

# Nano-Pass Compiler Architecture

**Eelco Visser**



**CS4200 | Compiler Construction | November 18, 2021**

# Essentials of Compilation: An Incremental Approach in Racket

"book" — 2021/11/8 — 16:47 — page ii — #2

## Essentials of Compilation An Incremental Approach in Racket

Jeremy G. Siek

The MIT Press  
Cambridge, Massachusetts  
London, England

"book" — 2021/11/8 — 16:47 — page v — #5

## Contents

Preface	ix
<b>1 Preliminaries</b>	<b>1</b>
1.1 Abstract Syntax Trees	1
1.2 Grammars	3
1.3 Pattern Matching	5
1.4 Recursive Functions	6
1.5 Interpreters	6
1.6 Example Compiler: a Partial Evaluator	9
<b>2 Integers and Variables</b>	<b>11</b>
2.1 The $\mathcal{L}_{Var}$ Language	11
2.2 The $x86_{int}$ Assembly Language	16
2.3 Planning the trip to x86	19
2.4 Uniquify Variables	23
2.5 Remove Complex Operands	24
2.6 Explicate Control	26
2.7 Select Instructions	28
2.8 Assign Homes	29
2.9 Patch Instructions	30
2.10 Generate Prelude and Conclusion	30
2.11 Challenge: Partial Evaluator for $\mathcal{L}_{Var}$	31
<b>3 Register Allocation</b>	<b>33</b>
3.1 Registers and Calling Conventions	34
3.2 Liveness Analysis	36
3.3 Build the Interference Graph	39
3.4 Graph Coloring via Sudoku	41
3.5 Patch Instructions	47
3.6 Prelude and Conclusion	47
3.7 Challenge: Move Biasing	48
3.8 Further Reading	51
<b>4 Booleans and Conditionals</b>	<b>55</b>

"book" — 2021/11/8 — 16:47 — page vi — #6

vi Contents

4.1 The $\mathcal{L}_H$ Language	56
4.2 Type Checking $\mathcal{L}_H$ Programs	57
4.3 The $\mathcal{C}_H$ Intermediate Language	62
4.4 The $x86_H$ Language	62
4.5 Shrink the $\mathcal{L}_H$ Language	64
4.6 Uniquify Variables	65
4.7 Remove Complex Operands	65
4.8 Explicate Control	66
4.9 Select Instructions	72
4.10 Register Allocation	73
4.11 Patch Instructions	74
4.12 Challenge: Optimize Blocks and Remove Jumps	75
4.13 Further Reading	78
<b>5 Loops and Dataflow Analysis</b>	<b>79</b>
5.1 The $\mathcal{L}_{While}$ Language	80
5.2 Cyclic Control Flow and Dataflow Analysis	81
5.3 Mutable Variables & Remove Complex Operands	86
5.4 Uncover <code>get!</code>	87
5.5 Remove Complex Operands	88
5.6 Explicate Control and $\mathcal{C}_O$	89
5.7 Select Instructions	90
5.8 Register Allocation	90
<b>6 Tuples and Garbage Collection</b>	<b>93</b>
6.1 The $\mathcal{L}_{Tup}$ Language	93
6.2 Garbage Collection	96
6.3 Shrink	104
6.4 Expose Allocation	104
6.5 Remove Complex Operands	105
6.6 Explicate Control and the $\mathcal{C}_{Tup}$ language	105
6.7 Select Instructions and the $x86_{Global}$ Language	106
6.8 Register Allocation	110
6.9 Prelude and Conclusion	110
6.10 Challenge: Simple Structures	113
6.11 Challenge: Arrays	115
6.12 Challenge: Generational Collection	119
<b>7 Functions</b>	<b>123</b>
7.1 The $\mathcal{L}_{Fun}$ Language	123
7.2 Functions in x86	125
7.3 Shrink $\mathcal{L}_{Fun}$	130
7.4 Reveal Functions and the $\mathcal{L}_{FunRef}$ language	131
7.5 Limit Functions	131
7.6 Remove Complex Operands	132

# From ChocoPy to RISC-V Assembly Language

```
x : int = 1
y : int = 2
x + y
```

```
.globl main
main:
    lui a0, 8192
    add s11, s11, a0
    jal heap.init
    mv gp, a0
    mv s10, gp
    add s11, s10, s11
    mv ra, zero
    mv fp, zero
    mv fp, zero
    mv ra, zero
    addi sp, sp, -@..main.size
    sw ra, @..main.size-4(sp)
    sw fp, @..main.size-8(sp)
    addi fp, sp, @..main.size
    jal initchars
    lw a0, $x
    sw a0, -12(fp)
    lw a0, $y
    lw t0, -12(fp)
    add a0, t0, a0
    .equiv @..main.size, 16
label_0:
    li a0, 10
    ecall

# Initialize heap size (in multiples of 4KB)
# Save heap size
# Call heap.init routine
# Initialize heap pointer
# Set beginning of heap
# Set end of heap (= start of heap + heap size)
# No normal return from main program.
# No preceding frame.
# Top saved FP is 0.
# No function return from top level.
# Reserve space for stack frame.
# return address
# control link
# New fp is at old SP.
# Initialize one-character strings.
# Load global: x
# Push on stack slot 3
# Load global: y
# Pop stack slot 3
# Operator +

# End of program
# Code for ecall: exit
```

# Motivation for Passes

## Operators

- x86 arithmetic instructions typically have two arguments and update the second argument in place. In contrast, LVar arithmetic operations take two arguments and produce a new value. An x86 instruction may have at most one memory-accessing argument. Furthermore, some x86 instructions place special restrictions on their arguments.

## Nested

- An argument of an LVar operator can be a deeply-nested expression, whereas x86 instructions restrict their arguments to be integer constants, registers, and memory locations.

## Order of execution

- The order of execution in x86 is explicit in the syntax: a sequence of instructions and jumps to labeled positions, whereas in LVar the order of evaluation is a left-to-right depth-first traversal of the abstract syntax tree.

## Limited Registers

- A program in LVar can have any number of variables whereas x86 has 16 registers and the procedure call stack.

## Shadowing

- Variables in LVar can shadow other variables with the same name. In x86, registers have unique names and memory locations have unique addresses.

# Passes

## uniquify

- deals with the shadowing of variables by renaming every variable to a unique name.

## remove\_complex\_operands

- ensures that each subexpression of a primitive operation or function call is a variable or integer, that is, an atomic expression. We refer to non-atomic expressions as complex. This pass introduces temporary variables to hold the results of complex subexpressions.

## explicate\_control

- makes the execution order of the program explicit. It converts the abstract syntax tree representation into a control-flow graph in which each node contains a sequence of statements and the edges between nodes say which nodes contain jumps to other nodes.

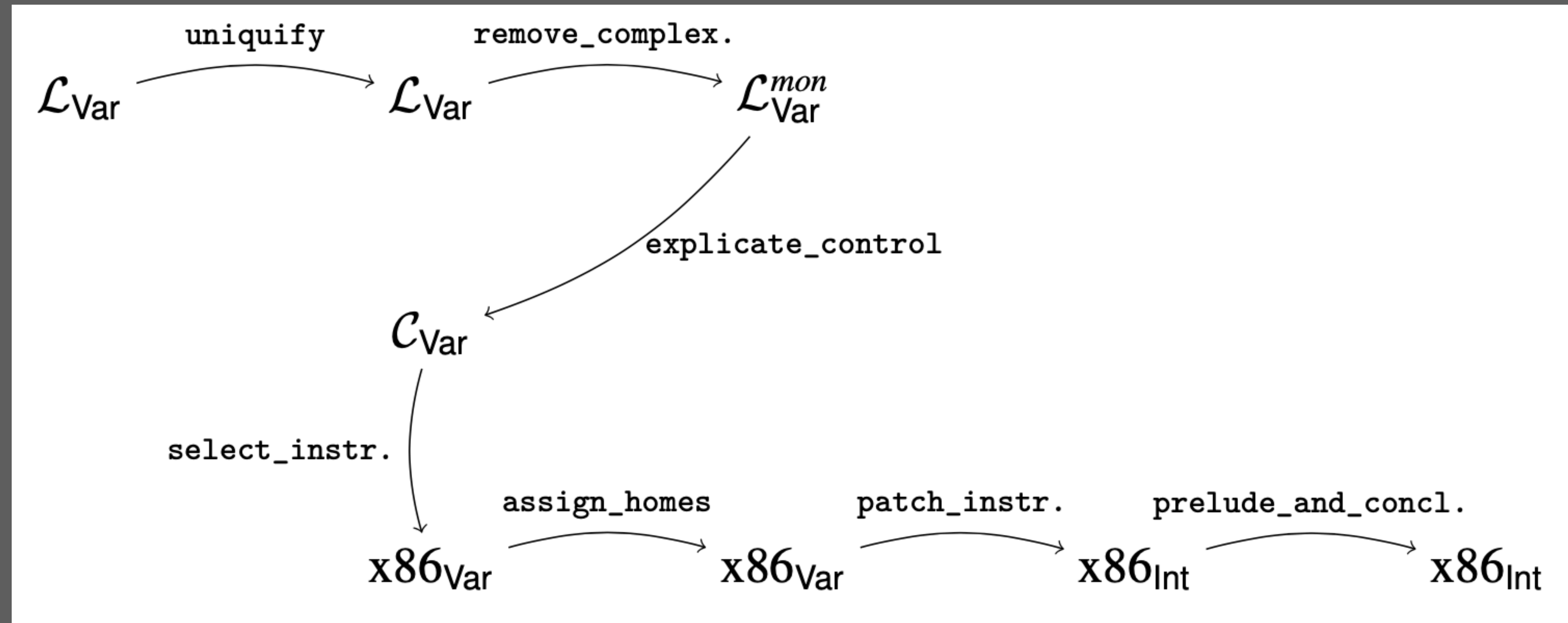
## select\_instructions

- handles the difference between LVar operations and x86 instructions. This pass converts each LVar operation to a short sequence of instructions that accomplishes the same task.

## assign\_homes

- replaces variables with registers or stack locations.

# Intermediate Languages

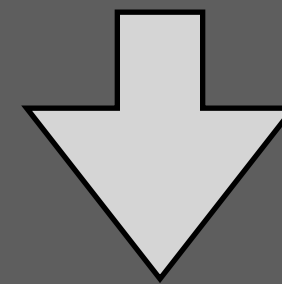


# uniquify

Does not seem necessary for ChocoPy yet; no shadowing allowed

# Remove Complex Operands

```
x : int = 2  
1 + x * 3
```



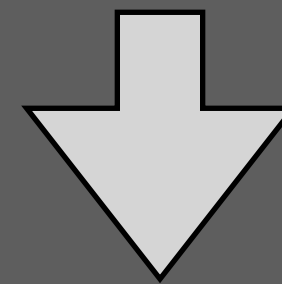
```
x : int = 2  
y : int = 0  
y = x * 3  
1 + y
```

Give every intermediate result a name



# Explicate Control: Translate to C-like language

```
x : int = 2  
y : int = 0  
y = x * 3  
1 + y
```



```
Main:  
var x : int = 2  
var y : int = 0  
y := x * 3  
return 1 + y
```

Translate to language with explicit instructions

# Select Instructions

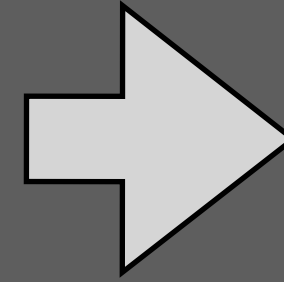
Main:

```
var x : int = 2
```

```
var y : int = 0
```

```
y := x * 3
```

```
return 1 + y
```



```
.globl main
```

```
main:
```

```
lw x, $x # Load global: x
```

```
li a, 3 # Load integer literal 3
```

```
mul y, x, a # Operator *
```

```
sw a0, $y, y # Assign global: y (using tmp register)
```

```
li b, 1 # Load integer literal 1
```

```
lw y, $y # Load global: y
```

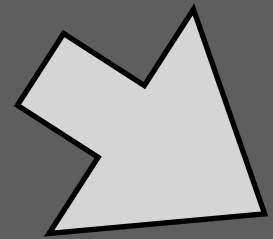
```
add c, b, y # Operator +
```

```
mv a0 c # return register
```

Translation to RISC-V instructions

# Assign Homes

```
.globl main
main:
    lw  x, $x      # Load global: x
    li  a, 3       # Load integer literal 3
    mul y, x, a    # Operator *
    sw  a0, $y, y  # Assign global: y (using tmp register)
    li  b, 1       # Load integer literal 1
    lw  y, $y      # Load global: y
    add c, b, y    # Operator +
    mv  a0 c       # return register
```



```
.globl main
main:
    lw  a0, $x      # Load global: x
    sw  a0, -12(fp) # Push on stack slot 3
    li  a0, 3       # Load integer literal 3
    lw  t0, -12(fp) # Pop stack slot 3
    mul a0, t0, a0  # Operator *
    sw  a0, $y, t0  # Assign global: y (using tmp register)
    li  a0, 1       # Load integer literal 1
    sw  a0, -12(fp) # Push on stack slot 3
    lw  a0, $y      # Load global: y
    lw  t0, -12(fp) # Pop stack slot 3
    add a0, t0, a0  # Operator +
```

Use stack and actual registers of the machine

# Composing Transformations with Strategy Combinators

**Eelco Visser**



**CS4200 | Compiler Construction | November 18, 2021**

**Rewriting =  
Matching & Building**

## **Atomic actions of program transformation**

1. Creating (building) terms from patterns
2. Matching terms against patterns

# Building and Matching Terms

## Atomic actions of program transformation

1. Creating (building) terms from patterns
2. Matching terms against patterns

## Build pattern

- Syntax:  $!p$
- Replace current term by instantiation of pattern  $p$
- A pattern is a term with *meta-variables*

```
stratego> :binding e
e is bound to Var("b")
stratego> !Plus(Var("a"), e)
Plus(Var("a"), Var("b"))
```

# Matching Terms

## Match pattern

- Syntax:  $?p$
- Match current term ( $t$ ) against pattern  $p$
- Succeed if there is a substitution  $\sigma$  such that  $\sigma(p) = t$

```
Plus(Var("a"), Int("3"))  
stratego> ?Plus(e, -)
```



# Matching Terms

## Match pattern

- Syntax:  $?p$
- Match current term ( $t$ ) against pattern  $p$
- Succeed if there is a substitution  $\sigma$  such that  $\sigma(p) = t$
- Wildcard  $_$  matches any term

```
Plus(Var("a"), Int("3"))  
stratego> ?Plus(e, _)
```

# Matching Terms

## Match pattern

- Syntax:  $?p$
- Match current term ( $t$ ) against pattern  $p$
- Succeed if there is a substitution  $\sigma$  such that  $\sigma(p) = t$
- Wildcard  $_$  matches any term
- Binds variables in  $p$  in the environment

```
Plus(Var("a"), Int("3"))  
stratego> ?Plus(e, _)  
stratego> :binding e  
e is bound to Var("a")
```

# Matching Terms

## Match pattern

- Syntax:  $?p$
- Match current term ( $t$ ) against pattern  $p$
- Succeed if there is a substitution  $\sigma$  such that  $\sigma(p) = t$
- Wildcard  $_$  matches any term
- Binds variables in  $p$  in the environment
- Fails if pattern does not match

```
Plus(Var("a"), Int("3"))  
stratego> ?Plus(e, _)  
stratego> :binding e  
e is bound to Var("a")  
stratego> ?Plus(Int(x), e2)  
command failed
```

# Recognizing Dubious Statements and Expressions

control-flow statement with empty statement

```
while ((c = inputStream.read()) != -1);  
    outputStream.write(c);
```

```
?While(_, Empty())
```

```
?If(_, Empty(), _)
```

```
?If(_, _, Empty())
```

equality operator with literal true operand; e.g. `e == true`

```
?Eq(_, Lit(Bool(True())))
```

```
?Eq(Lit(Bool(True())), _)
```

# Combining Match and Build

**Basic transformations are combinations of match and build**

**Combination requires**

1. Sequential composition of transformations
2. Restricting the scope of term variables

**Syntactic abstractions (sugar) for typical combinations**

1. Rewrite rules
2. Apply and match
3. Build and apply
4. Where
5. Conditional rewrite rules

## Sequential composition

- Syntax:  $s_1 ; s_2$
- Apply  $s_1$ , then  $s_2$
- Fails if either  $s_1$  or  $s_2$  fails
- Variable bindings are propagated

```
Plus(Var("a"), Int("3"))  
stratego> ?Plus(e1, e2); !Plus(e2, e1)  
Plus(Int("3"), Var("a"))
```

# Combining Match and Build

## Anonymous rewrite rule (sugar)

- Syntax:  $(p_1 \rightarrow p_2)$
- Match  $p_1$ , then build  $p_2$
- Equivalent to:  $?p_1; !p_2$

```
Plus(Var("a"), Int("3"))  
stratego> (Plus(e1, e2) -> Plus(e2, e1))  
Plus(Int("3"), Var("a"))
```

# Combining Match and Build

## Apply and match (sugar)

- Syntax:  $s \Rightarrow p$
- Apply  $s$ , then match  $p$
- Equivalent to:  $s; ?p$

## Build and apply (sugar)

- Syntax:  $\langle s \rangle p$
- Build  $p$ , then apply  $s$
- Equivalent to:  $!p; s$

```
stratego> <addS>("1","2") => x
"3"
stratego> :binding x
x is bound to "3"
```



# Combining Match and Build

## Assign (sugar)

- Syntax:  $p_2 := p_1$
- Build  $p_1$ , then match  $p_2$
- Equivalent to:  $!p_1; ?p_2$

```
stratego> x := <addS> ("1", "2")  
"3"  
stratego> :binding x  
x is bound to "3"
```

# Combining Match and Build

## Term variable scope

- Syntax:  $\{x_1, \dots, x_n : s\}$
- Restrict scope of variables  $x_1, \dots, x_n$  to  $s$

```
Plus(Var("a"), Int("3"))
stratego> (Plus(e1, e2) -> Plus(e2, e1))
Plus(Int("3"), Var("a"))
stratego> :binding e1
e1 is bound to Var("a")

stratego> {e3, e4 : (Plus(e3, e4) -> Plus(e4, e3))}
Plus(Var("a"), Int("3"))
stratego> :binding e3
e3 is not bound to a term
```

# Combining Match and Build

## Where (sugar)

- Syntax: `where(s)`
- Test and compute variable bindings
- Equivalent to:  $\{x: ?x; s; !x\}$   
for some fresh variable  $x$

```
Plus(Int("14"), Int("3"))
stratego> where(?Plus(Int(i), Int(j))); <addS>(i, j) => k)
Plus(Int("14"), Int("3"))
stratego> :binding i
i is bound to "14"
stratego> :binding k
k is bound to "17"
```

# Combining Match and Build

## Conditional rewrite rules (sugar)

- Syntax:  $(p_1 \rightarrow p_2 \text{ where } s)$
- Rewrite rule with condition  $s$
- Equivalent to:  $(?p_1; \text{where}(s); !p_2)$

```
Plus(Int("14"), Int("3"))
```

```
> (Plus(Int(i), Int(j)) -> Int(k) where <addS>(i, j) => k)
```

```
Int("17")
```

# Combining Match and Build

## Term Wrap

- Syntax:  $!p[\langle s \rangle]$
- Strategy application in pattern to current subterm
- Equivalent to:  $\{x: \text{where}(s \Rightarrow x); !p[x]\}$   
for some fresh variable  $x$

```
3
stratego> !(<id>, <id>)
(3, 3)
stratego> !(<Fst; inc>, <Snd>)
(4, 3)
```

```
"foobar"
stratego> !Call(<id>, [])
Call("foobar", [])
```

# Combining Match and Build

## Term Project

- Syntax:  $?p[\langle s \rangle]$
- Strategy application in pattern match
- Equivalent to:  $\{x: ?p[x]; \langle s \rangle x\}$   
for some fresh variable  $x$

```
[1,2,3]
stratego> ?[_|<id>]
[2,3]
```

```
Block([ExprStm(PostIncr(ExprName(Id("x")))),Return(None)])
stratego> ?Block(<length>)
2
```

