopeLkp.find(Type.name)
Var[2].symbol = Decl.scopeLkp.find(Var[2].name)
Decl.scopeAcc = new Scope(Var[1].symbol)
```

Example: qualified name

```
QualifiedName → QualifiedName . Name
Name.symbol = QualifiedName[2].type.find(Name.name)
QualifiedName[1].type = Name.symbol.type

QualifiedName → Name
Name.symbol = QualifiedName.scopeLkp.find(Name.name)
QualifiedName.type = Name.symbol.type
```

Where is this coming from?

Name and type analysis affect each other!

Other alternatives for scopes: manual imperative solutions

- 1. Symbol stack
  - changes dynamically
  - small memory footprint
  - must resolve everything in a single pass
- 2. Linked lists
  - static structure, does not change after it has been created
  - needs more memory
  - can handle more complex cases
  - (the previous examples with attribute grammars also built linked lists in a declarative way)

Typical scopes with linked lists

Example code:

Namespace

Import

Alias

```
namespace MovieApp.Service
{
  using MovieApp.Data;
  using C = MovieApp.Data.Cinema;

  public interface IMovieService
  {
    Movie GetMovie(int id);
  }

  public class MovieService : RestService, IMovieService
  {
    public Movie GetMovie(int id) { /*...*/ }
    public void ShowMovie(int id, C cinema) { if (true) { /*...*/ } }
  }
}
```

Current type (this)

Base type (base)

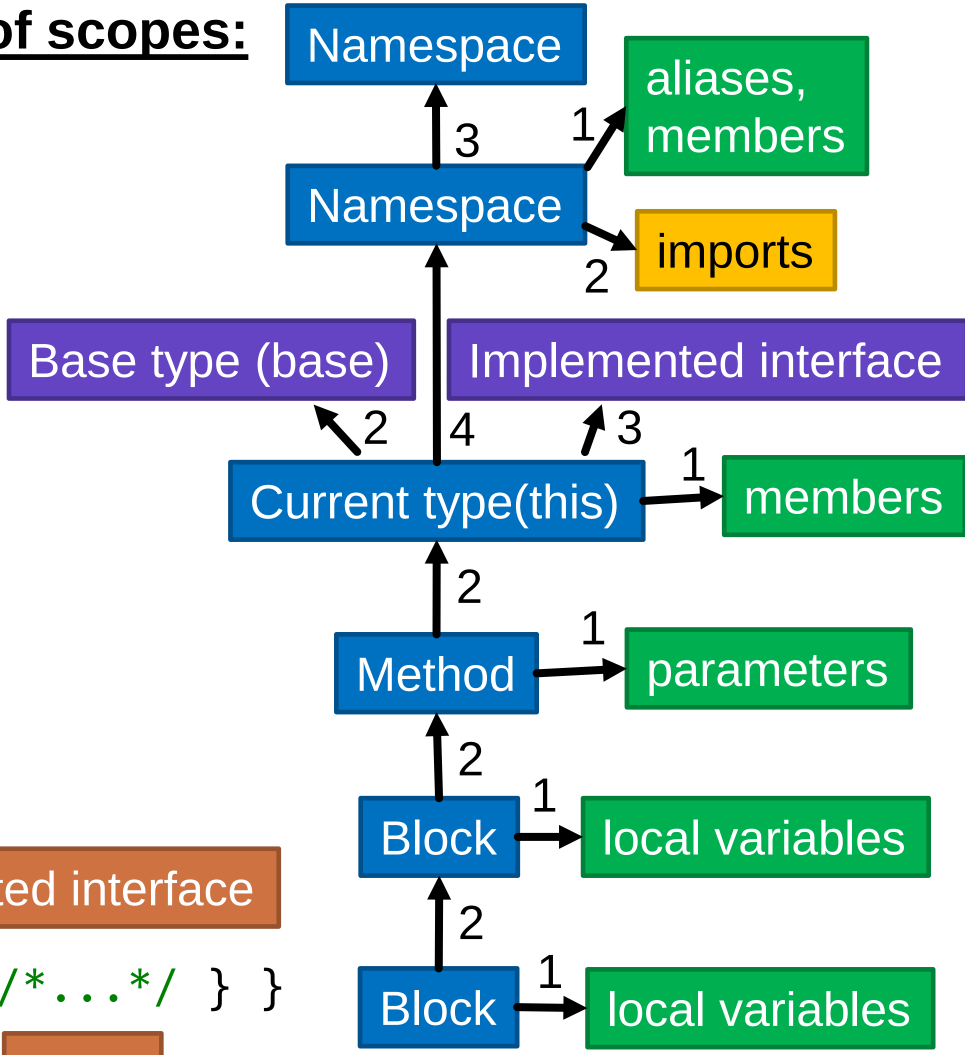
Implemented interface

Method

Block

Block

Linked list of scopes:



Errors detected by name analysis

- Multiple definition
  - the same name is defined in the same scope: won’t be unique in the symbol table
  - not always an error (e.g., overloading or different kinds of symbols)
- Undefined symbol
  - the referenced name is missing in the scope hierarchy
- Ambiguous symbol
  - the referenced name cannot be resolved unambiguously
  - e.g., because of a multiple definition, or because of a collision of a defined and imported symbol

This lecture: Semantic analysis

- I. Semantic analysis
- II. Attribute grammars
- III. Name analysis
- IV. Type analysis, operator identification
- V. Other language rules

Type analysis

- Type: set of values and operators defined on these values
  - primitive types (int, float, double, char, bool, etc.)
