COMP 410 Lecture 1

Kyle Dewey

About Me

- I research automated testing techniques and their intersection with CS education
- My dissertation used logic programming extensively
- This is my second semester at CSUN
- First time dedicating a whole course to logic programming

About this Class

- See something wrong? Want something improved? Email me about it! (kyle.dewey@csun.edu)
- I generally operate based on feedback

Bad Feedback

- This guy sucks.
- This class is boring.
- This material is useless.

-I can’t do anything in response to this

Good Feedback

- This guy sucks, *I can’t read his writing.*
- This class is boring, *it’s way too slow.*
- This material is useless, *I don’t see how it relates to anything in reality.*

- I can’t fix anything if I don’t know what’s wrong

-I can actually do something about this!

What is Logic Programming?

-Major programming paradigm - a way of thinking about problems
-Emphases thinking about exactly _what_ the problem is, as opposed to exactly _how_ to solve it. This is called declarative programming.
-For example: it's generally easier to say what constraints must hold for a valid Sudoku solution, as opposed to directly finding a valid Sudoku solution.
-Somewhat related to functional programming - we generally lack mutable state
-Unlike any other major paradigm, the distinction between inputs and outputs is intentionally blurred. You can take advantage of this.
-Basis in formal logic. It's the only major paradigm where "=" has the same meaning as it does in math.

What is Logic Programming?

- What, not how

-Major programming paradigm - a way of thinking about problems
-Emphases thinking about exactly _what_ the problem is, as opposed to exactly _how_ to solve it. This is called declarative programming.
-For example: it's generally easier to say what constraints must hold for a valid Sudoku solution, as opposed to directly finding a valid Sudoku solution.
-Somewhat related to functional programming - we generally lack mutable state
-Unlike any other major paradigm, the distinction between inputs and outputs is intentionally blurred. You can take advantage of this.
-Basis in formal logic. It's the only major paradigm where "=" has the same meaning as it does in math.

What is Logic Programming?

- What, not how
- No mutable state

-Major programming paradigm - a way of thinking about problems
-Emphases thinking about exactly _what_ the problem is, as opposed to exactly _how_ to solve it. This is called declarative programming.
-For example: it's generally easier to say what constraints must hold for a valid Sudoku solution, as opposed to directly finding a valid Sudoku solution.
-Somewhat related to functional programming - we generally lack mutable state
-Unlike any other major paradigm, the distinction between inputs and outputs is intentionally blurred. You can take advantage of this.
-Basis in formal logic. It's the only major paradigm where "=" has the same meaning as it does in math.

What is Logic Programming?

- What, not how
- No mutable state
- Basis in formal logic
  - = means =

-Major programming paradigm - a way of thinking about problems
-Emphases thinking about exactly _what_ the problem is, as opposed to exactly _how_ to solve it. This is called declarative programming.
-For example: it's generally easier to say what constraints must hold for a valid Sudoku solution, as opposed to directly finding a valid Sudoku solution.
-Somewhat related to functional programming - we generally lack mutable state
-Unlike any other major paradigm, the distinction between inputs and outputs is intentionally blurred. You can take advantage of this.
-Basis in formal logic. It's the only major paradigm where "=" has the same meaning as it does in math.

What is Logic Programming?

- What, not how
- No mutable state
- Basis in formal logic
  - = means =
- Line between input/output is blurry

-Major programming paradigm - a way of thinking about problems
-Emphases thinking about exactly _what_ the problem is, as opposed to exactly _how_ to solve it. This is called declarative programming.
-For example: it's generally easier to say what constraints must hold for a valid Sudoku solution, as opposed to directly finding a valid Sudoku solution.
-Somewhat related to functional programming - we generally lack mutable state
-Unlike any other major paradigm, the distinction between inputs and outputs is intentionally blurred. You can take advantage of this.
-Basis in formal logic. It's the only major paradigm where "=" has the same meaning as it does in math.

What is this Course?

-Strong emphasis on programming and using logic programming languages
-I want you to think in this paradigm, not merely force Java into it
-The ideas can be applied in non-logical languages, and your first assignment will force you to write in a logical way outside of a logic programming language (though you won't realize that's what you're doing yet)
-Little bit of theory

What is this Course?

- Programming, programming, programming

-Strong emphasis on programming and using logic programming languages
-I want you to think in this paradigm, not merely force Java into it
-The ideas can be applied in non-logical languages, and your first assignment will force you to write in a logical way outside of a logic programming language (though you won't realize that's what you're doing yet)
-Little bit of theory

What is this Course?

- Programming, programming, programming
- Thinking in a logic programming way

-Strong emphasis on programming and using logic programming languages
-I want you to think in this paradigm, not merely force Java into it
-The ideas can be applied in non-logical languages, and your first assignment will force you to write in a logical way outside of a logic programming language (though you won't realize that's what you're doing yet)
-Little bit of theory

What is this Course?

- Programming, programming, programming
- Thinking in a logic programming way
- Applying logic programming without a logic programming language

-Strong emphasis on programming and using logic programming languages
-I want you to think in this paradigm, not merely force Java into it
-The ideas can be applied in non-logical languages, and your first assignment will force you to write in a logical way outside of a logic programming language (though you won't realize that's what you're doing yet)
-Little bit of theory

What is this Course?

- Programming, programming, programming
- Thinking in a logic programming way
- Applying logic programming without a logic programming language
- Little bit of theory later on

-Strong emphasis on programming and using logic programming languages
-I want you to think in this paradigm, not merely force Java into it
-The ideas can be applied in non-logical languages, and your first assignment will force you to write in a logical way outside of a logic programming language (though you won't realize that's what you're doing yet)
-Little bit of theory

What this course **isn't**

What this course **isn’t**

- Artificial intelligence

–”Artificial intelligence” used to refer to search techniques, which is relevant to logic programming. Now the term largely refers to machine learning. What it means is a moving target.
–Machine learning (we won’t do any sort of statistics)
–You can spend a career on the theory behind this stuff. I know some, but it’s not my speciality.

What this course **isn't**

- Artificial intelligence
- Machine learning

-"Artificial intelligence" used to refer to search techniques, which is relevant to logic programming. Now the term largely refers to machine learning. What it means is a moving target.
-Machine learning (we won't do any sort of statistics)
-You can spend a career on the theory behind this stuff. I know some, but it's not my speciality.

What this course **isn't**

- Artificial intelligence
- Machine learning
- Deeply theoretical

-"Artificial intelligence" used to refer to search techniques, which is relevant to logic programming. Now the term largely refers to machine learning. What it means is a moving target.
-Machine learning (we won't do any sort of statistics)
-You can spend a career on the theory behind this stuff. I know some, but it's not my speciality.

Syllabus

Outline

- Abstract Syntax Trees and evaluation
- SAT and Semantic Tableau

Abstract Syntax Trees and Evaluation

Abstract Syntax Tree

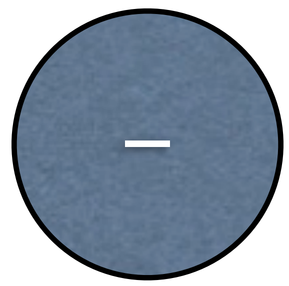
- Abbreviation: AST
- Unambiguous tree-based representation of a sentence in a language
- Very commonly used in compilers, interpreters, and related software

-Generally we work with ASTs instead of Strings or any other code representation

```
(1 + 2) - 3 * 4
```

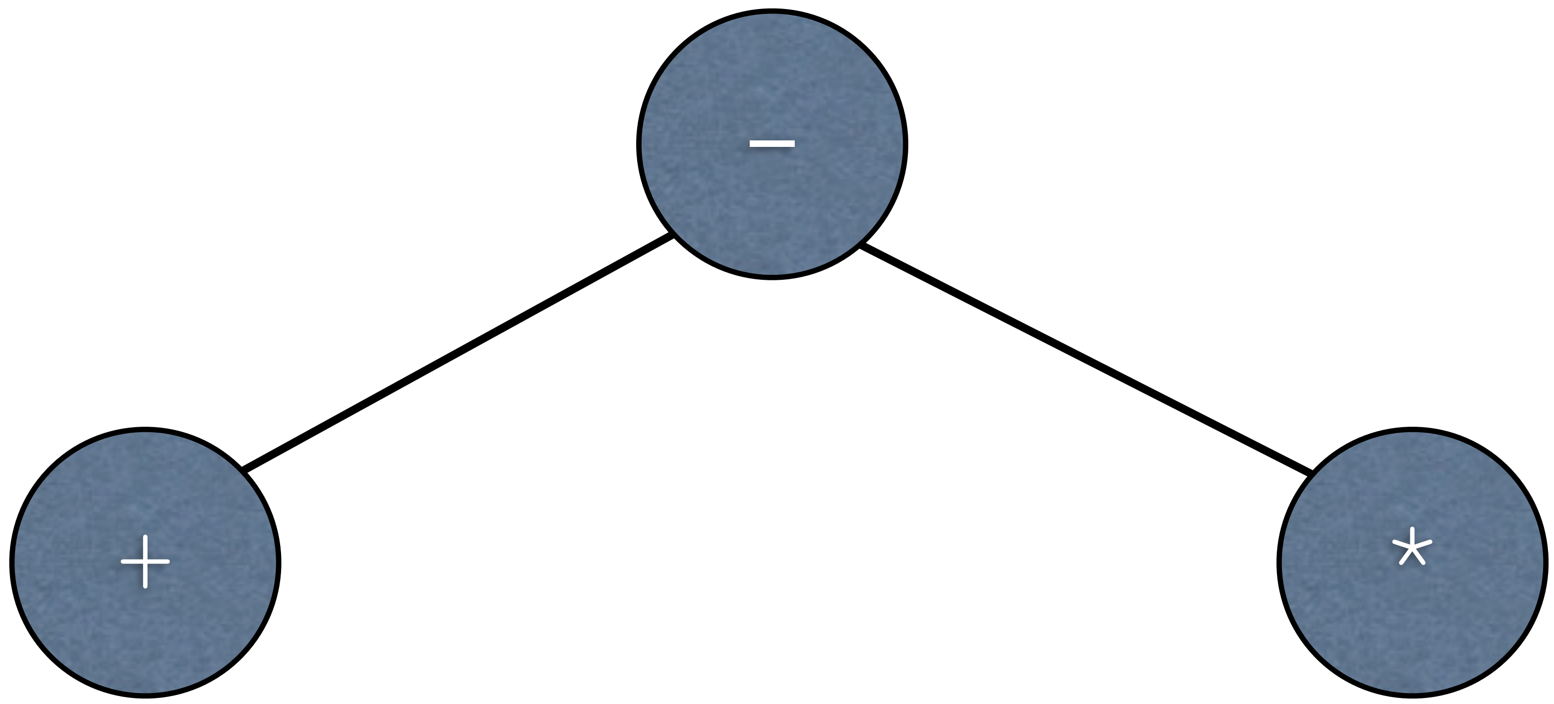
-Key parts: we need parentheses to direct that 1 + 2 happens first. We know that the 3 * 4 should happen after the part in parentheses from PEMDAS rules

`(1 + 2) - 3 * 4`



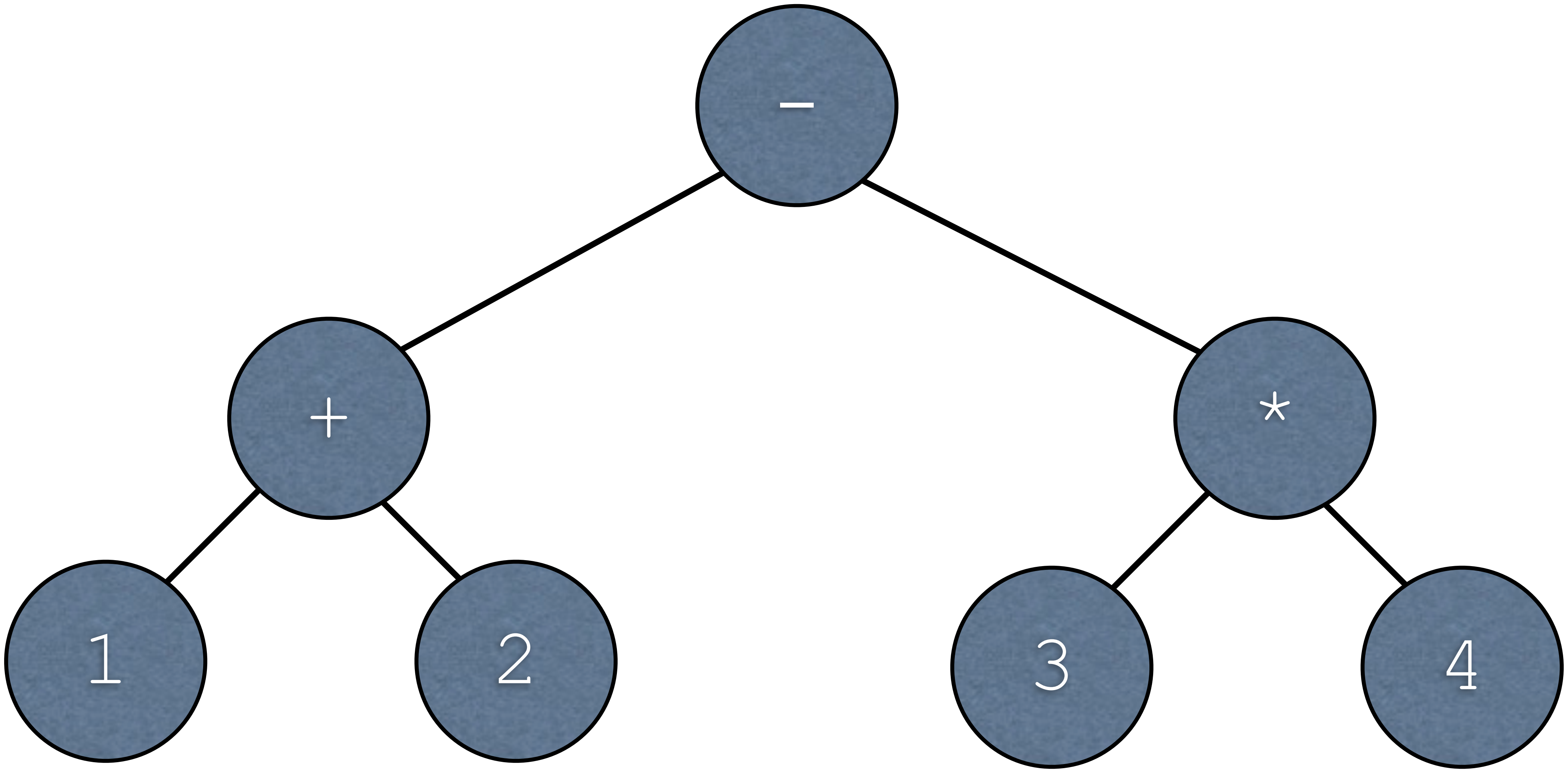
-Lowest priority thing ends up in the top of the tree

(1 + 2) - 3 * 4



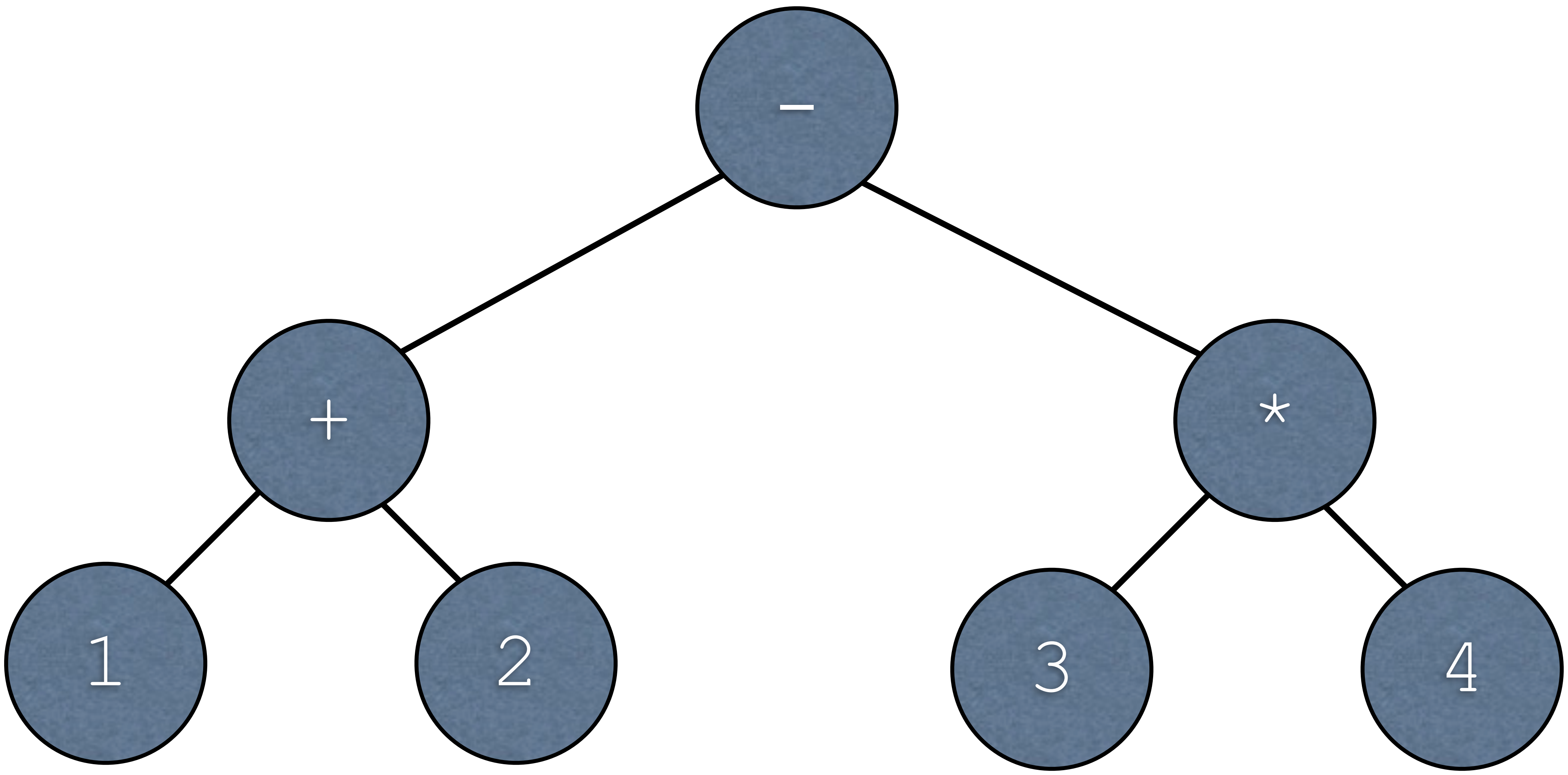
-Next level of priority

$(1 + 2) - 3 * 4$



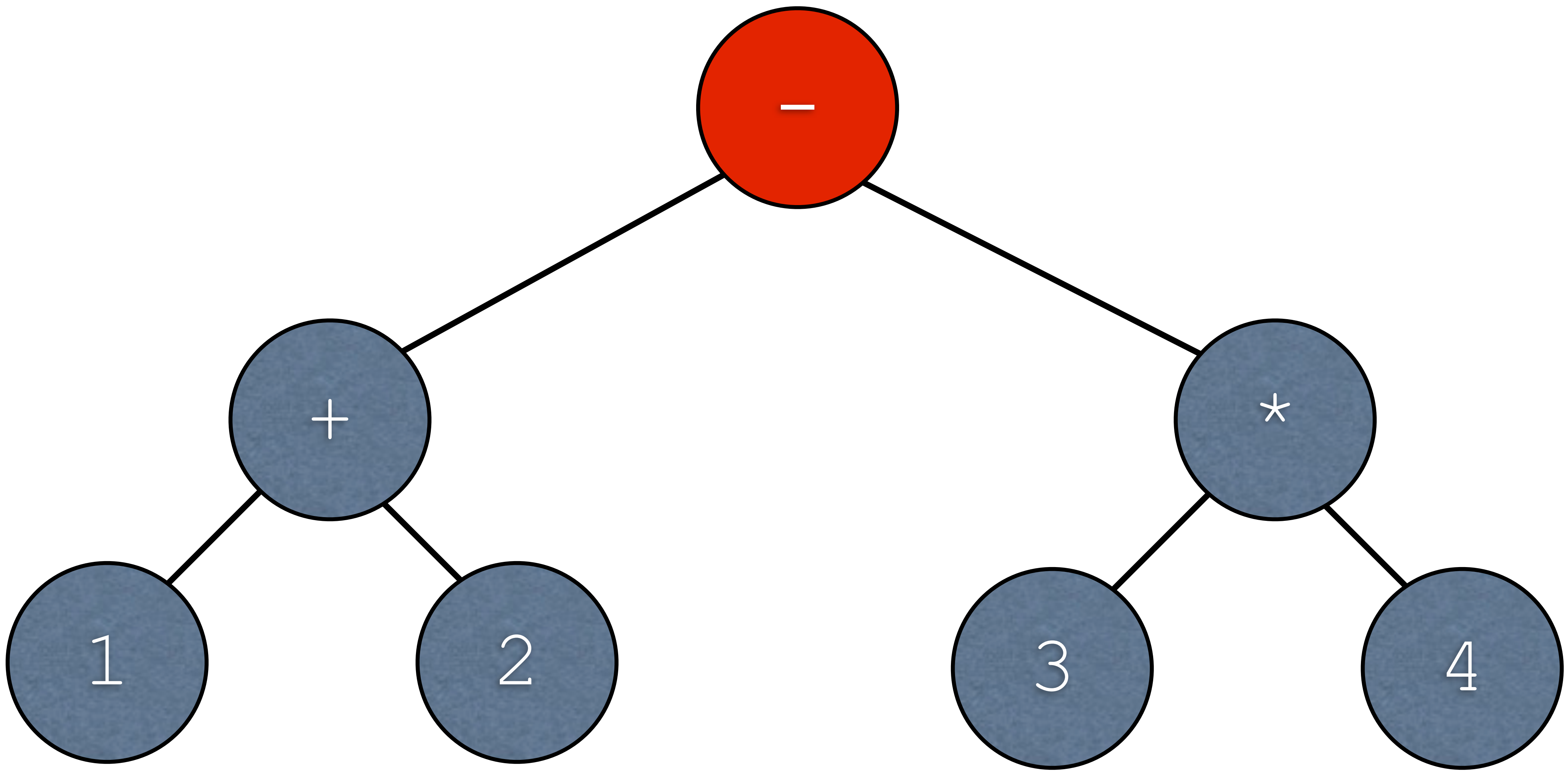
-Next level of priority

Evaluation



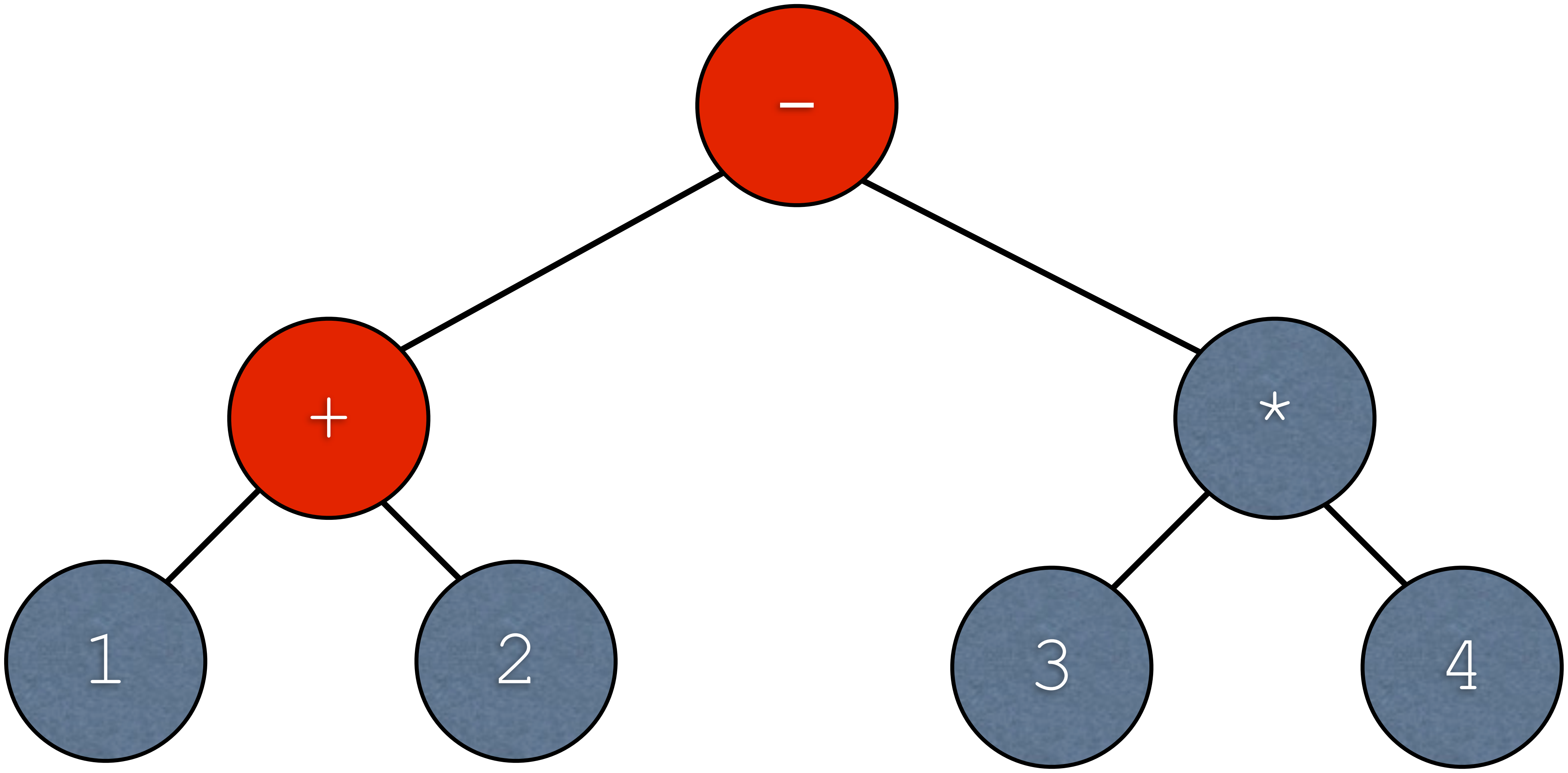
-Key point: bubble-up values from the leaves
-This can be implemented in code via a recursive function starting from the root (code in a bit later)