# Parameterized Rewrite Rules

# Parameterized Rewrite Rules

$f(x, y | a, b): lhs \rightarrow rhs$

- strategy or rule parameters  $x, y$
- term parameters  $a, b$
- no matching

$f(| a, b): lhs \rightarrow rhs$

- optional strategy parameters

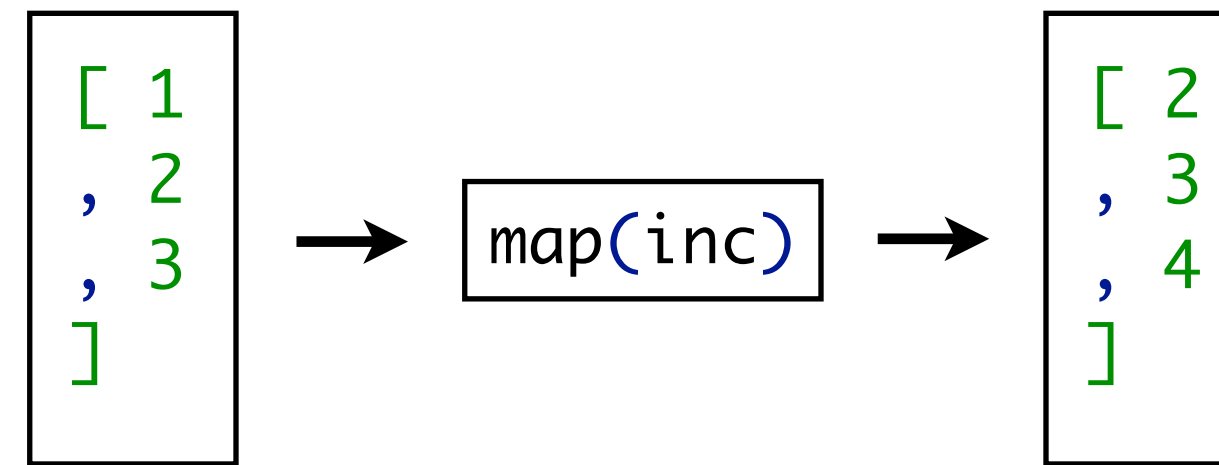
$f(x, y): lhs \rightarrow rhs$

- optional term parameters

$f: lhs \rightarrow rhs$

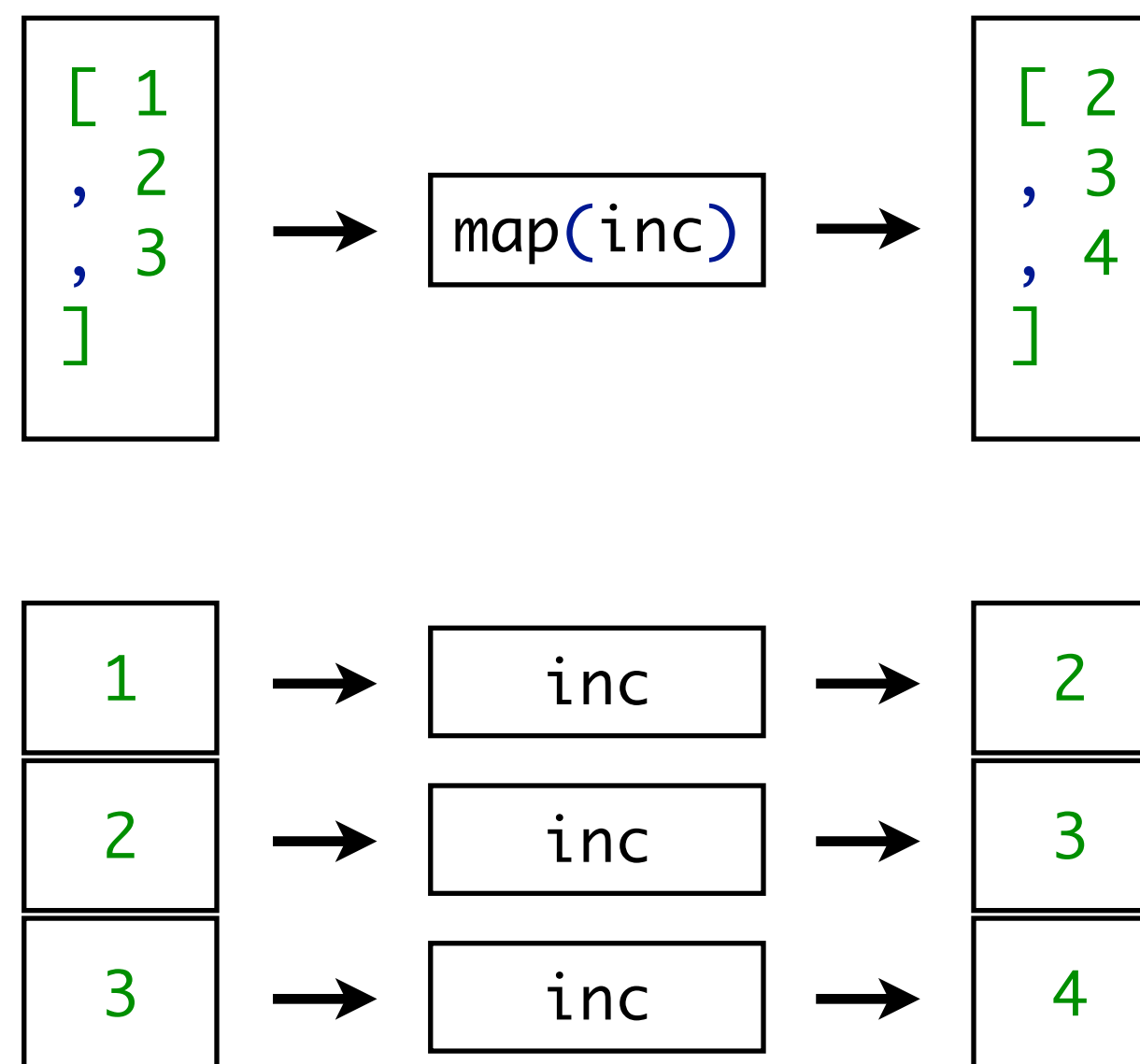


# Parameterized Rewrite Rules: Map



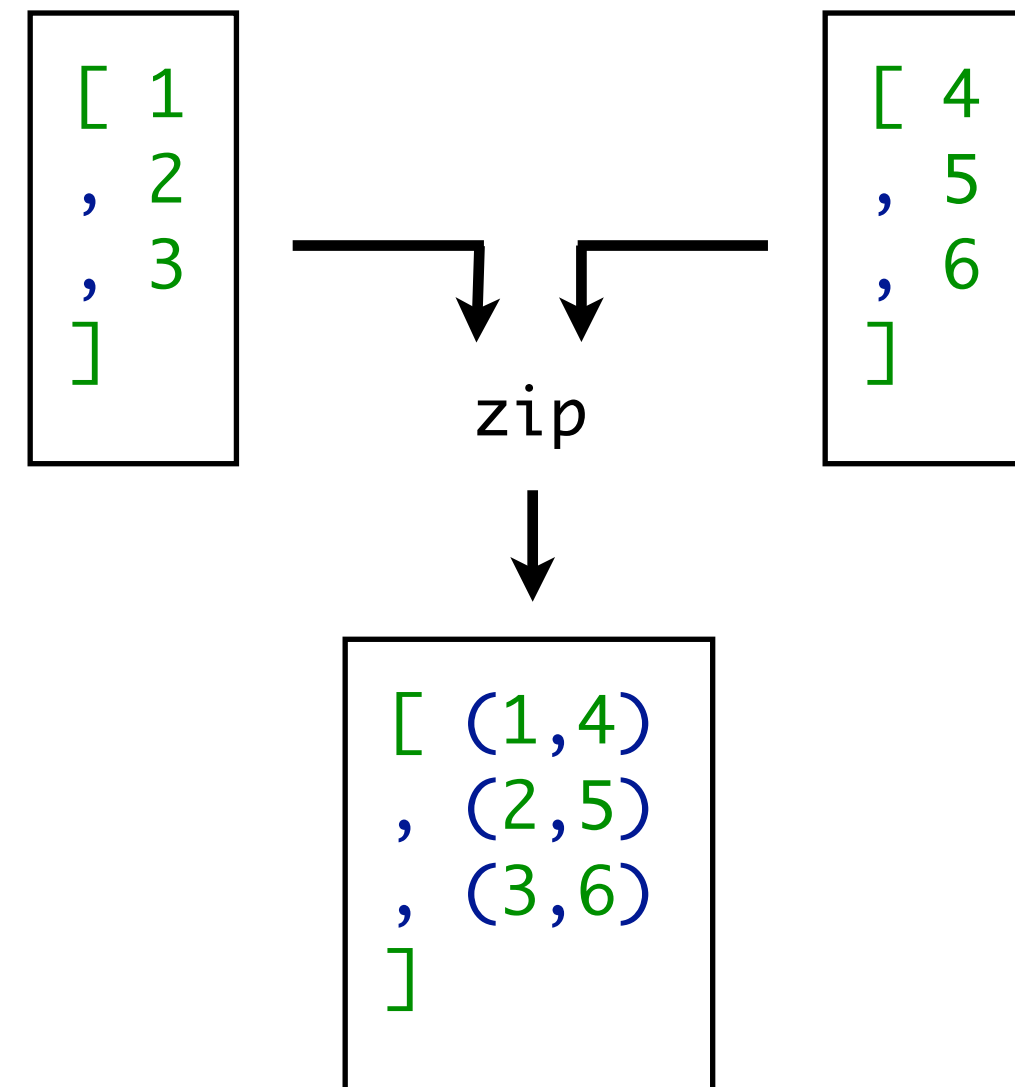
```
map(s): [] -> []  
map(s): [x | xs] -> [<s> x | <map(s)> xs]
```

# Parameterized Rewrite Rules: Map



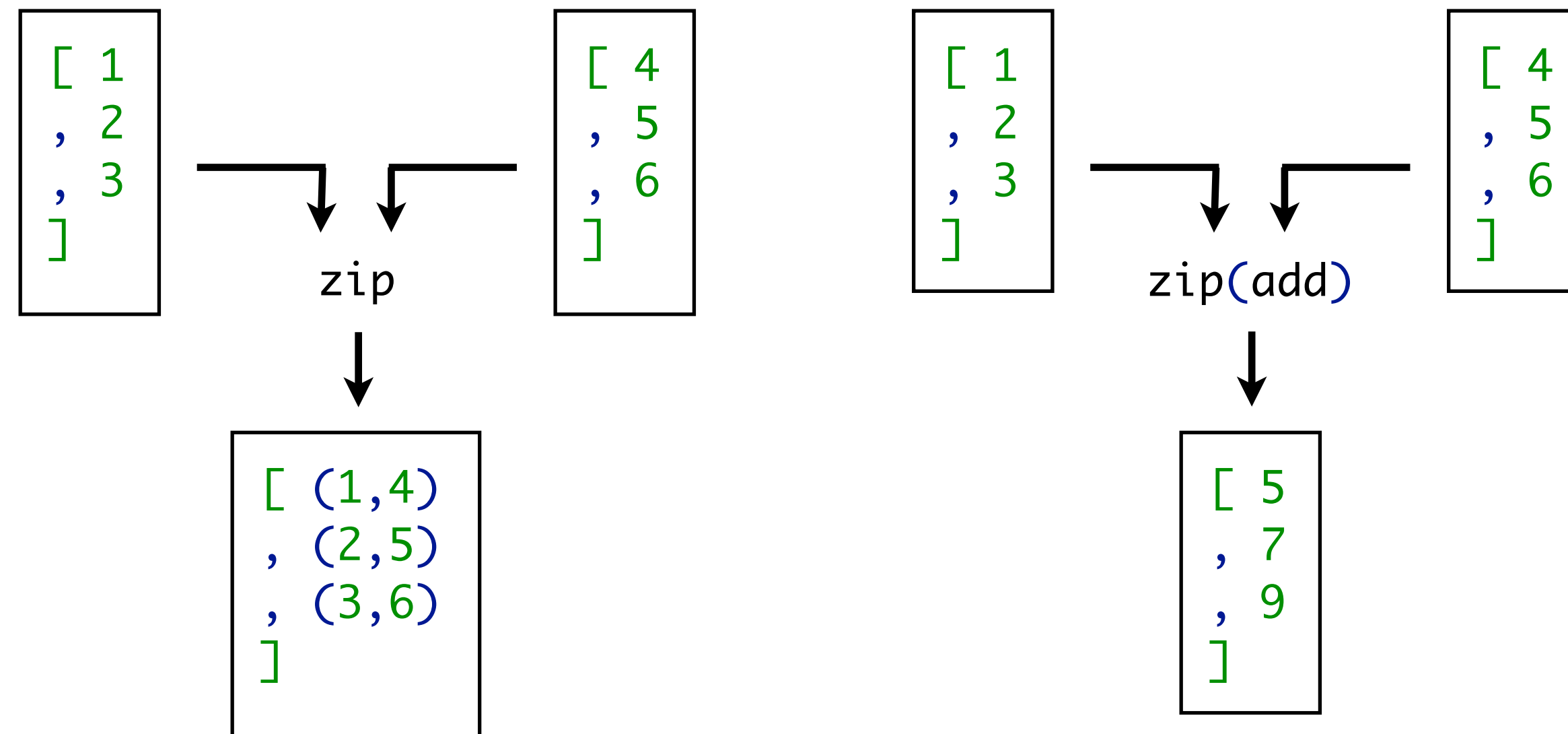
```
map(s): [] -> []  
map(s): [x | xs] -> [<s> x | <map(s)> xs]
```

# Parameterized Rewrite Rules: Zip



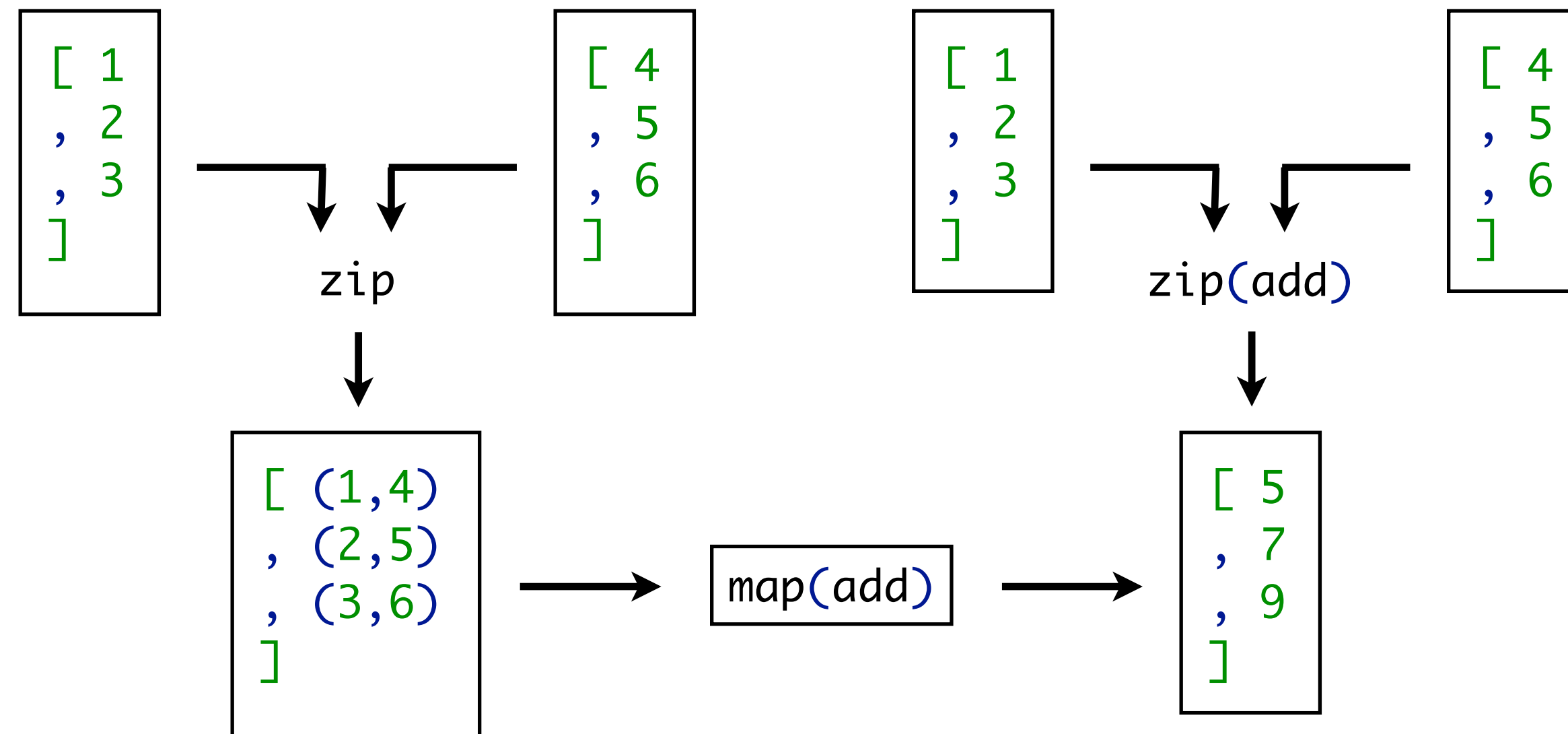
```
zip(s): ([],[]) -> []  
zip(s): ([x|xs],[y|ys]) -> [<s> (x,y) | <zip(s)> (xs,ys)]  
  
zip = zip(id)
```

# Parameterized Rewrite Rules: Zip



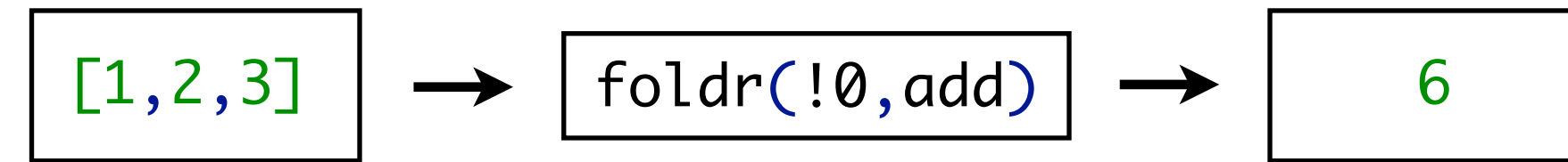
```
zip(s): ([],[]) -> []  
zip(s): ([x|xs],[y|ys]) -> [<s> (x,y) | <zip(s)> (xs,ys)]  
  
zip = zip(id)
```

# Parameterized Rewrite Rules: Zip



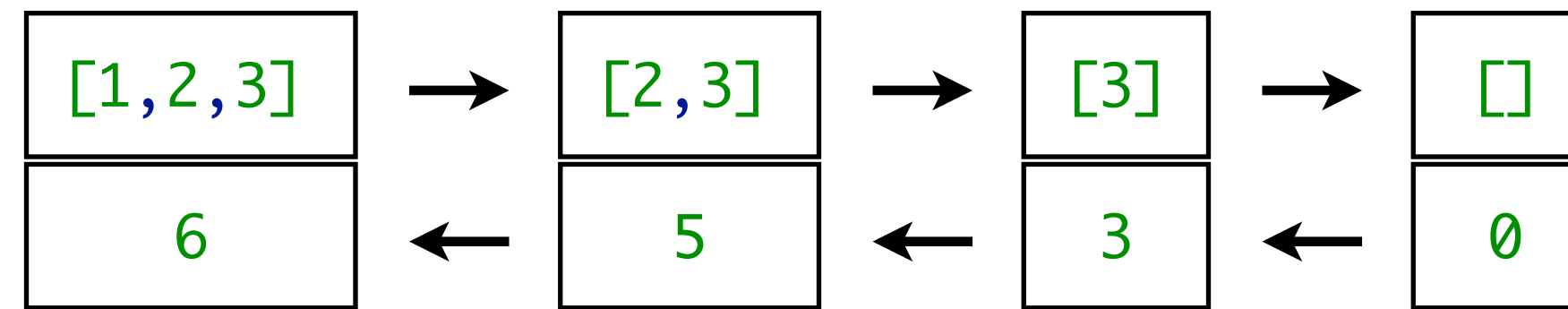
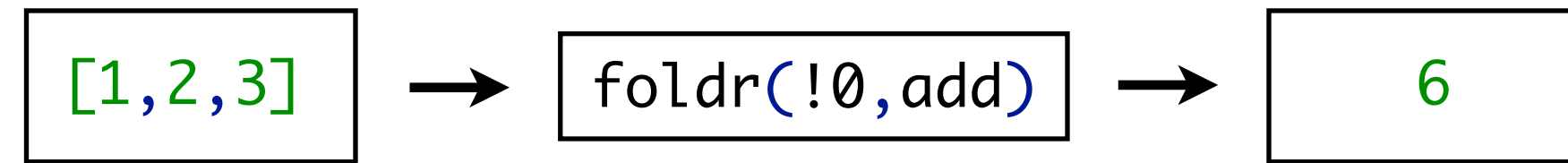
```
zip(s): ([],[]) -> []  
zip(s): ([x|xs],[y|ys]) -> [<s> (x,y) | <zip(s)> (xs,ys)]  
  
zip = zip(id)
```

# Parameterized Rewrite Rules: Fold



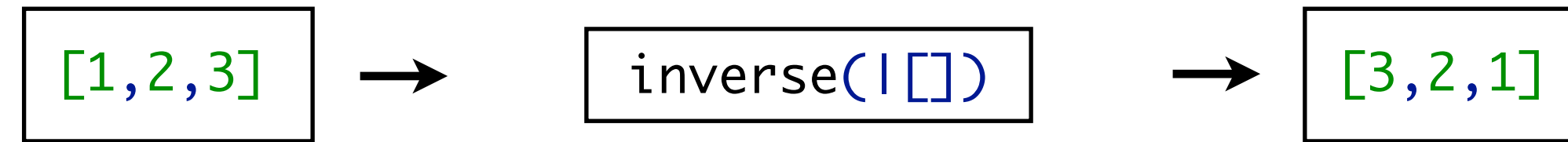
```
foldr(s1,s2): []      -> <s1>  
foldr(s1,s2): [x|xs] -> <s2> (x,<foldr(s1,s2)> xs)
```

# Parameterized Rewrite Rules: Fold



```
foldr(s1,s2): []      -> <s1>  
foldr(s1,s2): [x|xs] -> <s2> (x,<foldr(s1,s2)> xs)
```

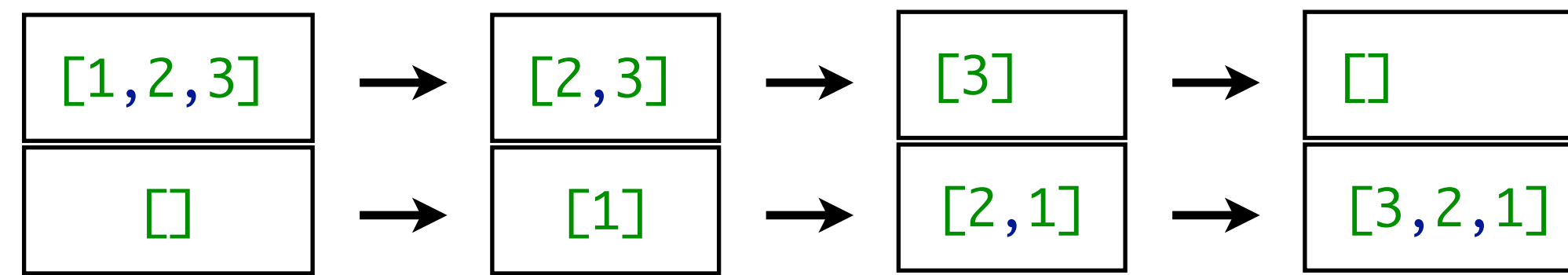
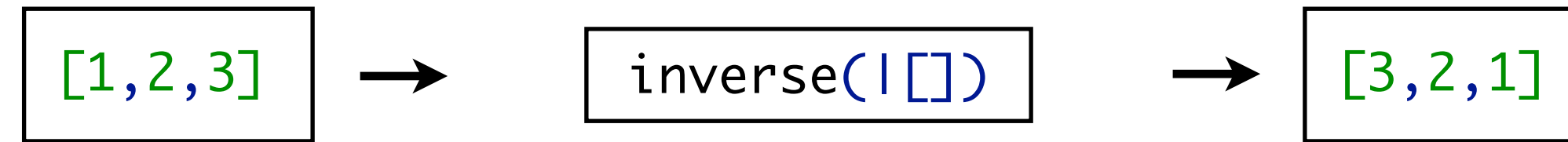
# Parameterized Rewrite Rules: Inverse



```
inverse(lis): [] -> is  
inverse(lis): [x|xs] -> <inverse(l[x|is])> xs
```



# Parameterized Rewrite Rules: Inverse



```
inverse(lis): [] -> is  
inverse(lis): [x|xs] -> <inverse(l[x|lis])> xs
```

# Traversal Combinators

# Realizing Term Traversal

## Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies

## Many ways to traverse a tree

- Bottom-up
- Top-down
- Innermost
- ...