  - user defined types (struct, union, class, array, etc.)
- Goal of type analysis:
  - determine the types of names, operands and expressions
  - type inference (e.g., `var` keyword in C#)
- Type analysis is necessary for:
  - name analysis and operator identification
  - value and type conversions
  - consistency checking

Type analysis

- Type error:
  - an operation is performed on a symbol which is invalid according to the symbol’s type
- Type checking:
  - before applying an operation the types of the symbols are checked whether they support this operation
  - at compile time: static type checking -> fast, but cannot handle all cases
  - at runtime: dynamic type checking -> slower, but more accurate
- Languages:
  - strongly typed language: all type errors can be detected at compile time
  - weakly typed language: type check at runtime -> type errors during execution are possible

Type equivalence

- Name equivalence:
  - two types are the same if they were defined with the same type definition
  - e.g., classes in C#
- Structural equivalence:
  - two types are the same if they were created by the same type constructor and type arguments
  - e.g., arrays, tuples, generic types
  - caution: recursive types can also be defined!

Covariance vs. Contravariance

```
virtual Base Method(Derived obj)
```

Covariance ⬇ ⬇ Contravariance

```
override Derived Method(Base obj)
```

Attributes for type analysis

- Actual type: type (synthesized)
- Expected type: expectedType (inherited)
- Implicit type conversion between them
- Example:

```
int x = 5;
double y = x;
```

```
x.expectedType = double
x.type = int
```

✓

```
if (x)
{
   ...
}
```

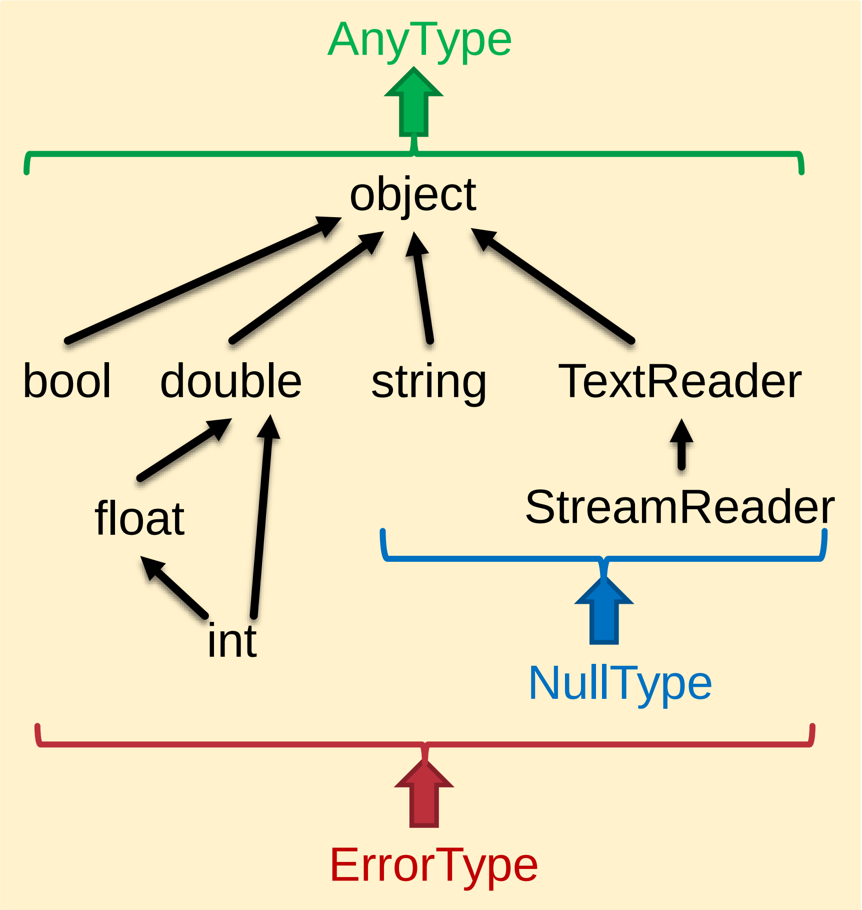
```
x.expectedType = bool
x.type = int
```

✗

Type hierarchy

- Implicit type conversion (type coercion): automatic
  - e.g., `int` -> `double`, derived -> base
- Explicit type conversion (type cast): not automatic
  - unsafe or leads to information loss
  - e.g., base -> derived, `double` -> `int`
- Type hierarchy:
  - ordering between types
  - direction of type conversion
- Types with special handling:
  - AnyType: anything allowed -> all types can be converted to it
  - NullType: type of `null` -> can be converted to reference types