Evaluation



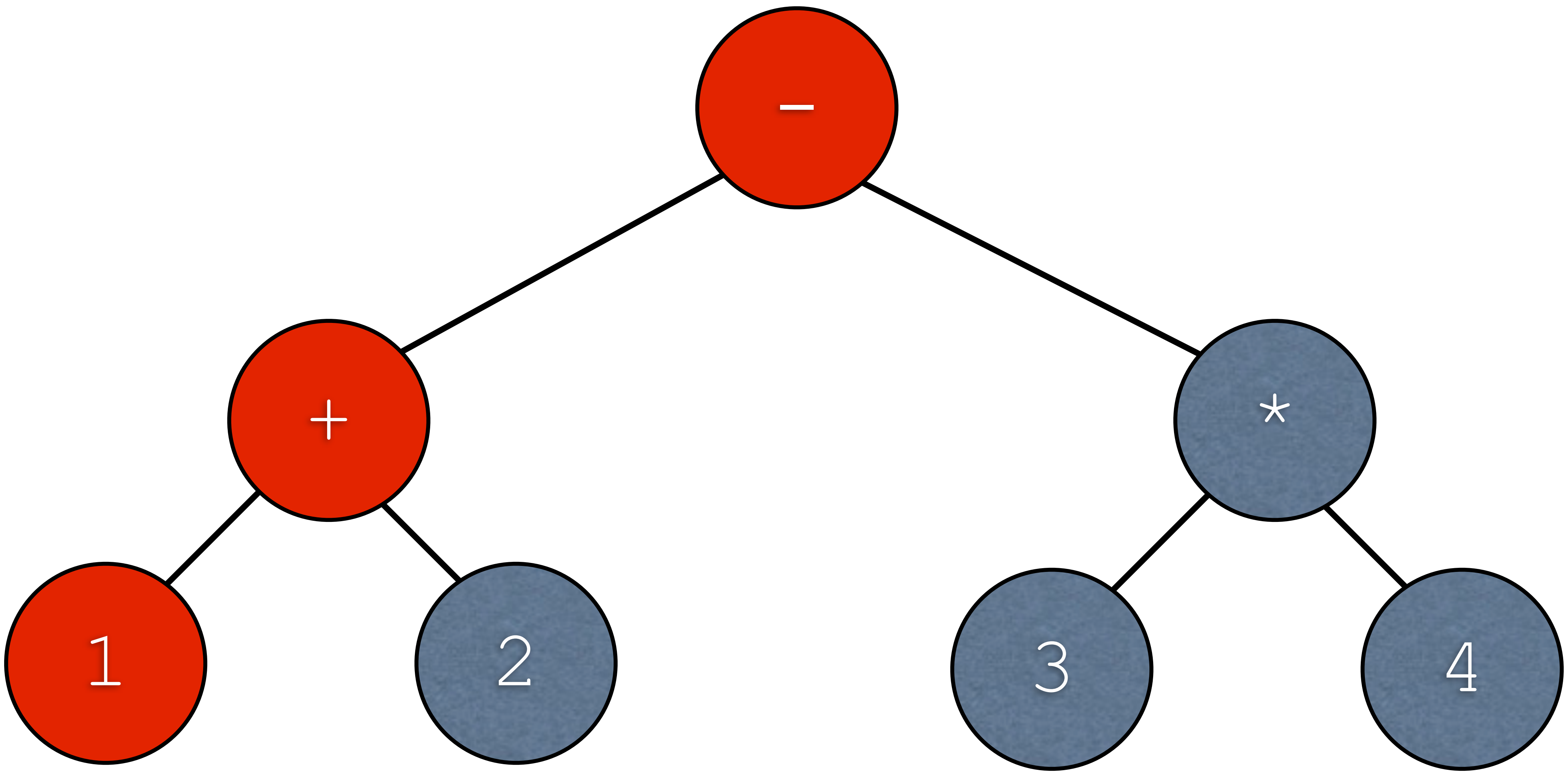
–We start evaluation from the root...

Evaluation



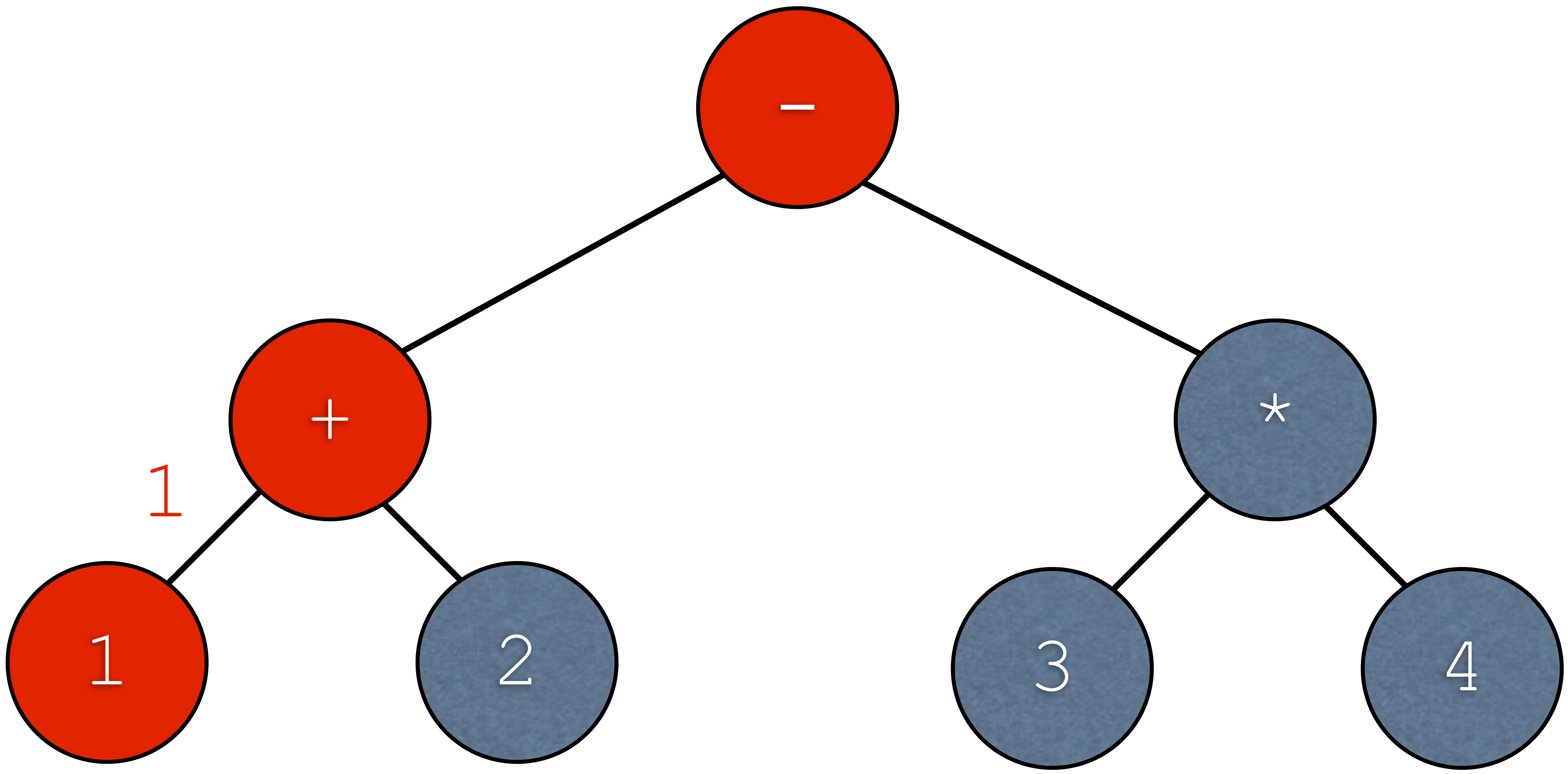
–In order to evaluate the root, we need to evaluate the left subtree of the root (+)

Evaluation



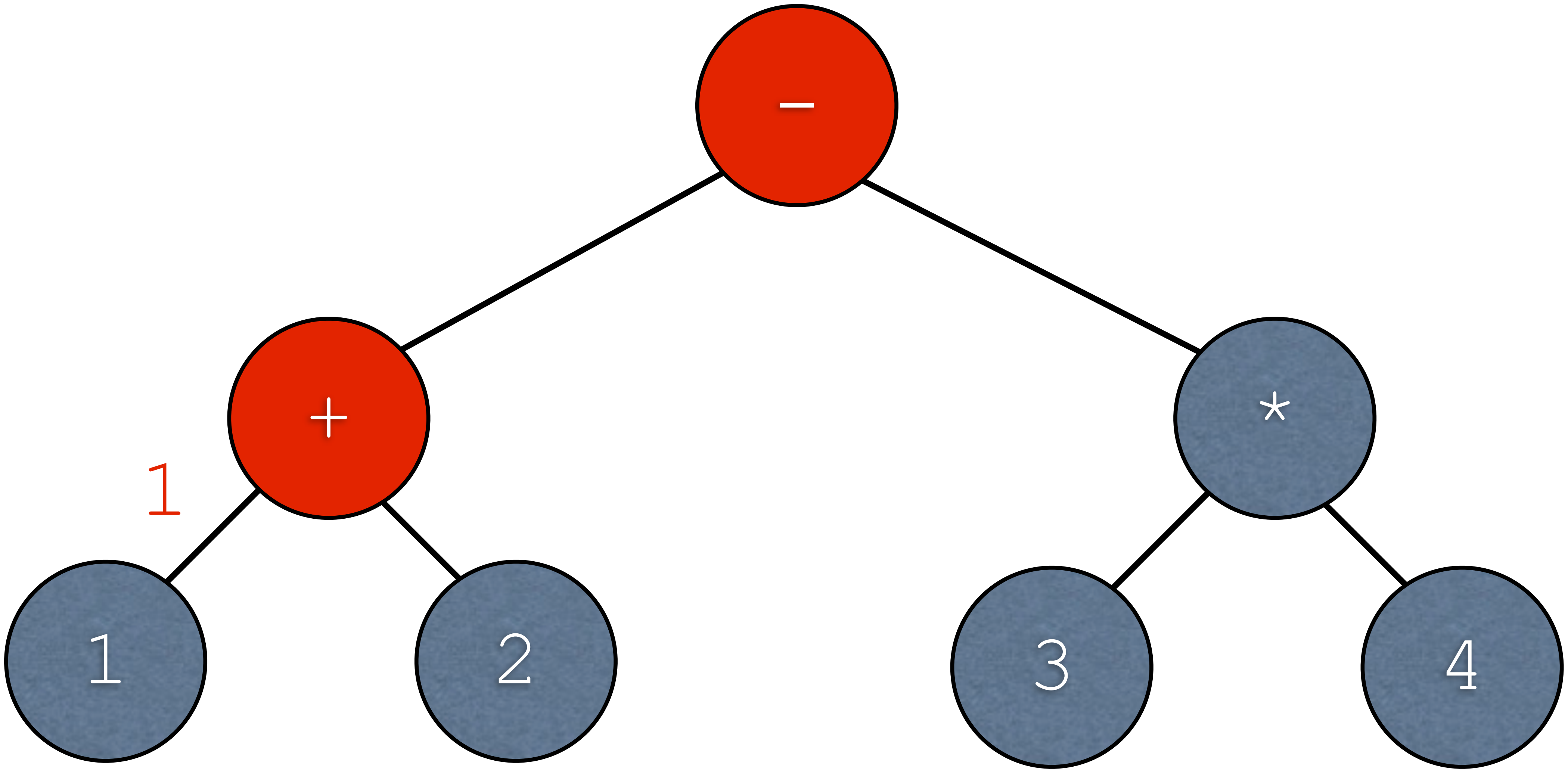
–In order to evaluate +, we need to evaluate the left subtree (as with the root)

Evaluation



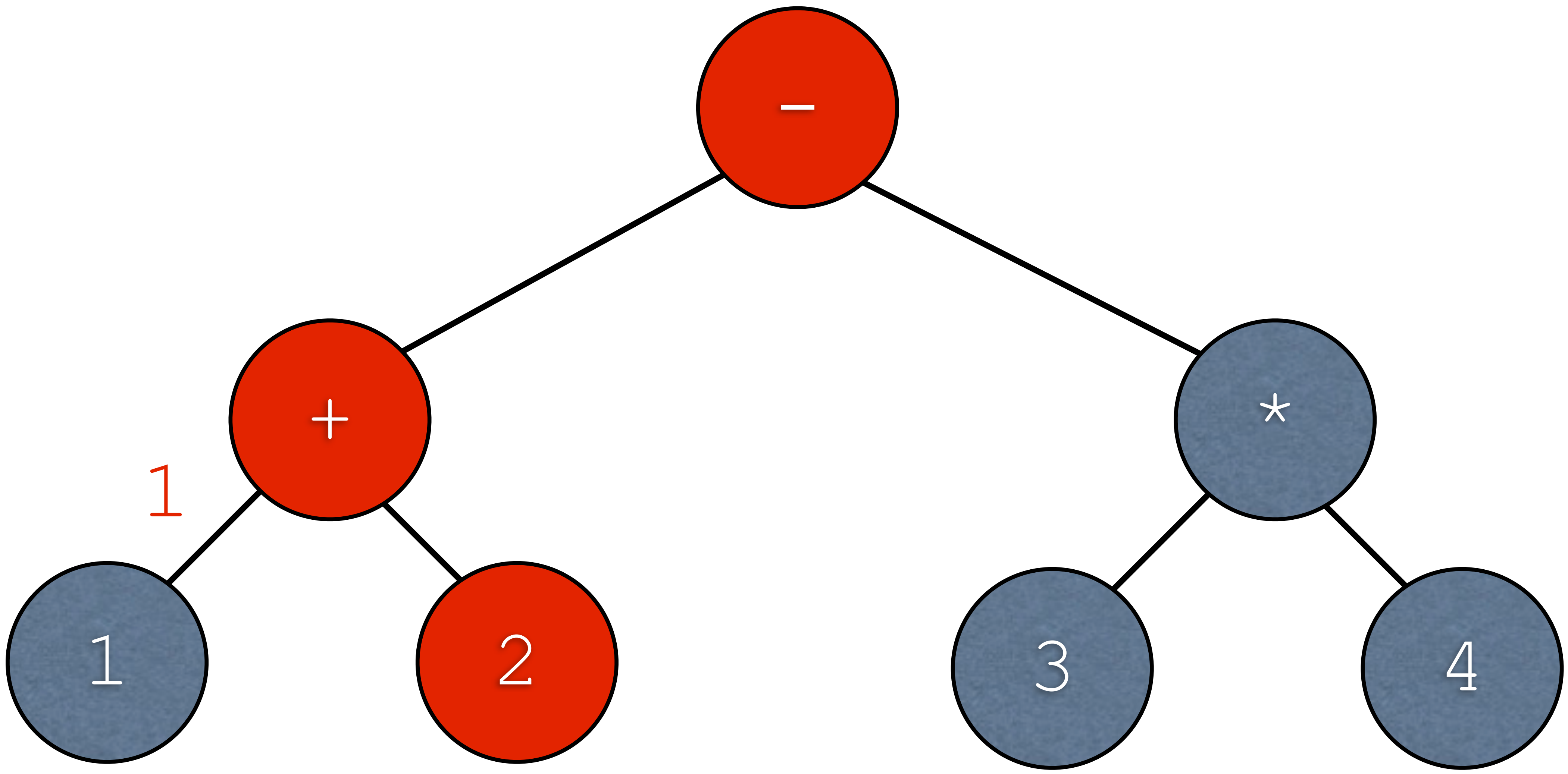
-For arithmetic, leaves are simply numbers
-Evaluating a leaf returns the number held within

Evaluation



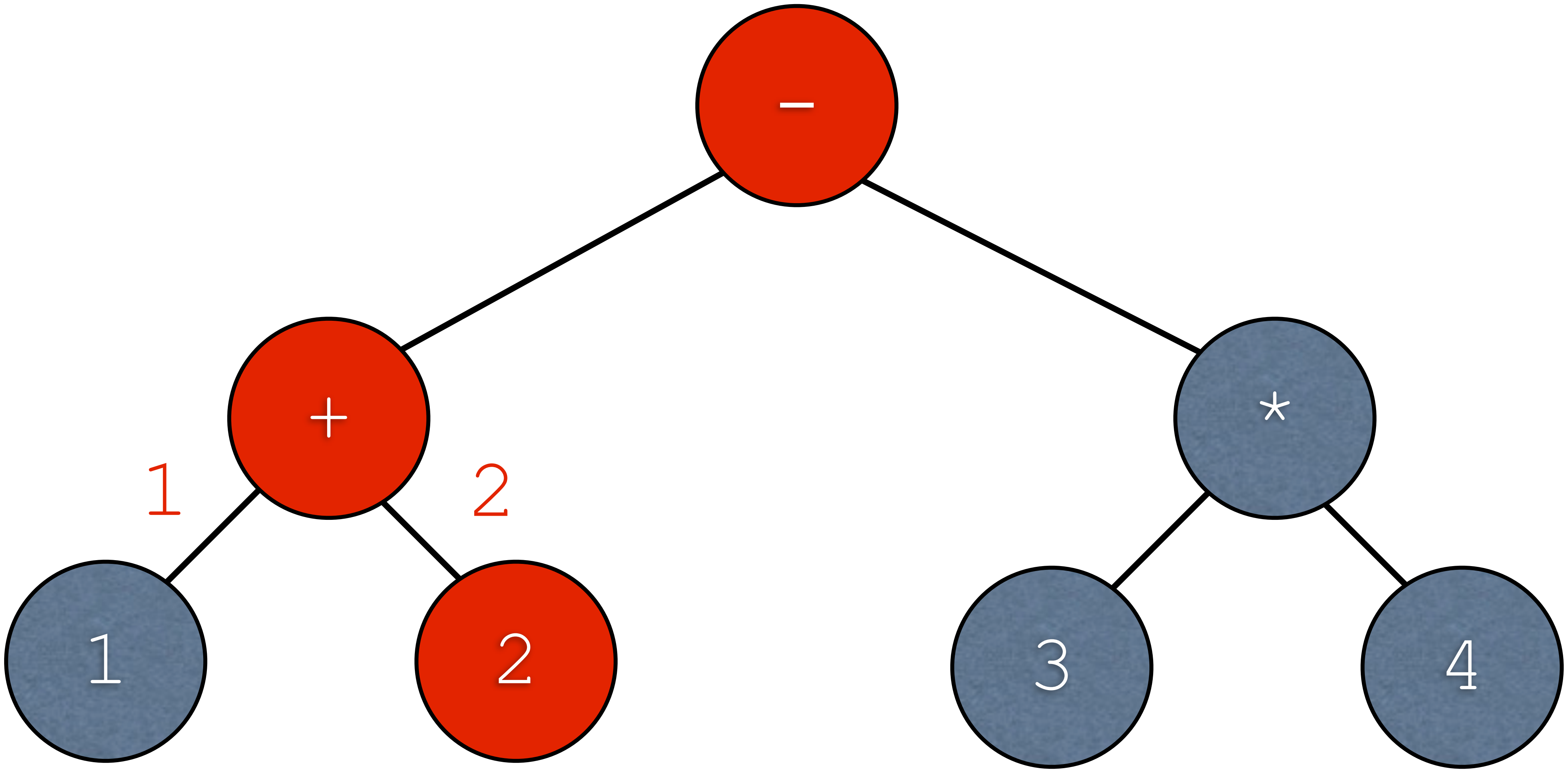
-The left subtree of + has now been evaluated
-Now + needs the value of the right subtree

Evaluation



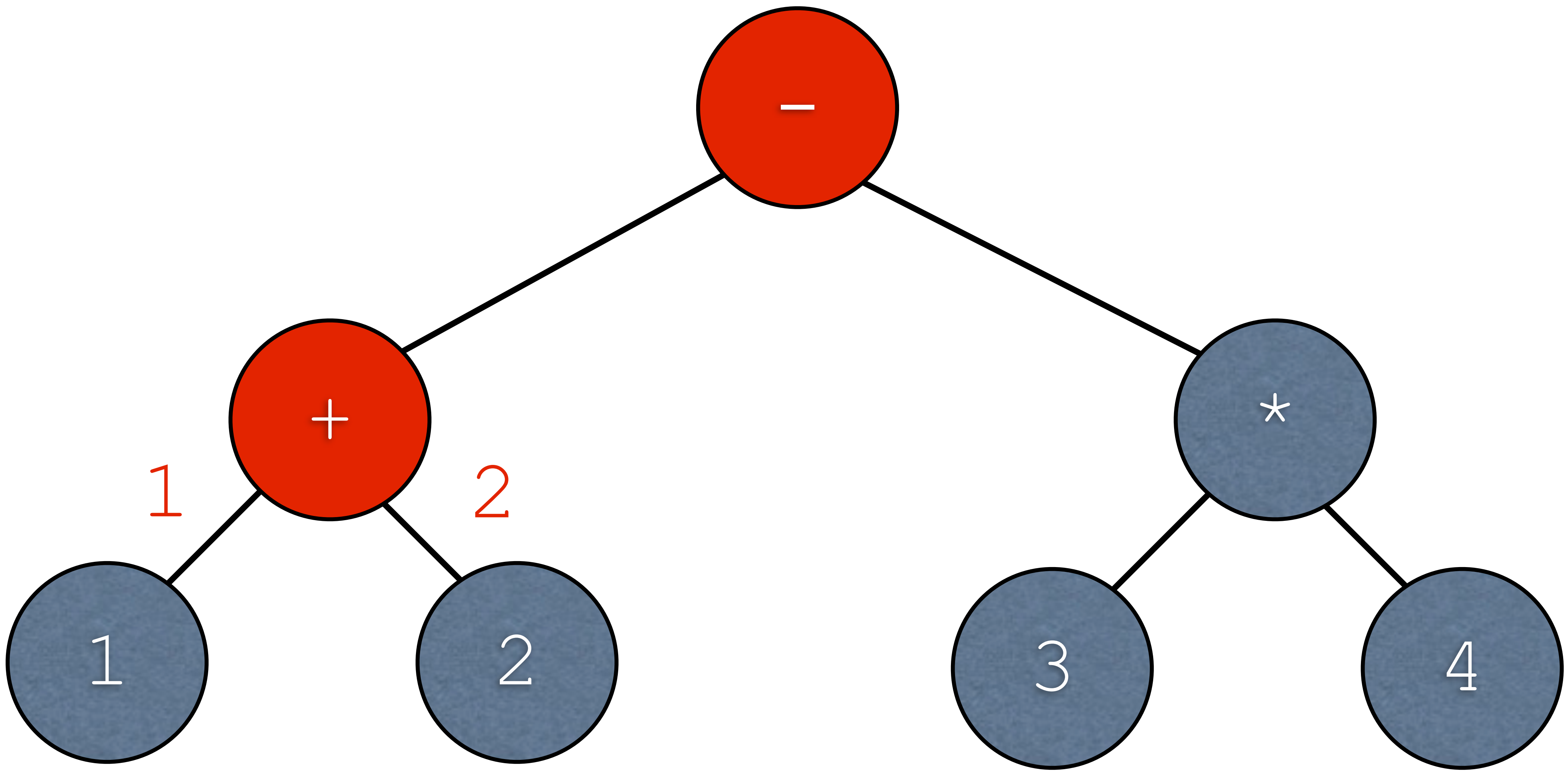
-The left subtree of + has now been evaluated
-Now + needs the value of the right subtree