**What are the primitives of traversal?**

## One-level traversal operators

- Apply a strategy to one or more direct subterms

## Congruence: data-type specific traversal

- Apply a different strategy to each argument of a specific constructor

## Generic traversal

- All: apply to all direct subterms
- One: apply to one direct subterm
- Some: apply to as many direct subterms as possible, and at least one

# Congruence Operators

## Congruence operator: data-type specific traversal

- Syntax:  $c(s_1, \dots, s_n)$  for each  $n$ -ary constructor  $c$
- Apply strategies to direct sub-terms of a  $c$  term

```
Plus(Int("14"), Int("3"))  
stratego> Plus(!Var("a"), id)  
Plus(Var("a"), Int("3"))
```

```
map(s) = [] + [s | map(s)]
```

```
fetch(s) = [s | id] <+ [id | fetch(s)]
```

```
filter(s) =
```

```
  [] + ([s | filter(s)] <+ ?[_|<id>]; filter(s))
```

Data-type specific traversal requires tedious enumeration of cases

Even if traversal behaviour is uniform

Generic traversal allows concise specification of default traversals

## Visiting all subterms

- Syntax: `all(s)`
- Apply strategy `s` to all direct sub-terms

```
Plus(Int("14"), Int("3"))  
stratego> all(!Var("a"))  
Plus(Var("a"), Var("a"))
```

## Visiting all subterms

- Syntax: `all(s)`
- Apply strategy `s` to all direct sub-terms

```
Plus(Int("14"), Int("3"))  
stratego> all(!Var("a"))  
Plus(Var("a"), Var("a"))
```

```
bottomup(s) = all(bottomup(s)); s  
topdown(s)  = s; all(topdown(s))  
downup(s)   = s; all(downup(s)); s  
alltd(s)    = s <+ all(alltd(s))
```



# Generic Traversal

## Visiting all subterms

- Syntax: `all(s)`
- Apply strategy `s` to all direct sub-terms

```
Plus(Int("14"), Int("3"))  
stratego> all(!Var("a"))  
Plus(Var("a"), Var("a"))
```

```
bottomup(s) = all(bottomup(s)); s  
topdown(s)  = s; all(topdown(s))  
downup(s)   = s; all(downup(s)); s  
alltd(s)    = s <+ all(alltd(s))
```

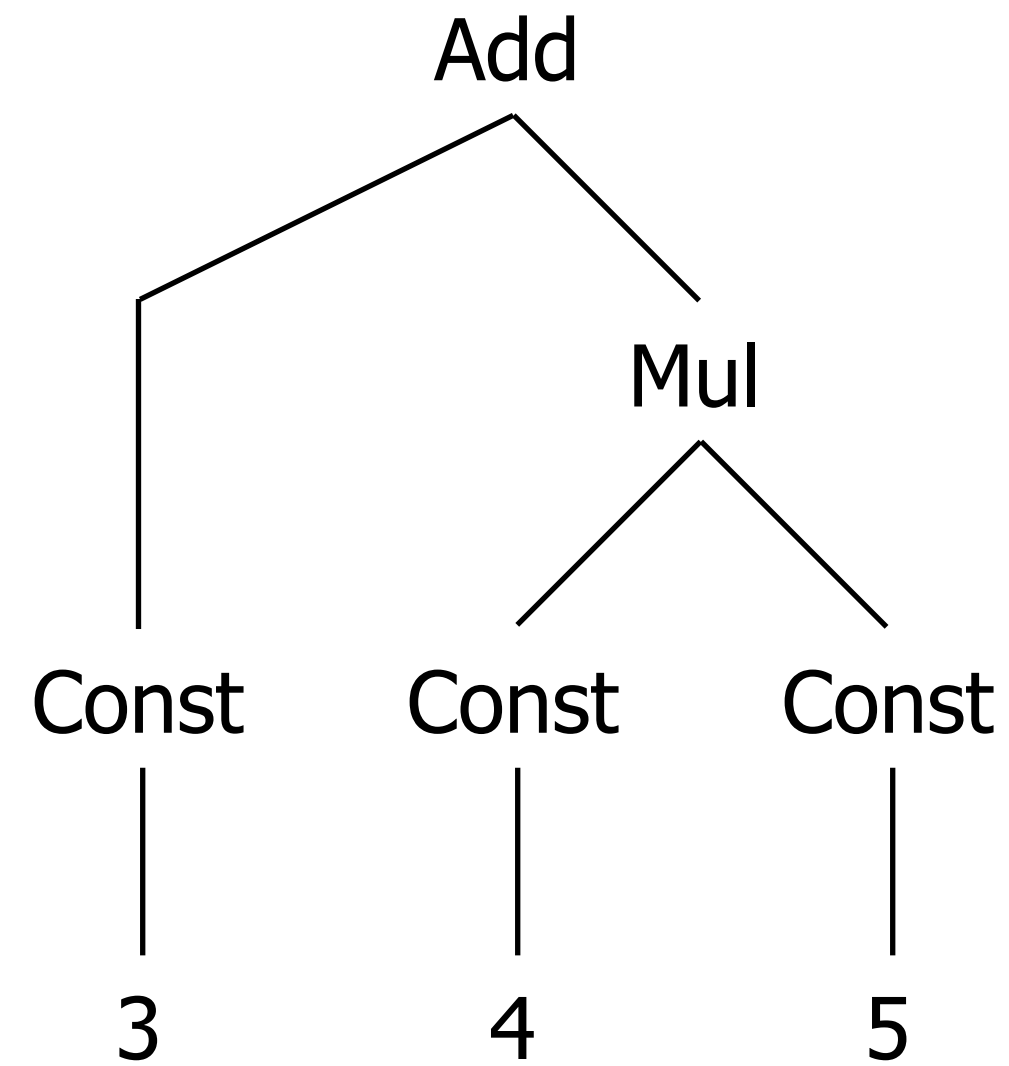
```
const-fold =  
  bottomup(try(EvalBinOp <+ EvalCall <+ EvalIf))
```

# Traversal: Topdown

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(switch)
```

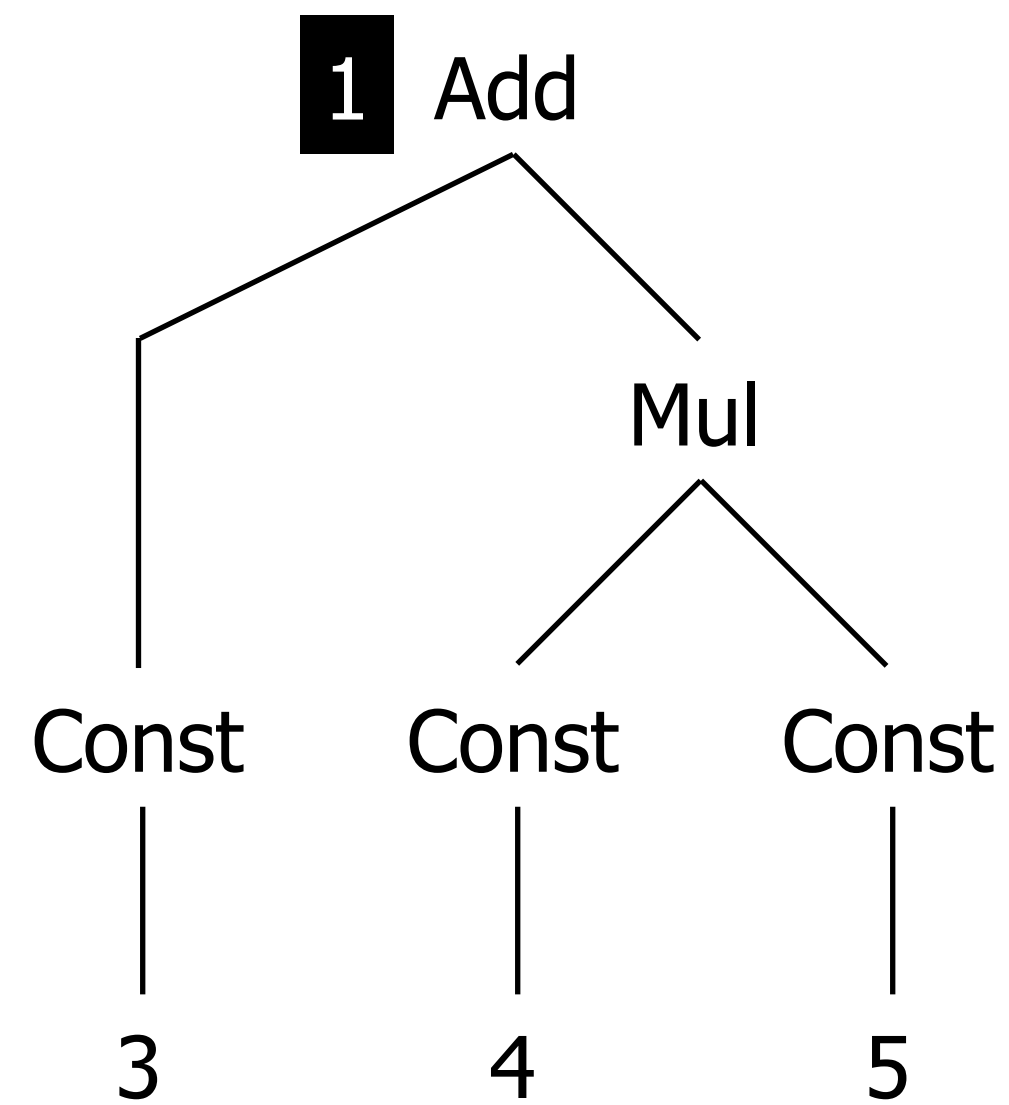


# Traversal: Topdown

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(switch)
```

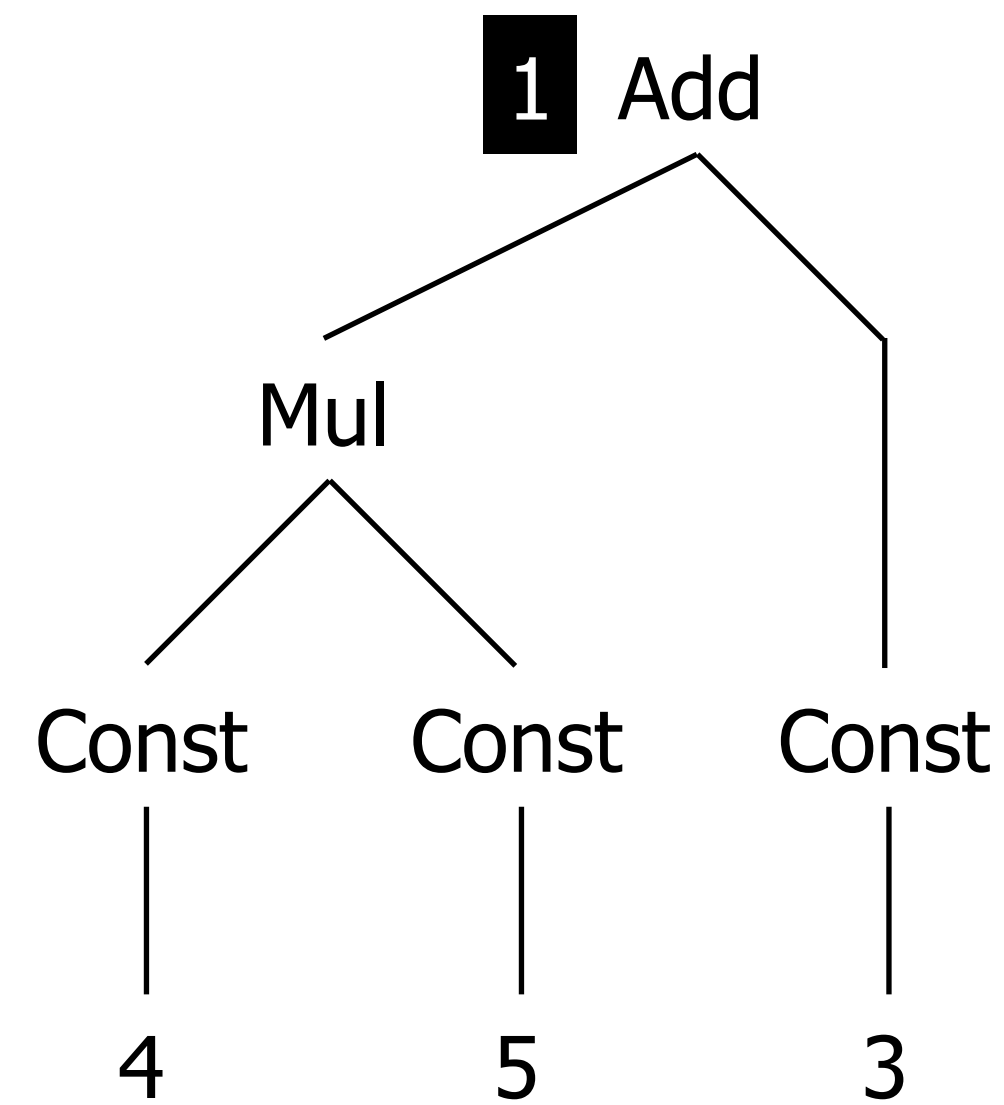


# Traversal: Topdown

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(switch)
```

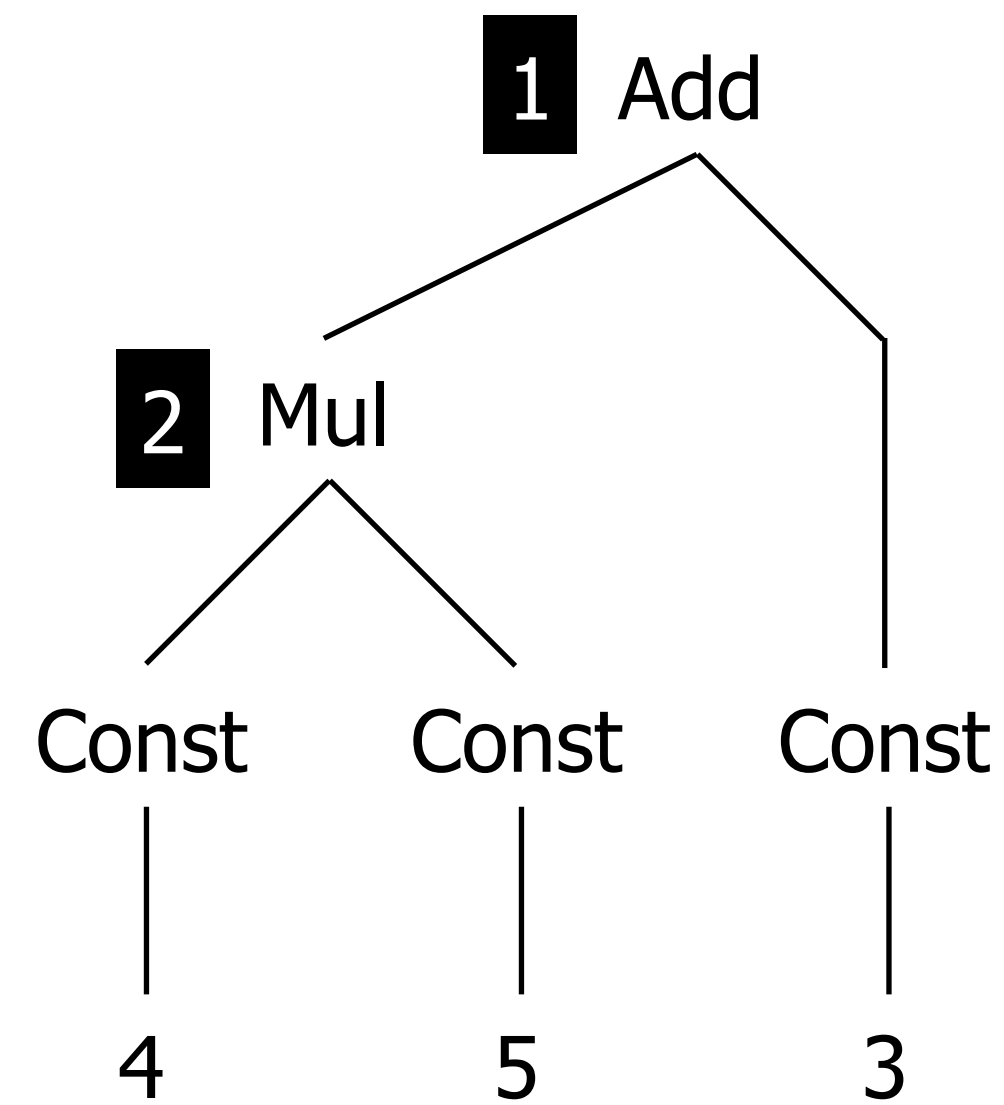


# Traversal: Topdown

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(switch)
```