  - ErrorType: type errors -> can be converted to all types

Type hierarchy example (C#):



Operations for type analysis

- GetBaseType: Type -> Type
  - returns the innermost type, e.g., `int?[] -> int`
- GetInnerType: Type -> Type
  - returns the directly contained type, e.g., `int?[] -> int?`
- AreEquivalent: Type x Type -> bool
  - whether two types are structurally equivalent, e.g.,
    `(int,bool),ValueTuple<int,bool> -> true`
- GetImplicitConversion: Type x Type -> Operator
  - returns the operator converting the first type to the second type implicitly
- GetExplicitConversion: Type x Type -> Operator
  - returns the operator converting the first type to the second type explicitly

either built-in or user defined

Operations for type analysis

- GetOperator: OpKind x Type -> Operator
  - unary operator with given kind and operand type
- GetOperator: OpKind x Type x Type -> Operator
  - binary operator with given kind and operand type

either built-in or
user defined

Steps for selecting an operator

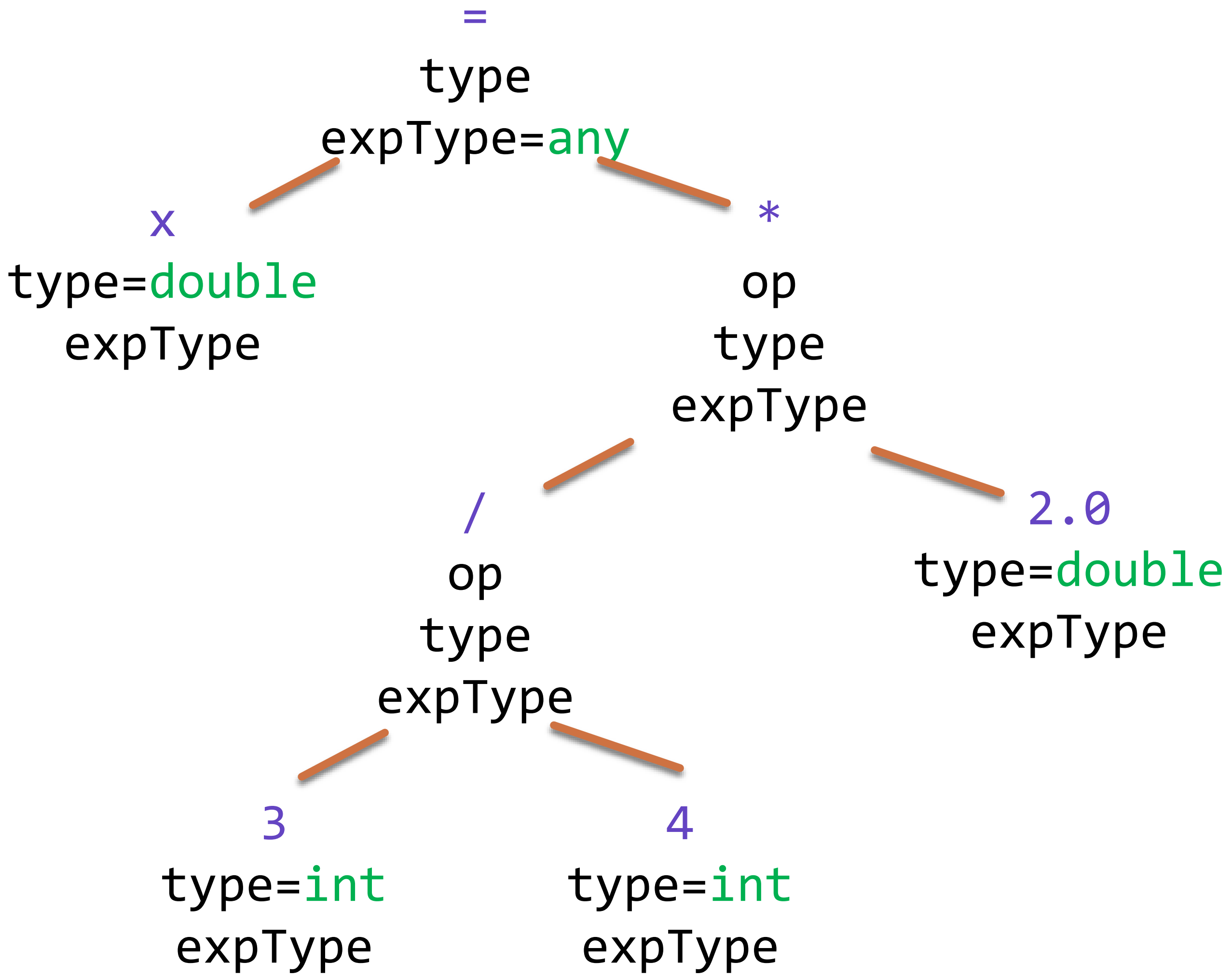
- 1. determine the actual types of the operands