Evaluation



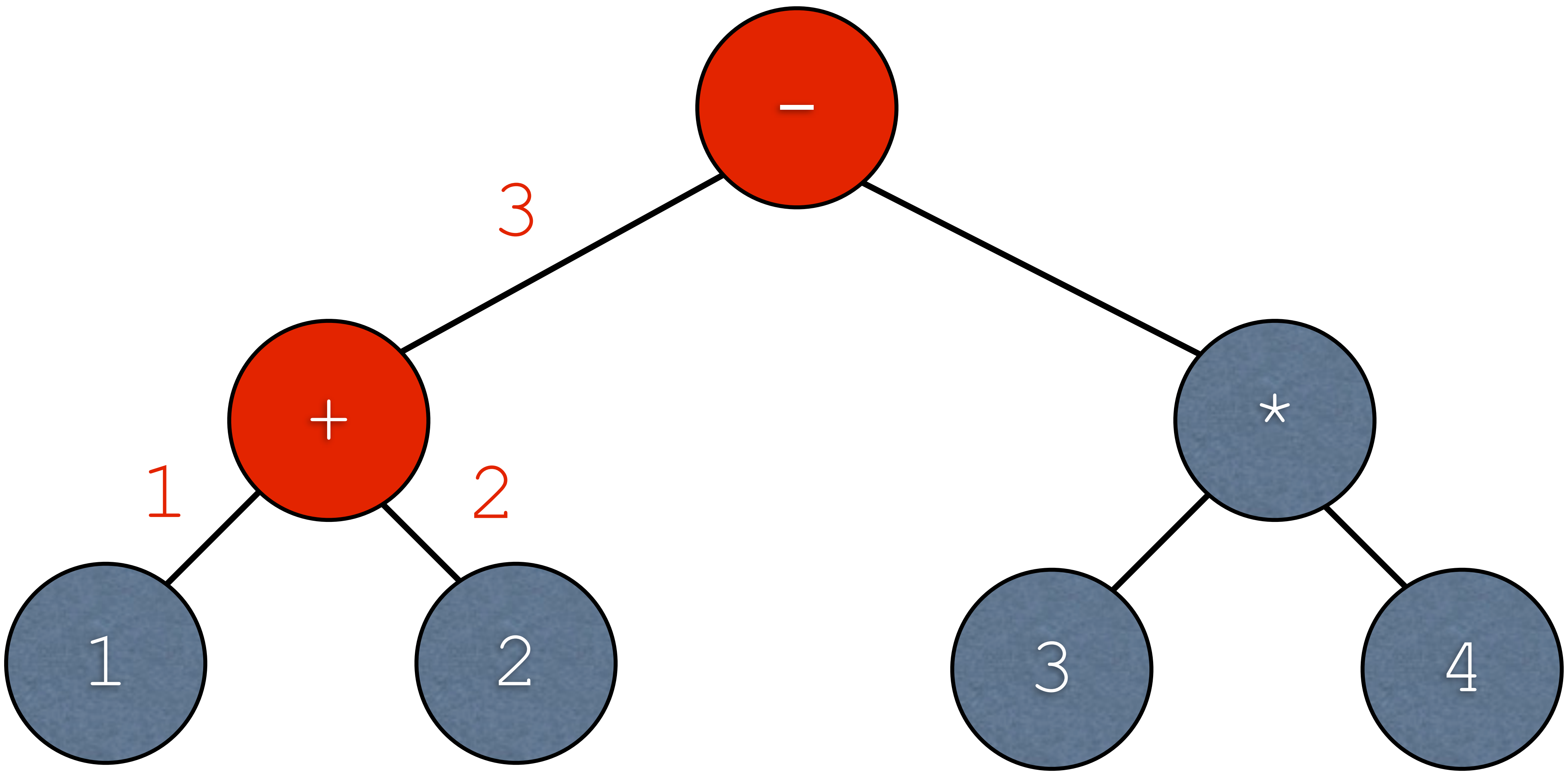
–As before, leaves just return the value held within

Evaluation



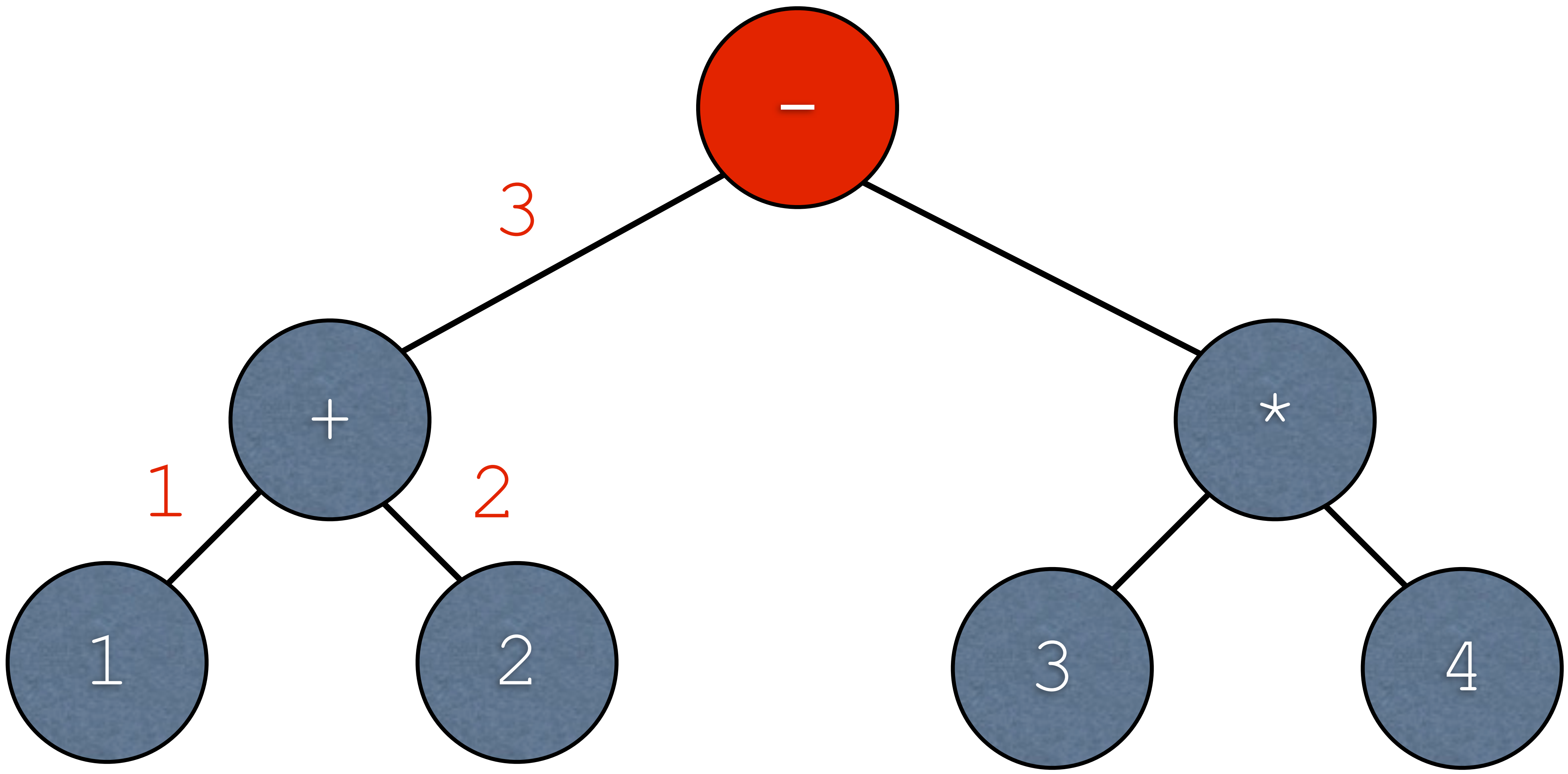
-Subtrees of + are now taken care of
-Now + has two values that it needs to work with...

Evaluation



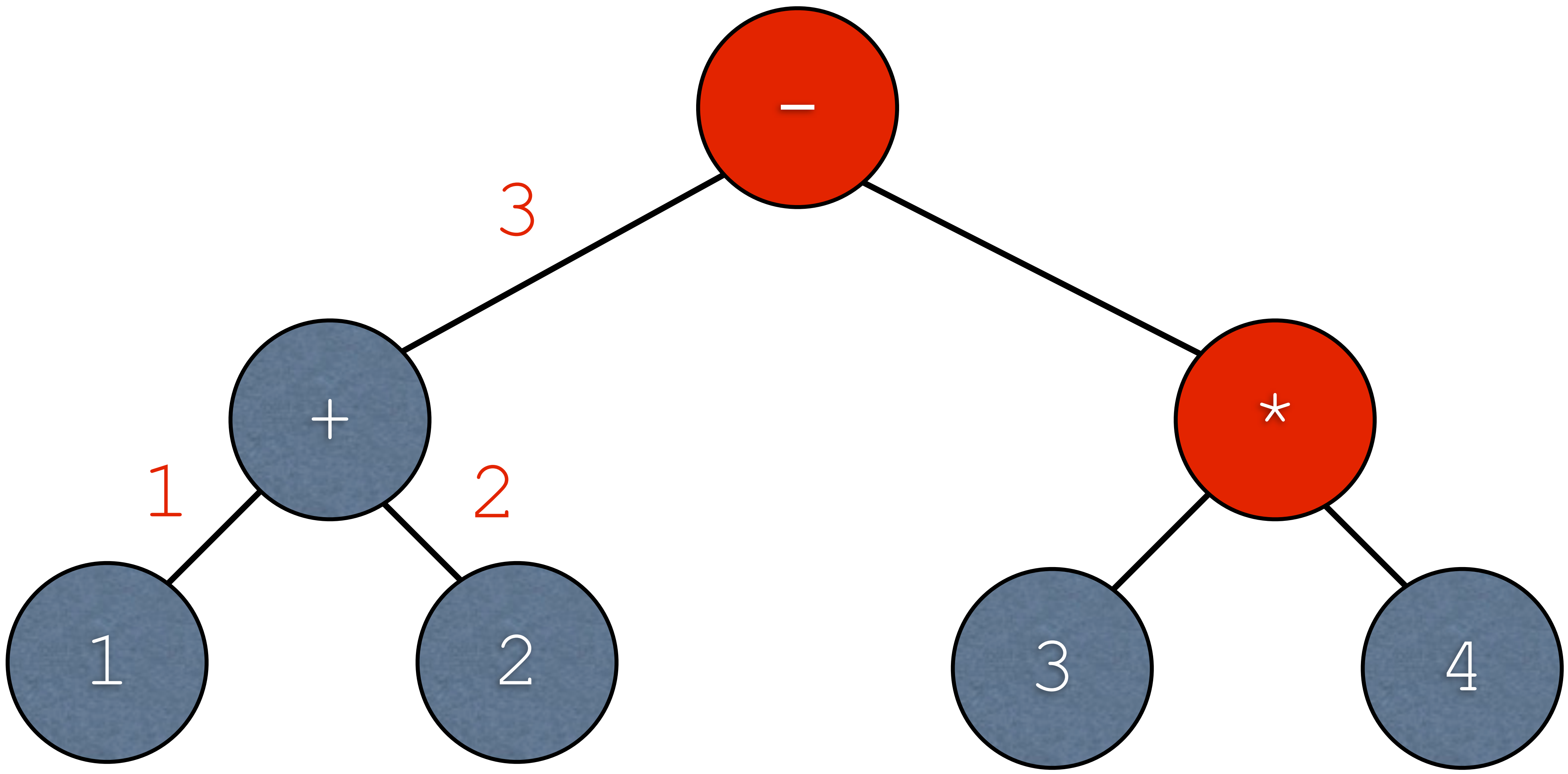
-+ performs the actual addition

Evaluation



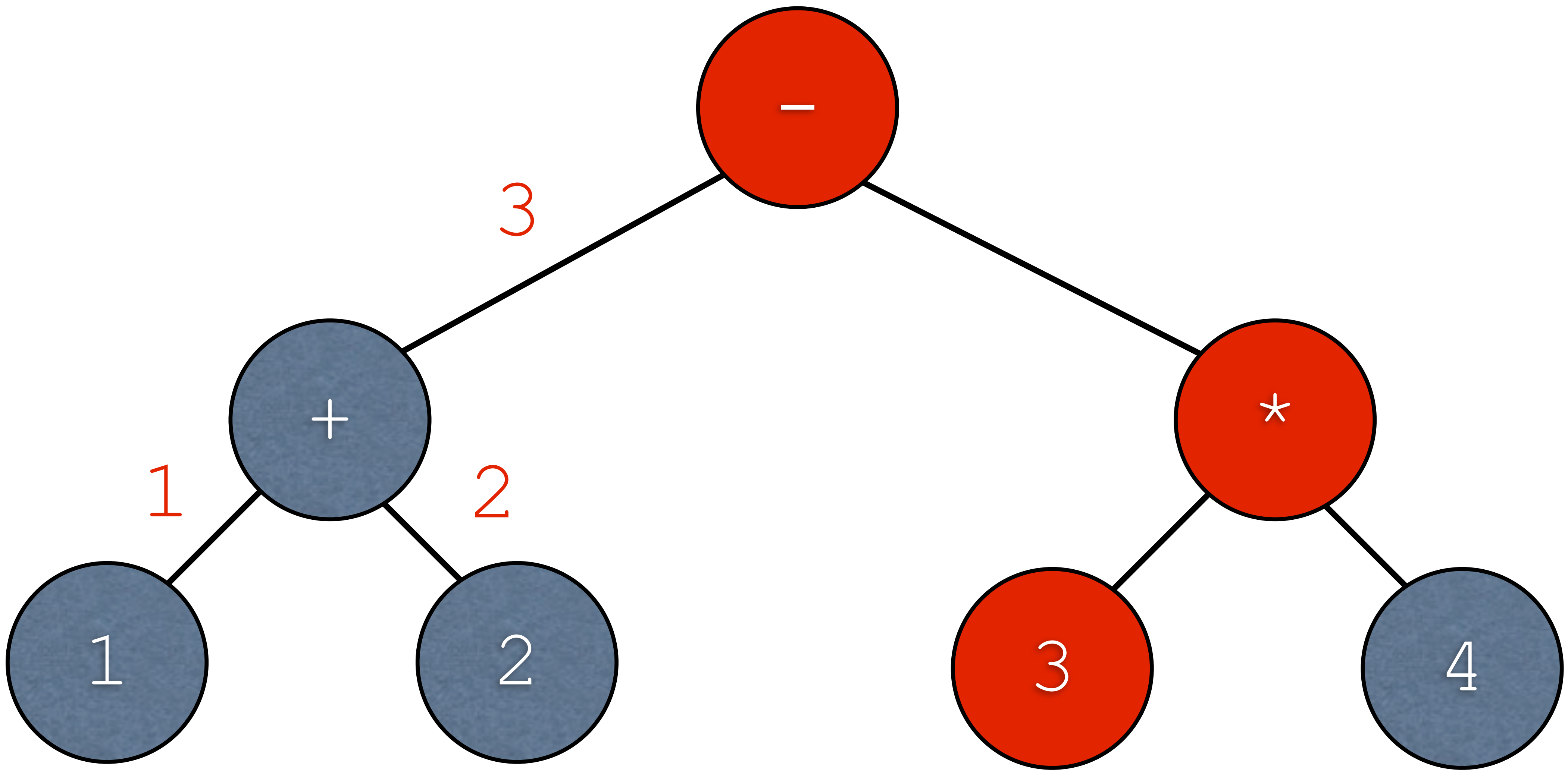
-Now + is taken care of
-Going back to -, - now has the value of the left subtree, and it needs the value of the right subtree

Evaluation



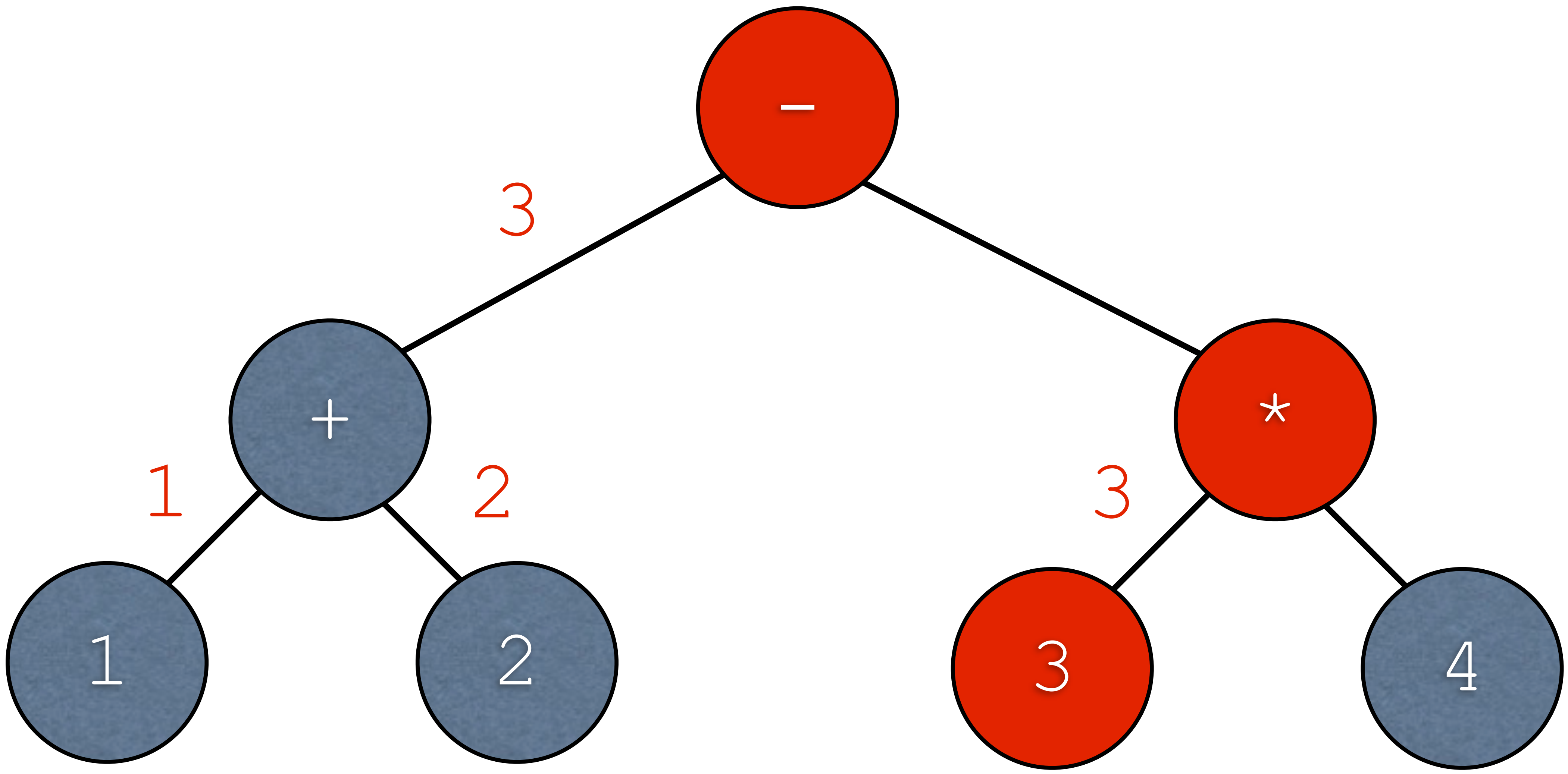
–Now we’re on *, which needs the value of the left subtree...

Evaluation



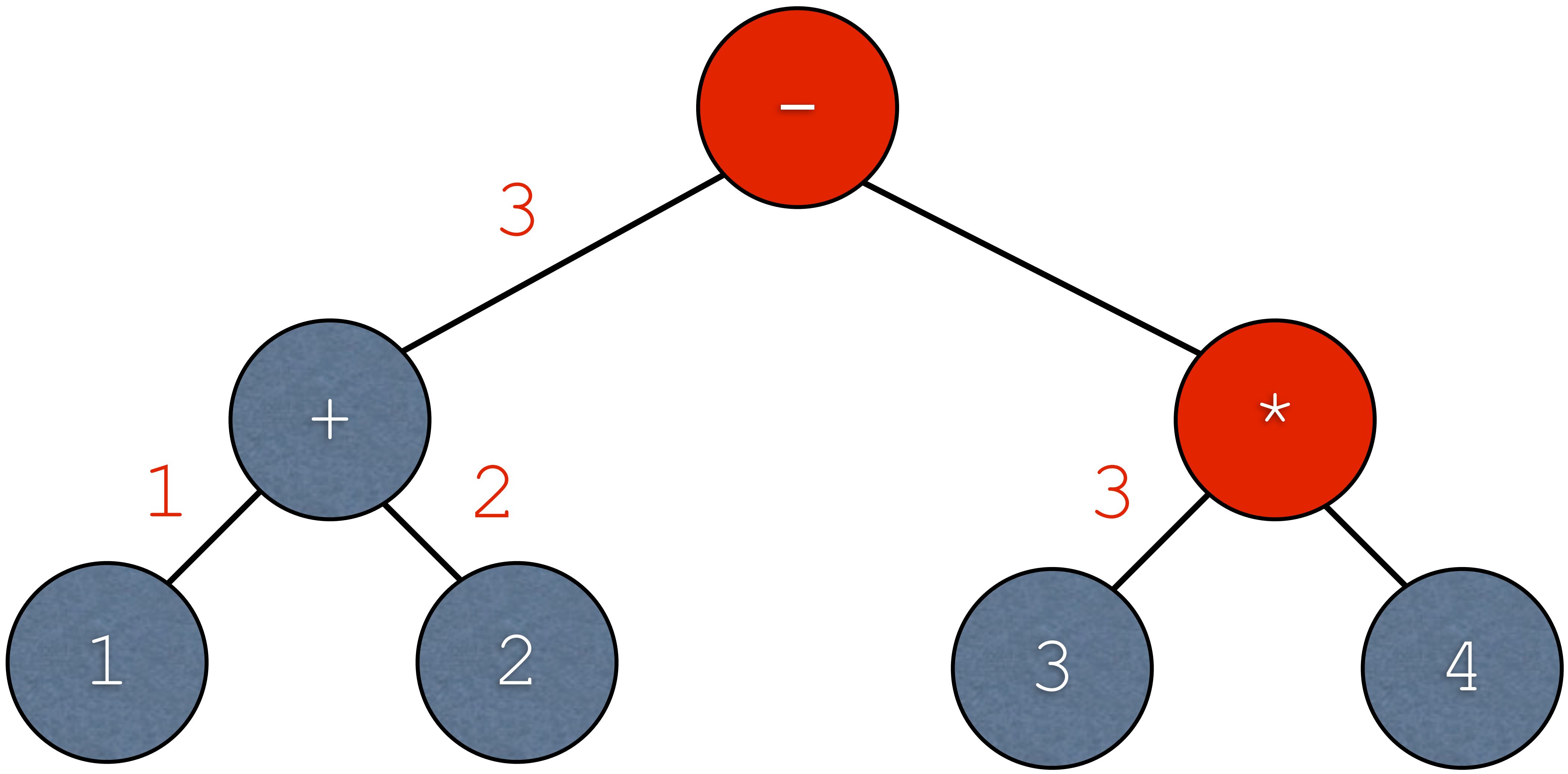
–Now we're on *, which needs the value of the left subtree...

Evaluation



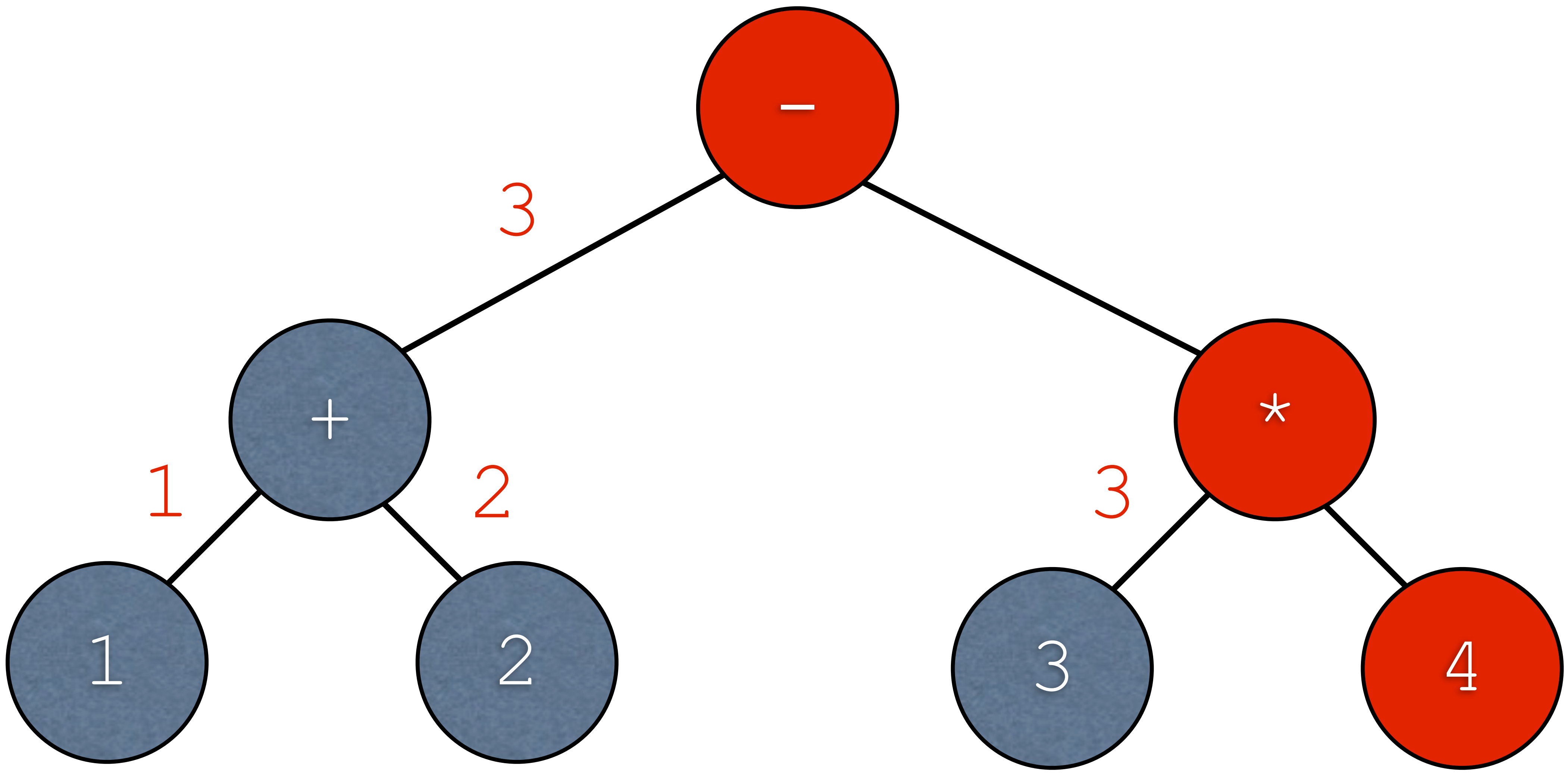
–Leaves again return the values held within...