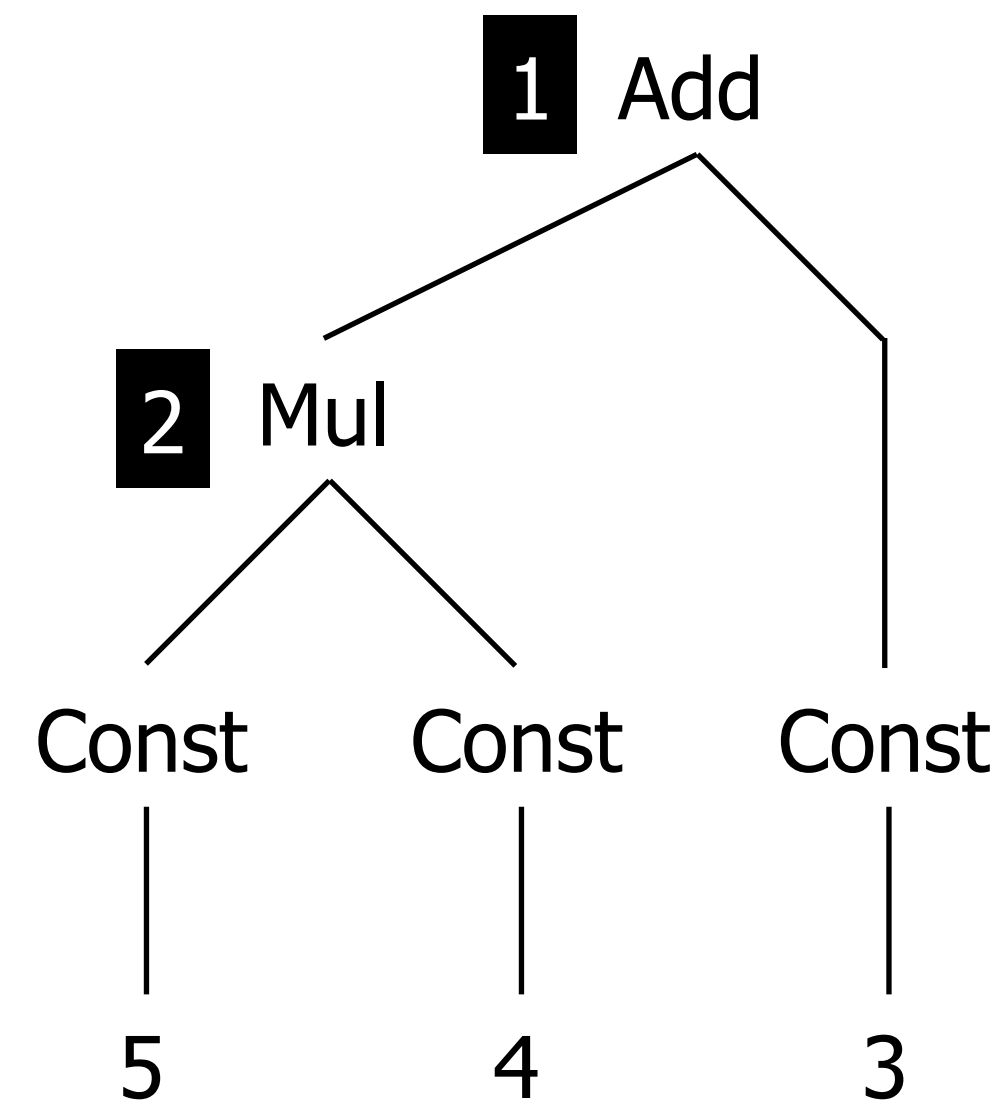


# Traversal: Topdown

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(switch)
```

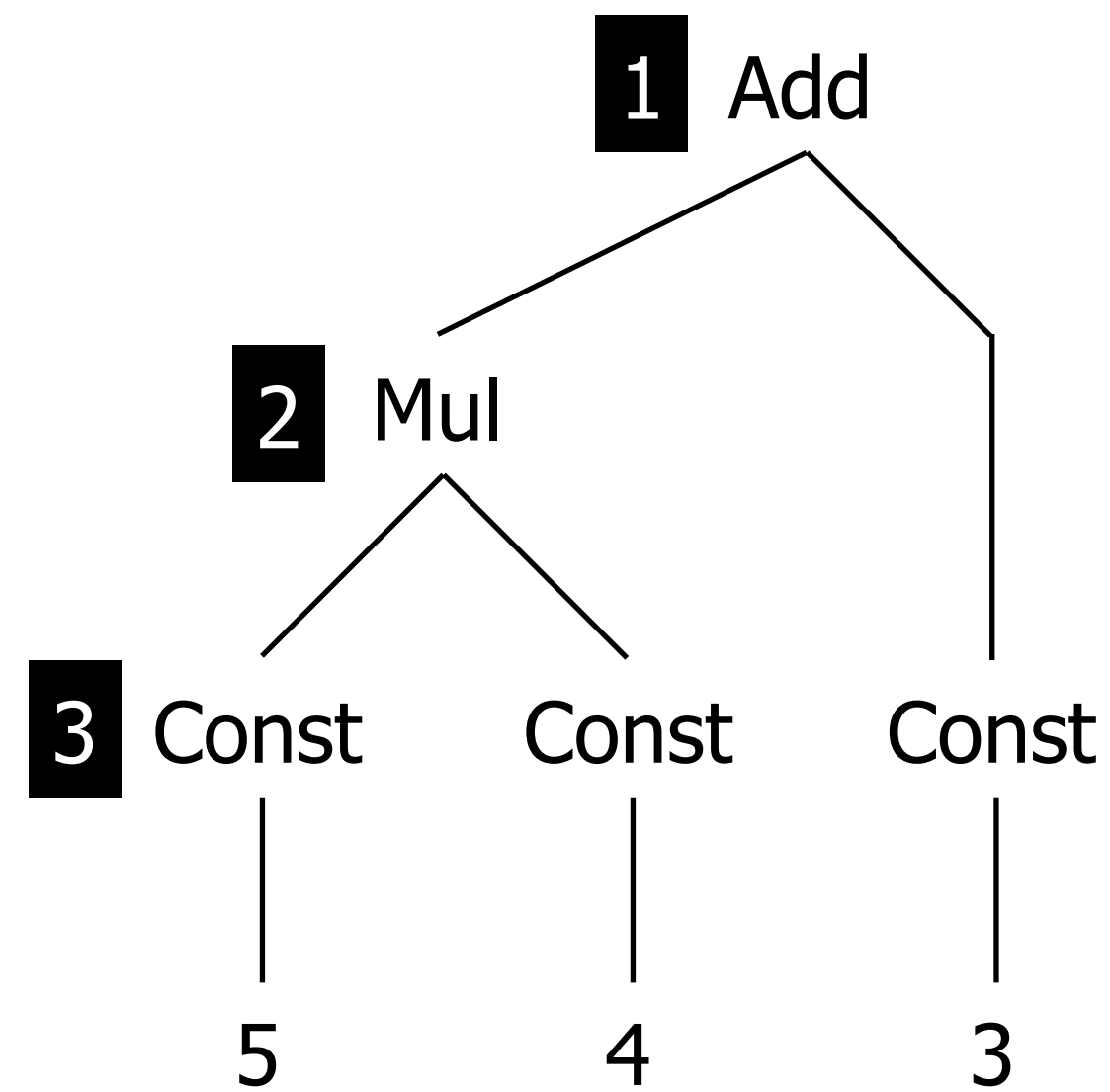


# Traversal: Topdown

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(switch)
```

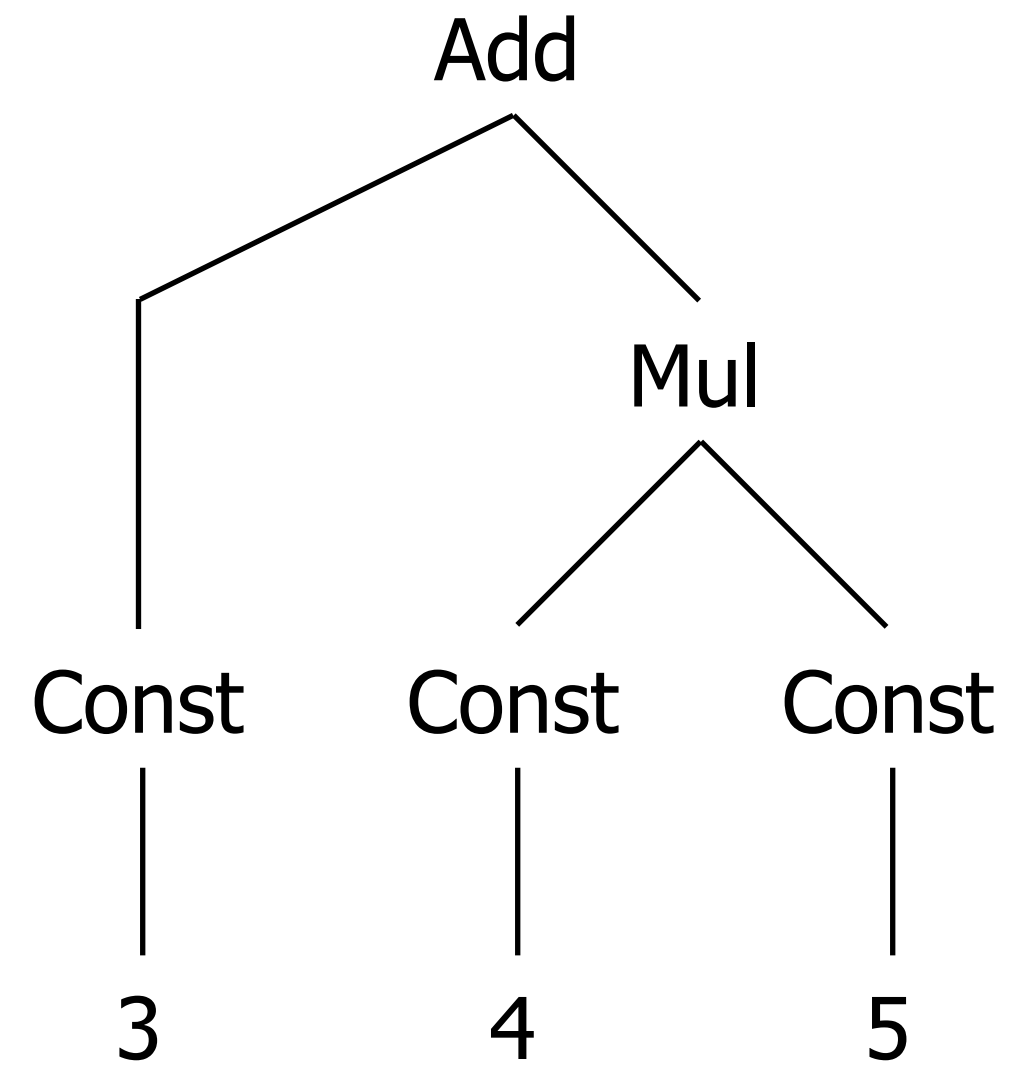


# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(try(switch))
```





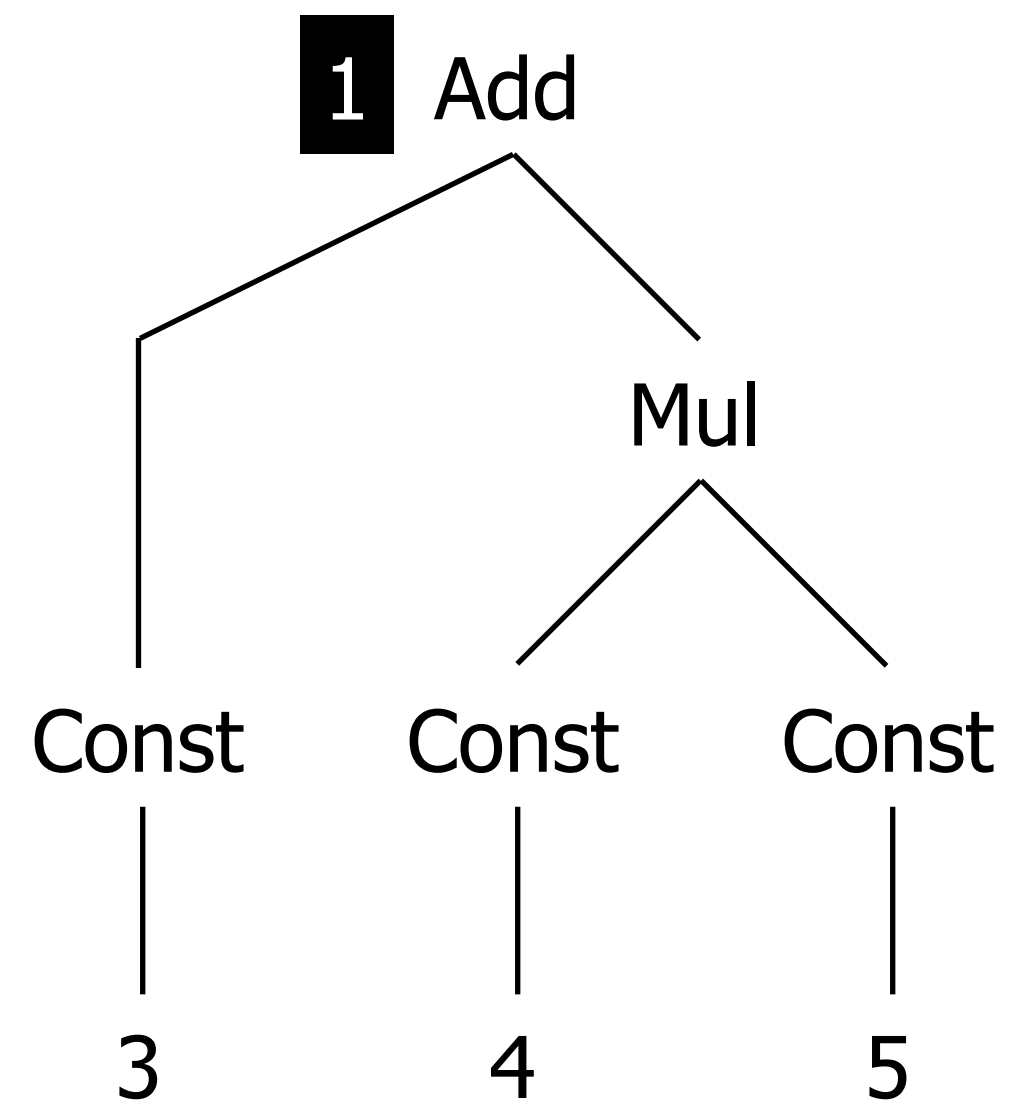
# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
topdown(s) = s ; all(topdown(s))
```

```
topdown(try(switch))
```



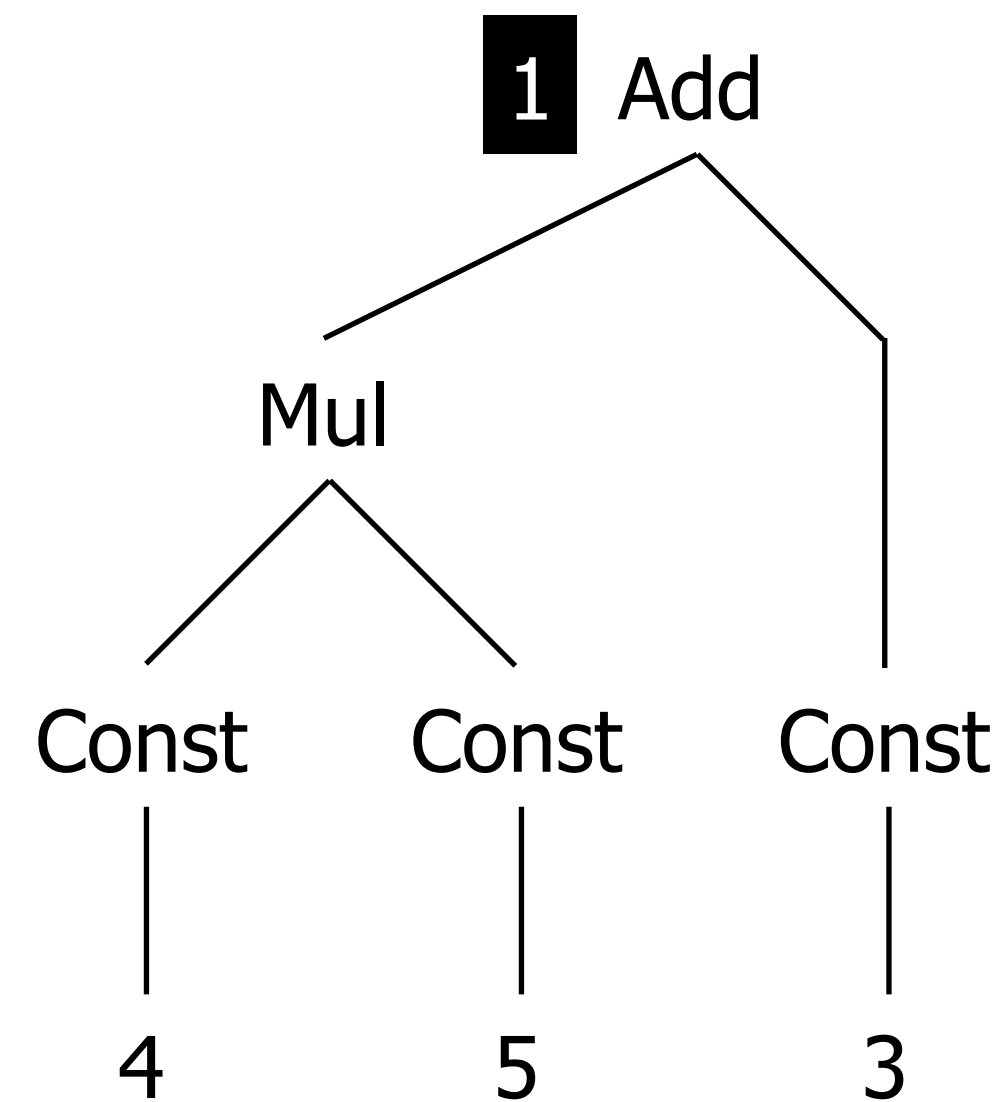
# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
topdown(s) = s ; all(topdown(s))
```

```
topdown(try(switch))
```



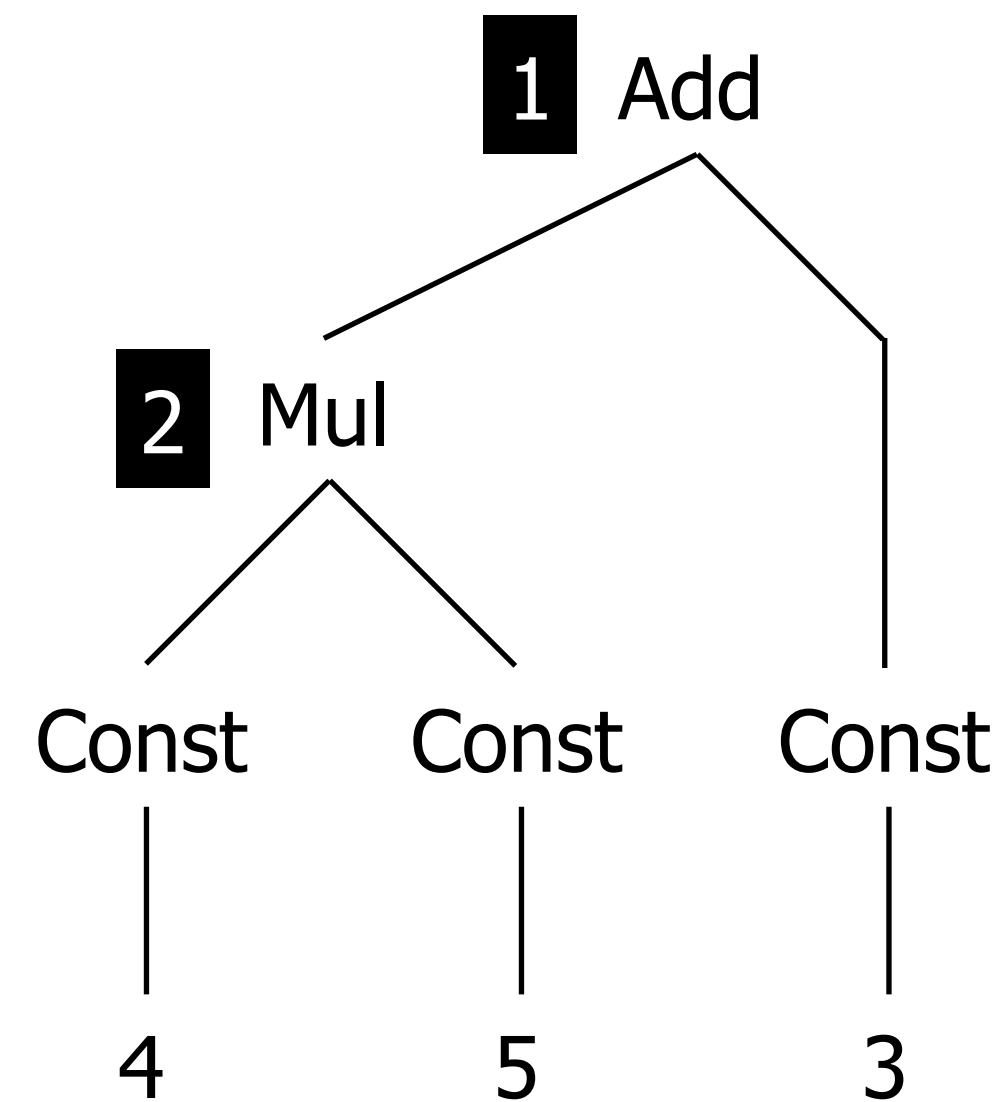
# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
topdown(s) = s ; all(topdown(s))
```

```
topdown(try(switch))
```



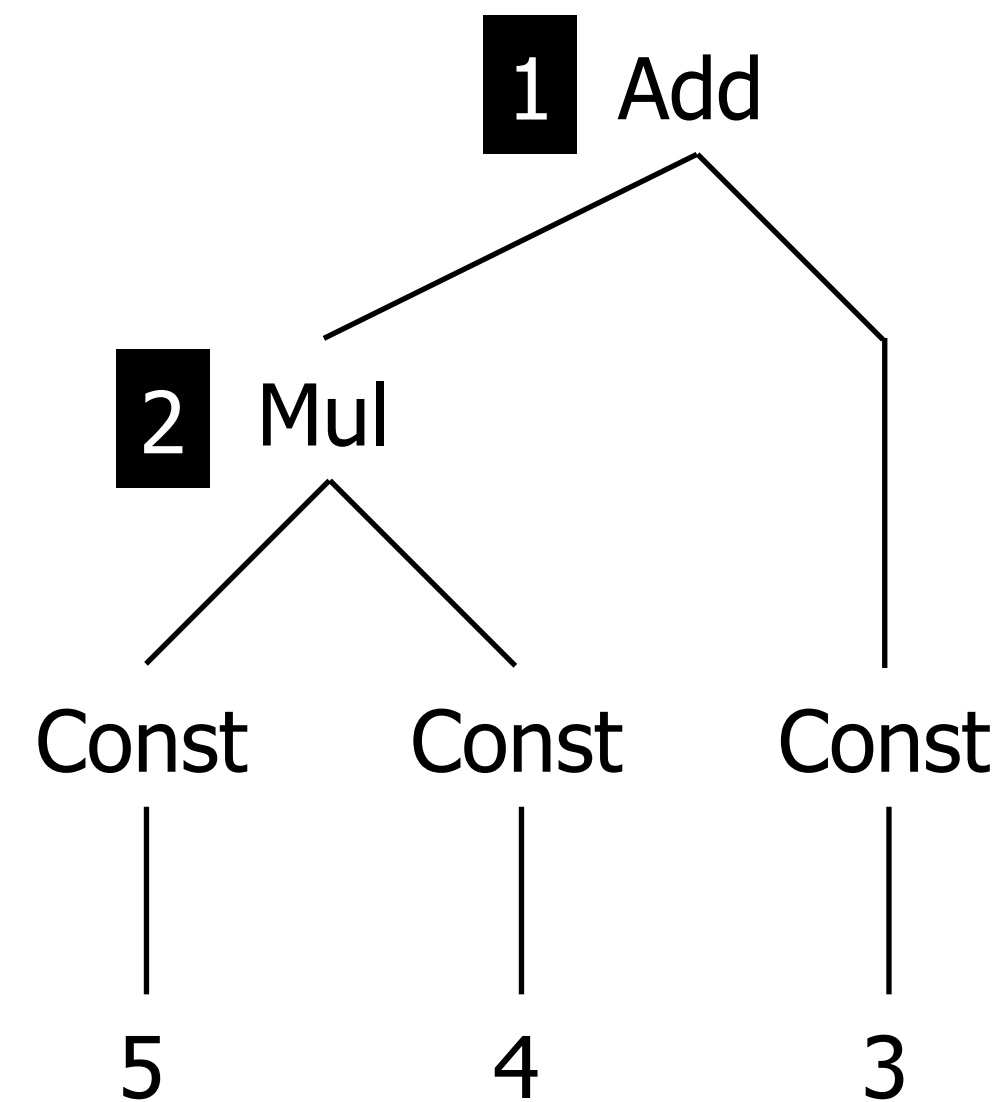
# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
topdown(s) = s ; all(topdown(s))
```