- 2. collecting operator definition candidates
- 3. selecting the most appropriate candidate
  - error: if none is found, or more than one possibilities are found
- 4. determine the expected type for the operands
- 5. check the compatibility of actual and expected types
  - error: on type conversion incompatibility

- Resolution of function overloading is similar:
  - operand -> argument
  - operator -> function

Example: selecting operators

Statement:

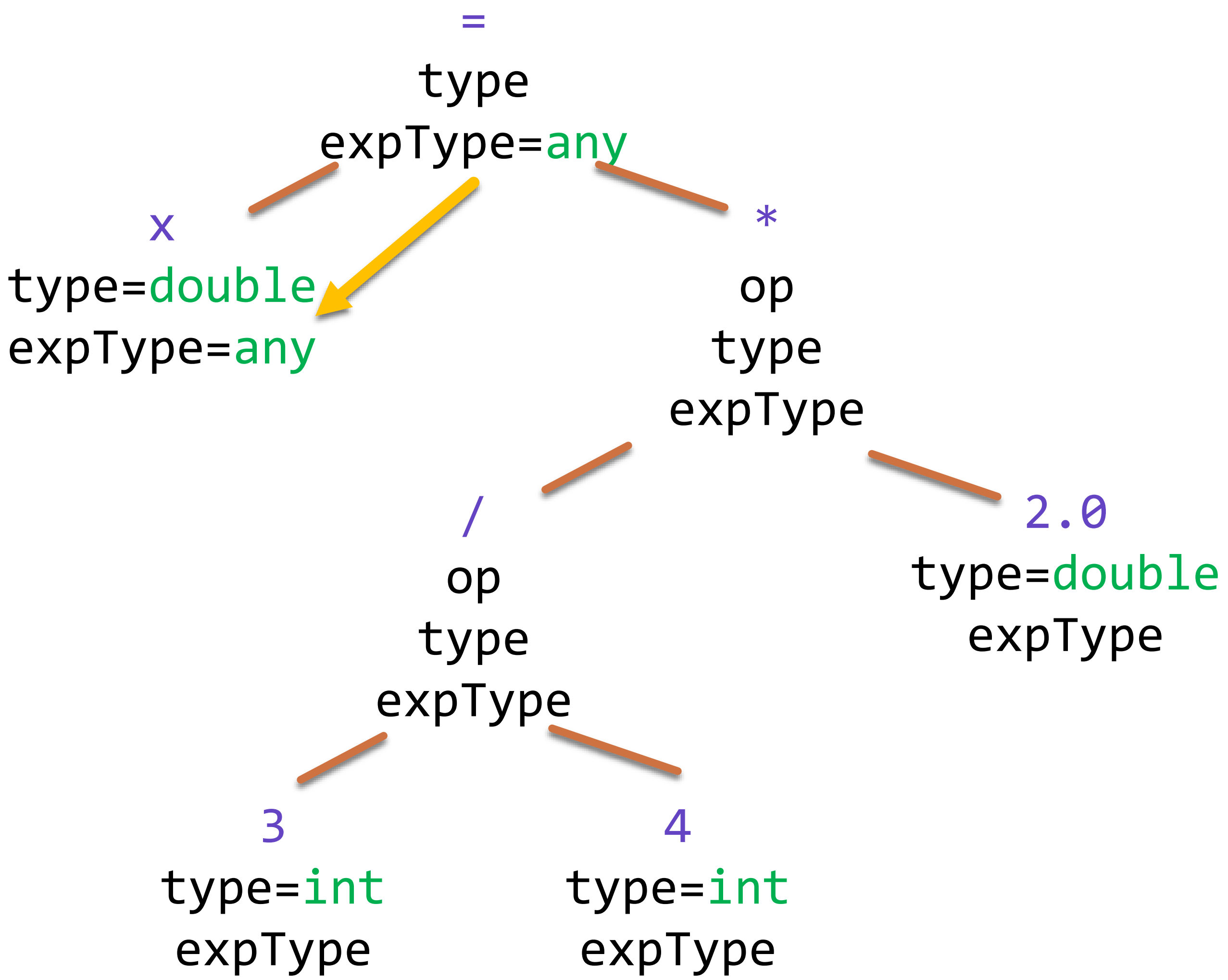
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

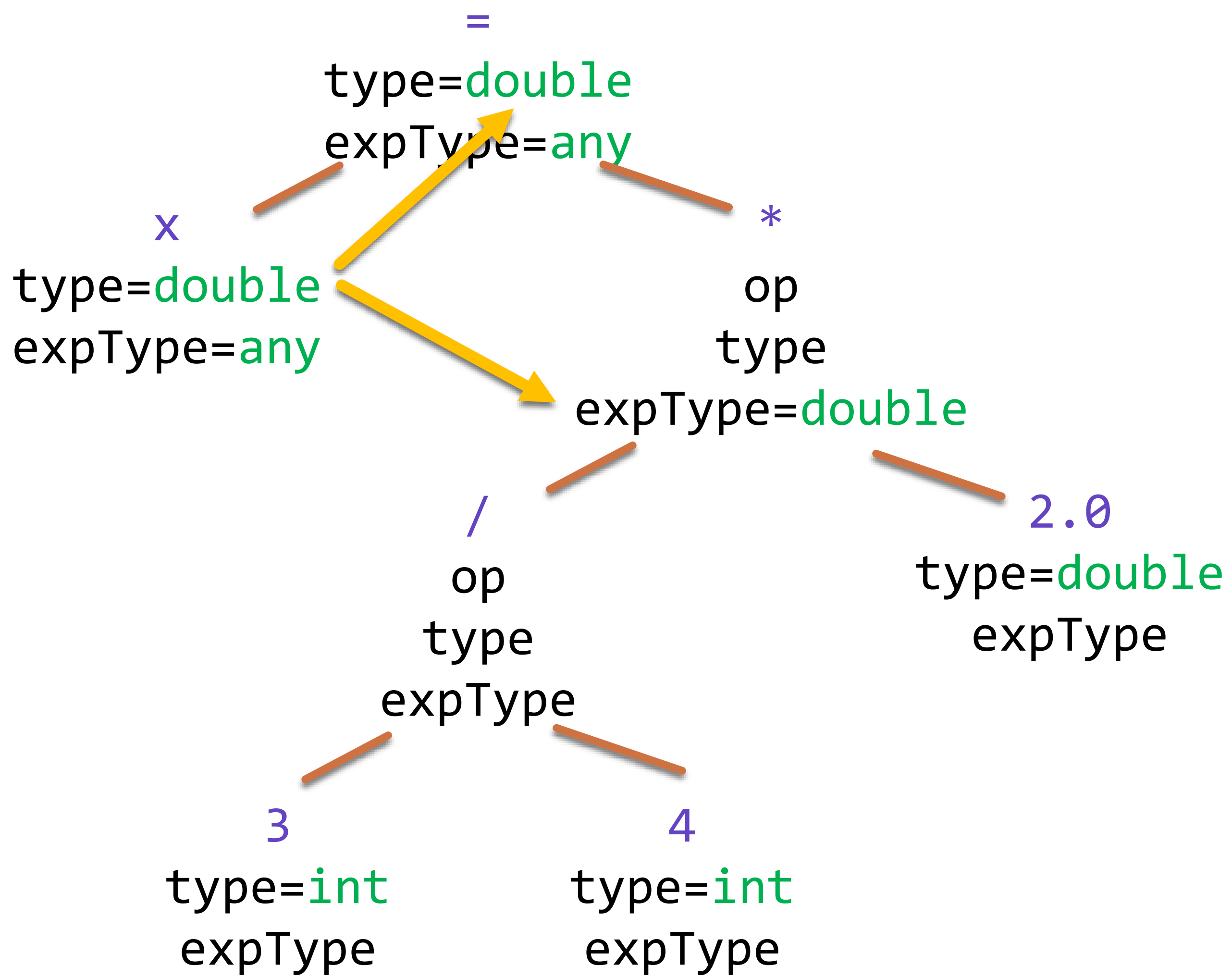
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

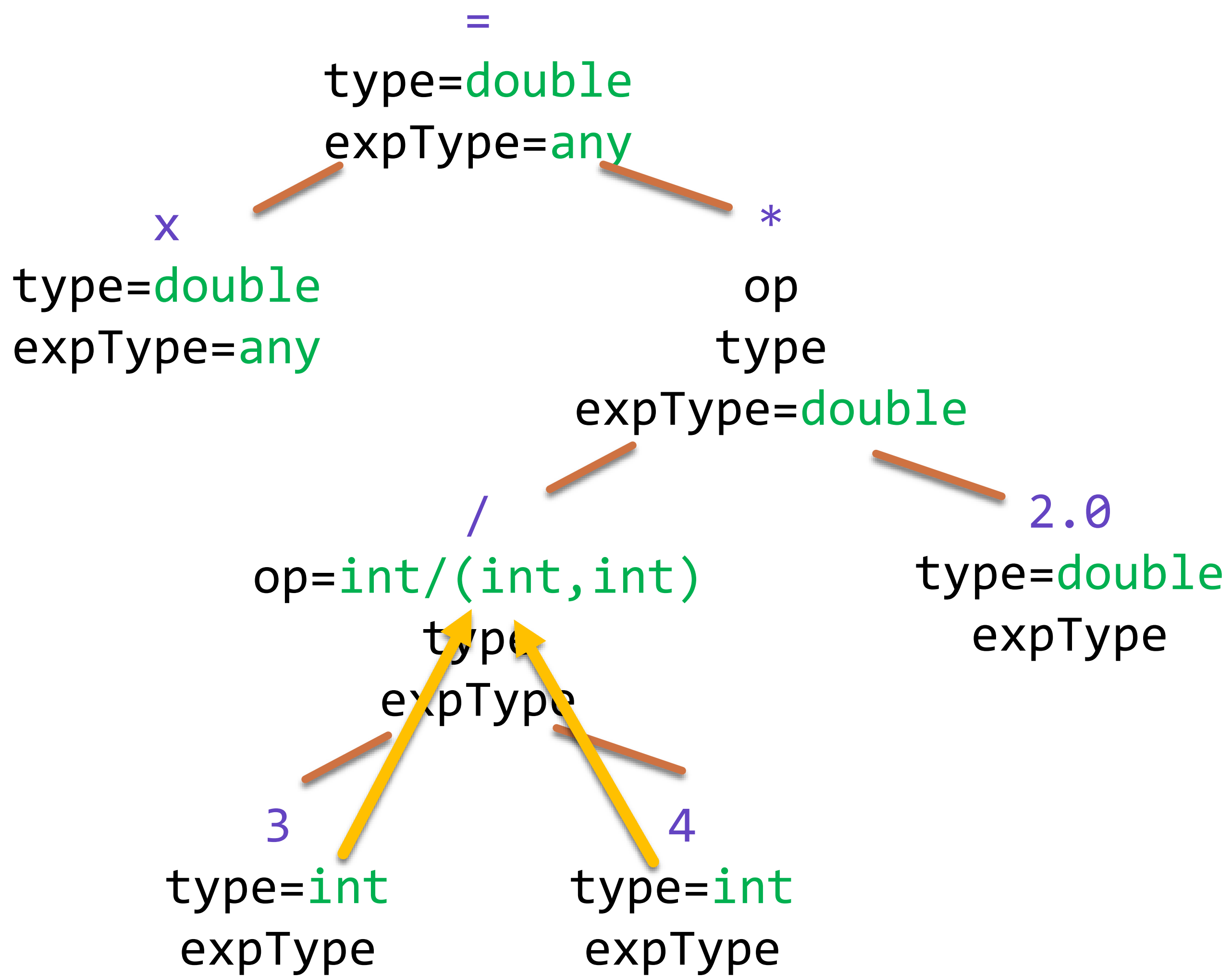
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

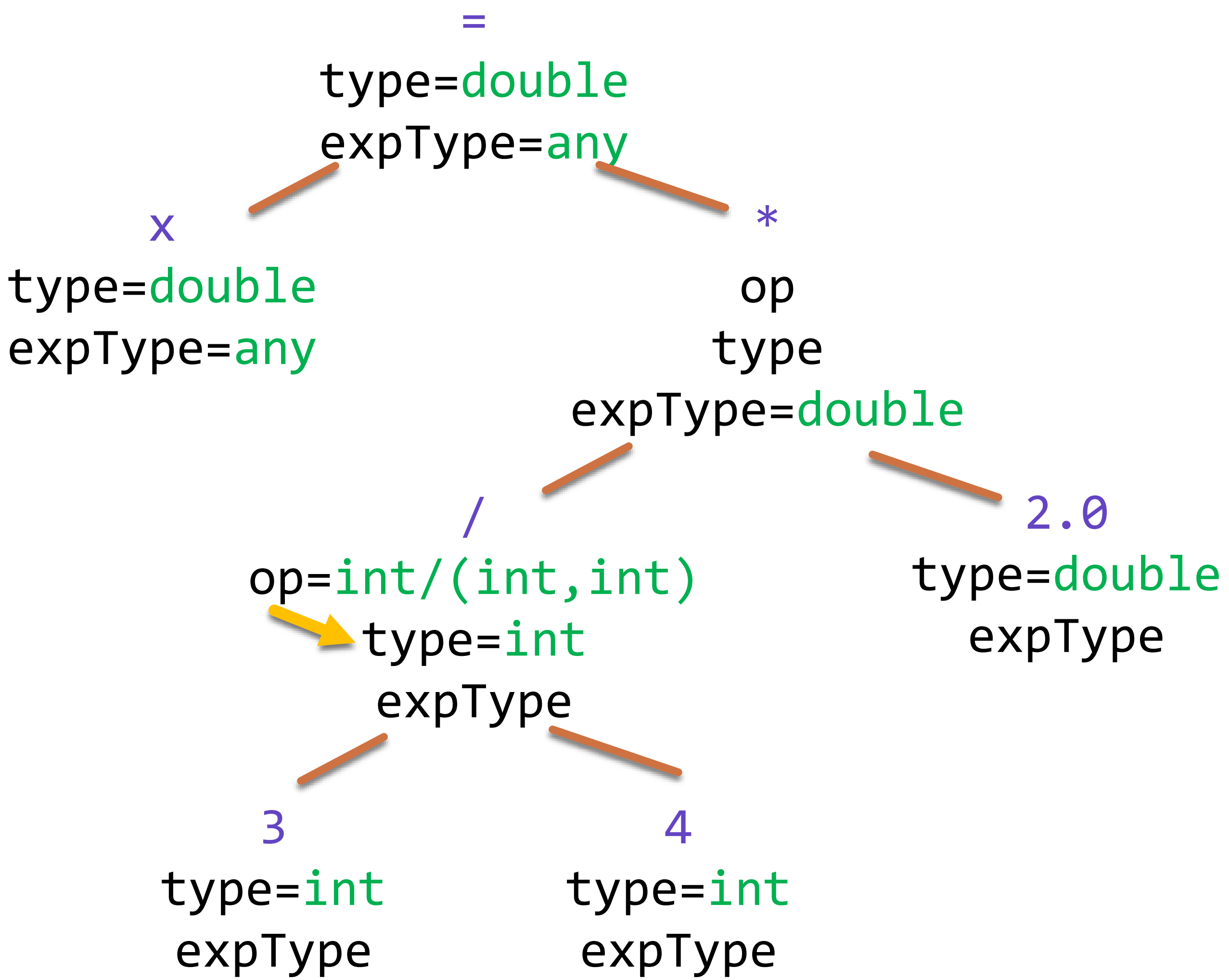
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

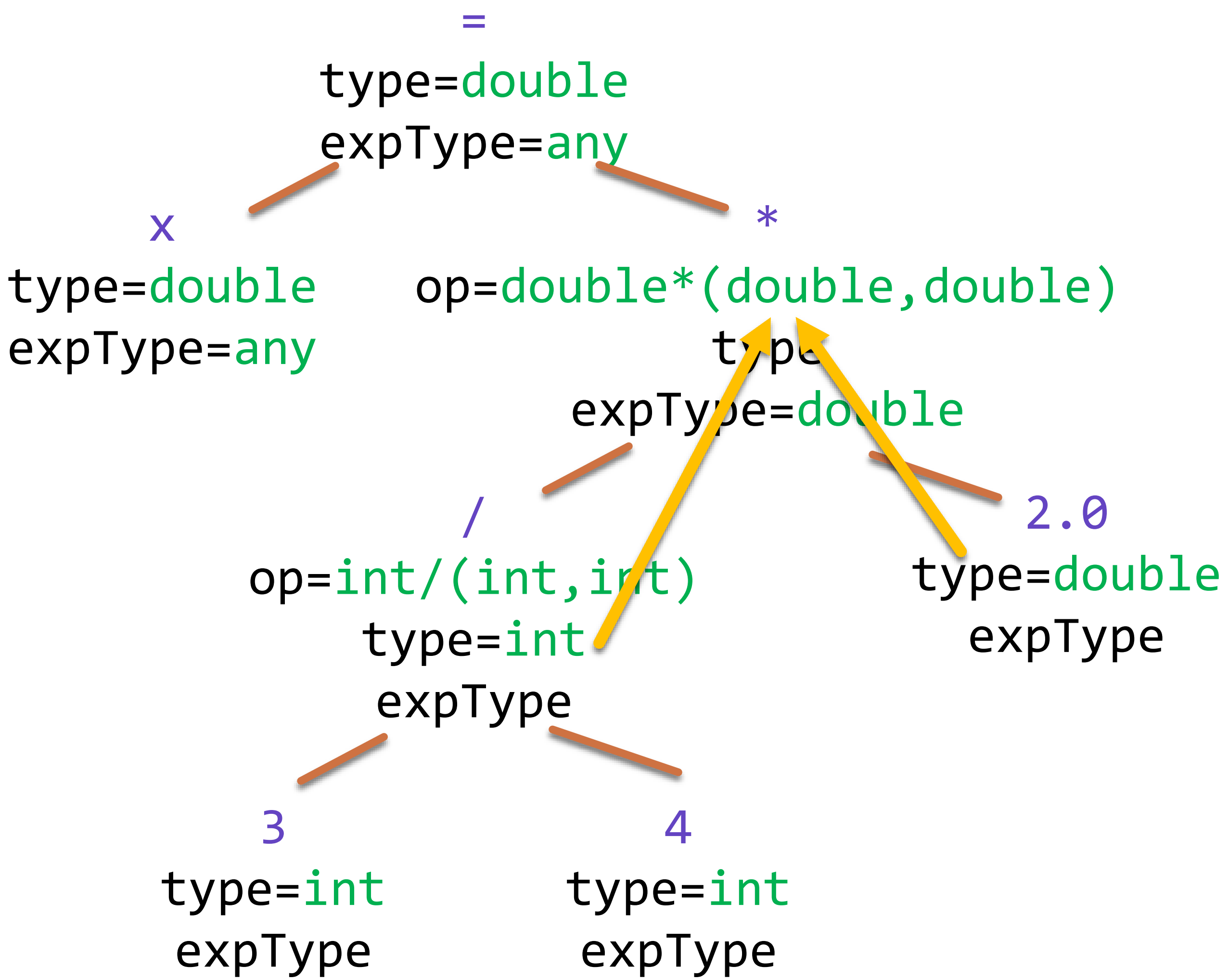
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

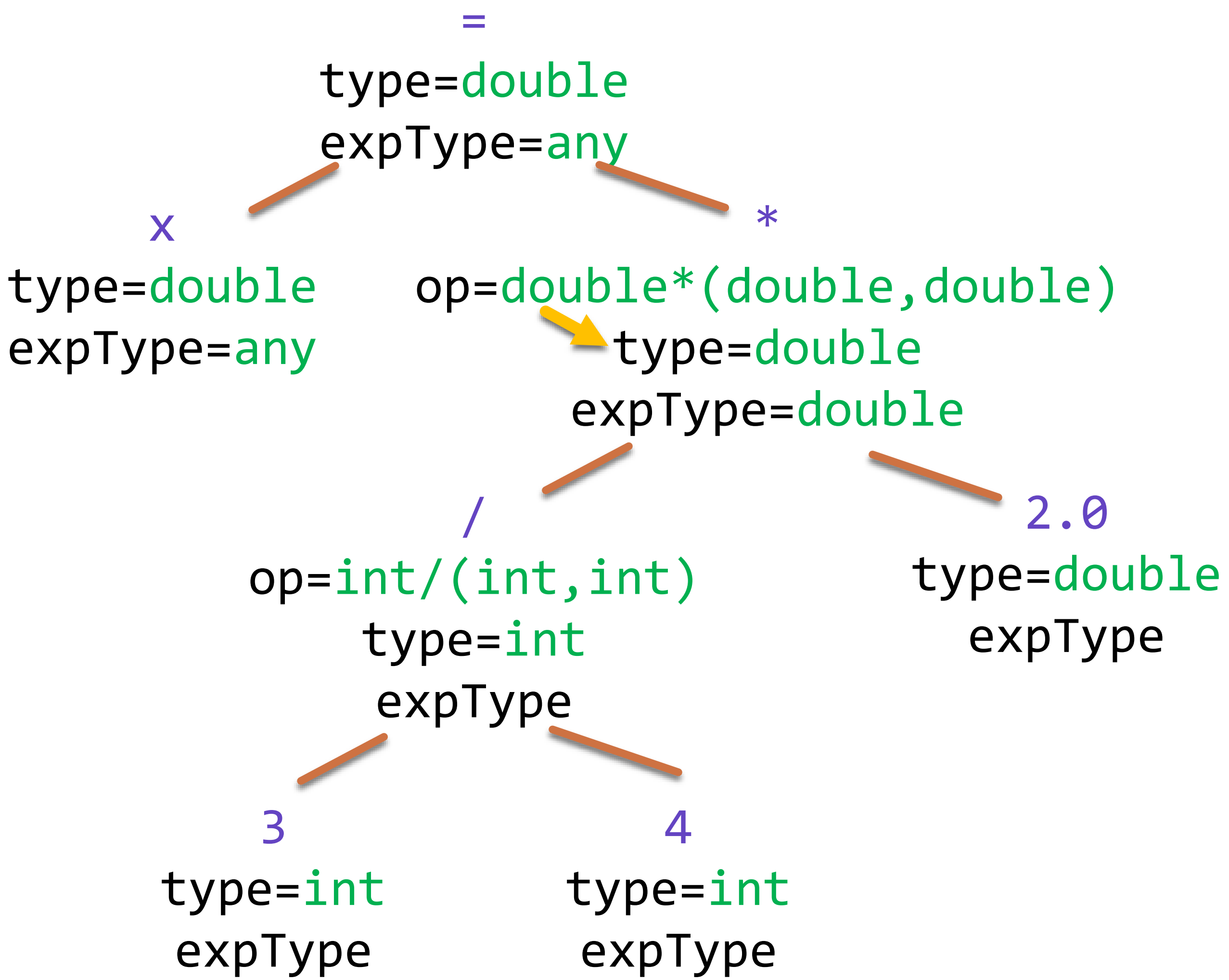
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

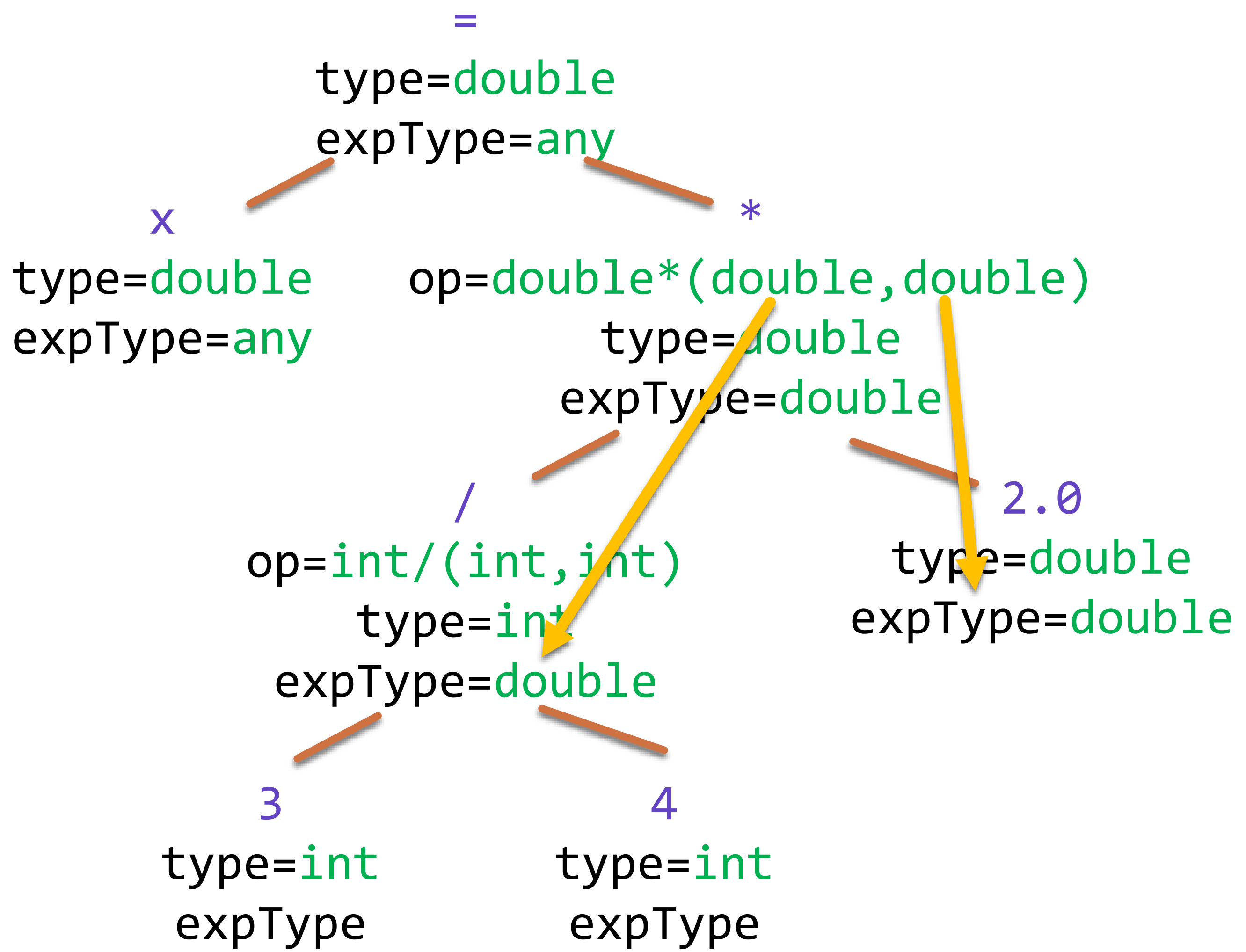
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