Evaluation



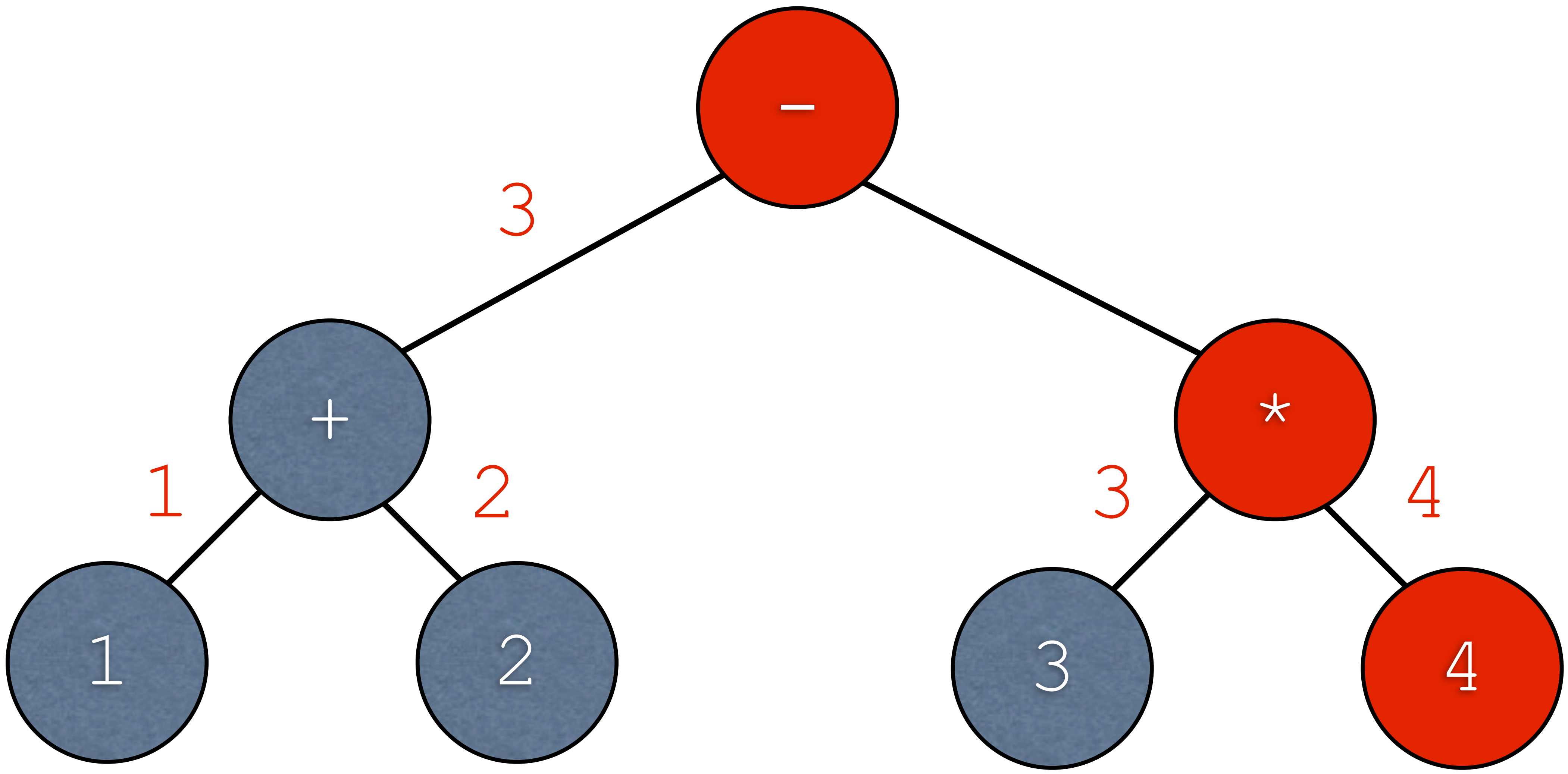
–Left subtree done; * now needs the value of the right subtree...

Evaluation



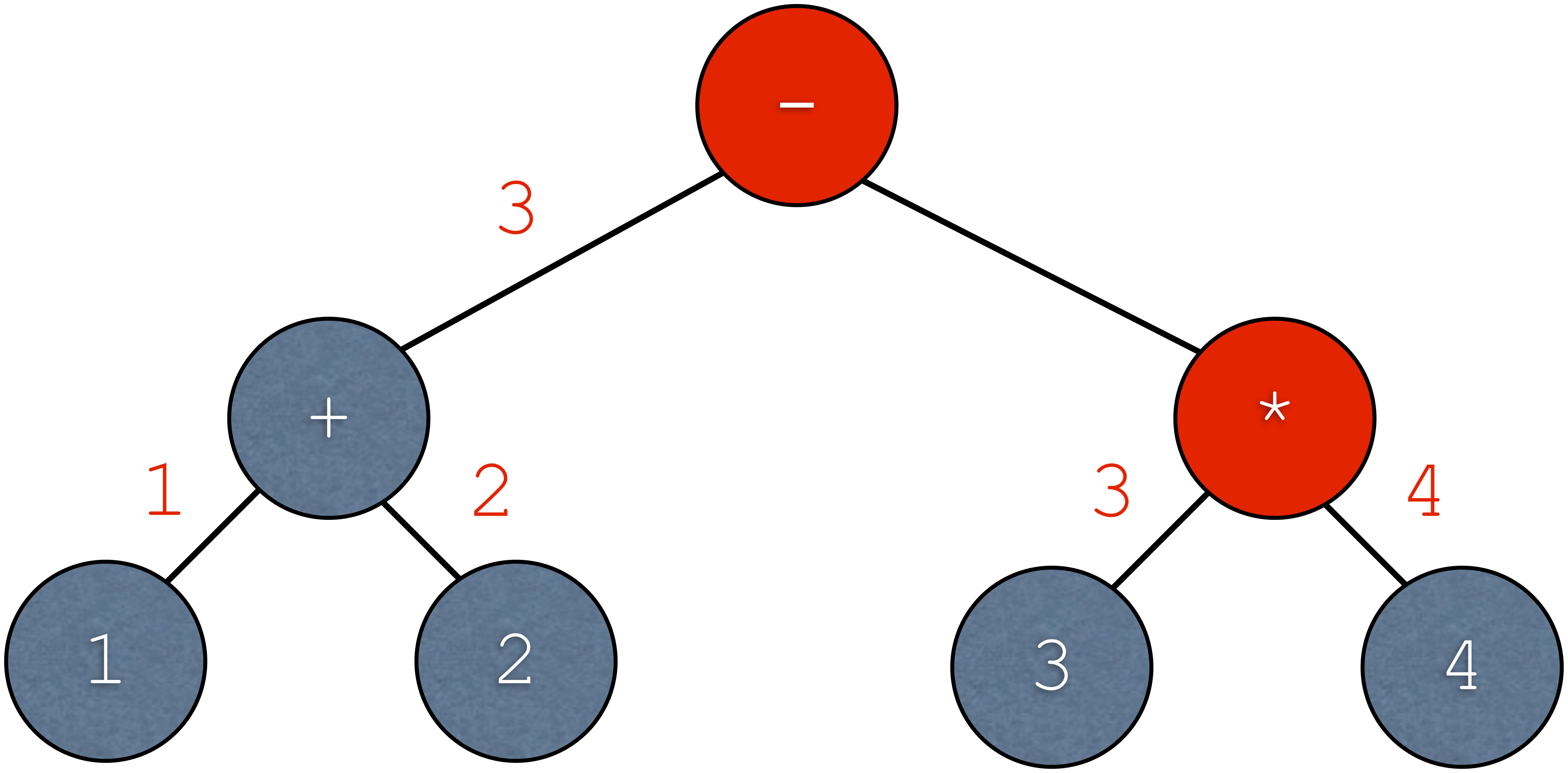
–Left subtree done; * now needs the value of the right subtree...

Evaluation



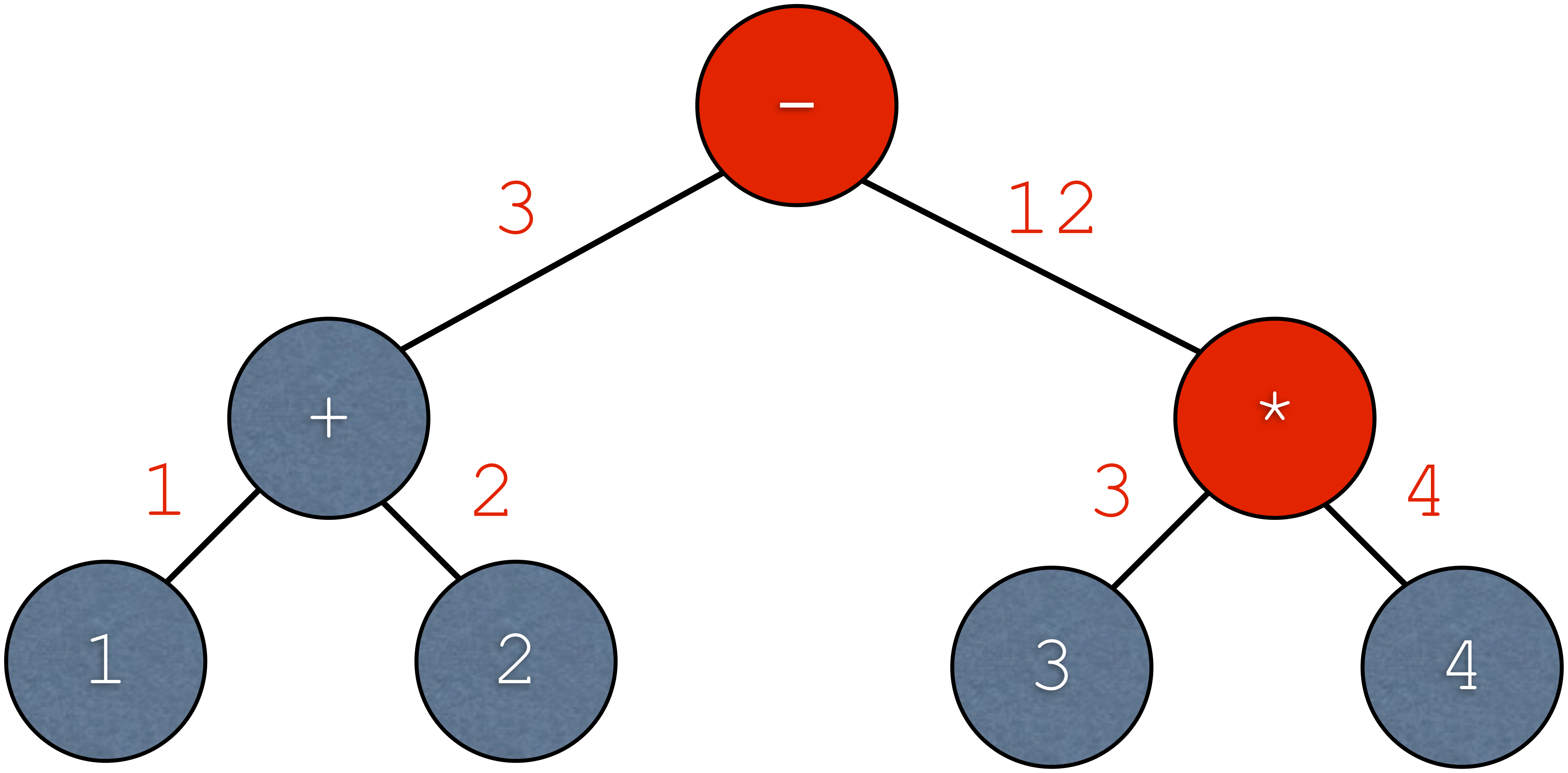
-Leaf returns value held within

Evaluation



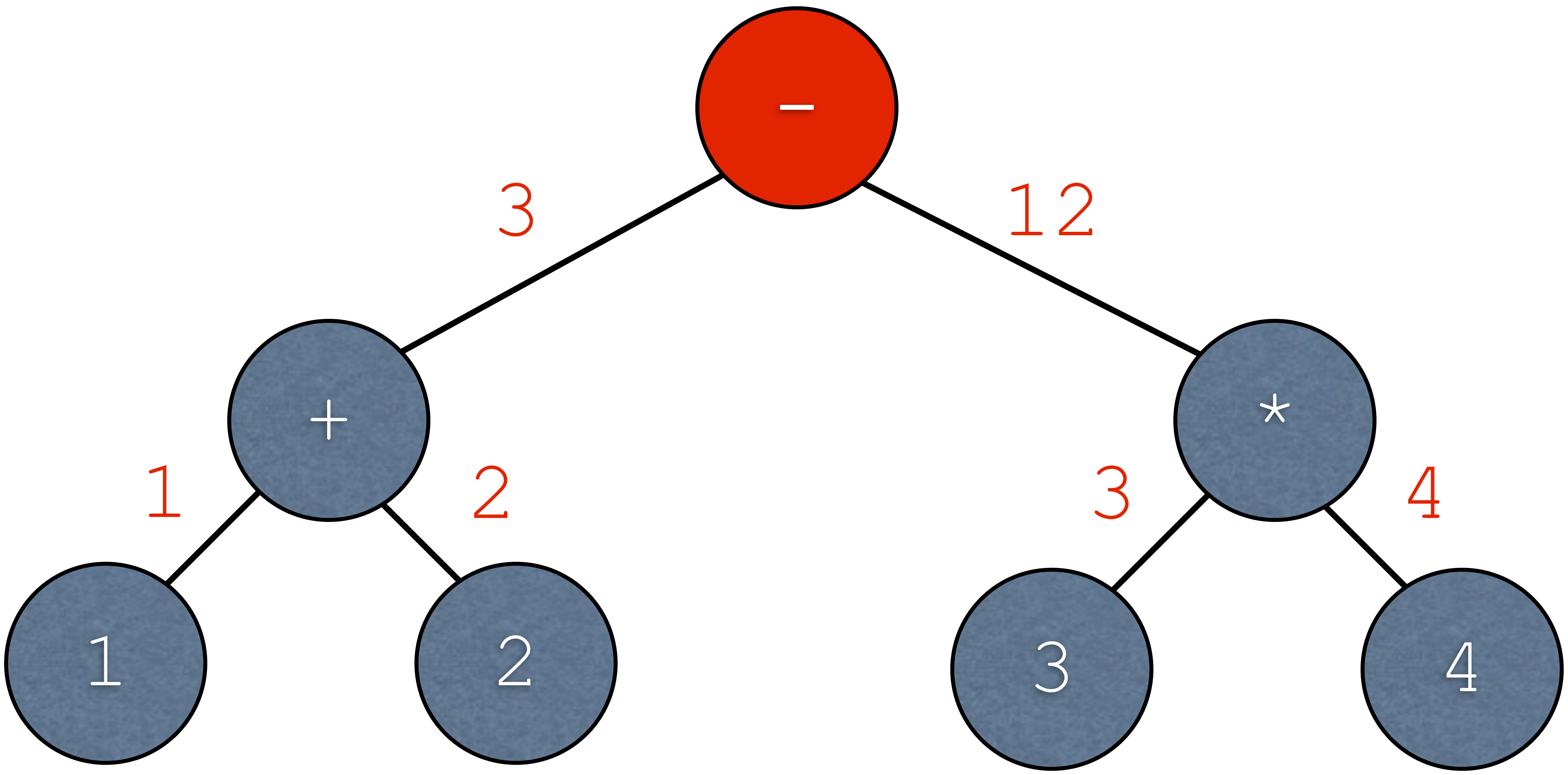
-Leaf is done. * now has both operands it needs...

Evaluation



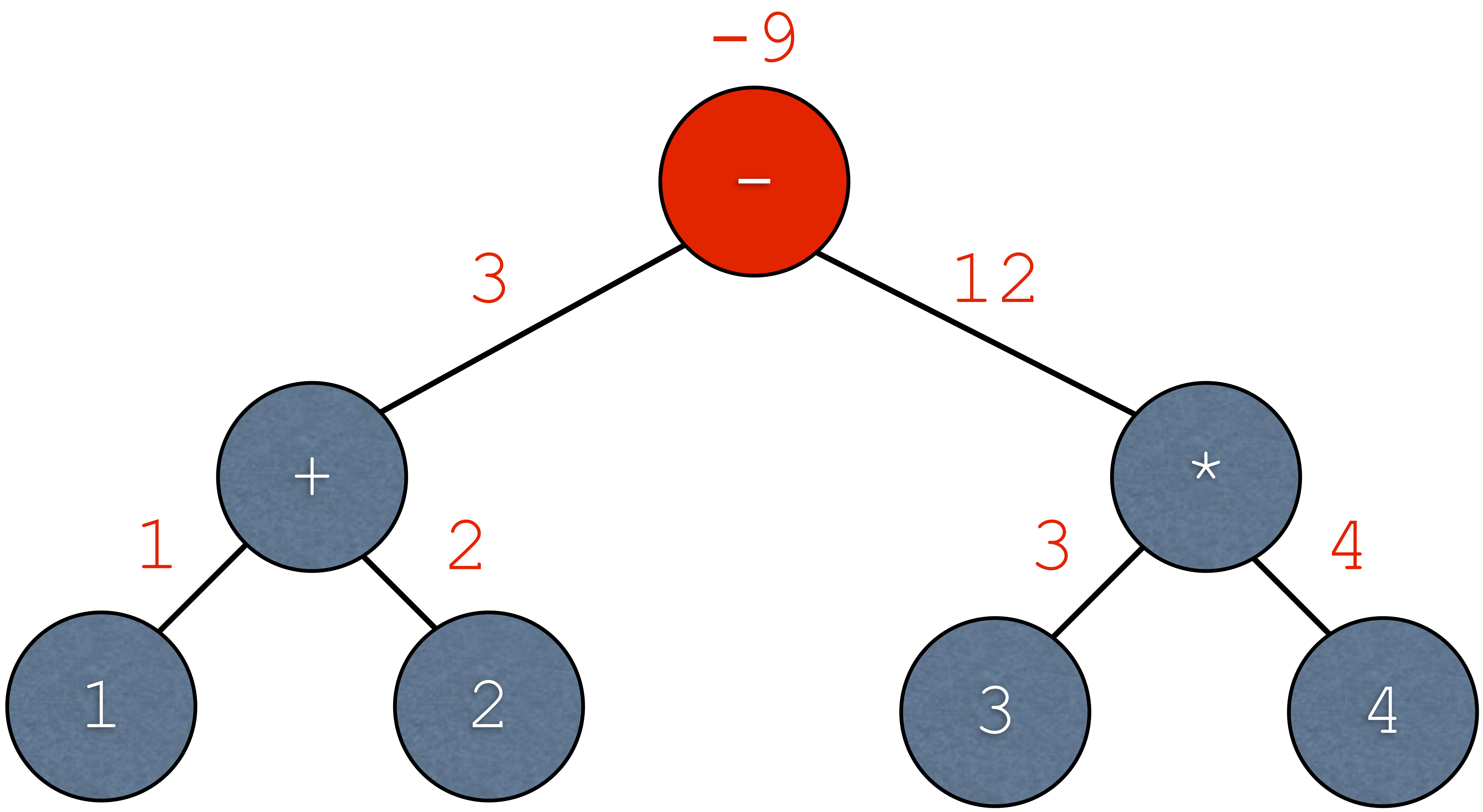
–* performs the multiplication and returns the value

Evaluation



-The root - node now has both operands...

Evaluation



–...and it returns the result of the subtraction

Evaluator Example:

`arithmetic_evaluator.rkt`

-Complete example online; we'll live-code this in class

SAT and Semantic Tableau

SAT Background

SAT

- Short for the Boolean satisfiability problem
- Given a Boolean formula with variables, is there an assignment of true/false to the variables which makes the formula true?

SAT

- Short for the Boolean satisfiability problem
- Given a Boolean formula with variables, is there an assignment of true/false to the variables which makes the formula true?

$$(x \lor \neg y) \land (\neg x \lor z)$$

SAT

- Short for the Boolean satisfiability problem
- Given a Boolean formula with variables, is there an assignment of true/false to the variables which makes the formula true?

$(x \lor \neg y) \land (\neg x \lor z)$

Yes: x is true, z is true

SAT

- Short for the Boolean satisfiability problem
- Given a Boolean formula with variables, is there an assignment of true/false to the variables which makes the formula true?

$$(x \lor \neg y) \land (\neg x \lor z)$$

Yes: x is true, z is true

$$(x \land \neg x)$$

SAT

- Short for the Boolean satisfiability problem
- Given a Boolean formula with variables, is there an assignment of true/false to the variables which makes the formula true?

$(x \lor \neg y) \land (\neg x \lor z)$

Yes: x is true, z is true

$(x \land \neg x)$

No

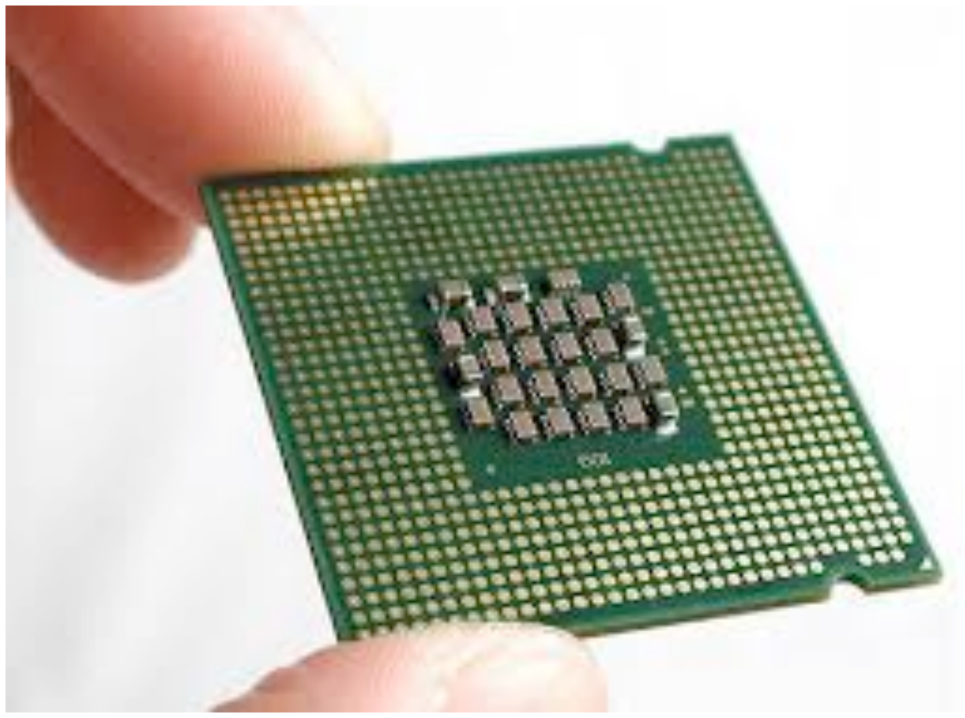
Relevance

Widespread usage in hardware and software verification

-Verification as in _proving_ the system does what we intend
-Much stronger guarantees than testing
-Testing can prove the existence of a bug (a failed test), whereas verification proves the absence of bugs (relative to the theorems proven)

Relevance

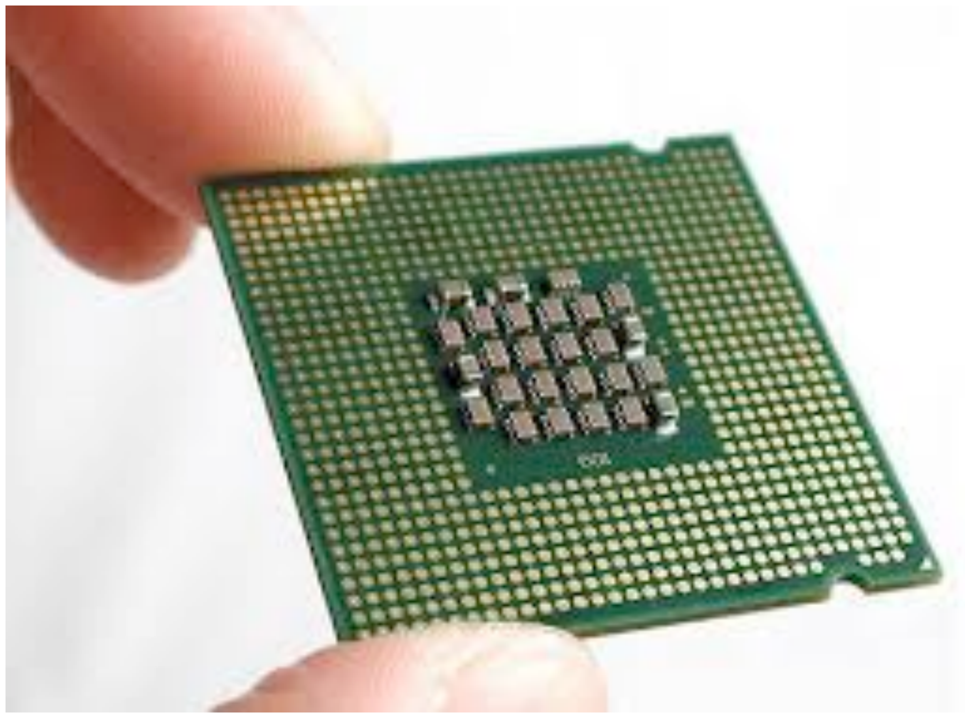
Widespread usage in hardware and software verification



-Circuits can be represented as Boolean formulas
-Can basically phrase proofs as Circuit $\wedge$ BadThing. If unsatisfiable, then BadThing cannot occur. If satisfiable, then the solution gives the circumstance under which BadThing occurs.
-Many details omitted (entire careers are based on this stuff)