```
topdown(try(switch))
```



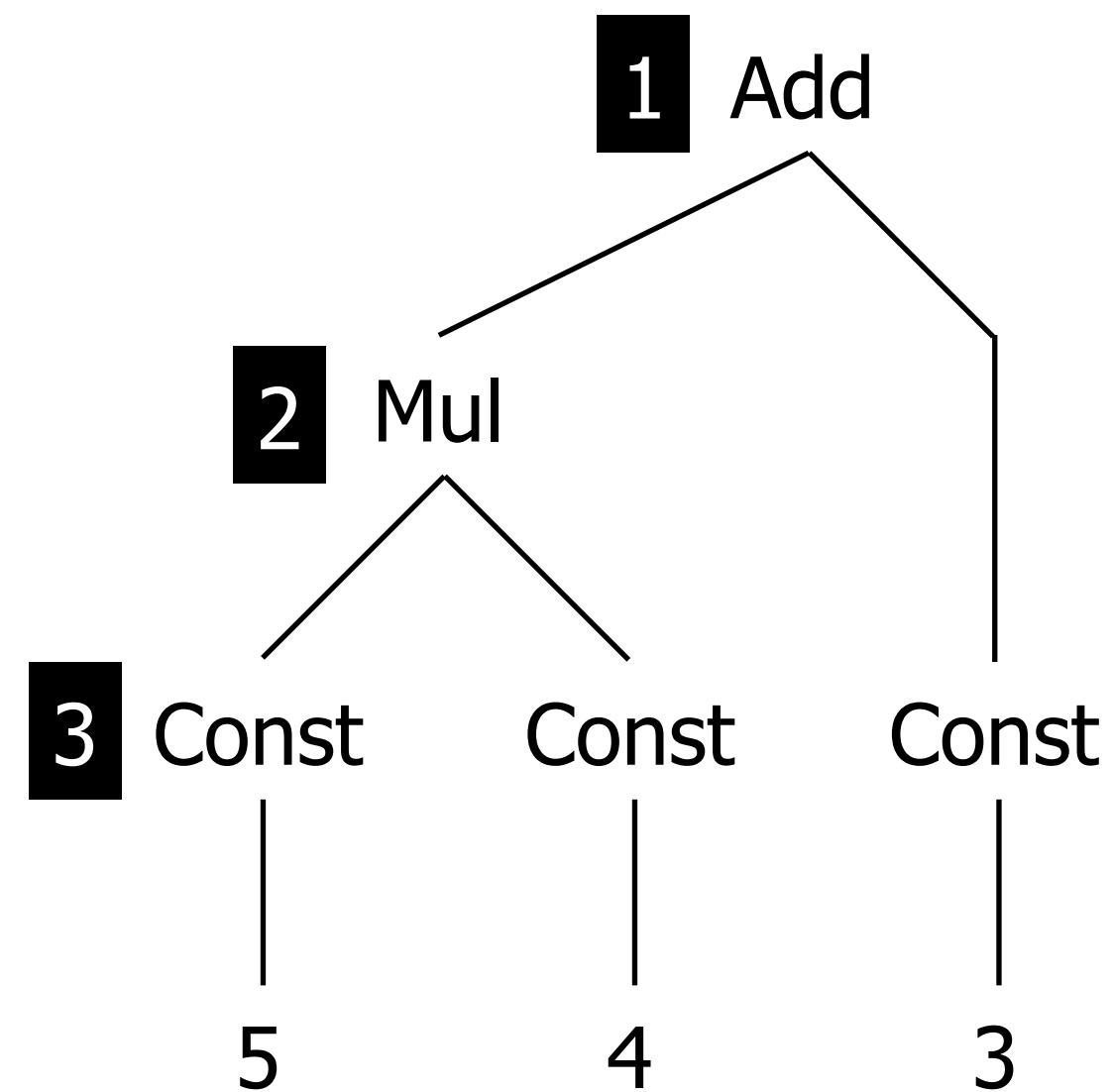
# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
topdown(s) = s ; all(topdown(s))
```

```
topdown(try(switch))
```

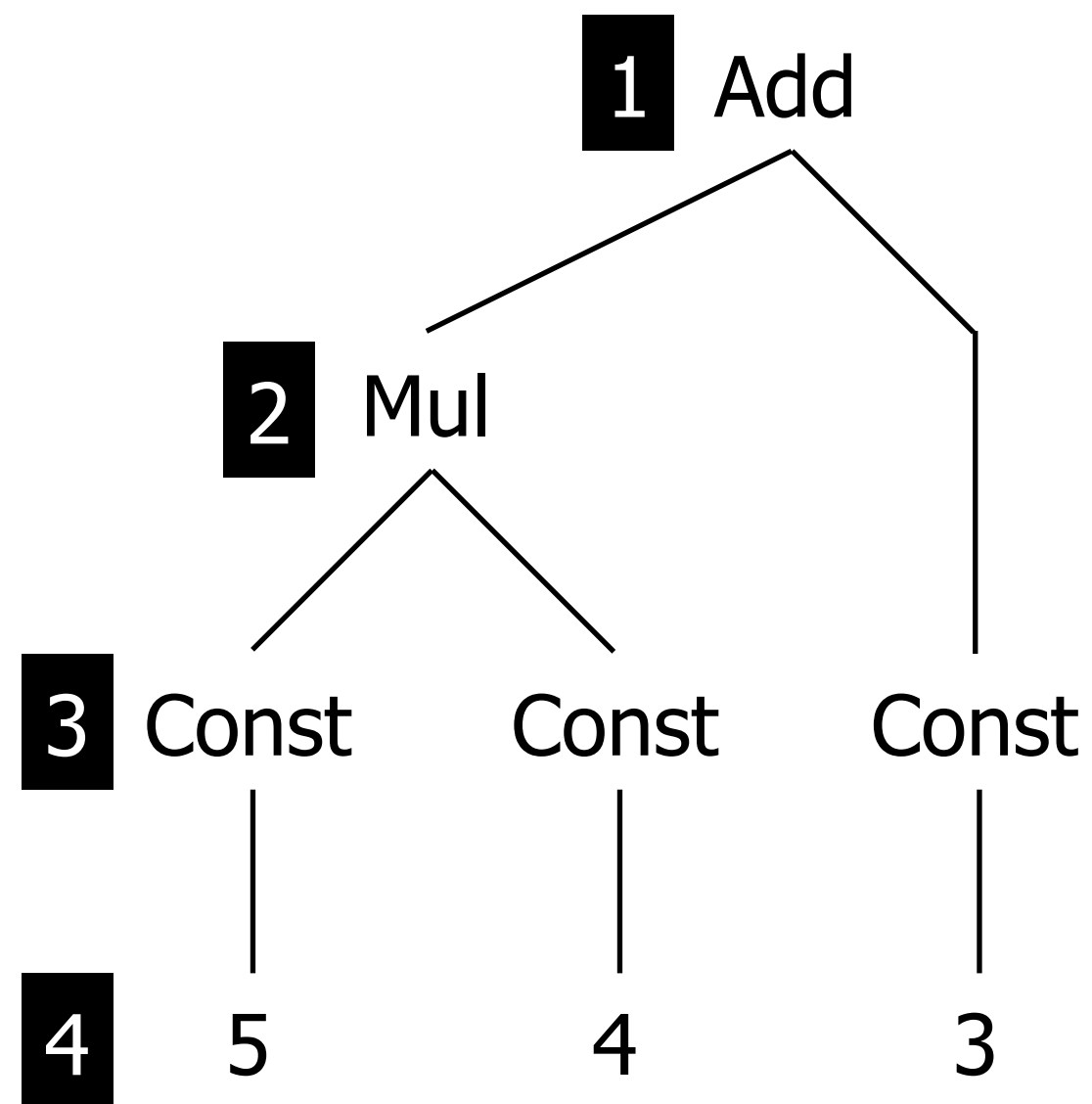


# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(try(switch))
```

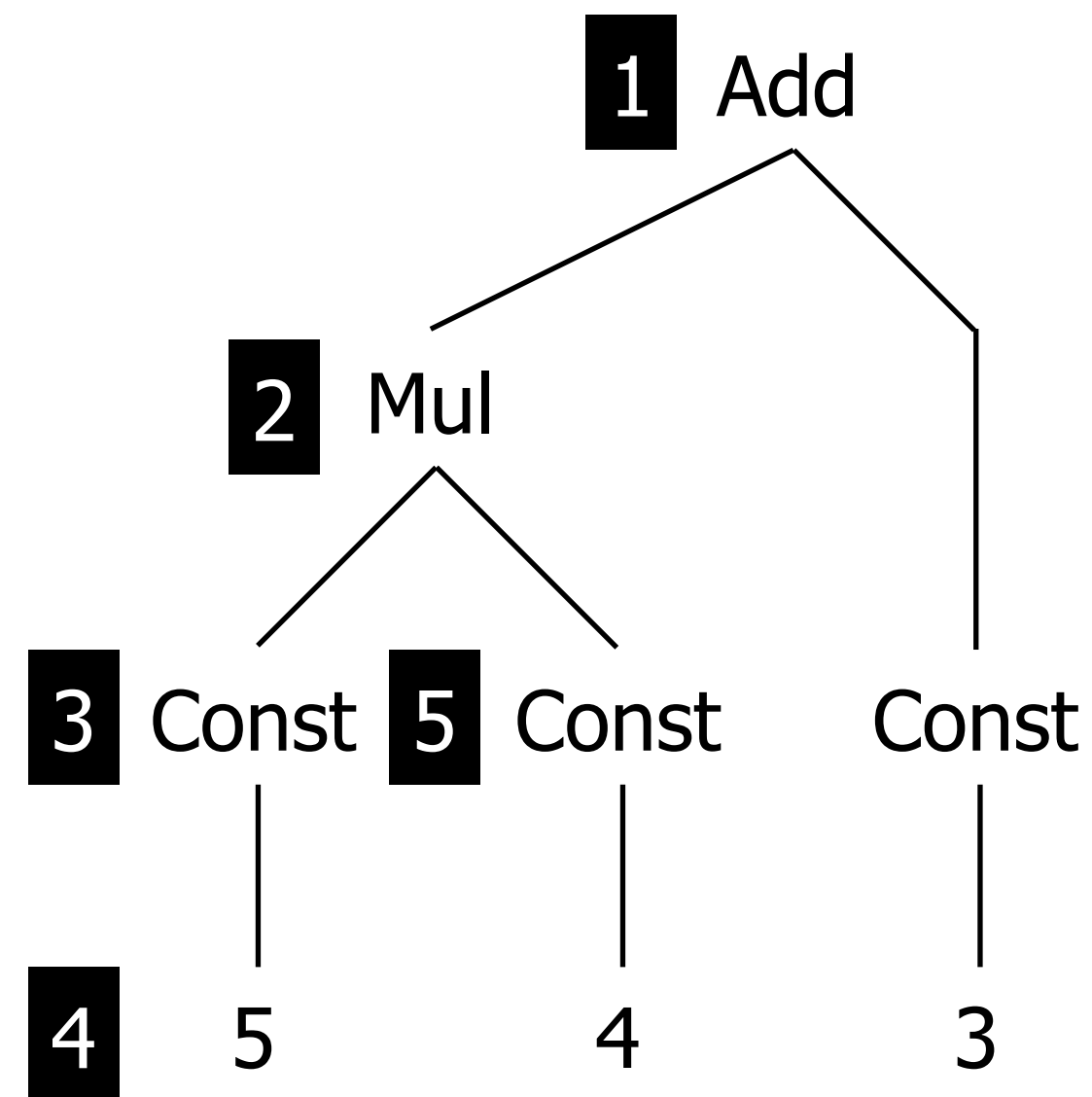


# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(try(switch))
```

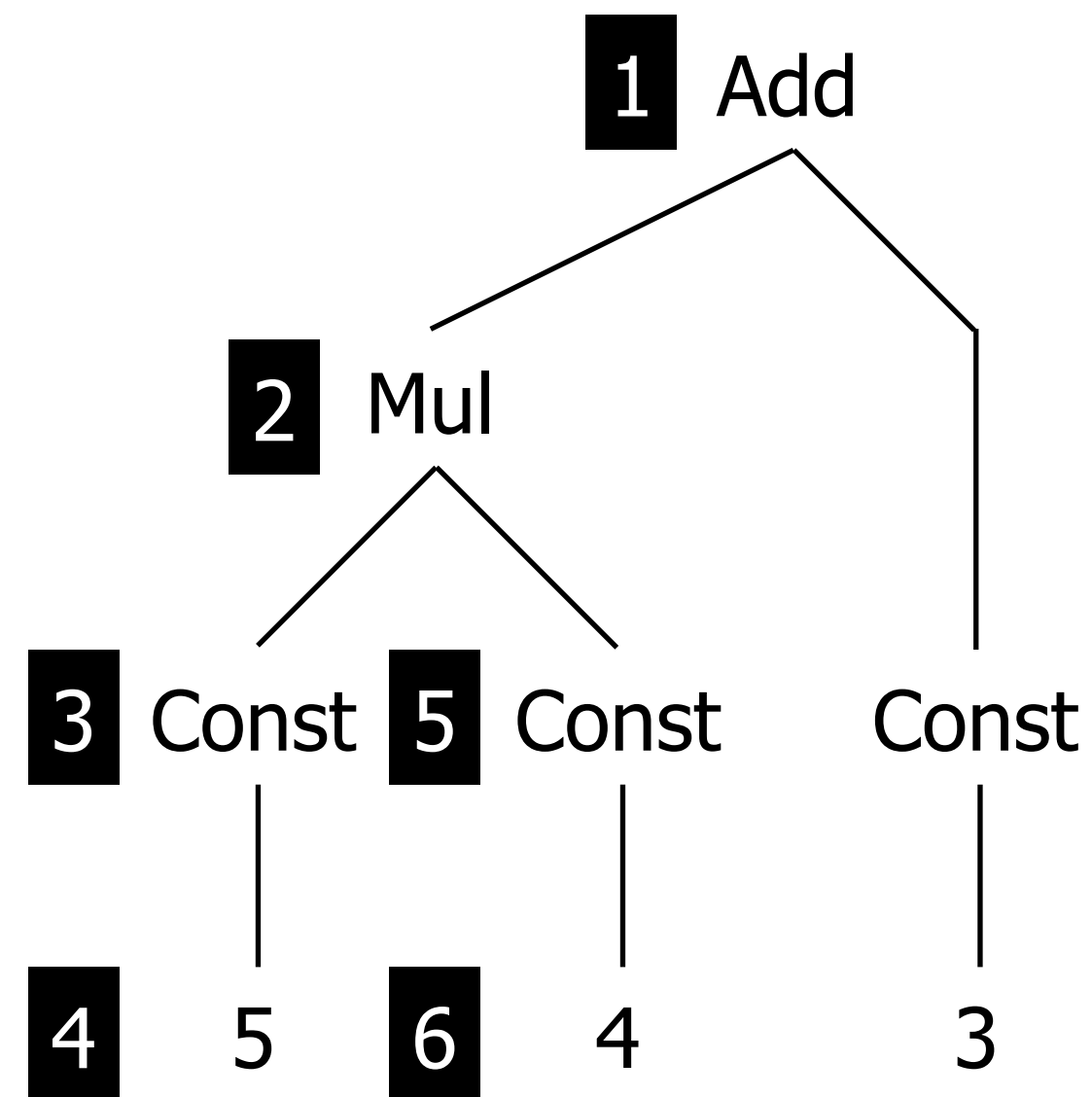


# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(try(switch))
```



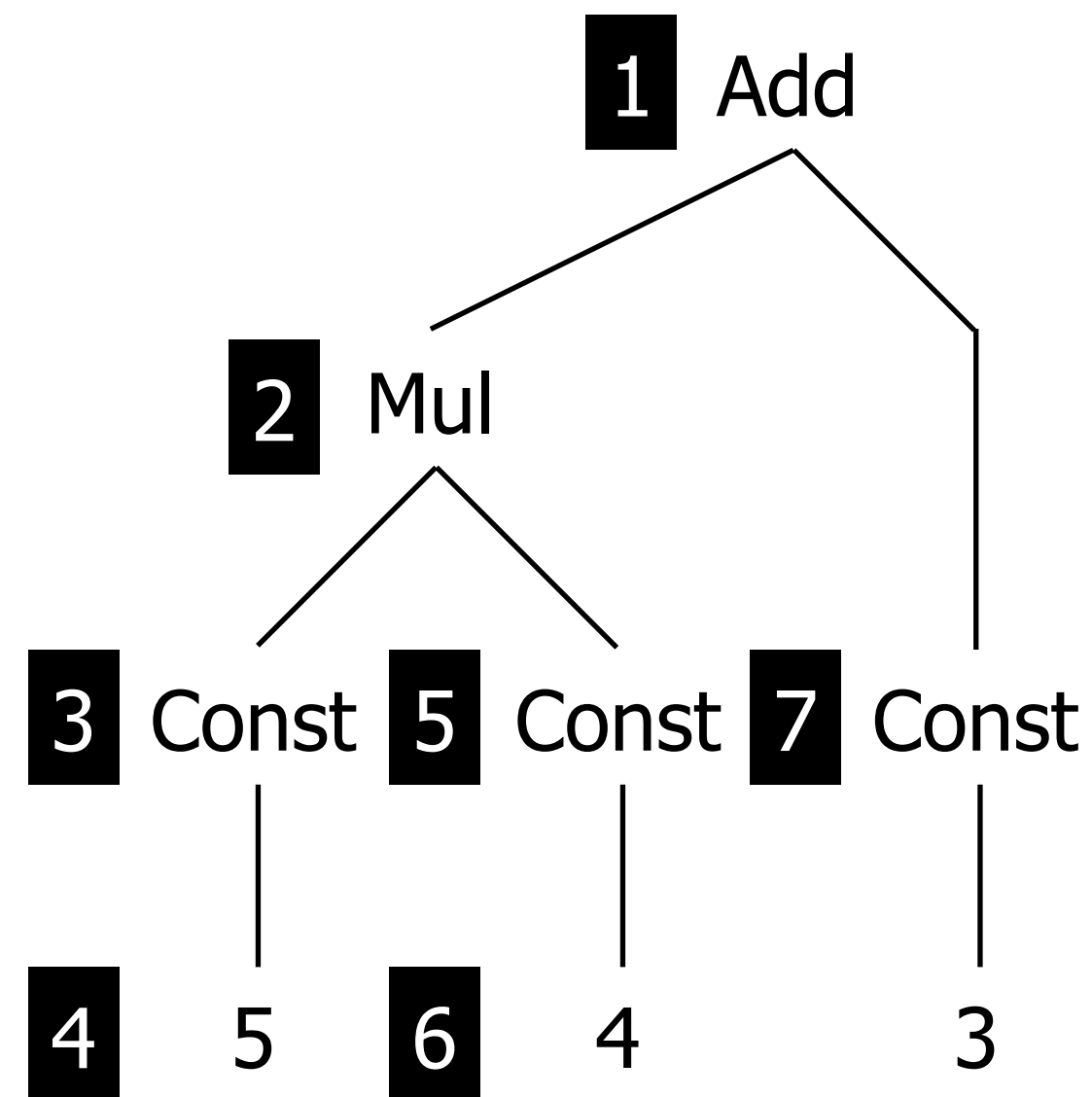


# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(try(switch))
```