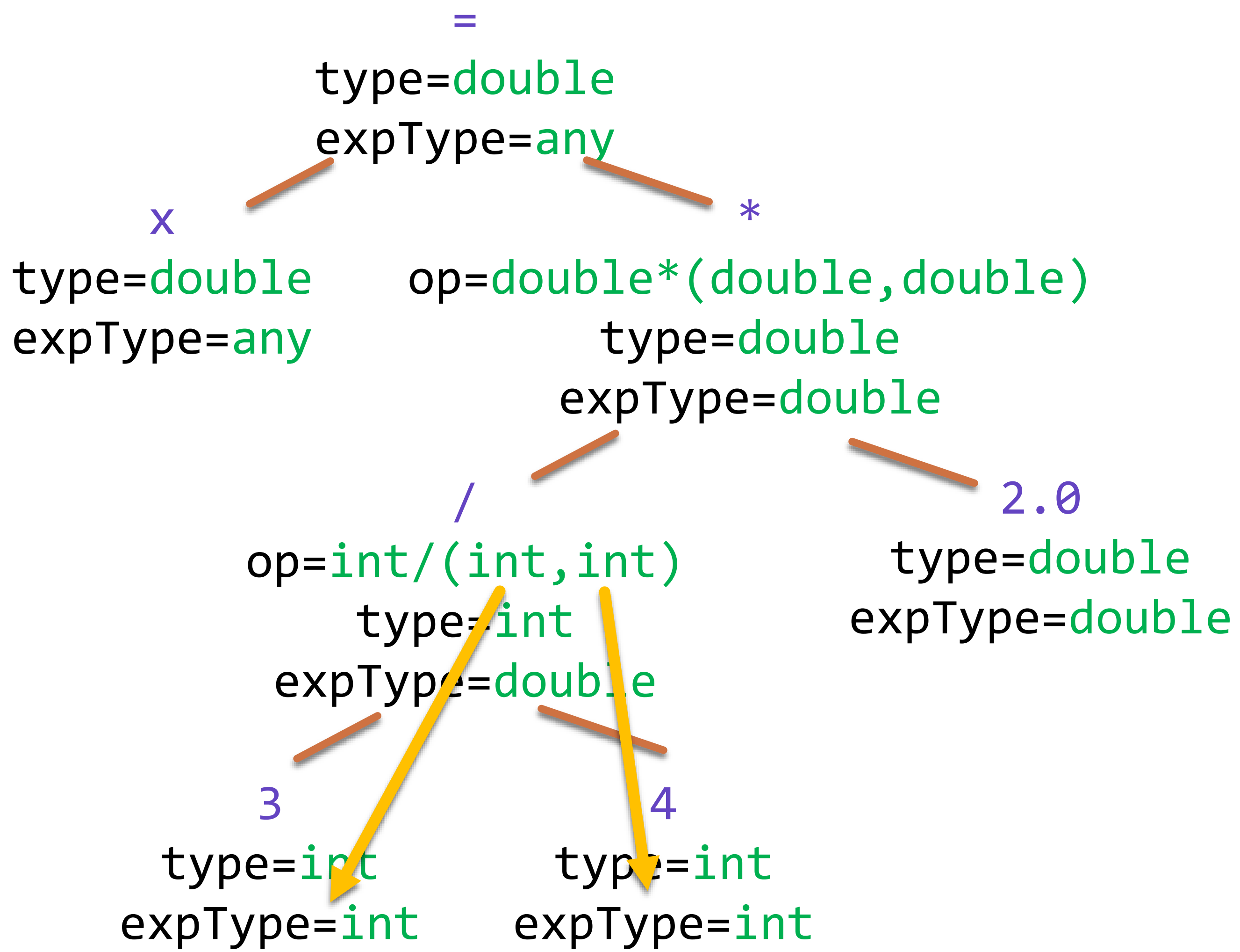
`double x = 3/4*2.0;`



Example: selecting operators

Statement:

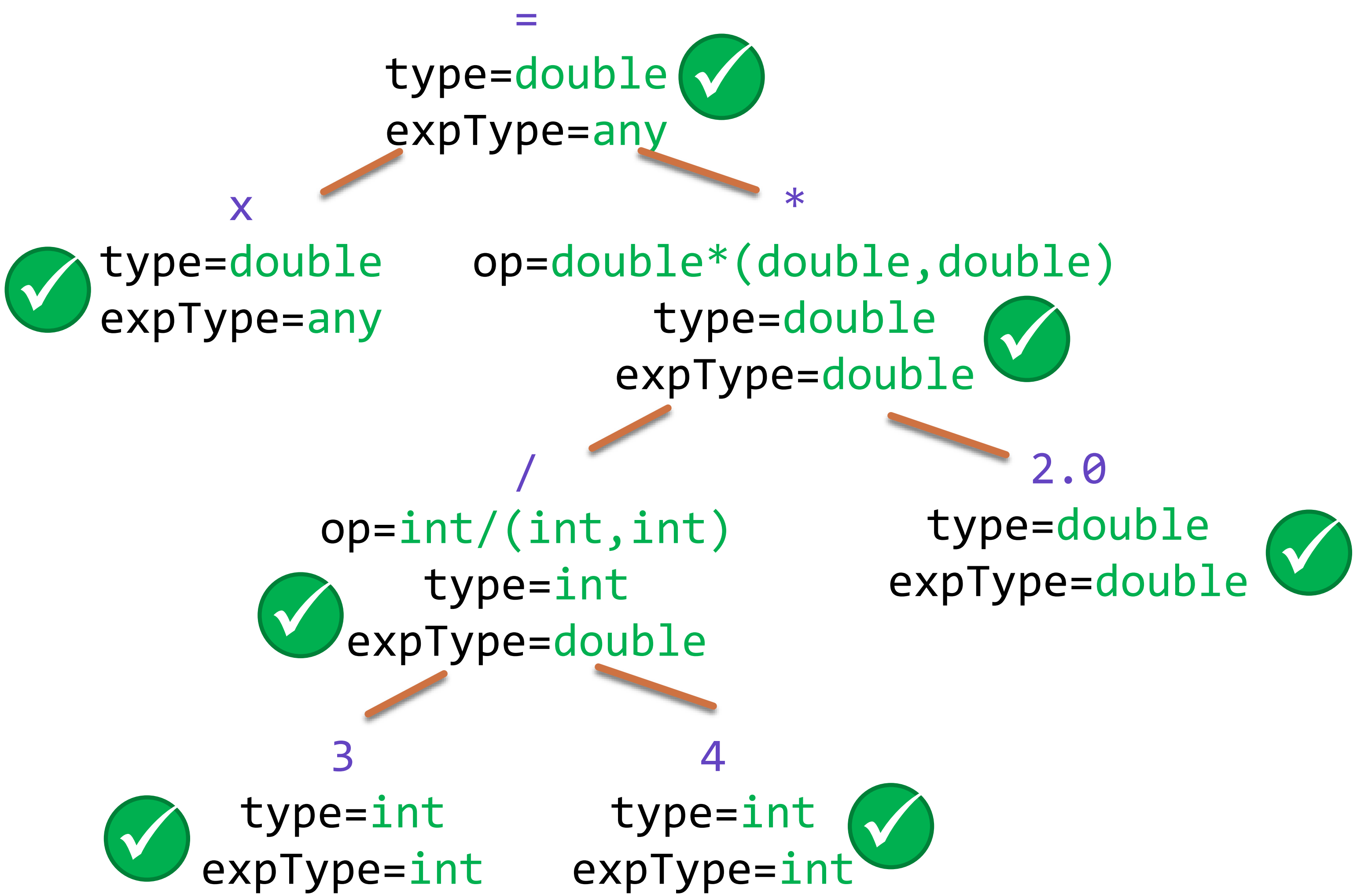
```
double x = 3/4*2.0;
```



Example: selecting operators

Statement:

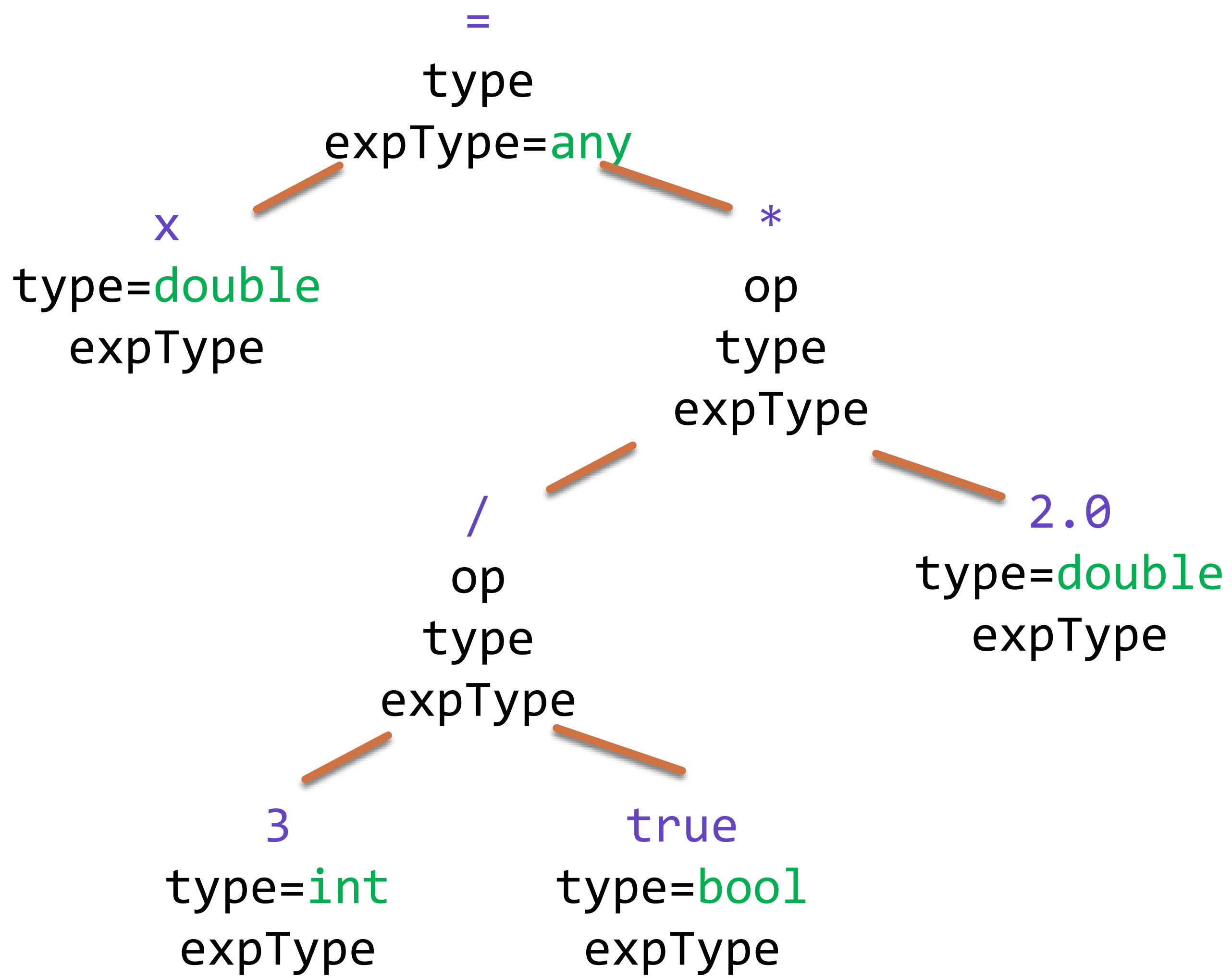
```
double x = 3/4*2.0;
```



Example: type error

Statement:

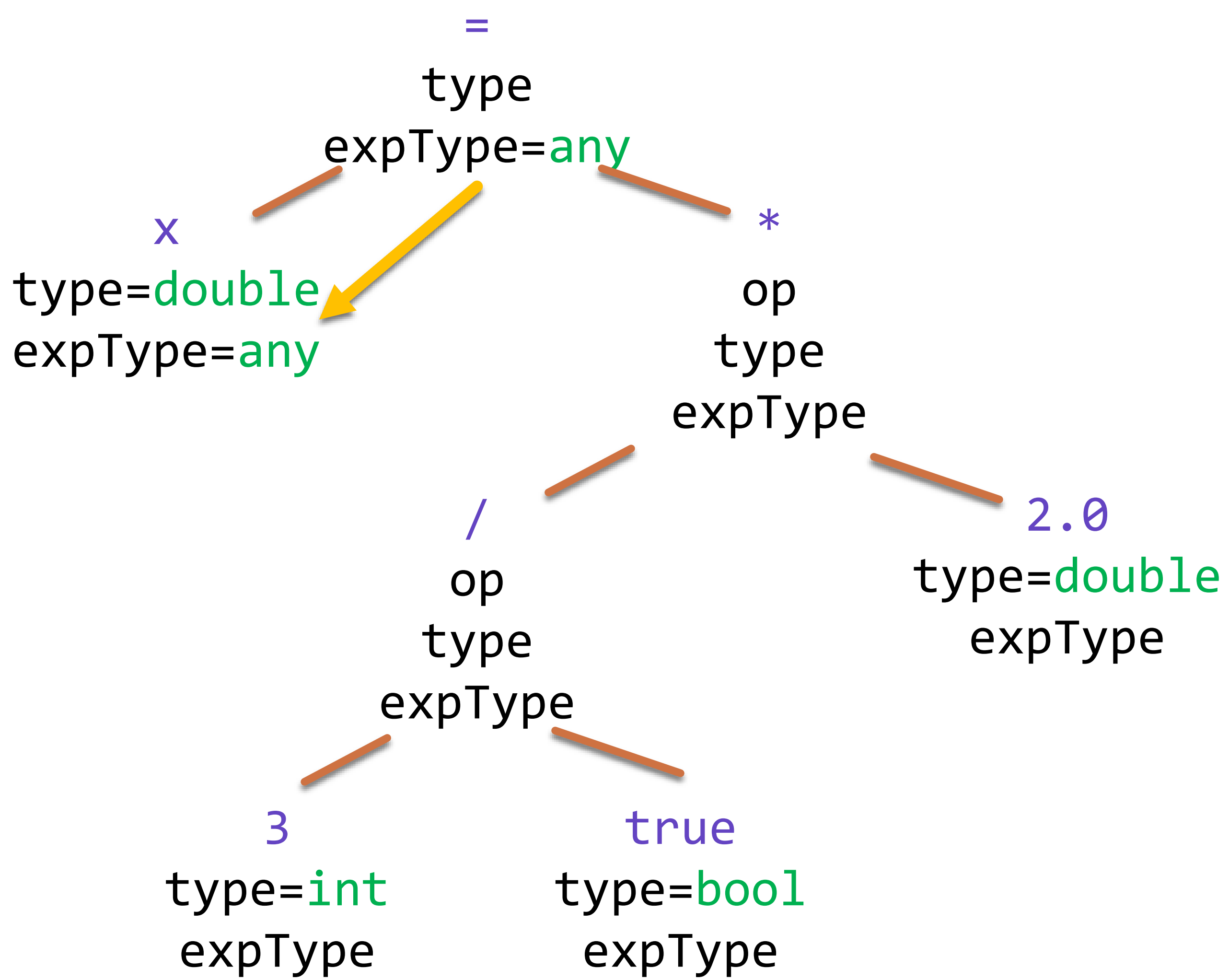
```
double x = 3/true*2.0;
```



Example: type error

Statement:

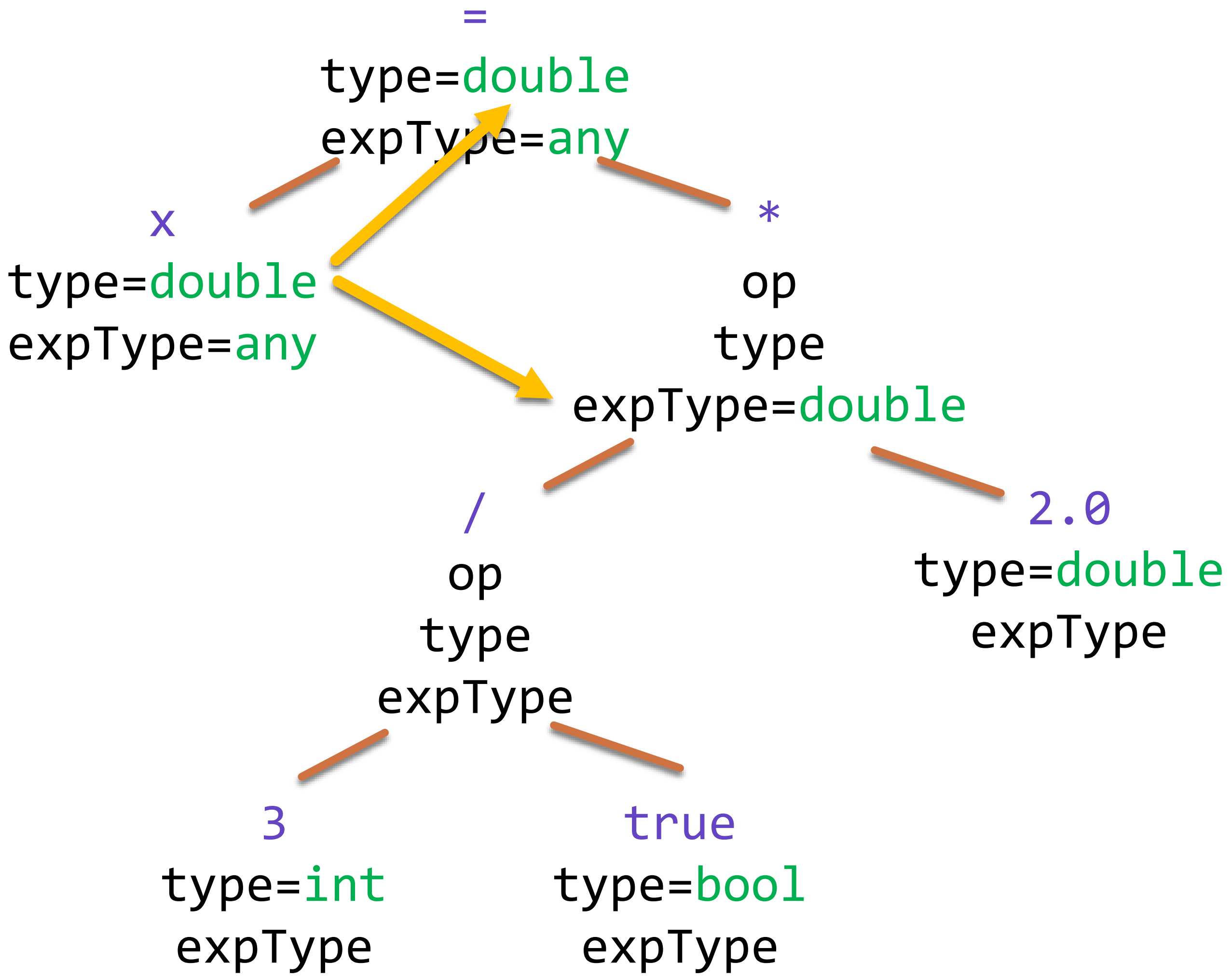
```
double x = 3/true*2.0;
```



Example: type error

Statement:

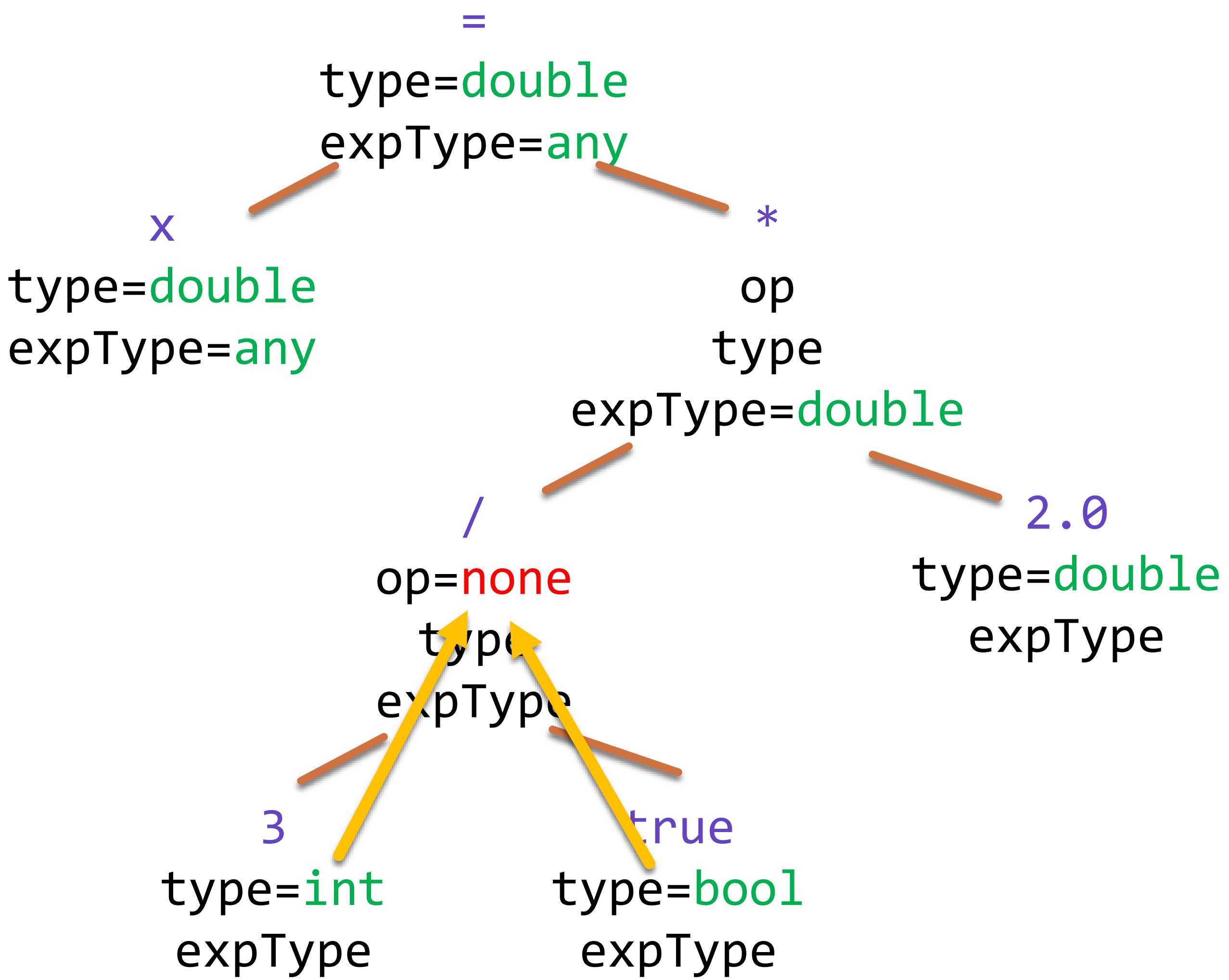
```
double x = 3/true*2.0;
```



Example: type error

Statement:

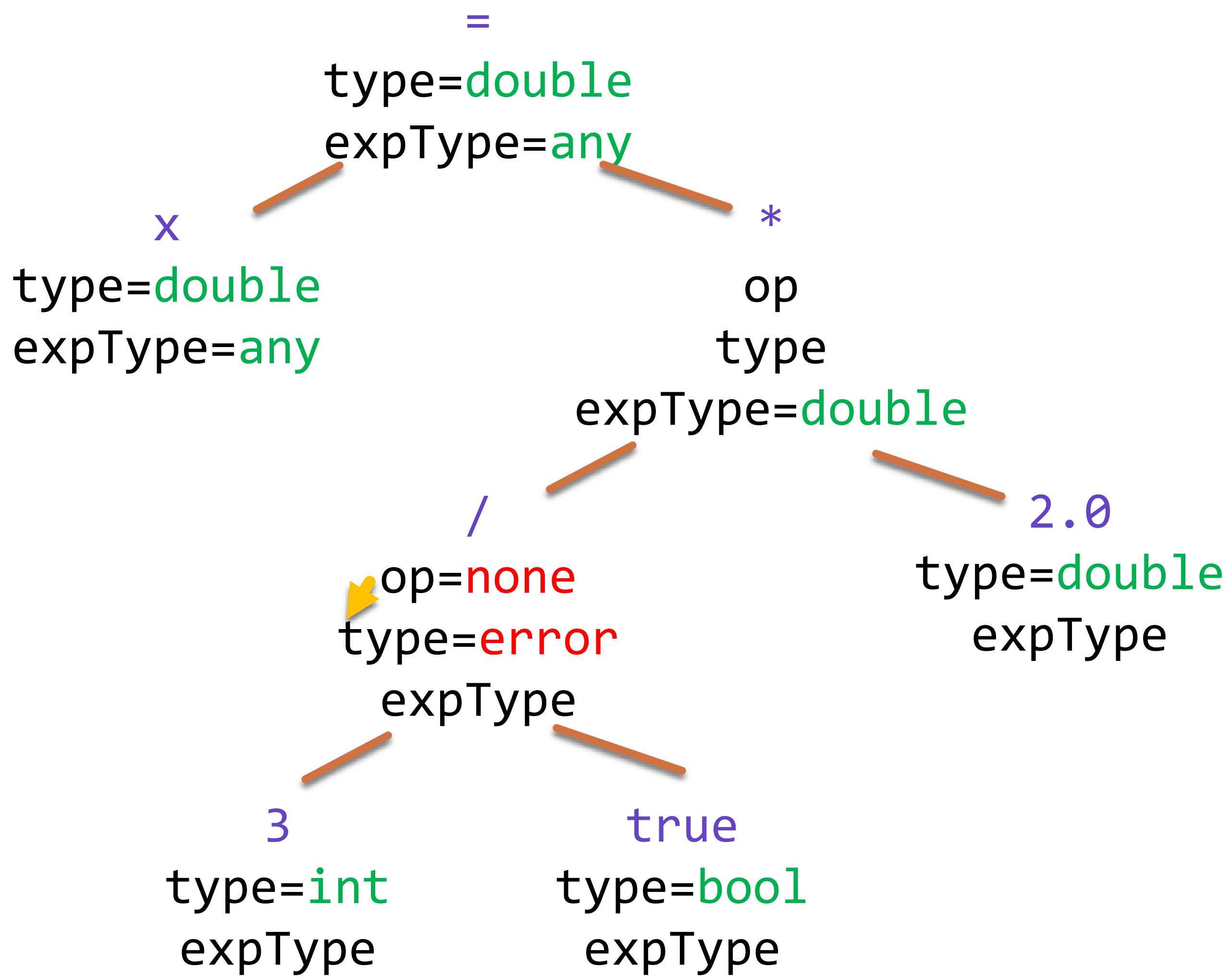
`double x = 3/true*2.0;`



Example: type error

Statement:

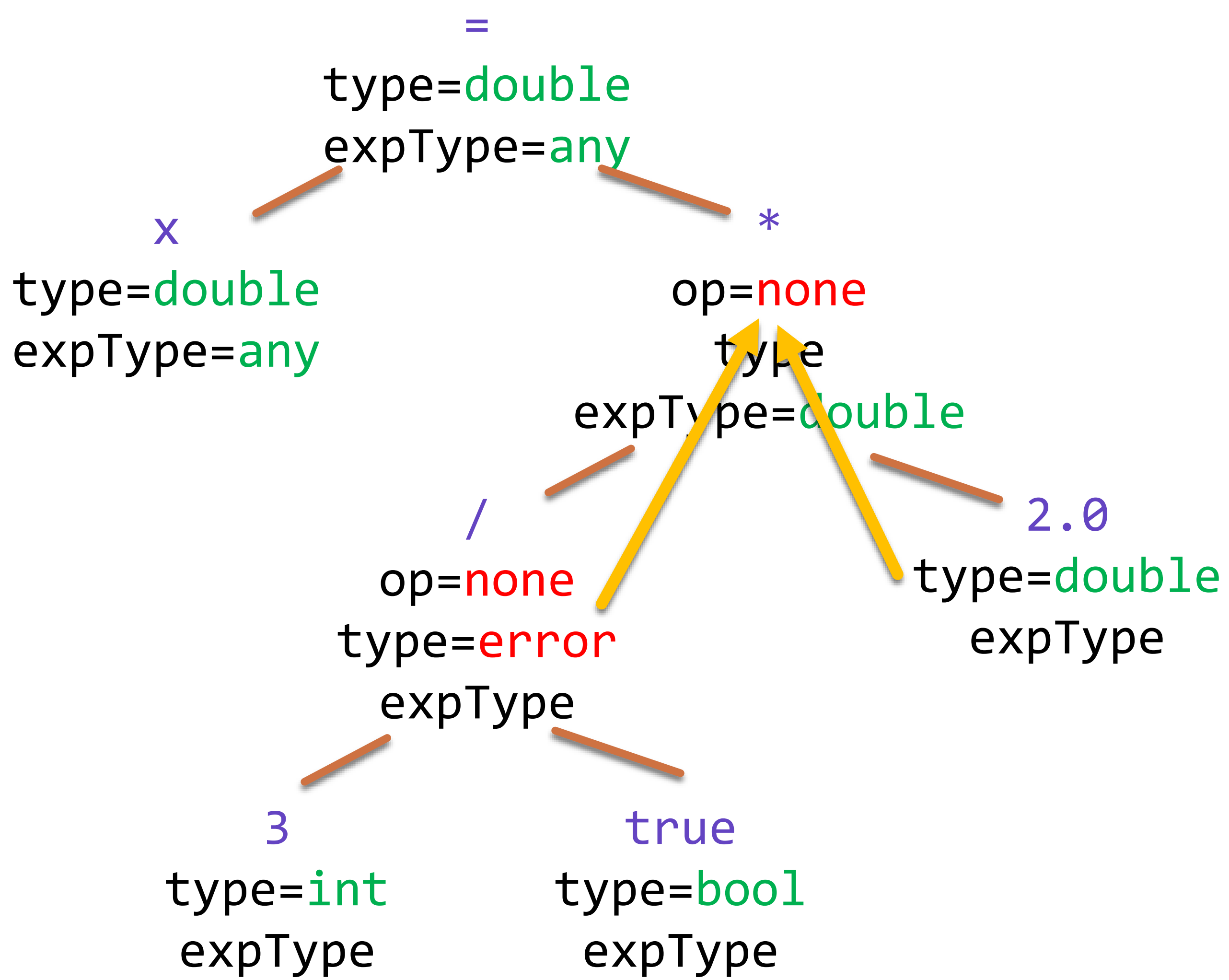
`double x = 3/true*2.0;`



Example: type error

Statement:

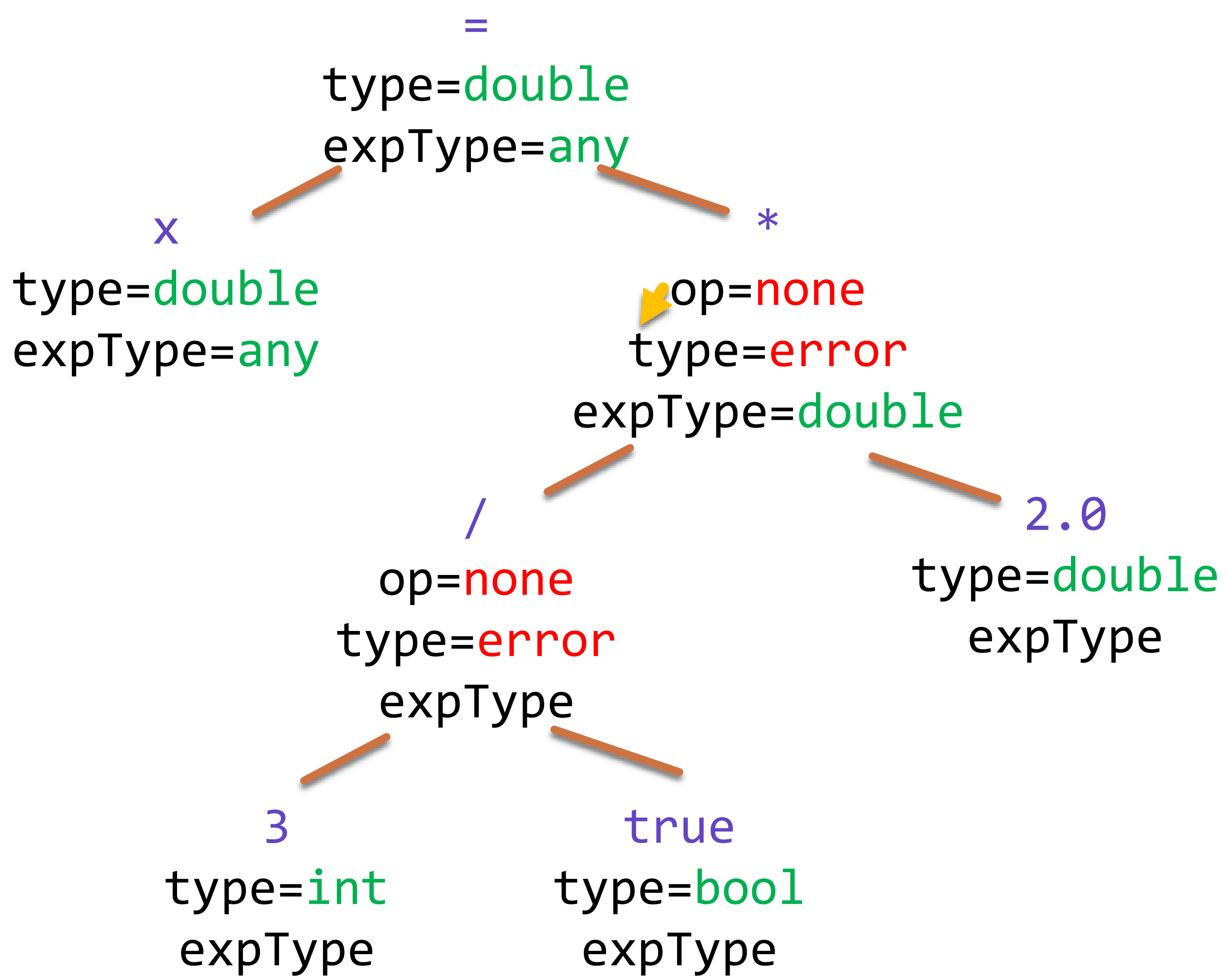
`double x = 3/true*2.0;`



Example: type error

Statement:

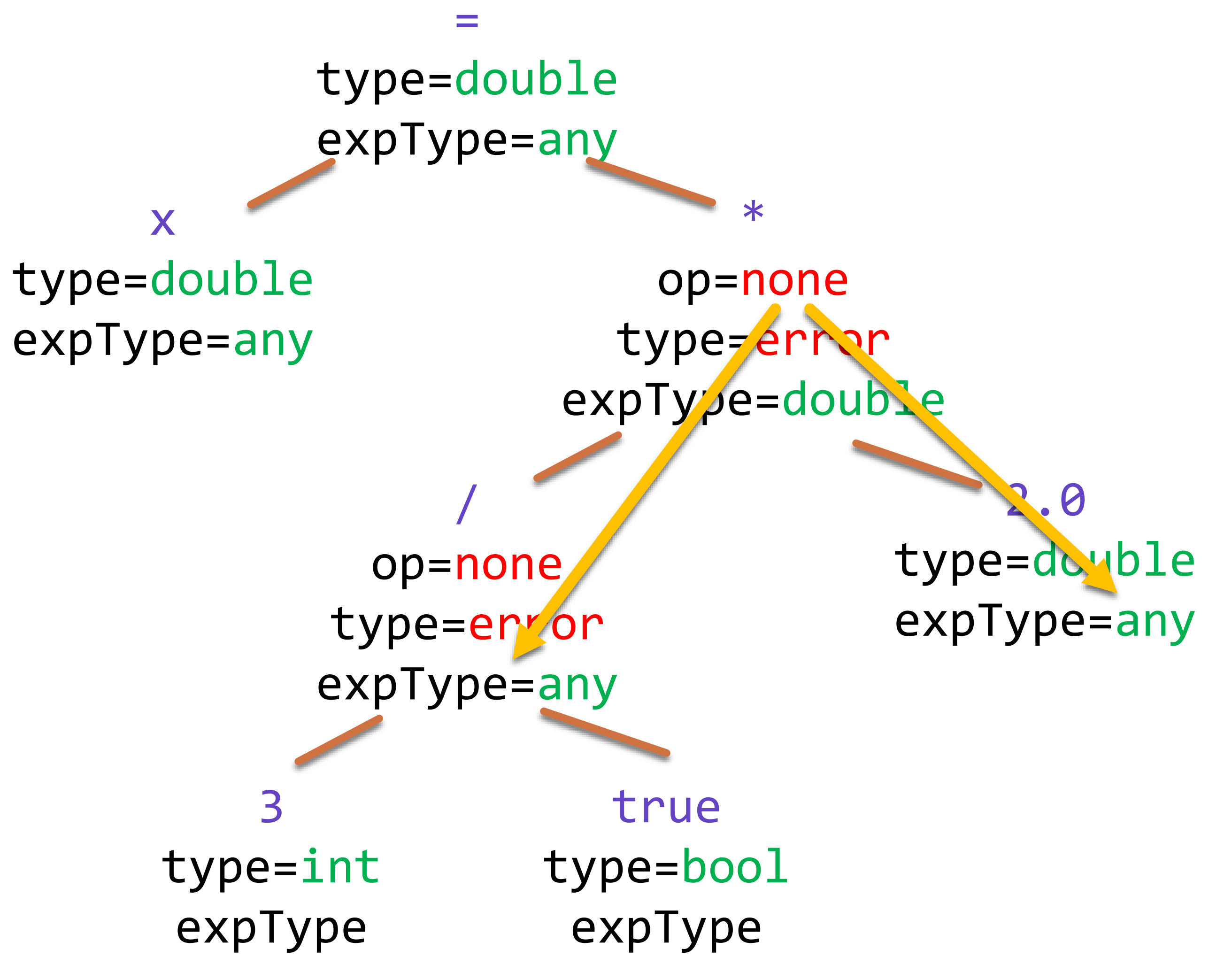
```
double x = 3/true*2.0;
```



Example: type error

Statement:

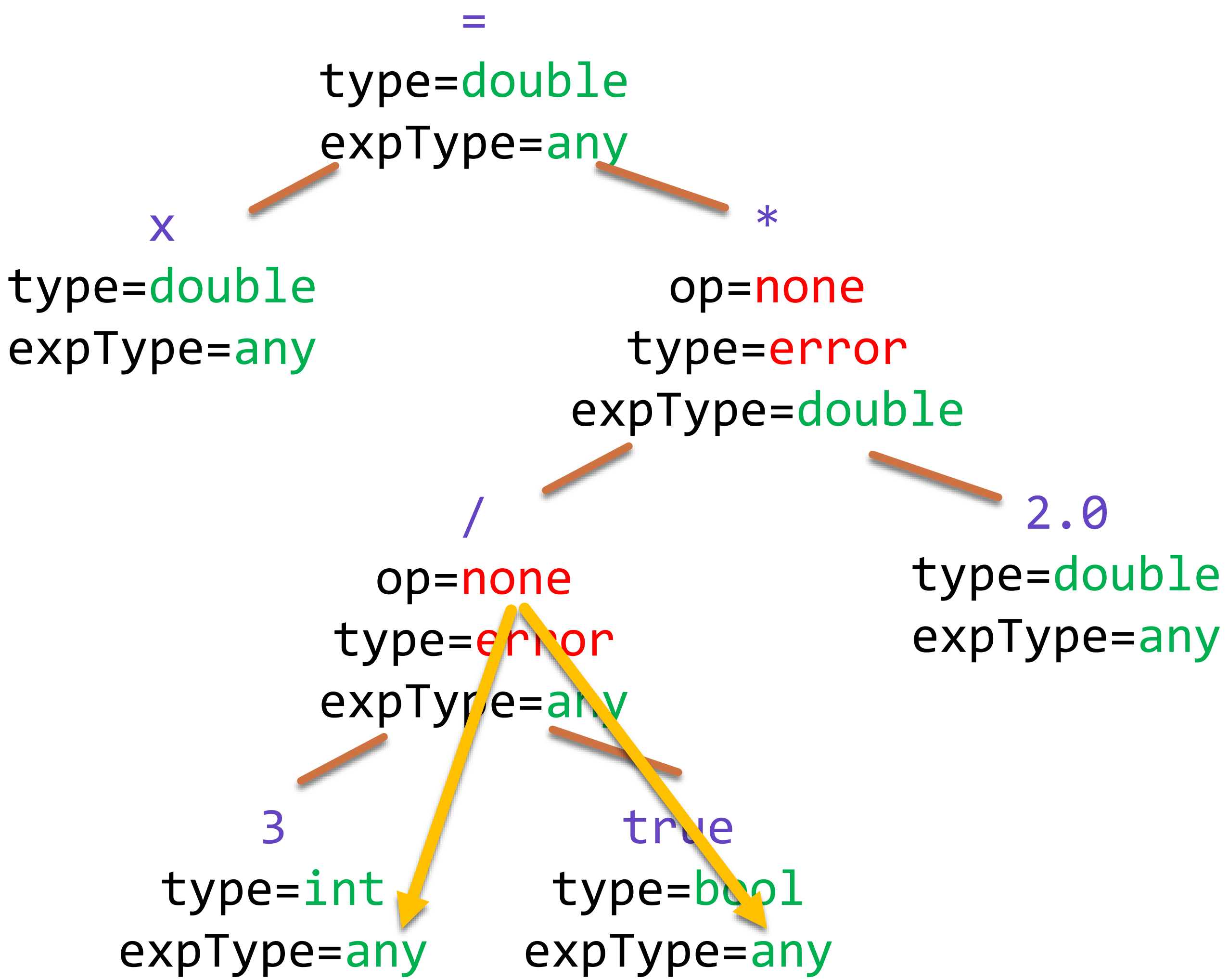
```