Relevance

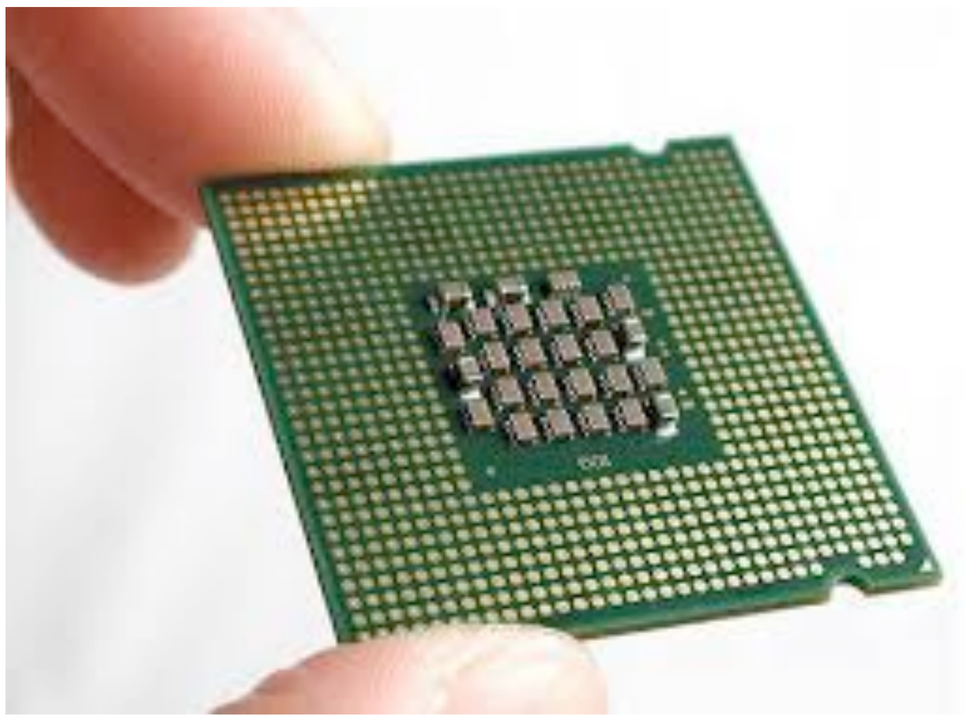
Widespread usage in hardware and software verification



-(Likely) used by AirBus to verify that flight control software does the right thing
-Lots of proprietary details so it's not 100% clear how this verification works, but SAT is still relevant to the problem

Relevance

Widespread usage in hardware and software verification



-Nasa uses software verification for a variety of tasks; SAT is relevant, though other techniques are used, too

Relevance to Logic Programming

- Methods for solving SAT can be used to execute logic programs
- Logic programming can be phrased as SAT with some additional stuff

Semantic Tableau

- One method for solving SAT instances
- Basic idea: iterate over the formula
  - Maintain subformulas that must be true
  - Learn which variables must be true/false
  - Stop at conflicts (unsatisfiable), or when no subformulas remain (have solution)

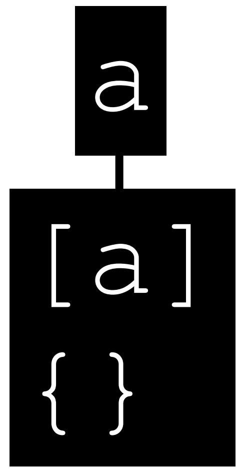
-There are many methods to this

Positive Literals

a

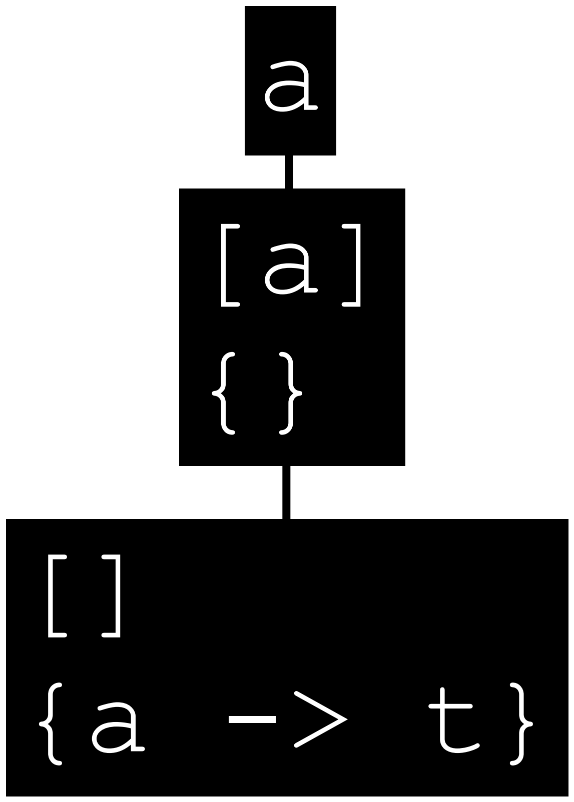
–As in, the input formula is simply “a”

Positive Literals



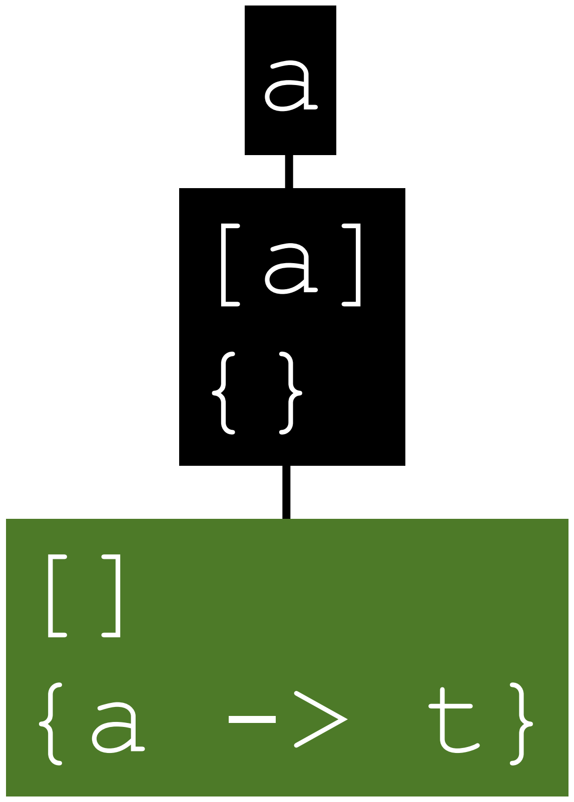
-One subformula must be true: a
-Initially, we don't know what any variables must map to

Positive Literals



–For formula “a” to be true, it must be the case that a is true

Positive Literals



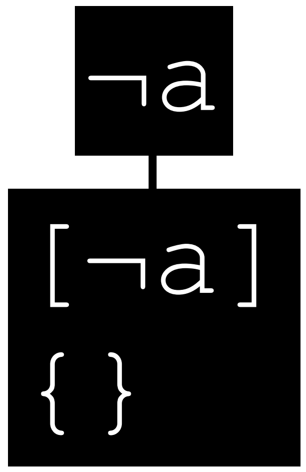
-No subformulas remain, so we are done. The satisfying solution is that a must be true.

Negative Literals

¬a

–As in, the input formula is simply “¬a”

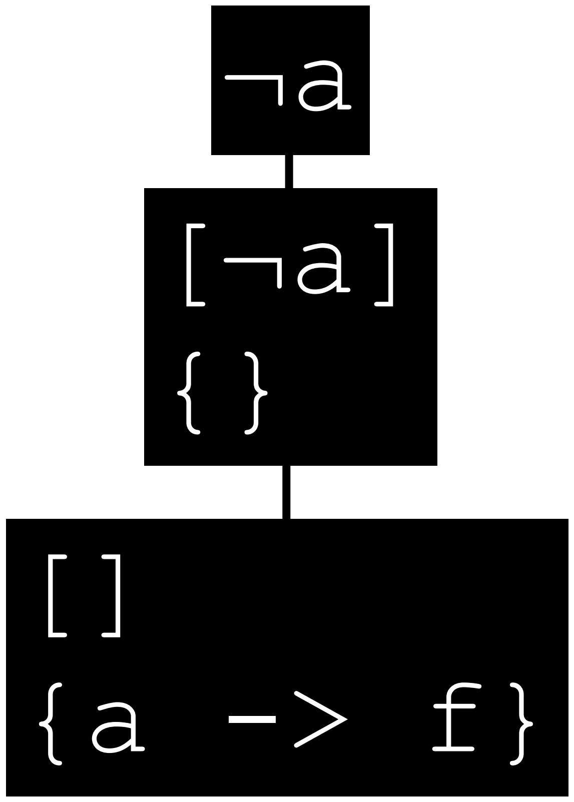
Negative Literals



-One subformula must be true: ¬a

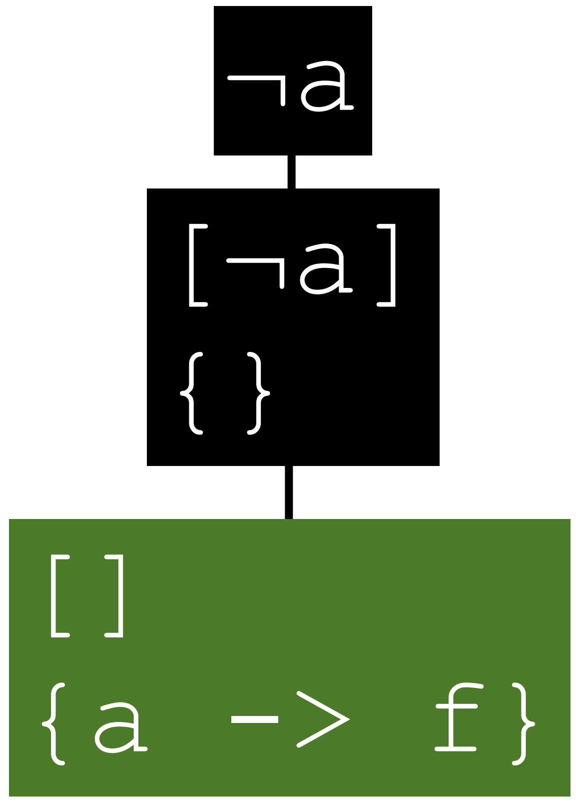
-Initially, we don't know what any variables must map to

Negative Literals



–For subformula “¬a” to be true, it must be the case that a is false

Negative Literals



–No subformulas remain, so we are done. The satisfying solution is that “a” must be false.

Logical And

$a \wedge b$

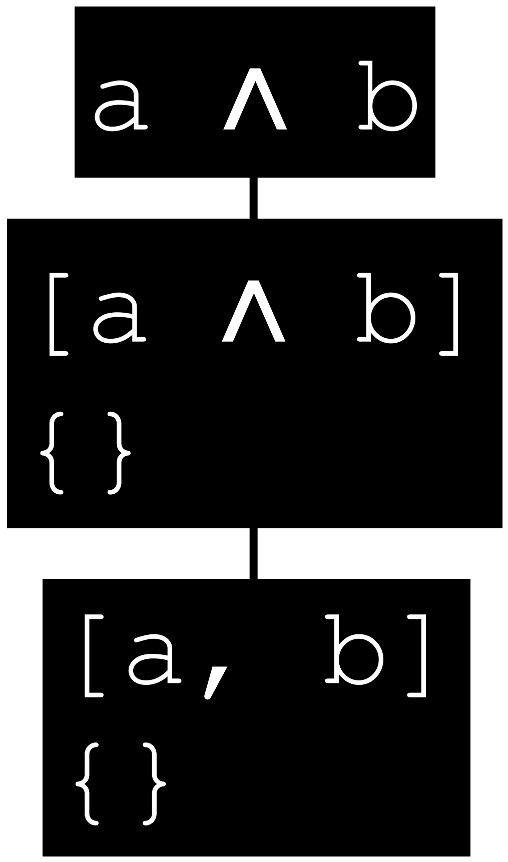
Logical And

a ∧ b

[a ∧ b]
{ }

-Initially, one subformula must be true: a ∧ b
-Initially, we don't know what any variable must map to

Logical And



–For a ∧ b to be true, subformulas a and b must both be true

Logical And

```
a ∧ b
  |
[a ∧ b]
{}
  |
[a, b]
{}
  |
[b]
{a -> t}
```

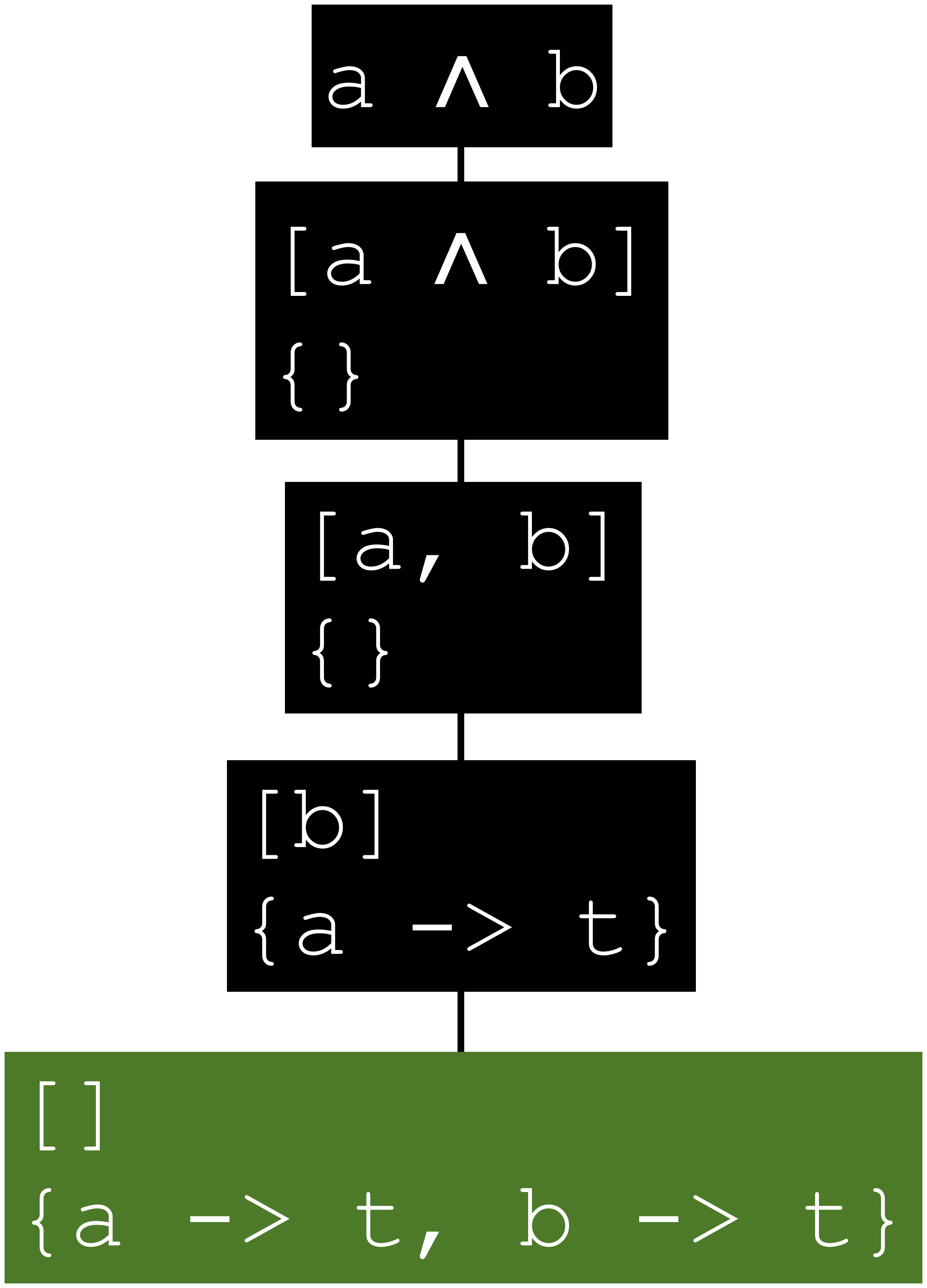
-From the positive literal case, for formula a to be true, variable a must be true

Logical And

```
a ∧ b

[a ∧ b]
{}

[a, b]
{}

[b]
{a -> t}

[]
{a -> t, b -> t}
```

-From the positive literal case, for formula b to be true, variable b must be true

Logical And



-No subformulas remain, so we are done with the solution that both a and b must be true

Logical And

$a \wedge \neg a$

–Alternative example, showing a conflict

Logical And

a ∧ ¬a

[a ∧ ¬a]

{ }

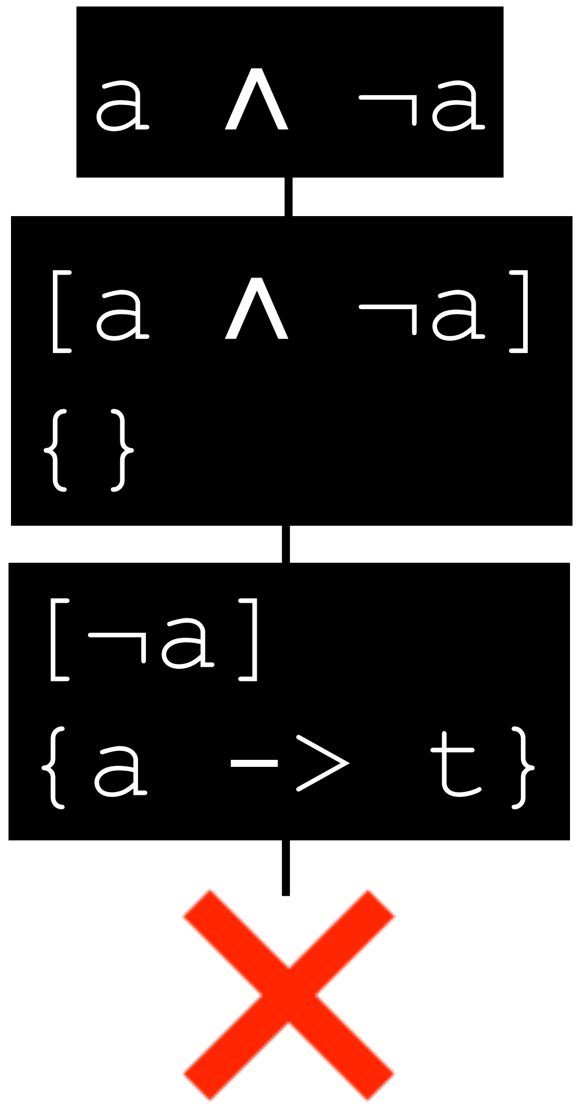
Logical And

```
a ∧ ¬a
```

```
[a ∧ ¬a]
{}
```

```
[¬a]
{a -> t}
```

Logical And



-Now we have a problem: for formula ¬a to be true, it must be the case that variable a is false
-We've already recorded that variable a must be true, which is the opposite of what we expect.
-As such, we have a conflict - this formula is unsatisfiable

Logical Or

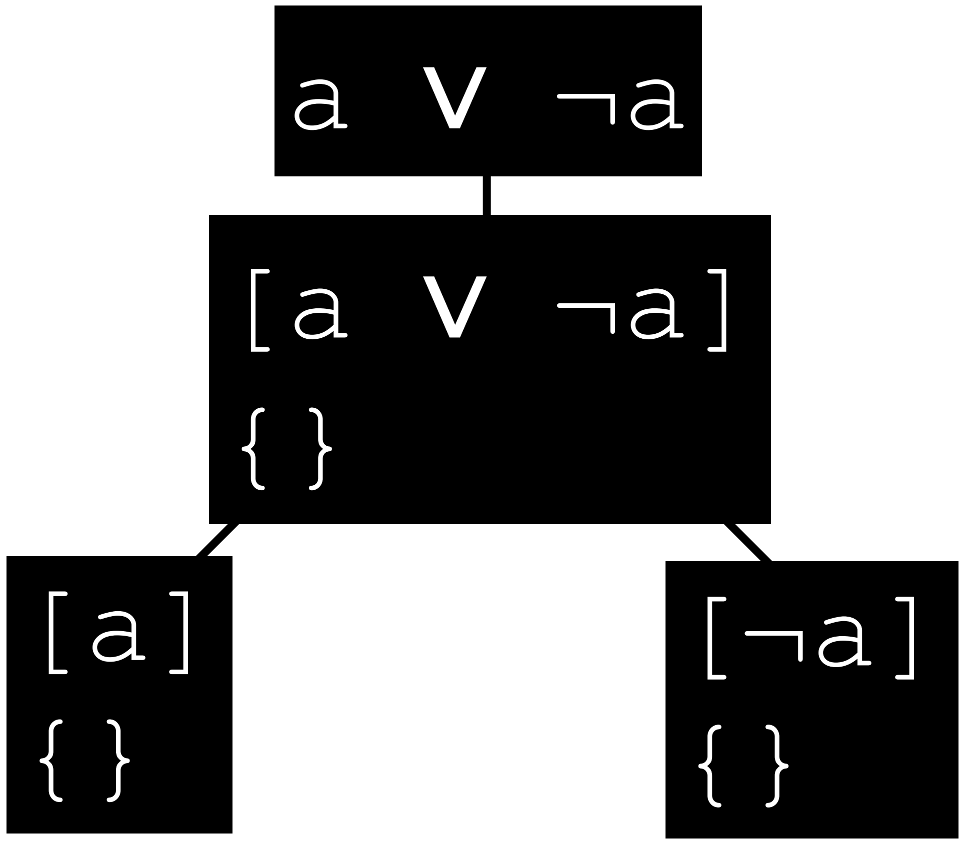
$a \vee \neg a$

Logical Or

a ∨ ¬a

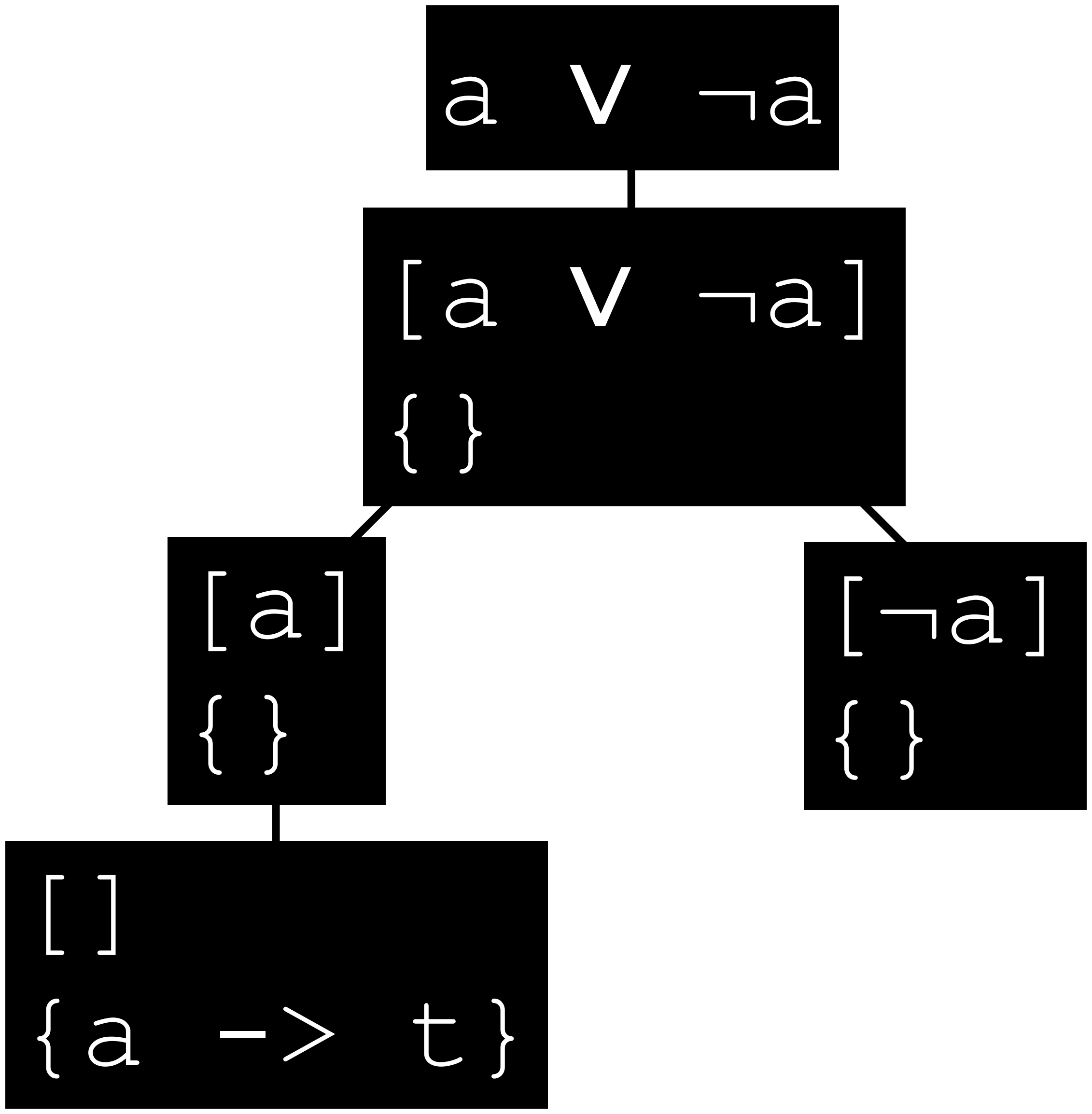
[a ∨ ¬a]
{ }

Logical Or



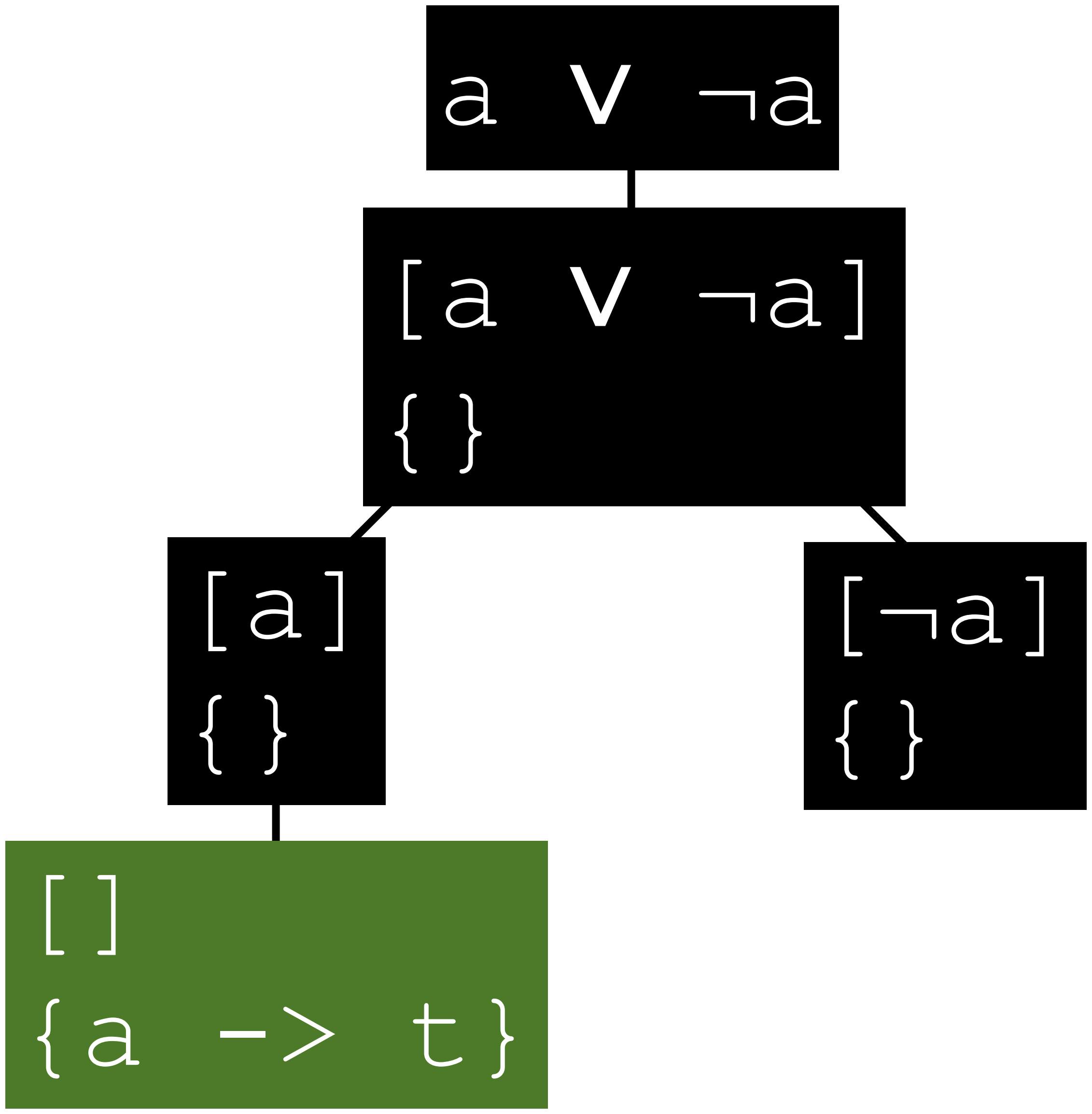
–World splits on or: in one world we pick the left subformula, and in another we pick the right