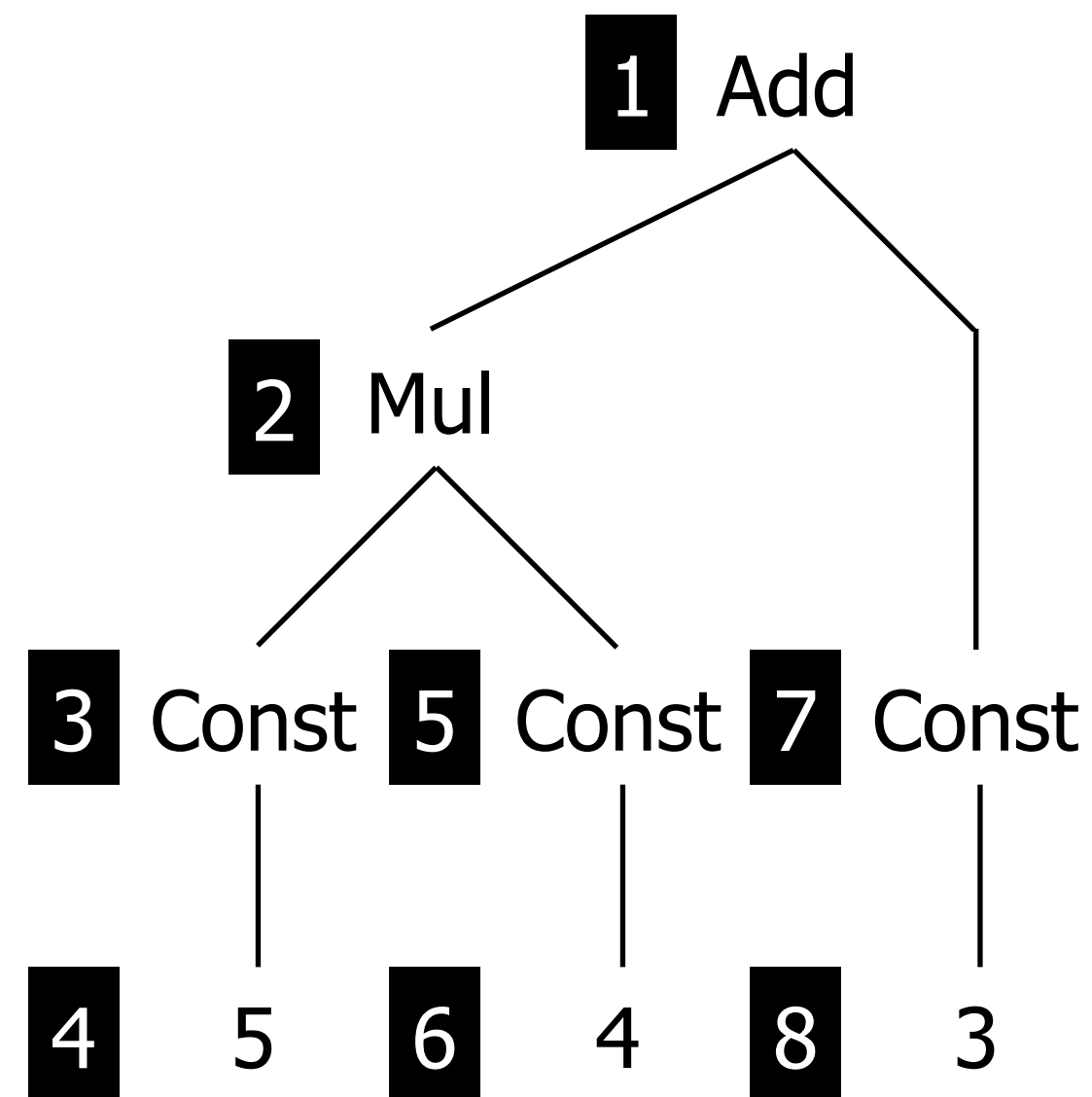


# Traversal: Topdown/Try

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

topdown(s) = s ; all(topdown(s))

topdown(try(switch))
```



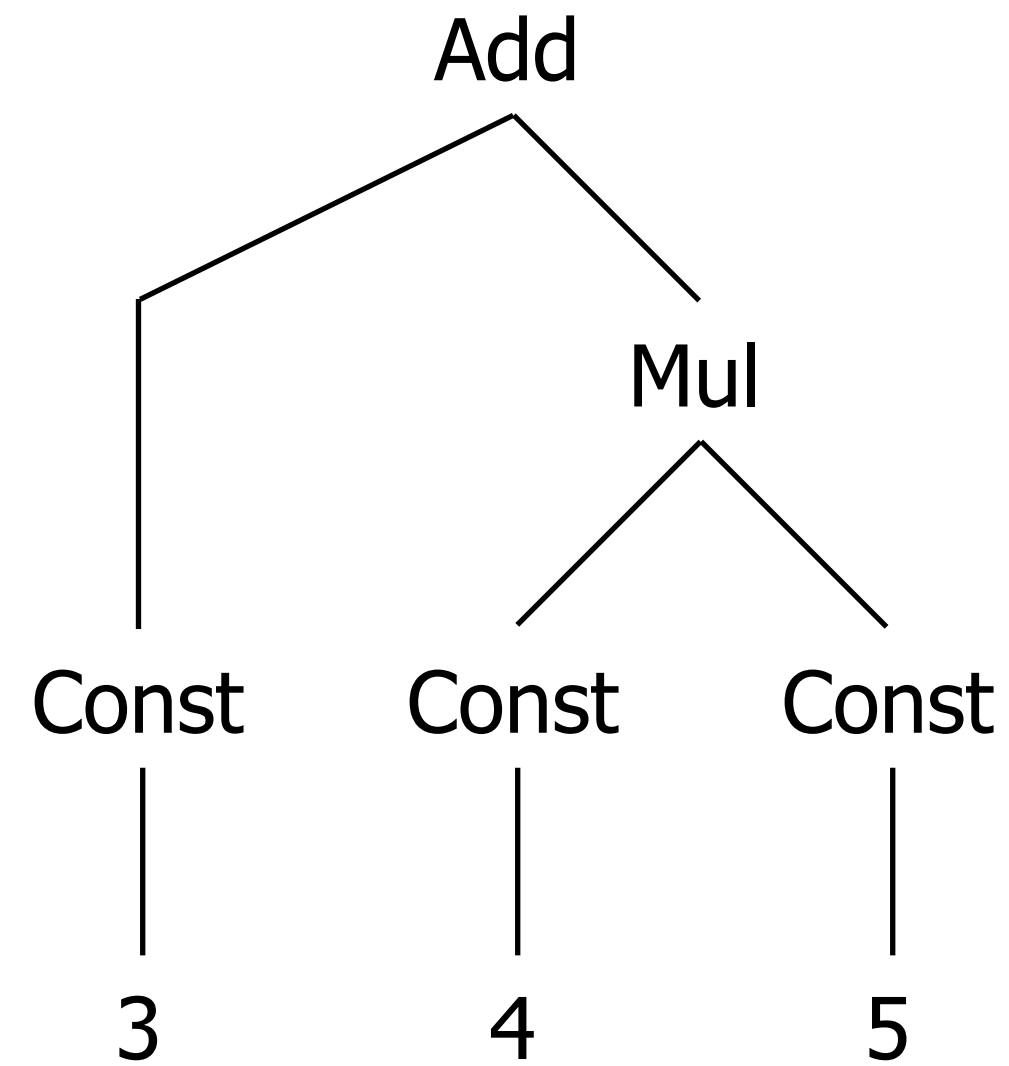
# Traversal: Alltd

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
alltd(s) = s <+ all(alltd(s))
```

```
alltd(switch)
```



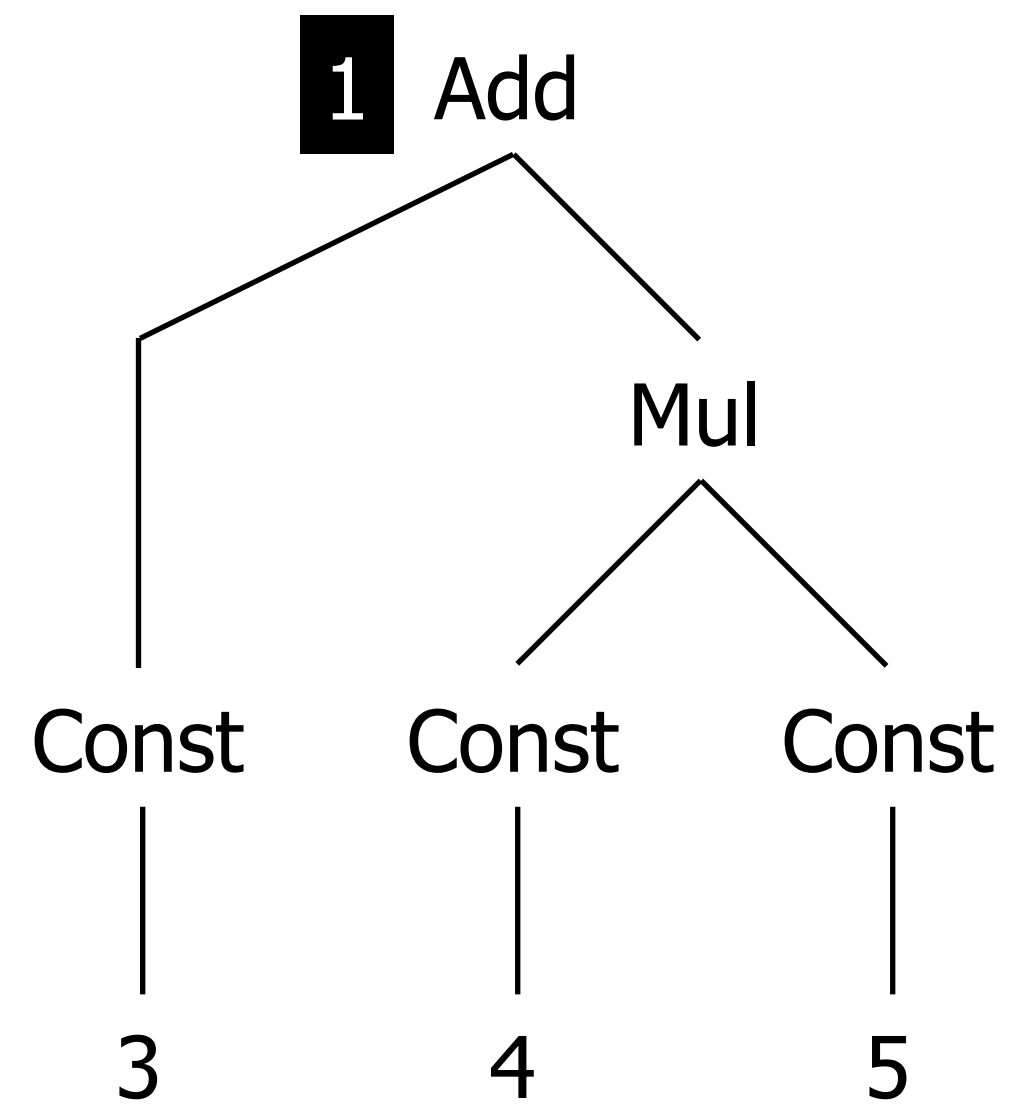
# Traversal: Alltd

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
alltd(s) = s <+ all(alltd(s))
```

```
alltd(switch)
```



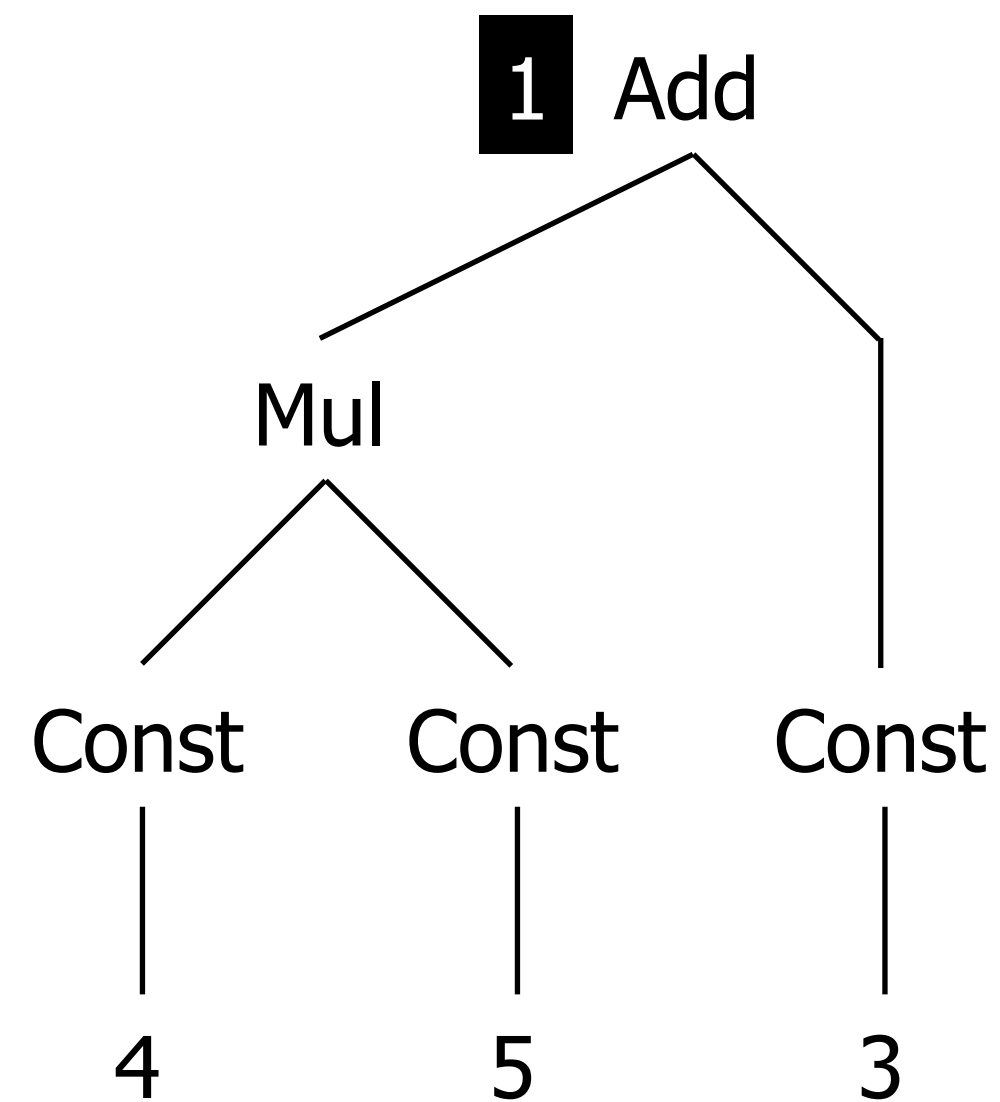
# Traversal: Alltd

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
alltd(s) = s <+ all(alltd(s))
```

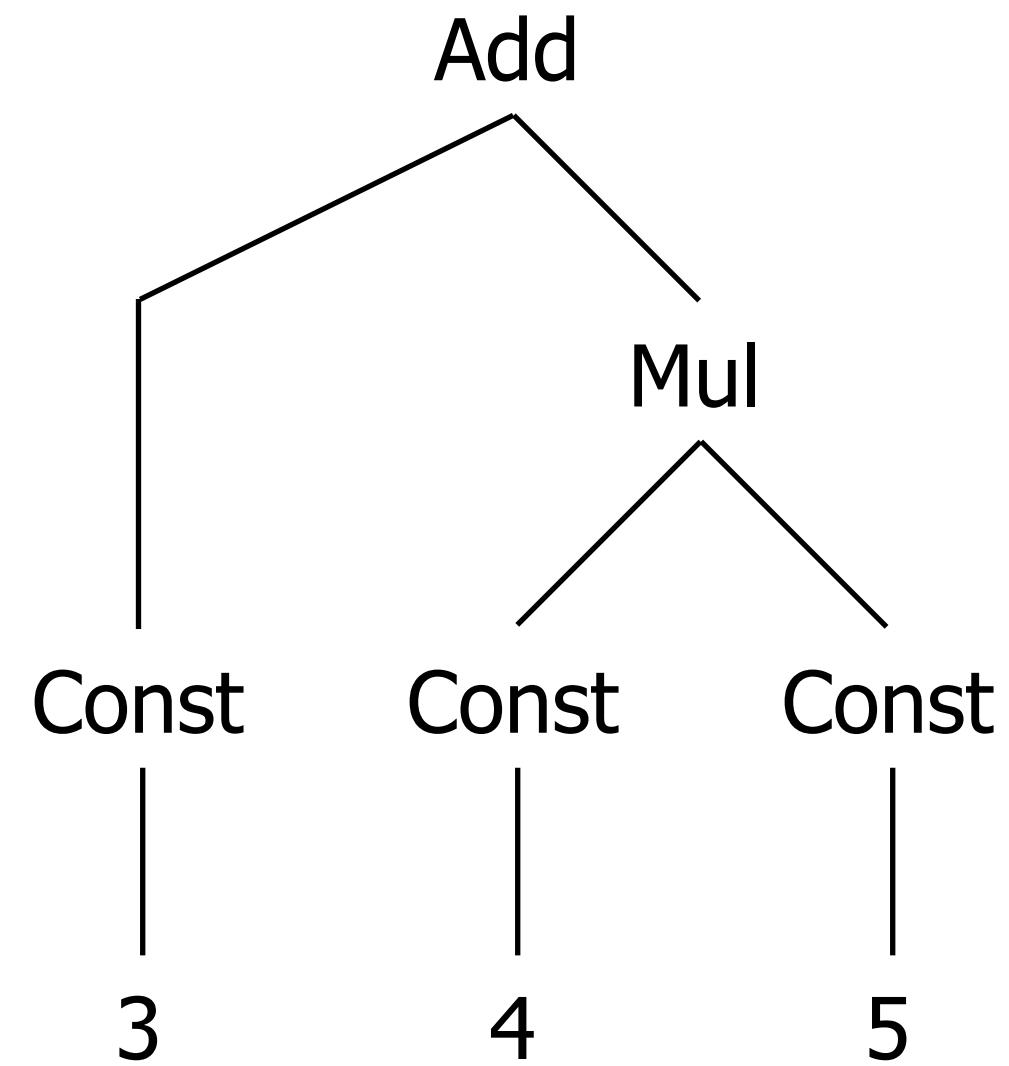
```
alltd(switch)
```



# Traversal: bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

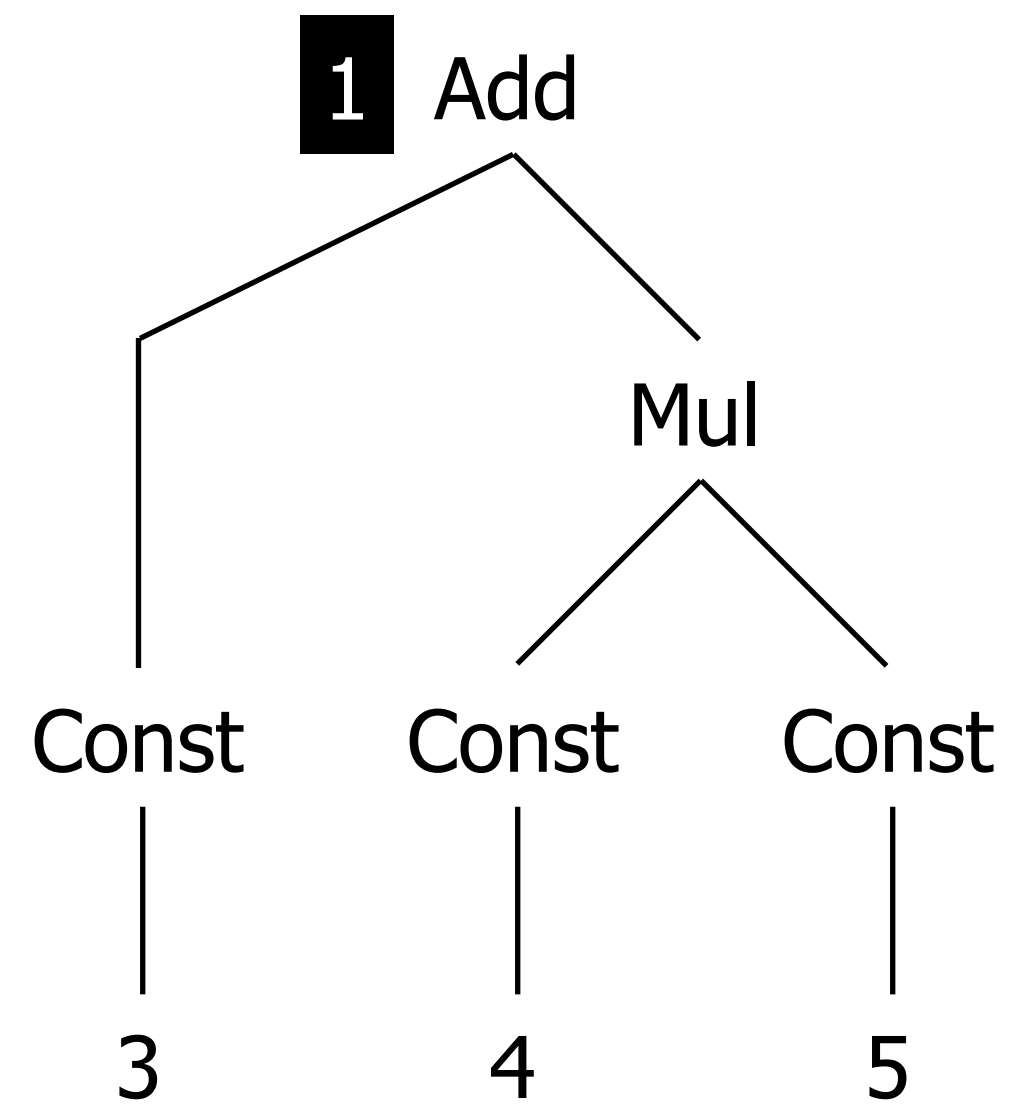
bottomup(s) = all(bottomup(s)) ; s
bottomup(switch)
```



# Traversal: bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

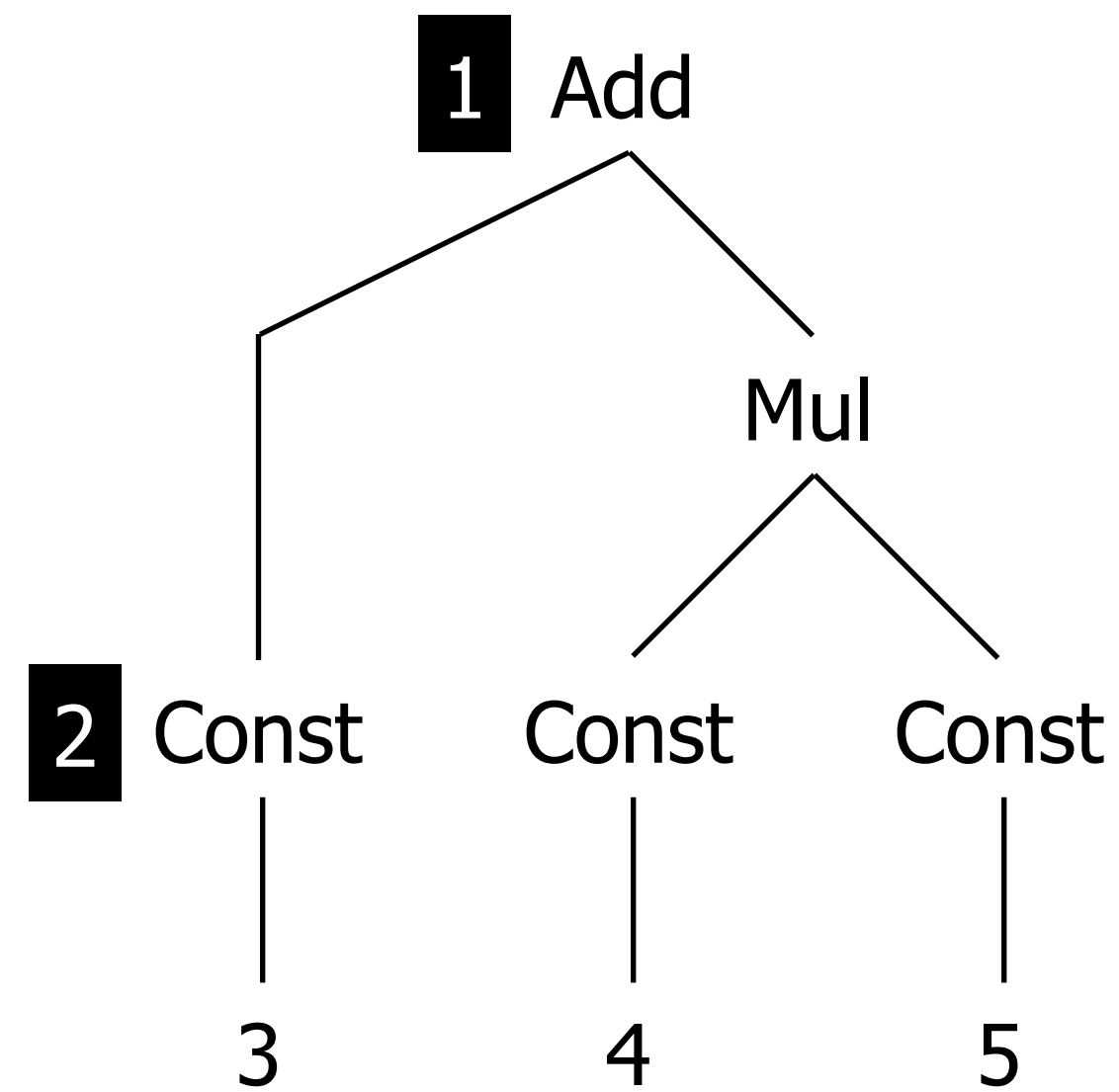
bottomup(s) = all(bottomup(s)) ; s
bottomup(switch)
```