double x = 3/true*2.0;
```



Example: type error

Statement:

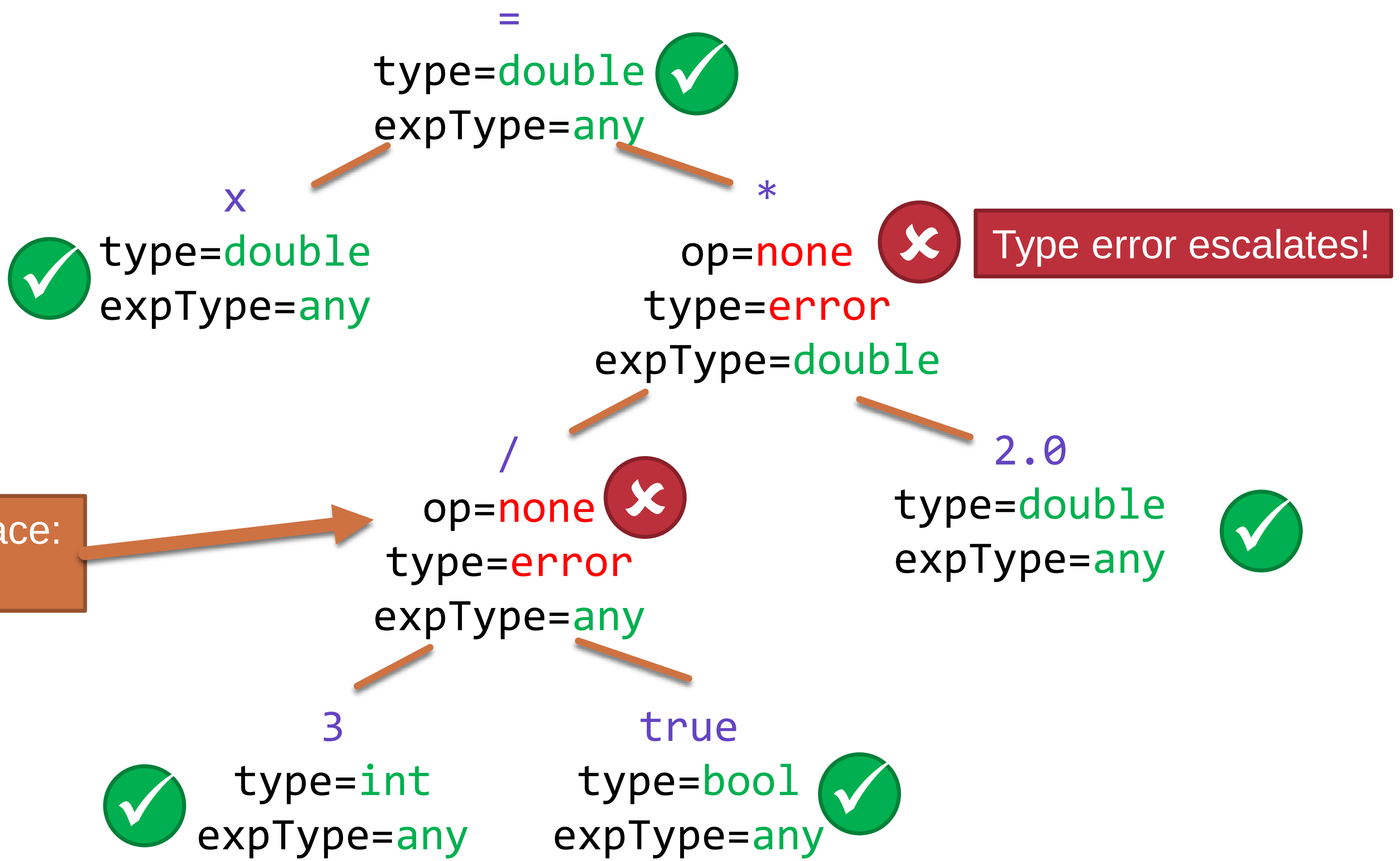
```
double x = 3/true*2.0;
```



Example: type error

Statement:

`double x = 3/true*2.0;`



Enough to signal at the first place: where the original error stems

This lecture: Semantic analysis

- I. Semantic analysis
- II. Attribute grammars
- III. Name analysis
- IV. Type analysis, operator identification
- **V. Other language rules**

Other language rules

- Specific for the given programming language
- Examples:
  - visibility (`private`, `protected`, `public`, etc.)
  - virtual methods (`abstract`, `virtual`, `override`, `sealed`, etc.)
  - implementing an interface, multiple inheritance
  - `this`, `base`, `super` keywords
  - memory management
  - arrays
  - nullable types
  - generic types
  - type inference
  - dynamic type checking

Semantic analysis summary

- Attribute grammars
- Symbol table
- Name analysis
- Type analysis
- Operator identification
- Consistency checking
- Other language rules

If the semantic analysis reveals no errors:
the program code is correct -> can be compiled!

Thank you!