Logical Or



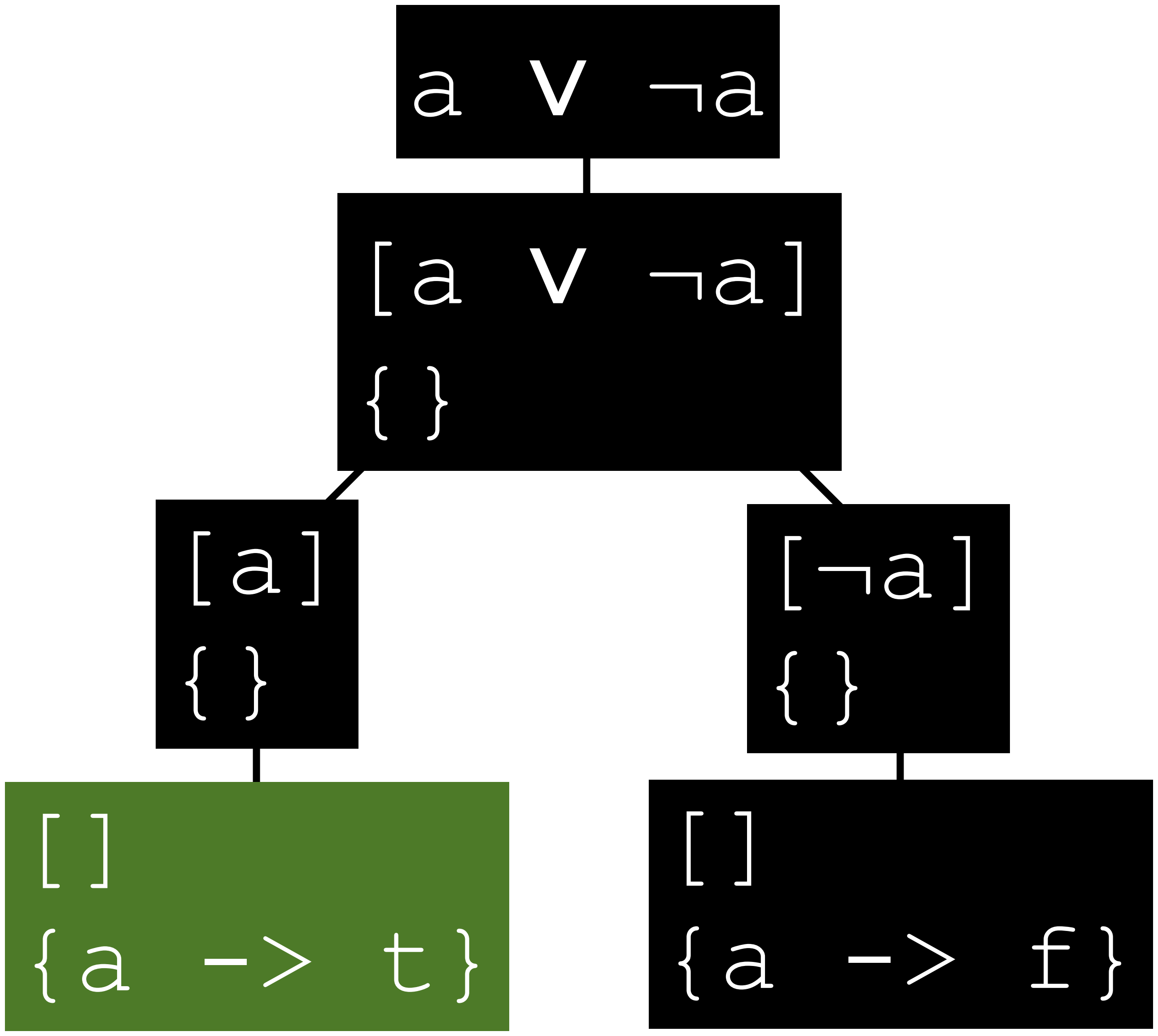
–World splits on or: in one world we pick the left subformula, and in another we pick the right

Logical Or



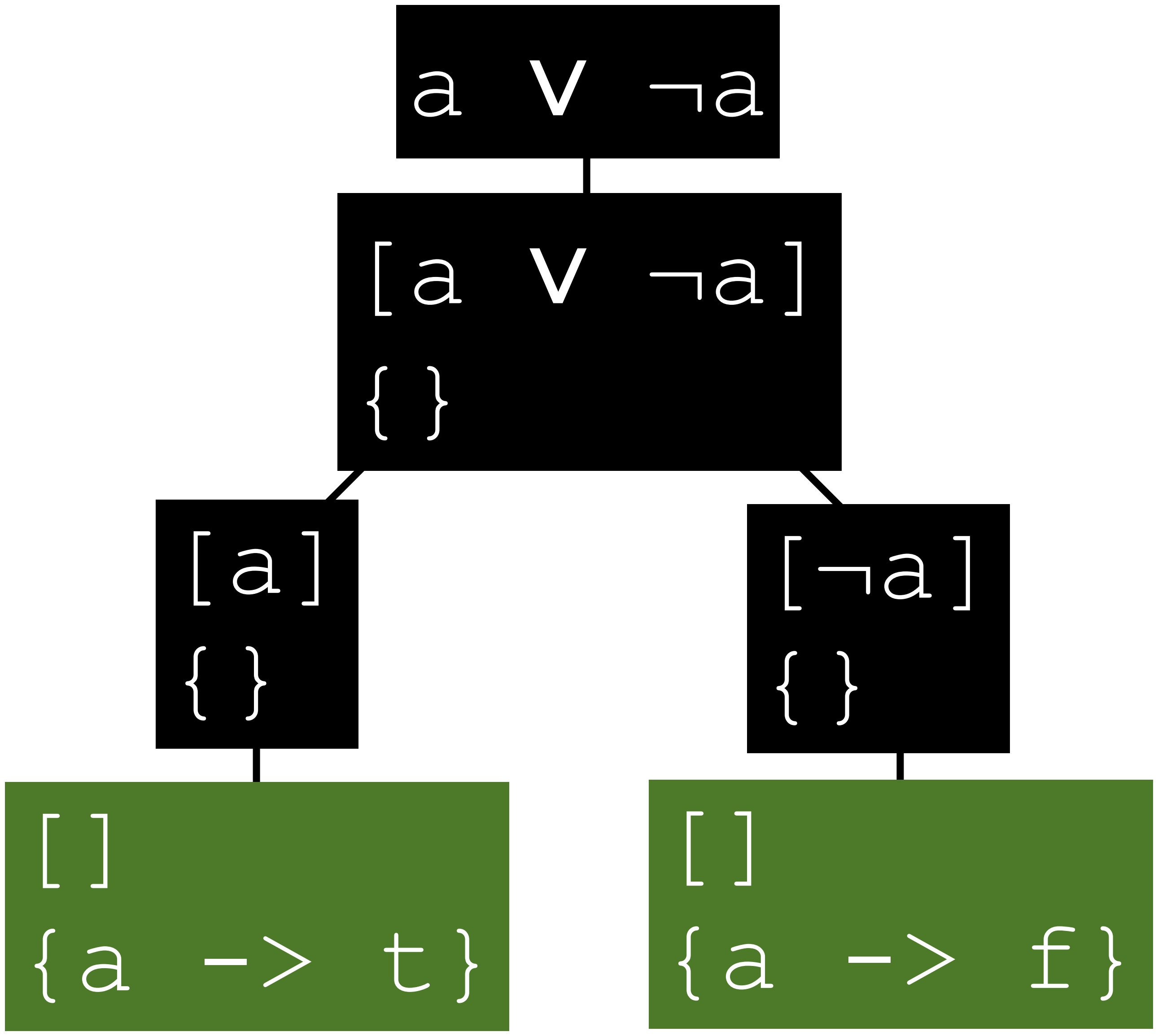
–World splits on or: in one world we pick the left subformula, and in another we pick the right

Logical Or



–World splits on or: in one world we pick the left subformula, and in another we pick the right

Logical Or



-World splits on or: in one world we pick the left subformula, and in another we pick the right

Examples

Example 1:

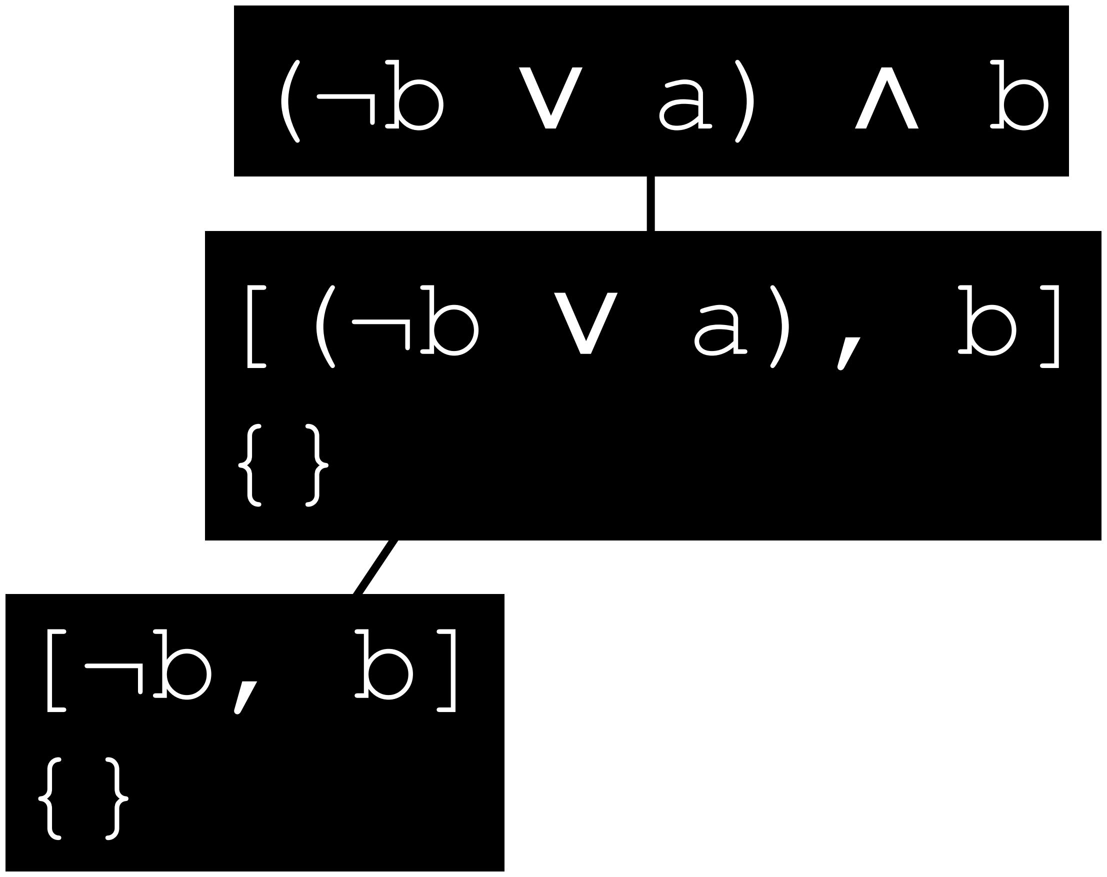
$$(\neg b \lor a) \land b$$

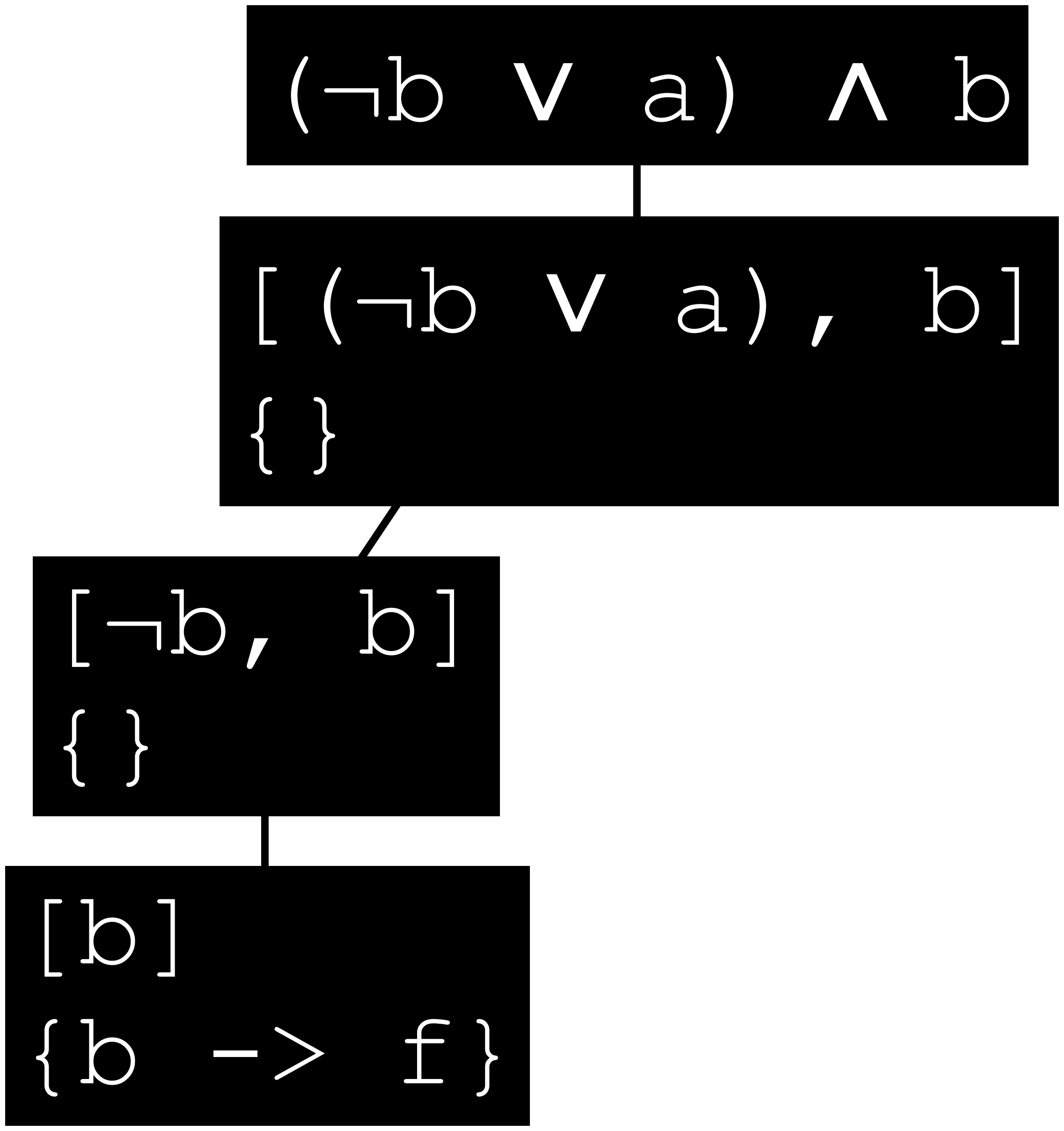
$(\neg b \lor a) \land b$

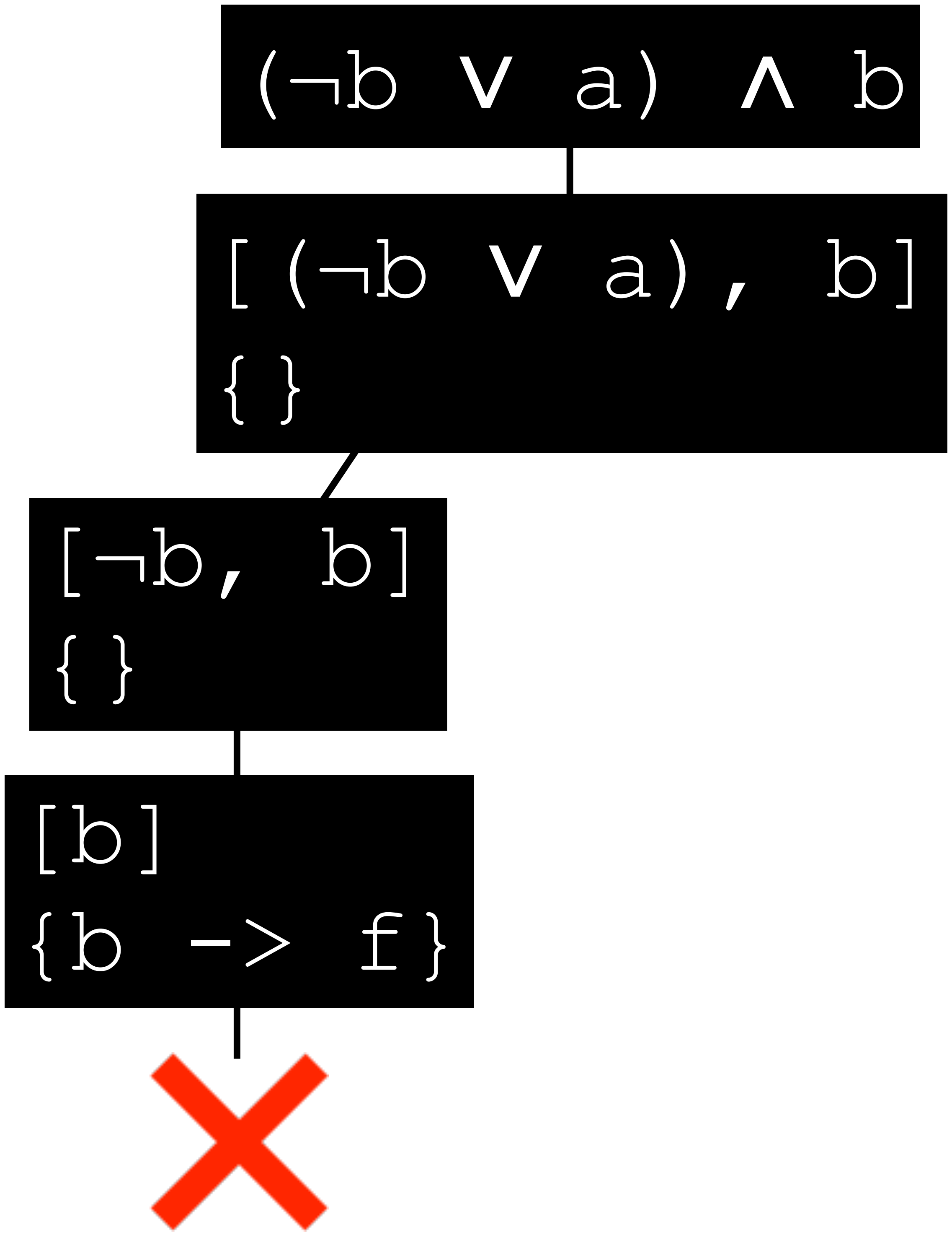
$(\neg b \lor a) \land b$

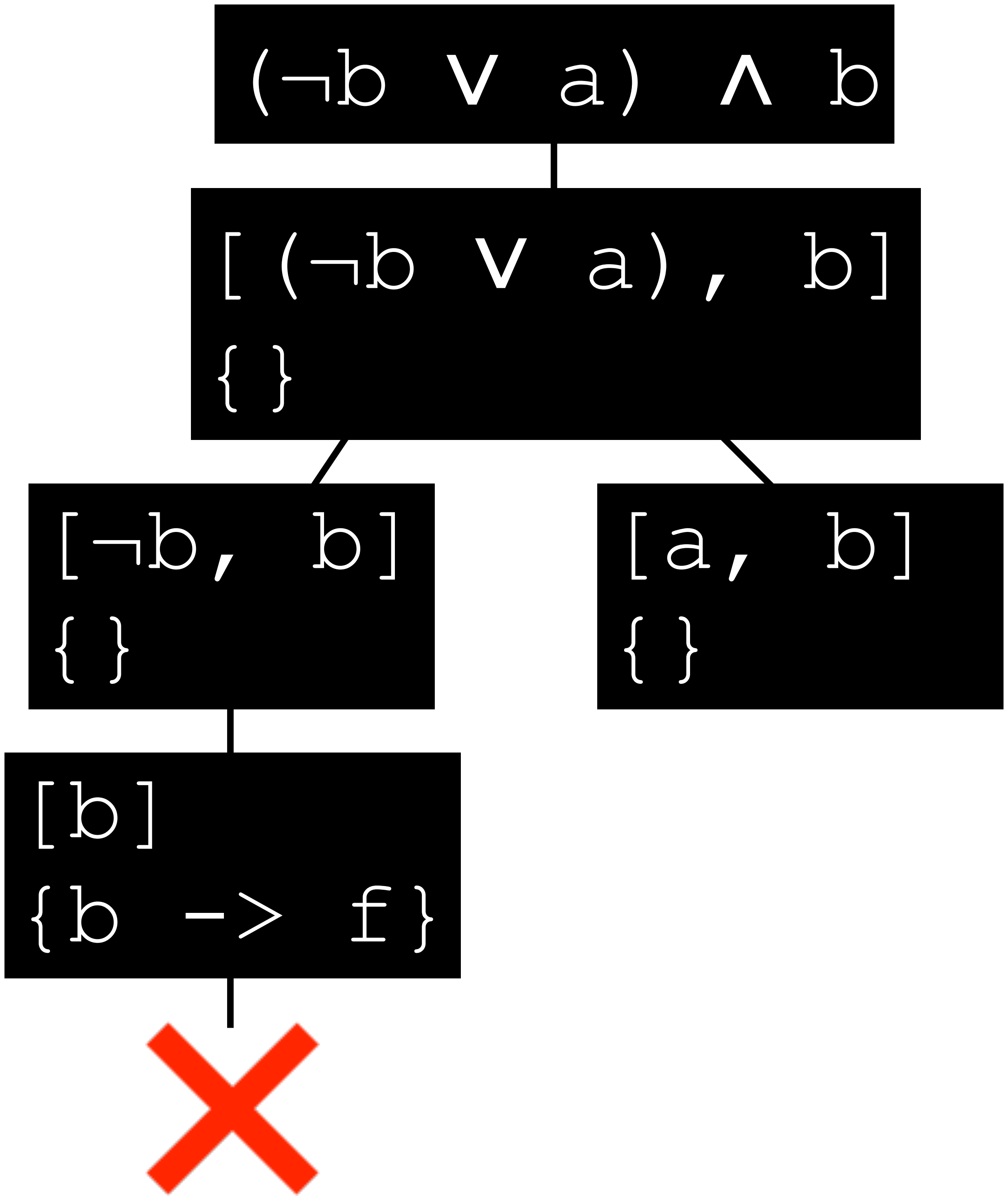
$[(\neg b \lor a),\ b]$

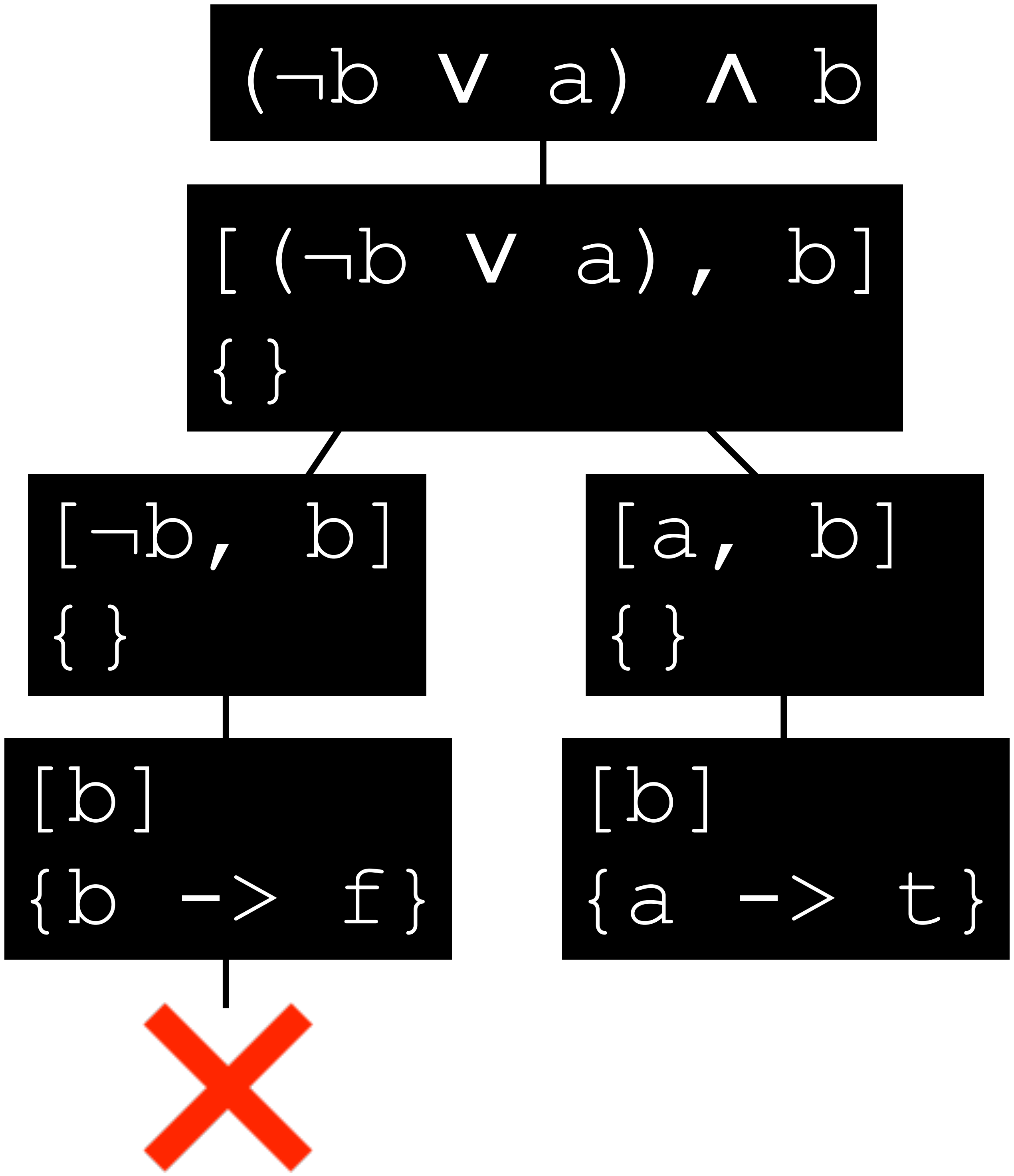
$\{\}$


(¬b ∨ a) ∧ b
[(¬b ∨ a), b]
{}
[¬b, b]
{}

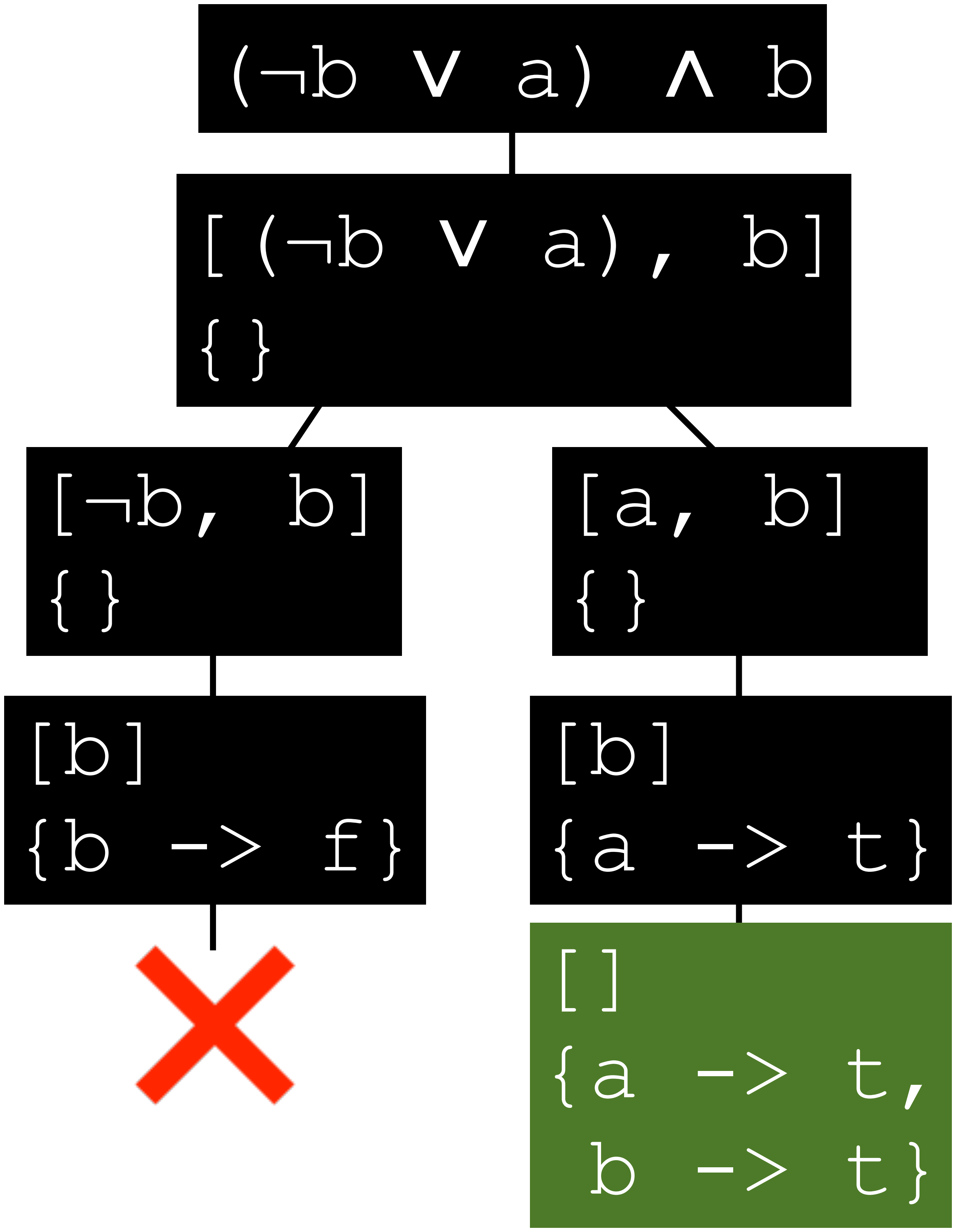

(¬b ∨ a) ∧ b
[(¬b ∨ a), b]
{}
[¬b, b]
{}
[b]
{b -> f}


(¬b ∨ a) ∧ b
[(¬b ∨ a), b]
{}
[¬b, b]
{}
[b]
{b -> f}


(¬b ∨ a) ∧ b
[(¬b ∨ a), b]
{}
[¬b, b]
{}
[a, b]
{}
[b]
{b -> f}



(¬b ∨ a) ∧ b
[(¬b ∨ a), b]
{}
[¬b, b]
{}
[a, b]
{}
[b]
{b -> f}
[b]
{a -> t}



(¬b ∨ a) ∧ b
[(¬b ∨ a), b]
{}
[¬b, b]
{}
[a, b]
{}
[b]
{b -> f}
[b]
{a -> t}
[]
{a -> t,
b -> t}

Example 2:

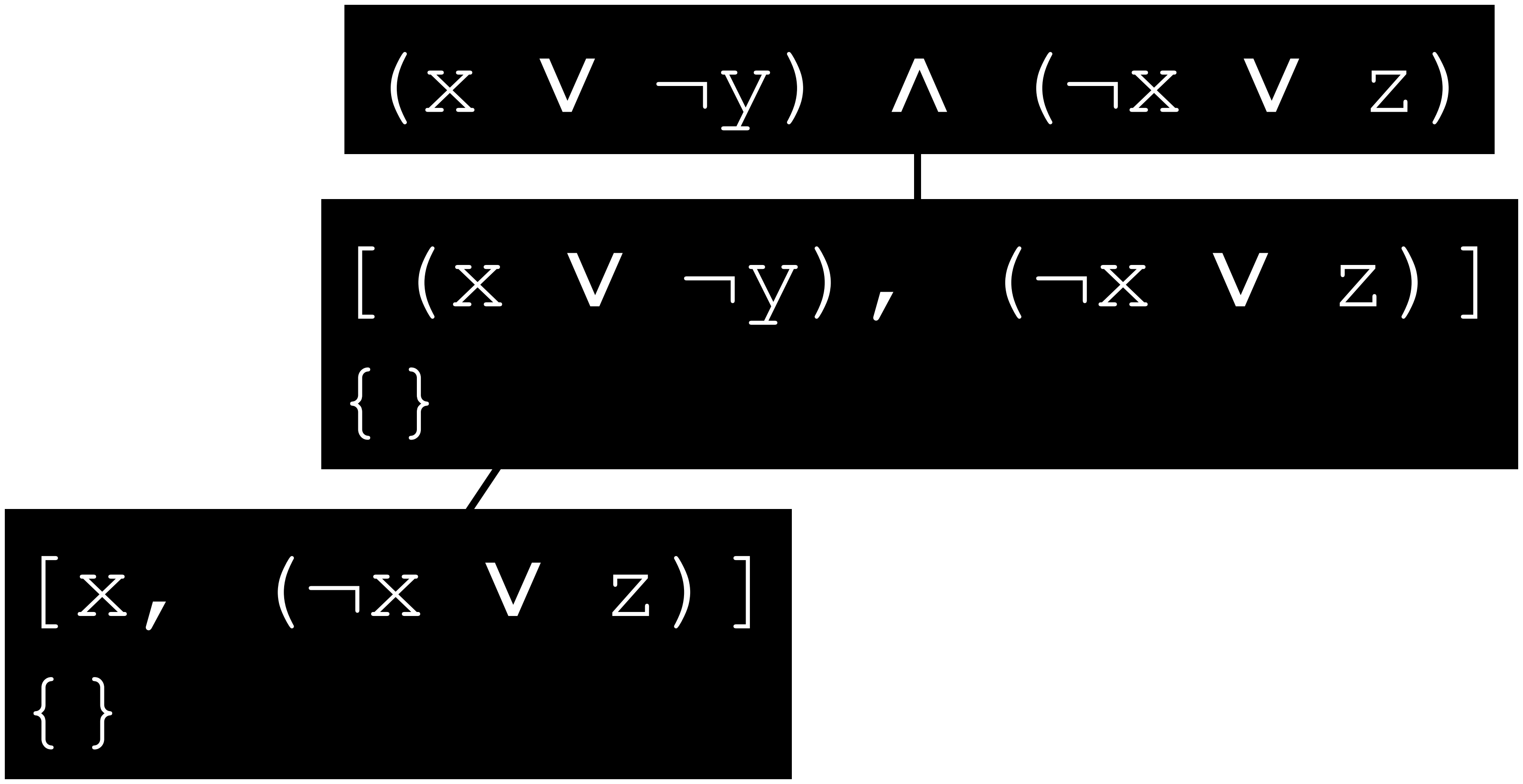
$$(x \lor \neg y) \land (\neg x \lor z)$$

$(x \lor \neg y) \land (\neg x \lor z)$

$(x \lor \neg y) \land (\neg x \lor z)$

$[(x \lor \neg y), (\neg x \lor z)]$

$\{\}$

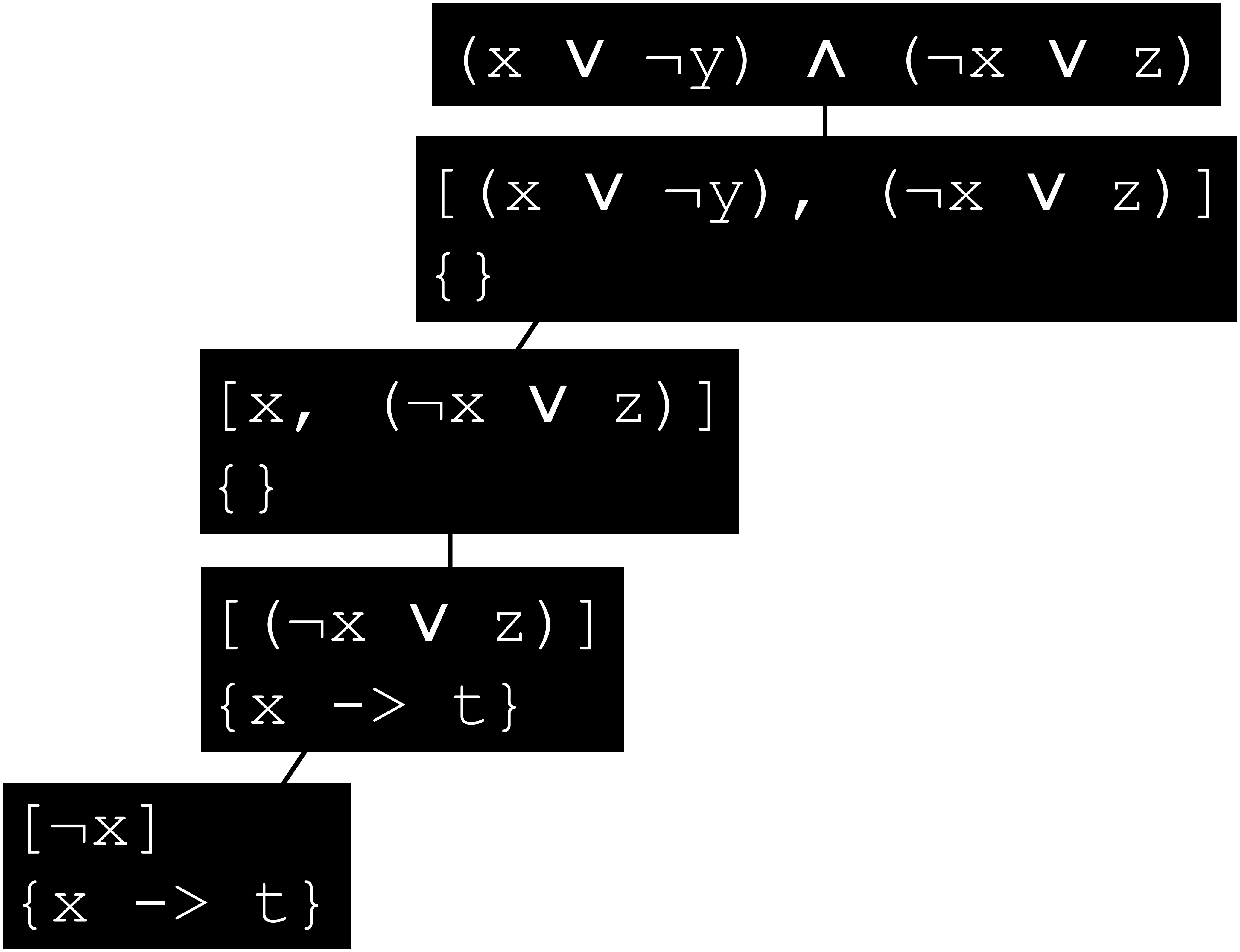

(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}

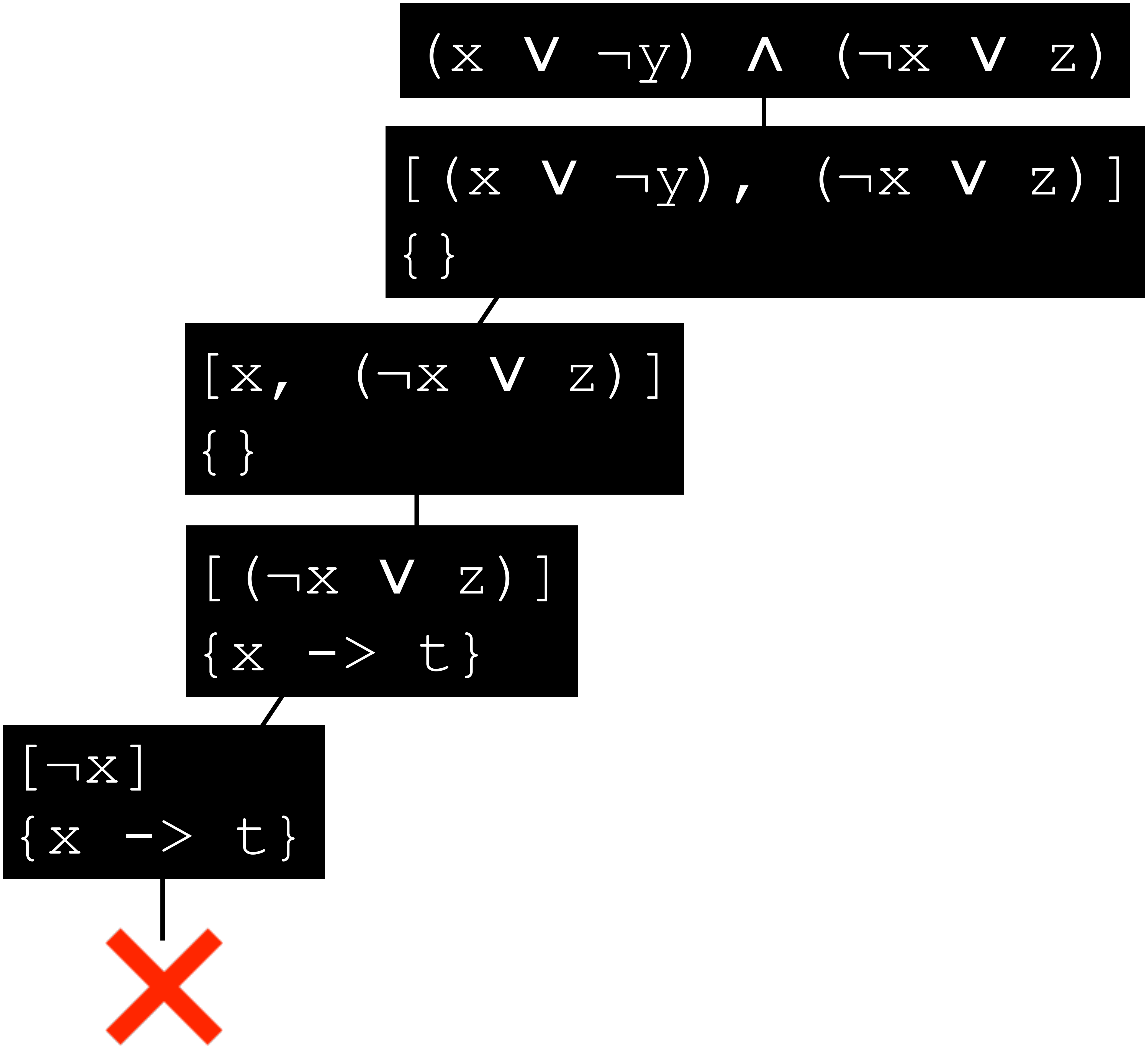
(x ∨ ¬y) ∧ (¬x ∨ z)