# Traversal: bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

bottomup(s) = all(bottomup(s)) ; s
bottomup(switch)
```

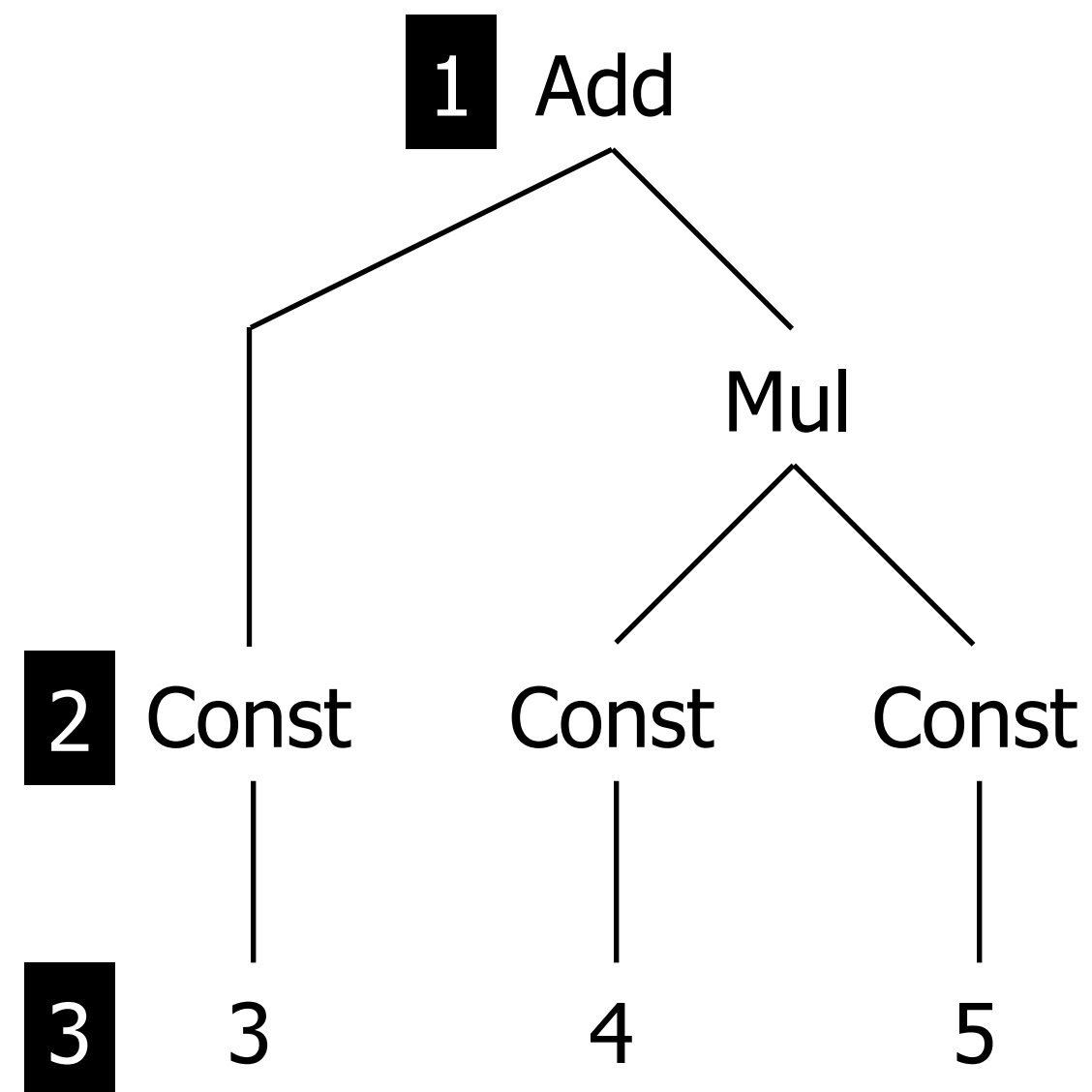




# Traversal: bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

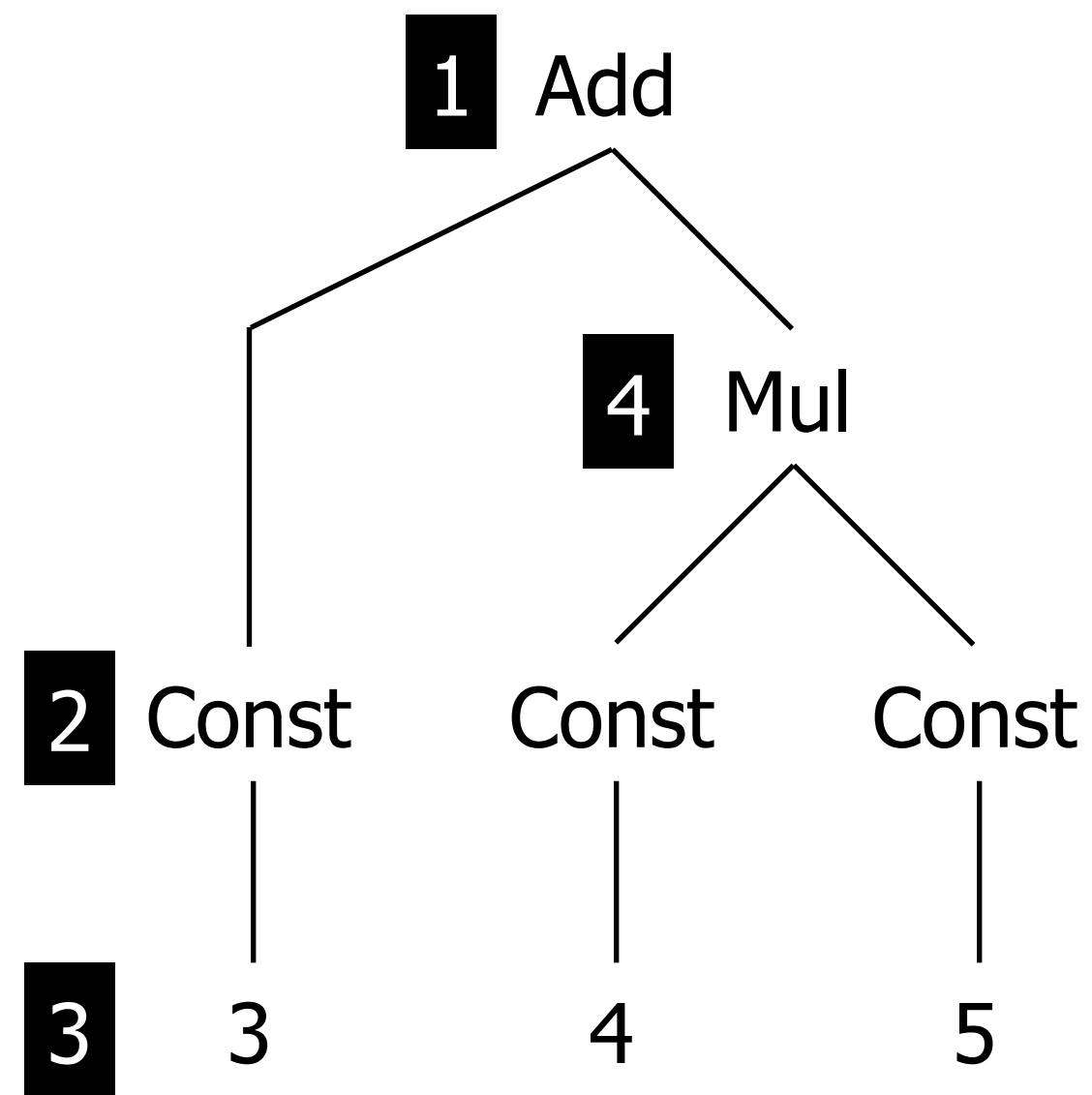
bottomup(s) = all(bottomup(s)) ; s
bottomup(switch)
```



# Traversal: bottomup

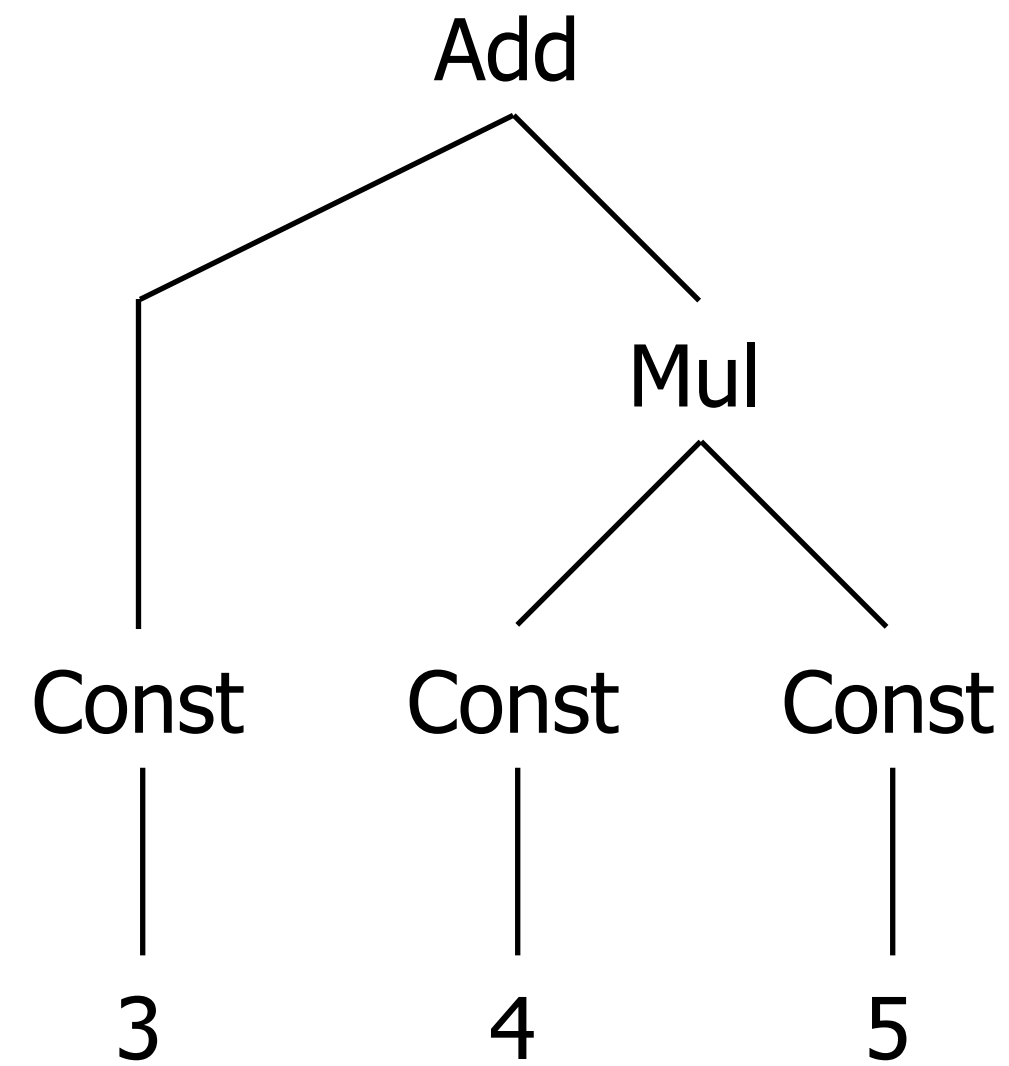
```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

bottomup(s) = all(bottomup(s)) ; s
bottomup(switch)
```



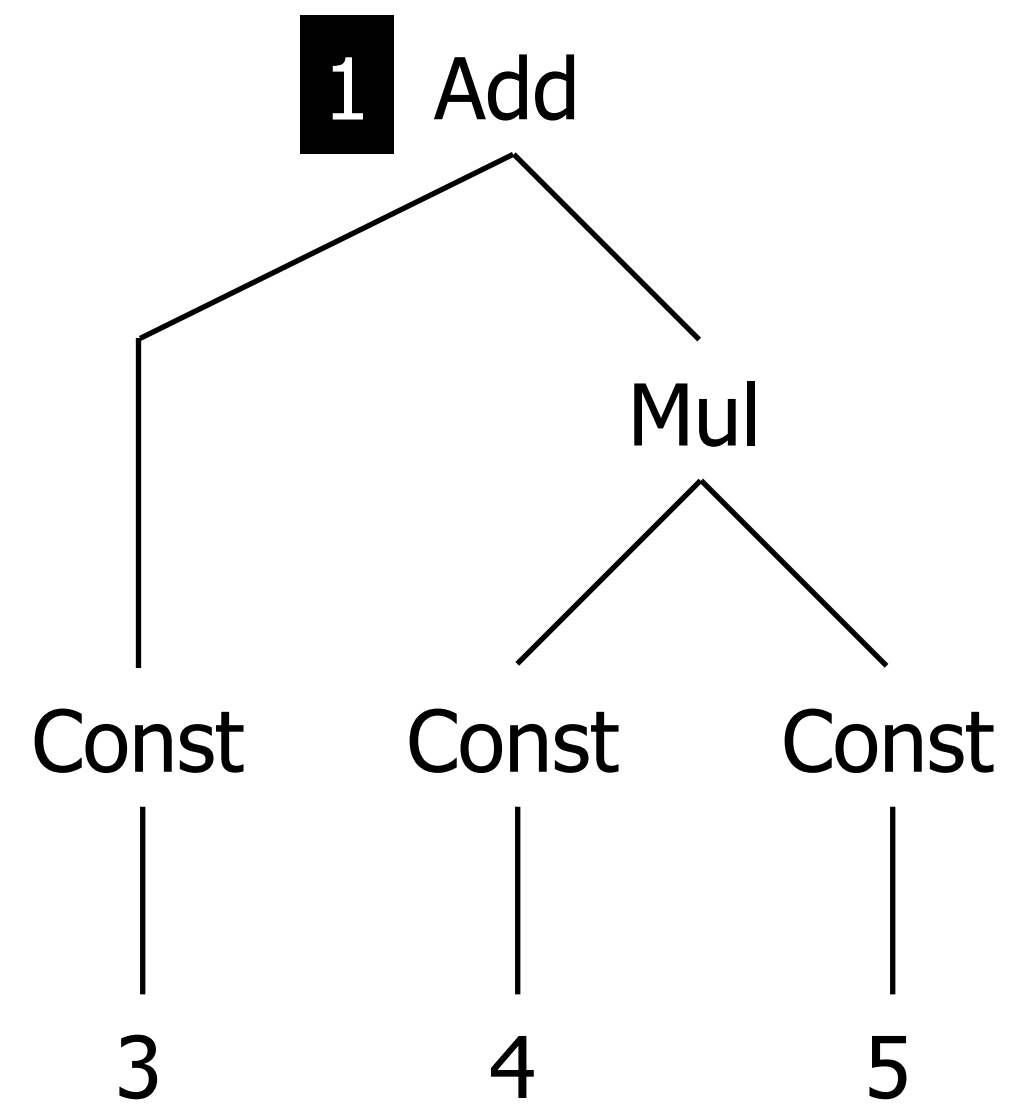
# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



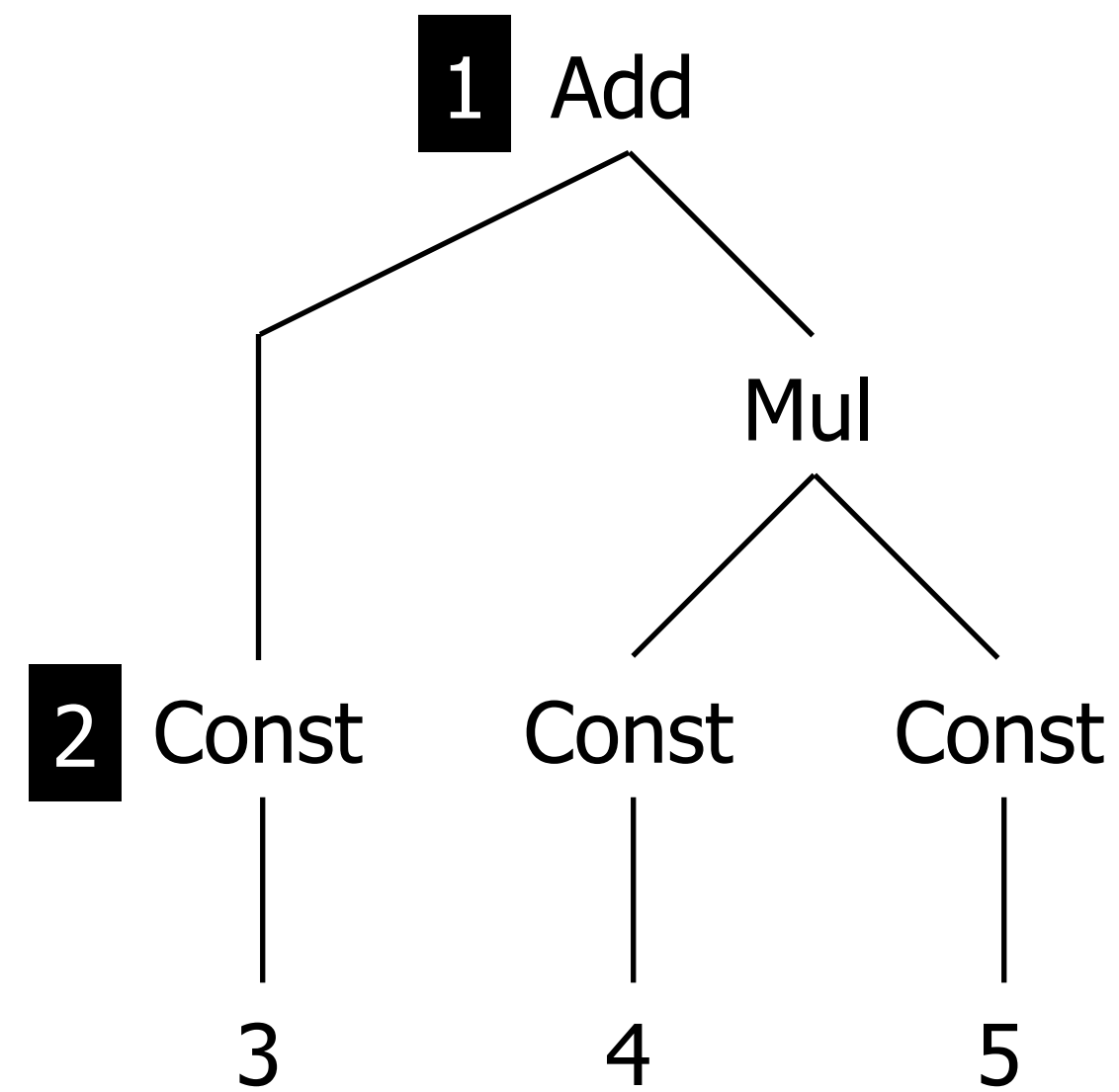
# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)  
switch: Mul(e1, e2) -> Mul(e2, e1)  
bottomup(s) = all(bottomup(s)) ; s  
bottomup(try(switch))
```



# Traversal: Bottomup

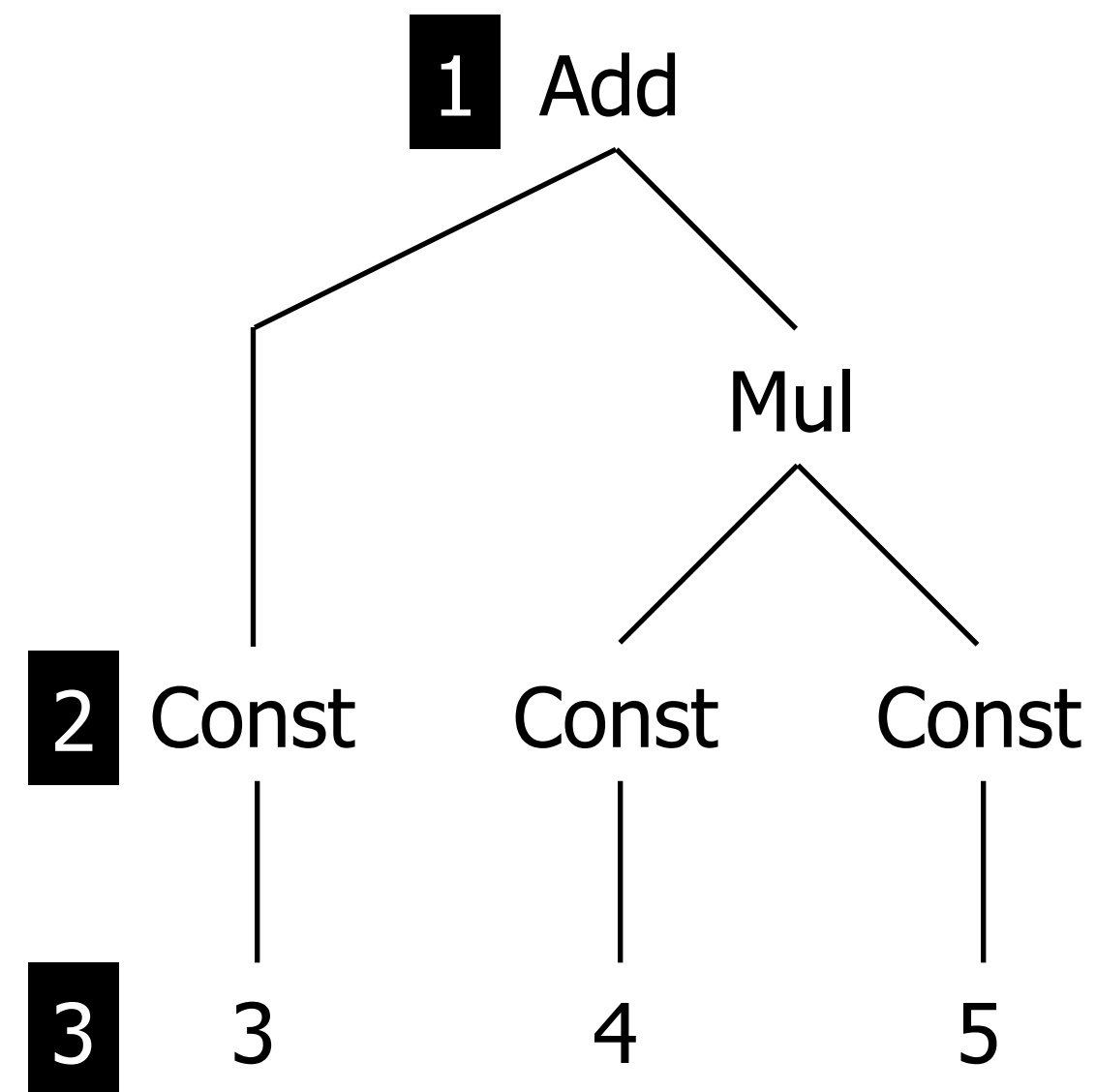
```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