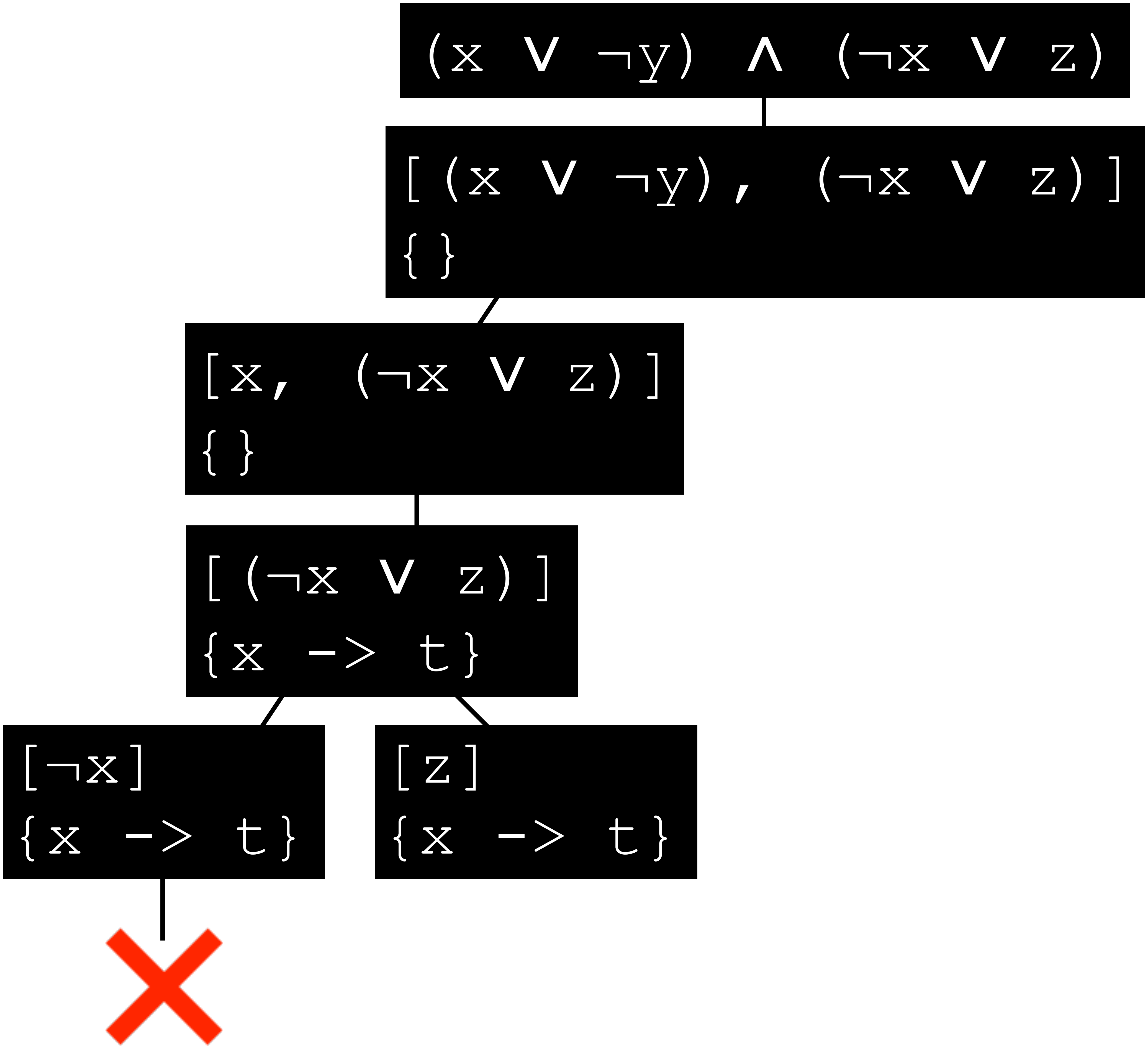
[(x ∨ ¬y), (¬x ∨ z)]
{}

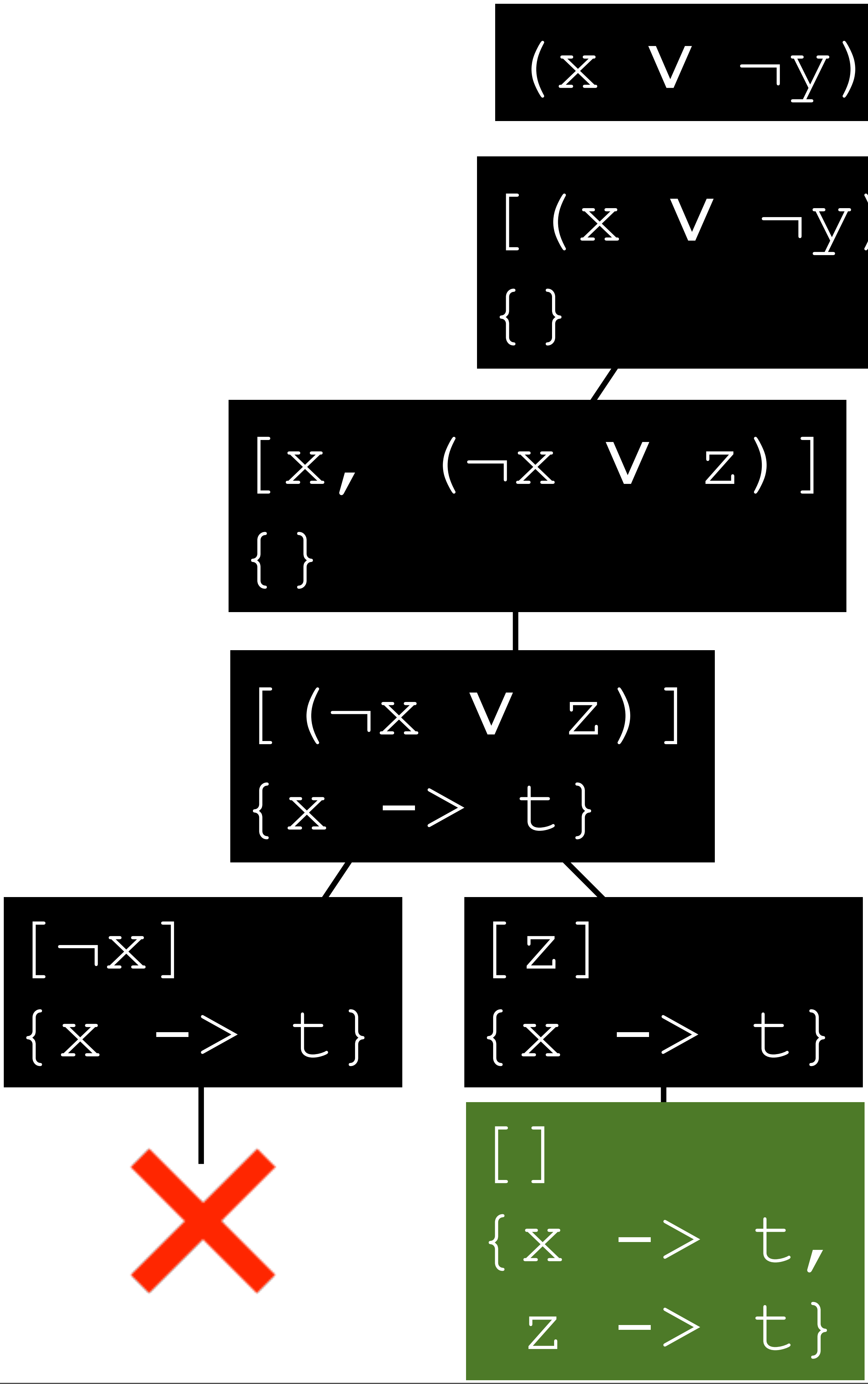
[x, (¬x ∨ z)]
{}

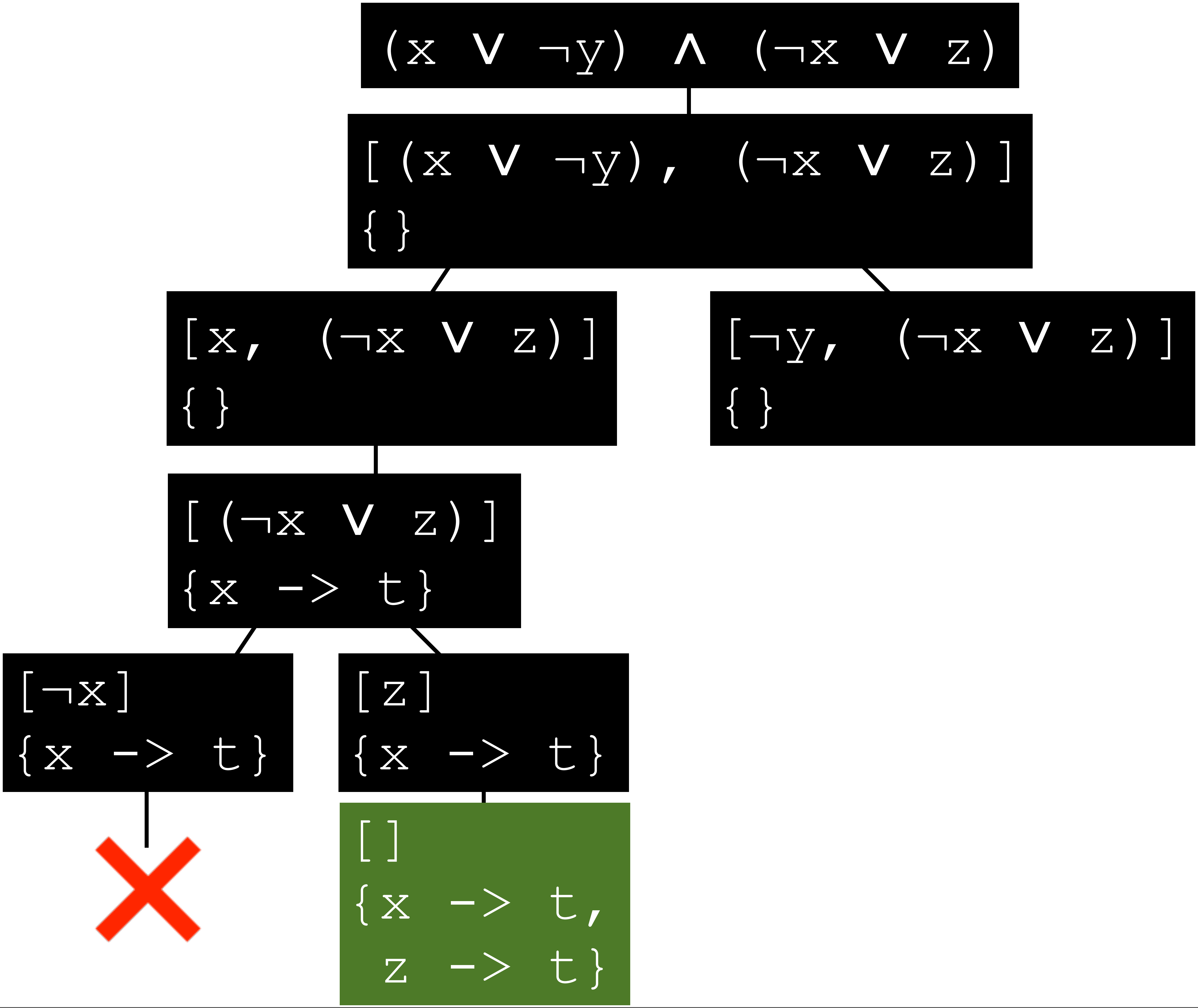
[(¬x ∨ z)]
{x -> t}

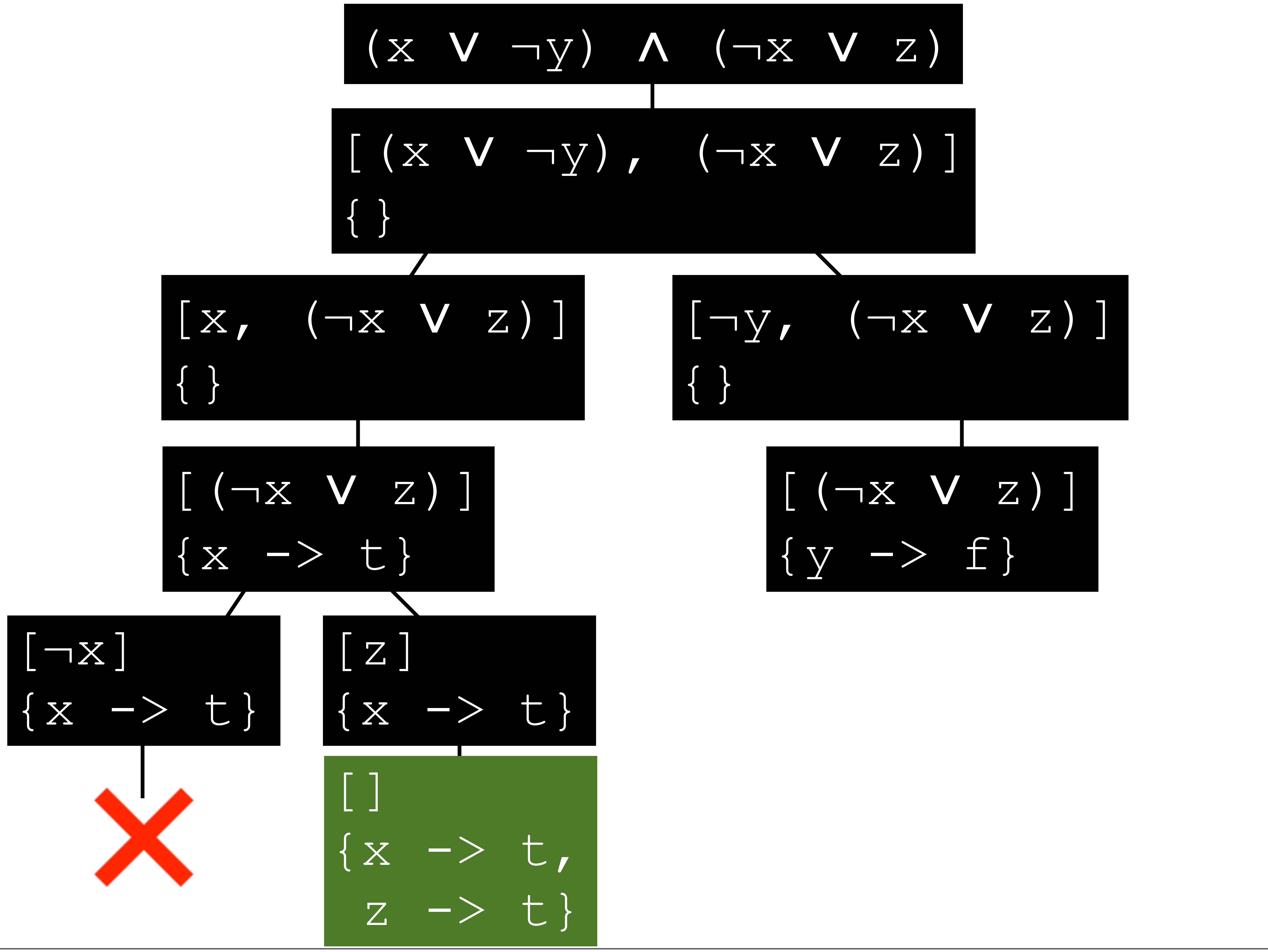

(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[¬x]
{x -> t}


(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[¬x]
{x -> t}

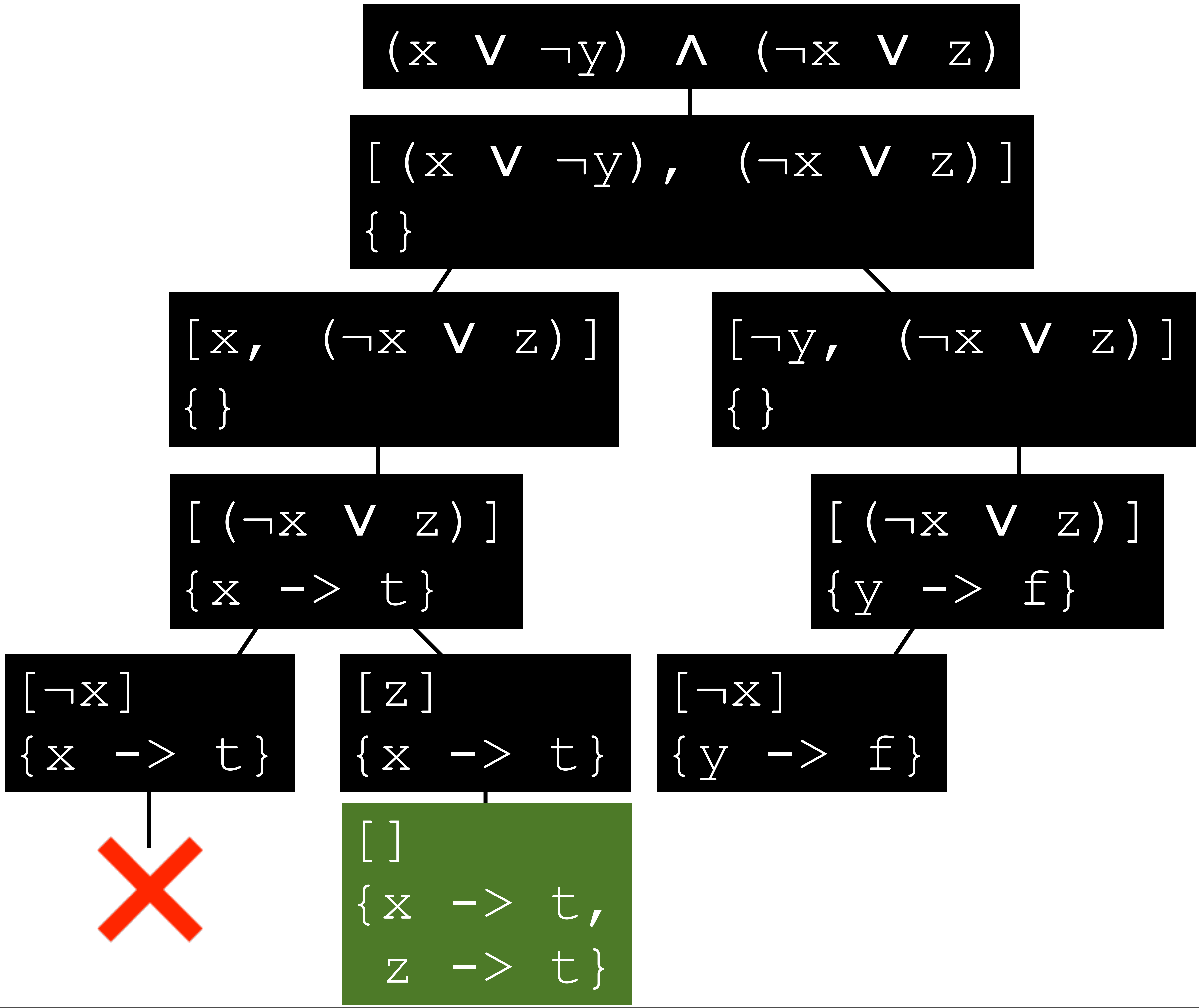

(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[¬x]
{x -> t}
[z]
{x -> t}

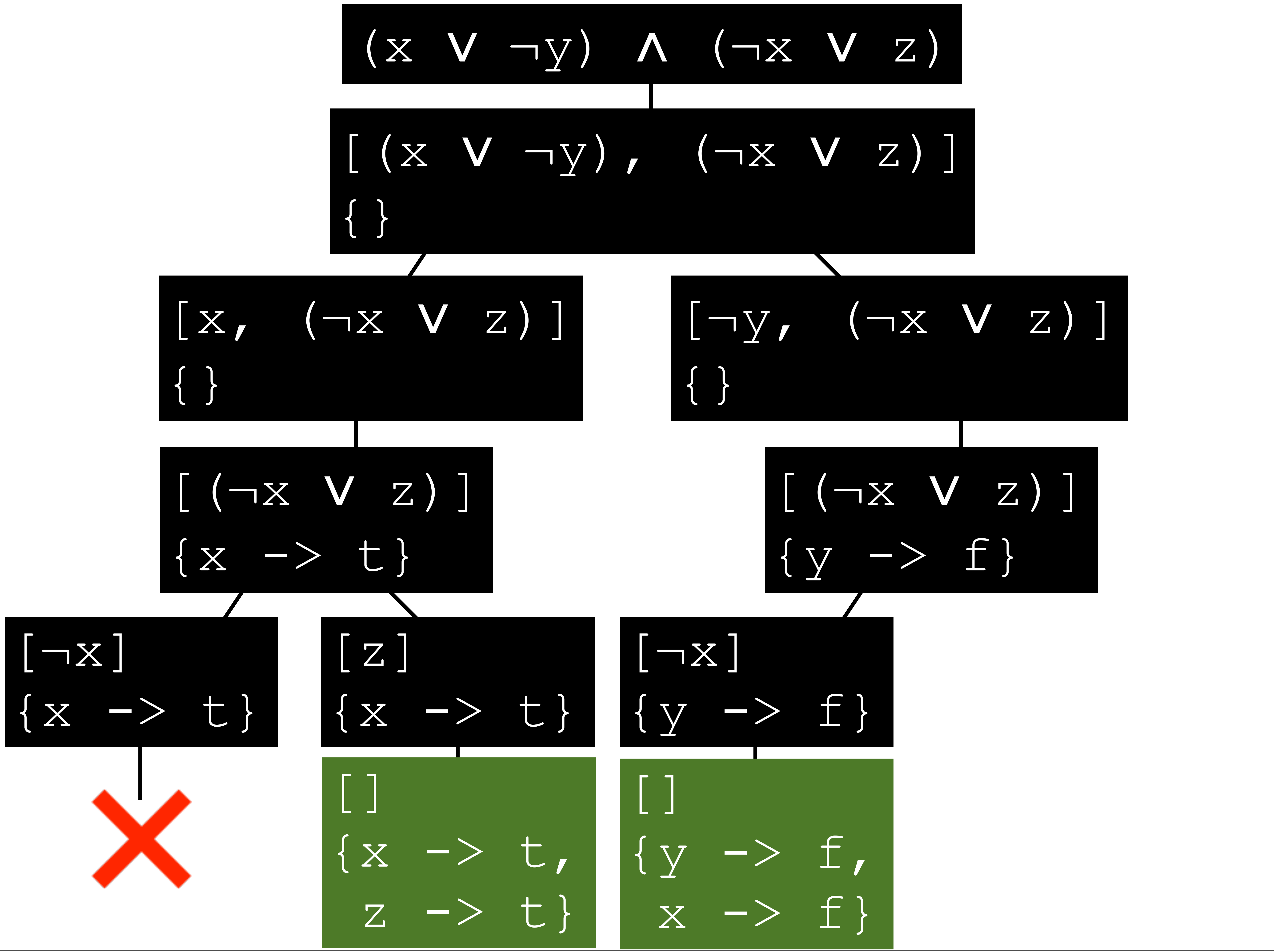

(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[¬x]
{x -> t}
[z]
{x -> t}
[]
{x -> t,
z -> t}

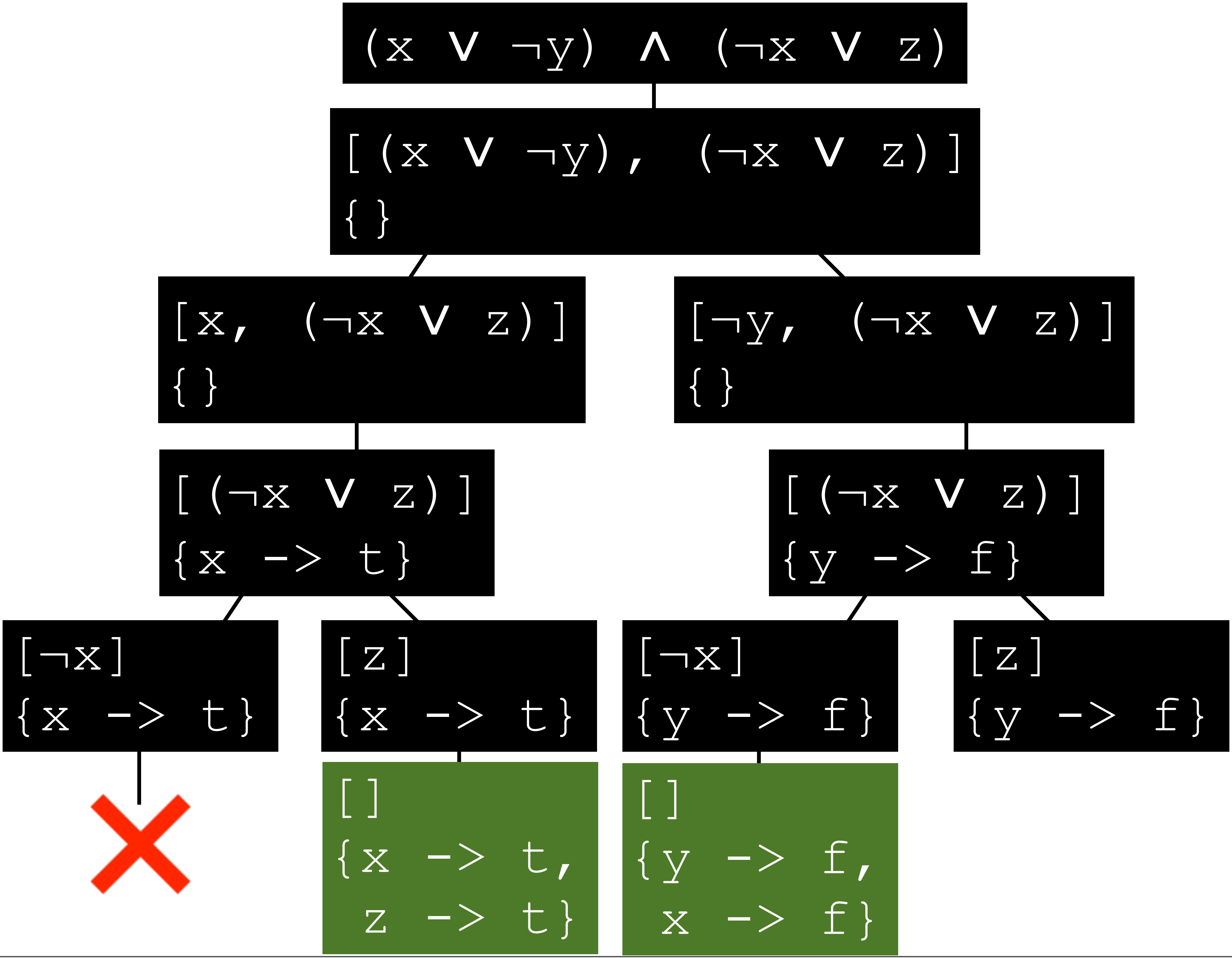

(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[¬y, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[¬x]
{x -> t}
[z]
{x -> t}
[]
{x -> t,
z -> t}

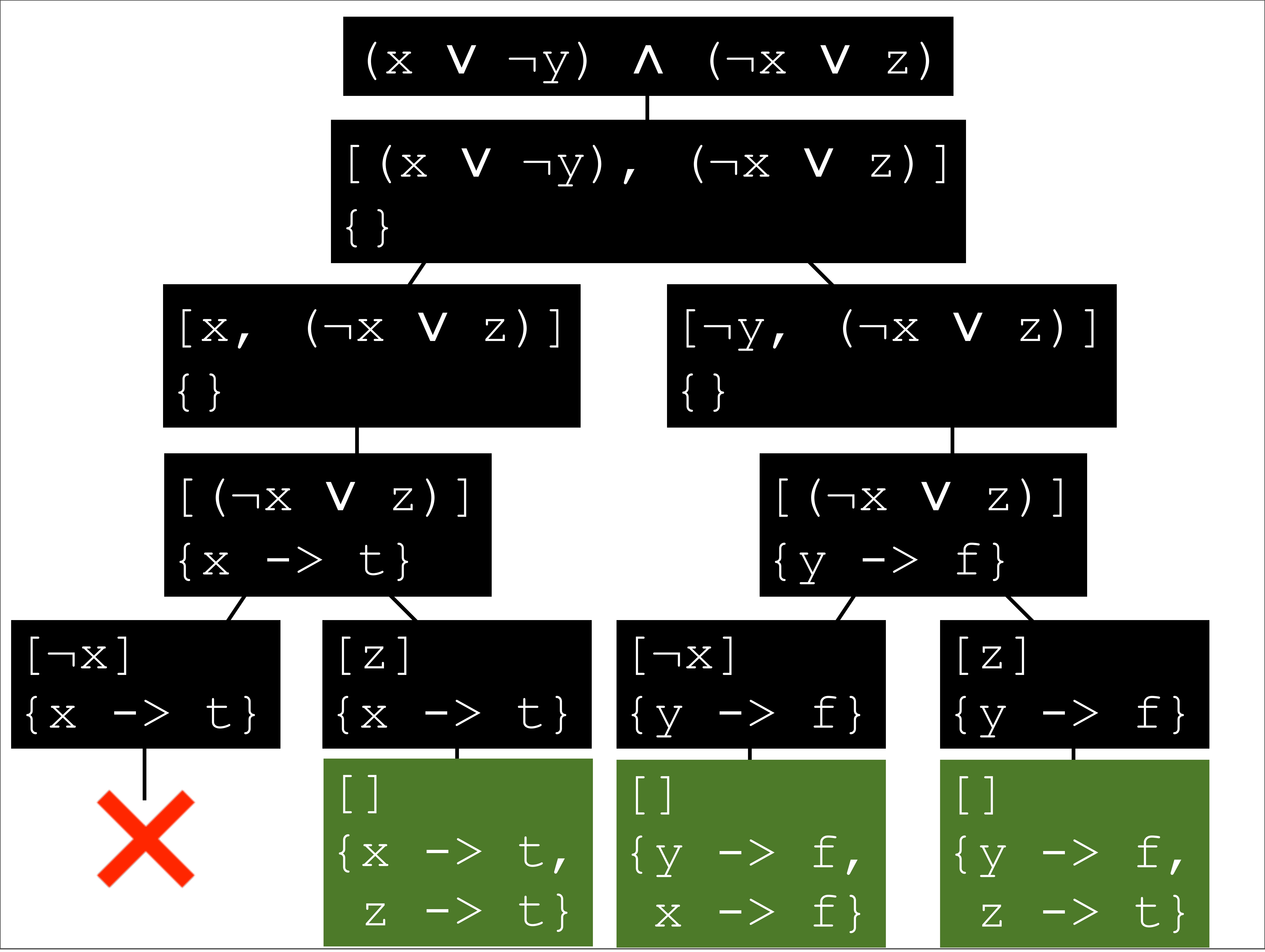


(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[¬y, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[(¬x ∨ z)]
{y -> f}
[¬x]
{x -> t}
[z]
{x -> t}
[]
{x -> t,
z -> t}


(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[¬y, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[(¬x ∨ z)]
{y -> f}
[¬x]
{x -> t}
[z]
{x -> t}
[¬x]
{y -> f}
[]
{x -> t,
z -> t}


(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[¬y, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[(¬x ∨ z)]
{y -> f}
[¬x]
{x -> t}
[z]
{x -> t}
[¬x]
{y -> f}
[]
{x -> t, z -> t}
[]
{y -> f, x -> f}


(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[¬y, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[(¬x ∨ z)]
{y -> f}
[¬x]
{x -> t}
[z]
{x -> t}
[¬x]
{y -> f}
[z]
{y -> f}
[]
{x -> t,
z -> t}
[]
{y -> f,
x -> f}


(x ∨ ¬y) ∧ (¬x ∨ z)
[(x ∨ ¬y), (¬x ∨ z)]
{}
[x, (¬x ∨ z)]
{}
[¬y, (¬x ∨ z)]
{}
[(¬x ∨ z)]
{x -> t}
[(¬x ∨ z)]
{y -> f}
[¬x]
{x -> t}
[z]
{x -> t}
[¬x]
{y -> f}
[z]
{y -> f}
[]
{x -> t, z -> t}
[]
{y -> f, x -> f}
[]
{y -> f, z -> t}