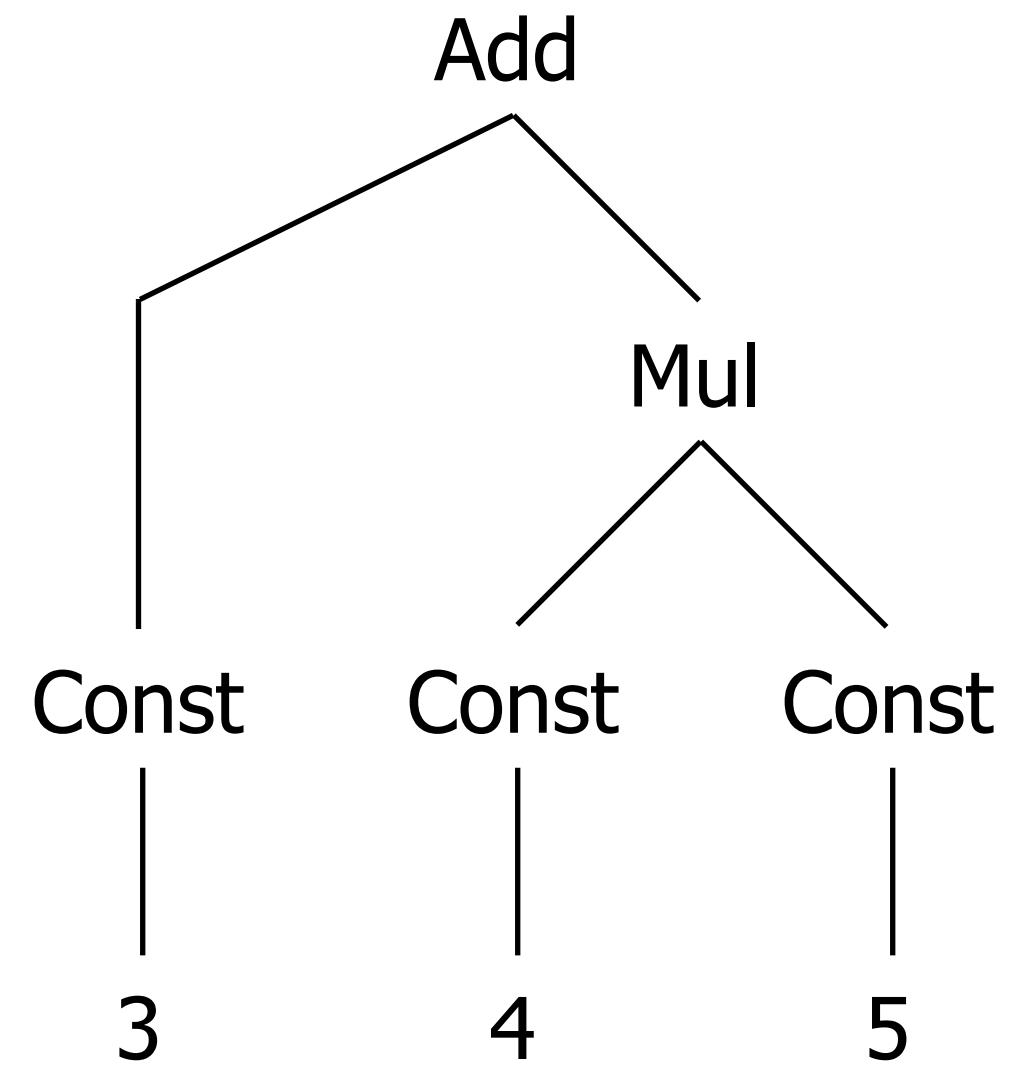
```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

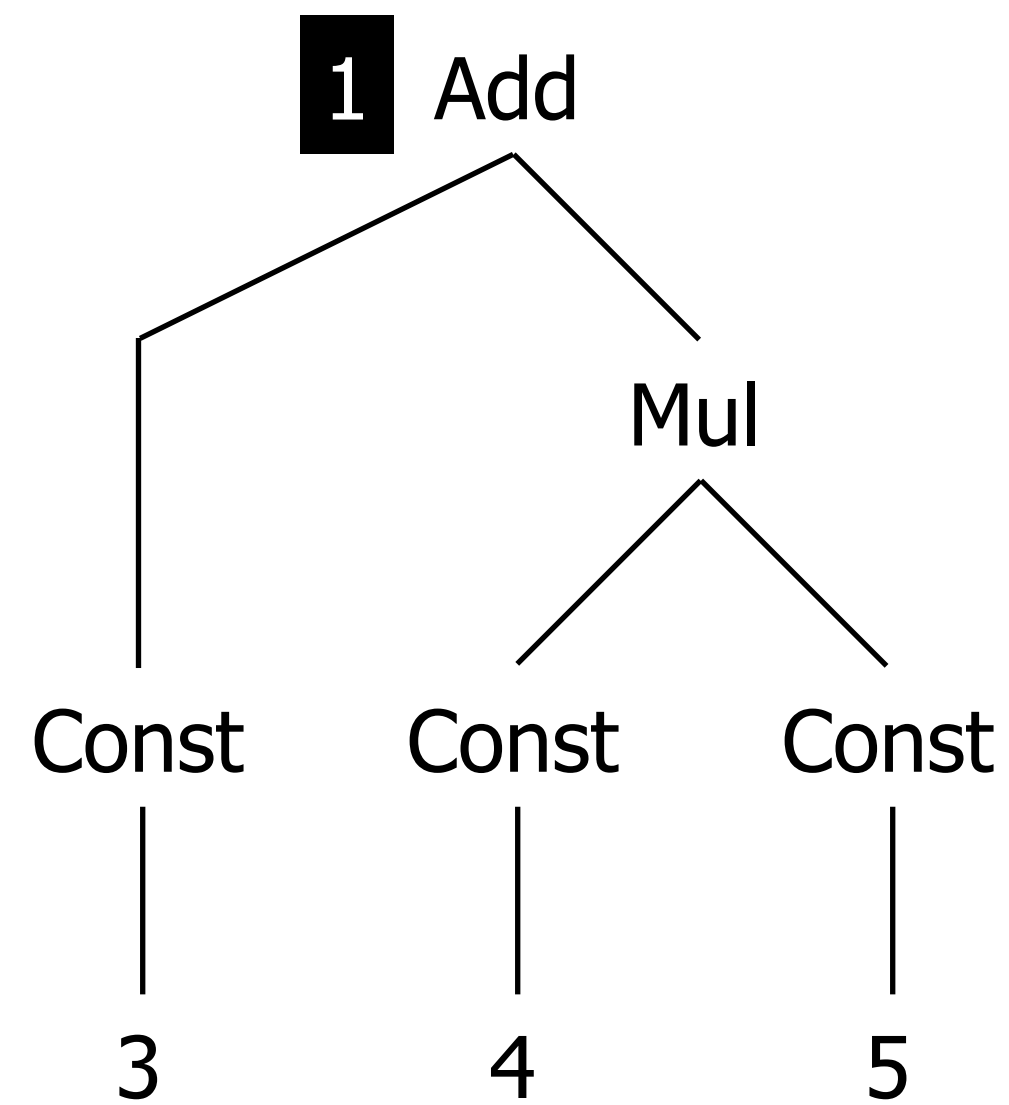
```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

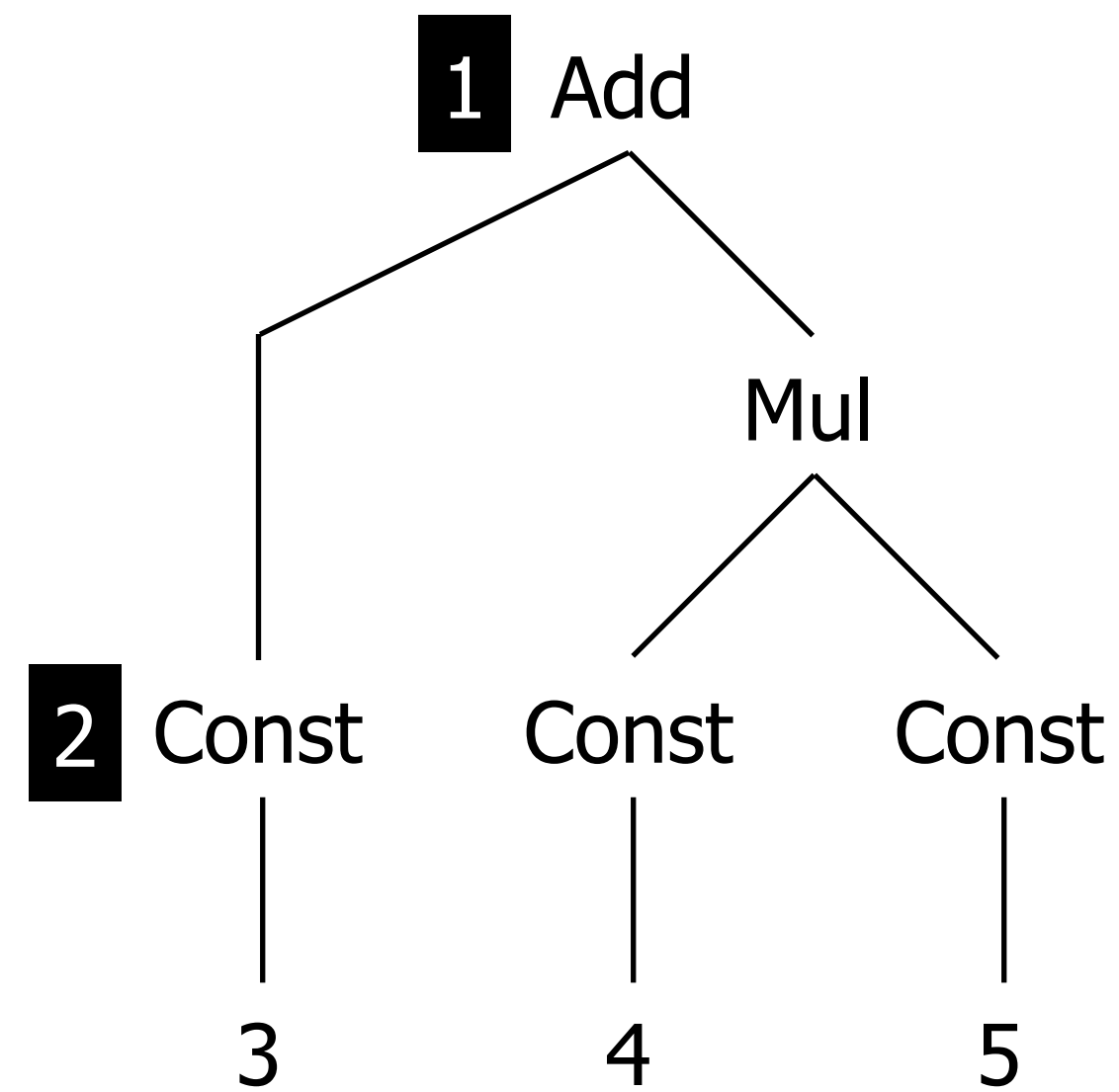
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```





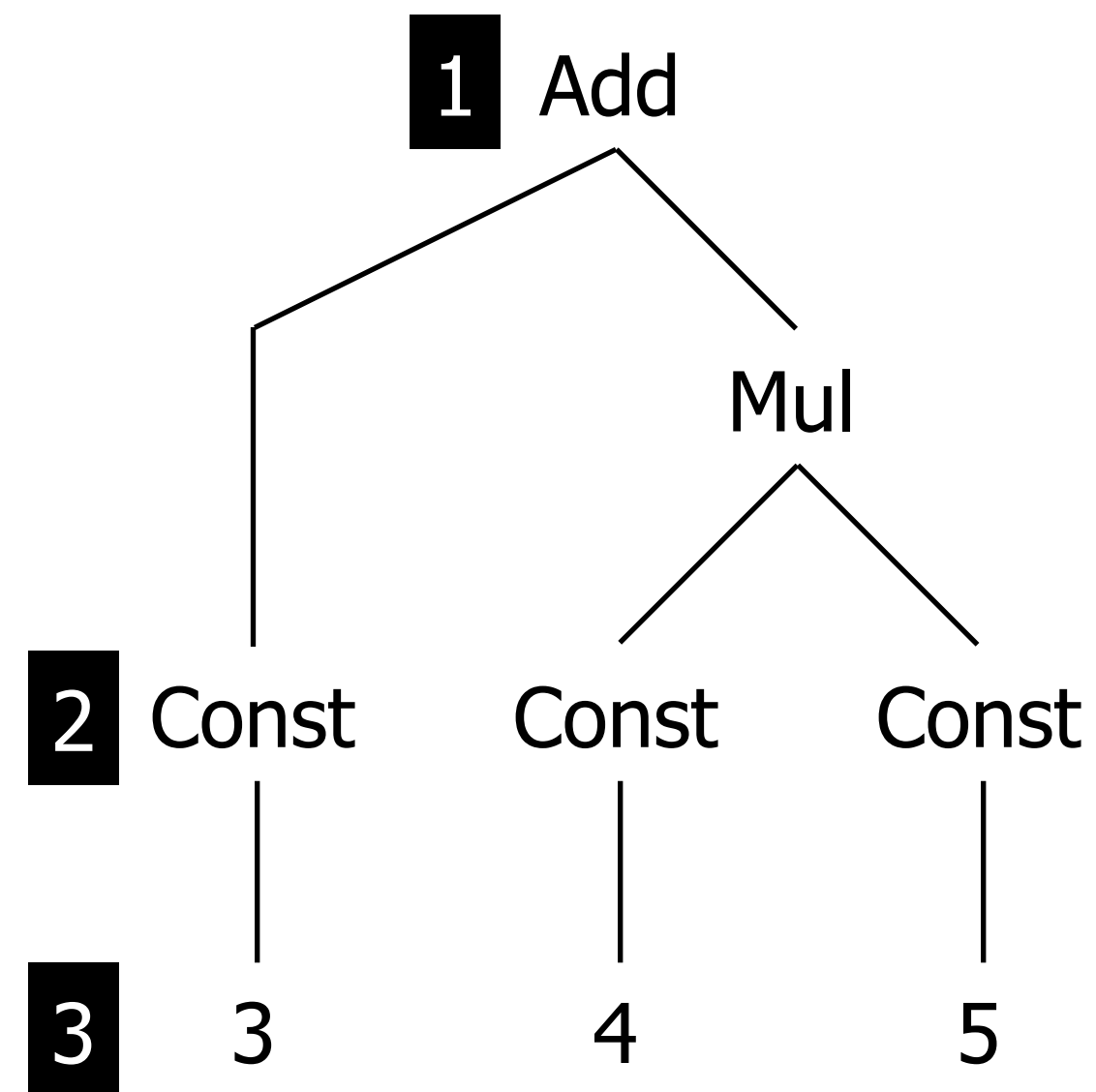
# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

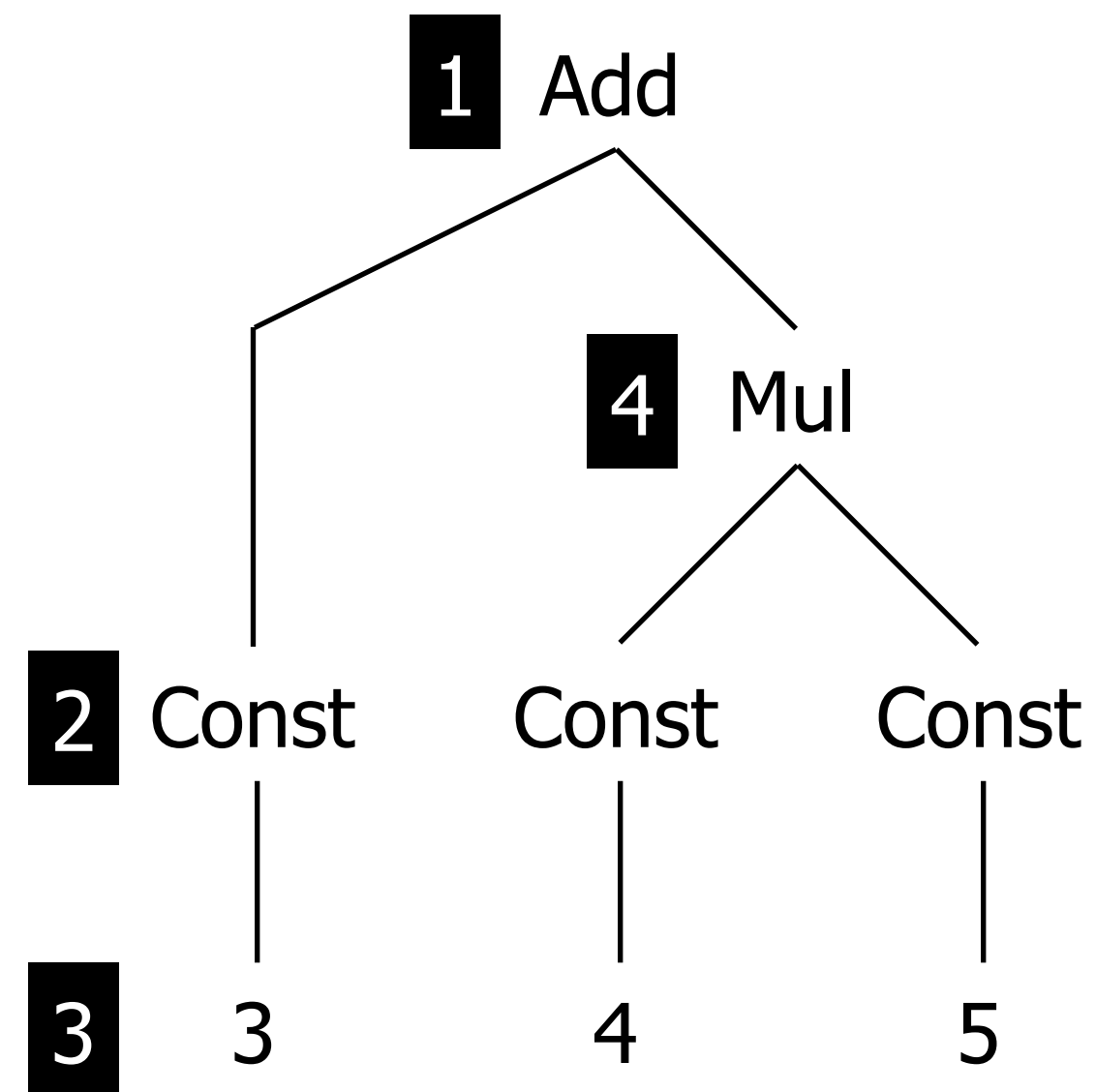
```
switch: Add(e1, e2) -> Add(e2, e1)  
switch: Mul(e1, e2) -> Mul(e2, e1)  
bottomup(s) = all(bottomup(s)) ; s  
bottomup(try(switch))
```



# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

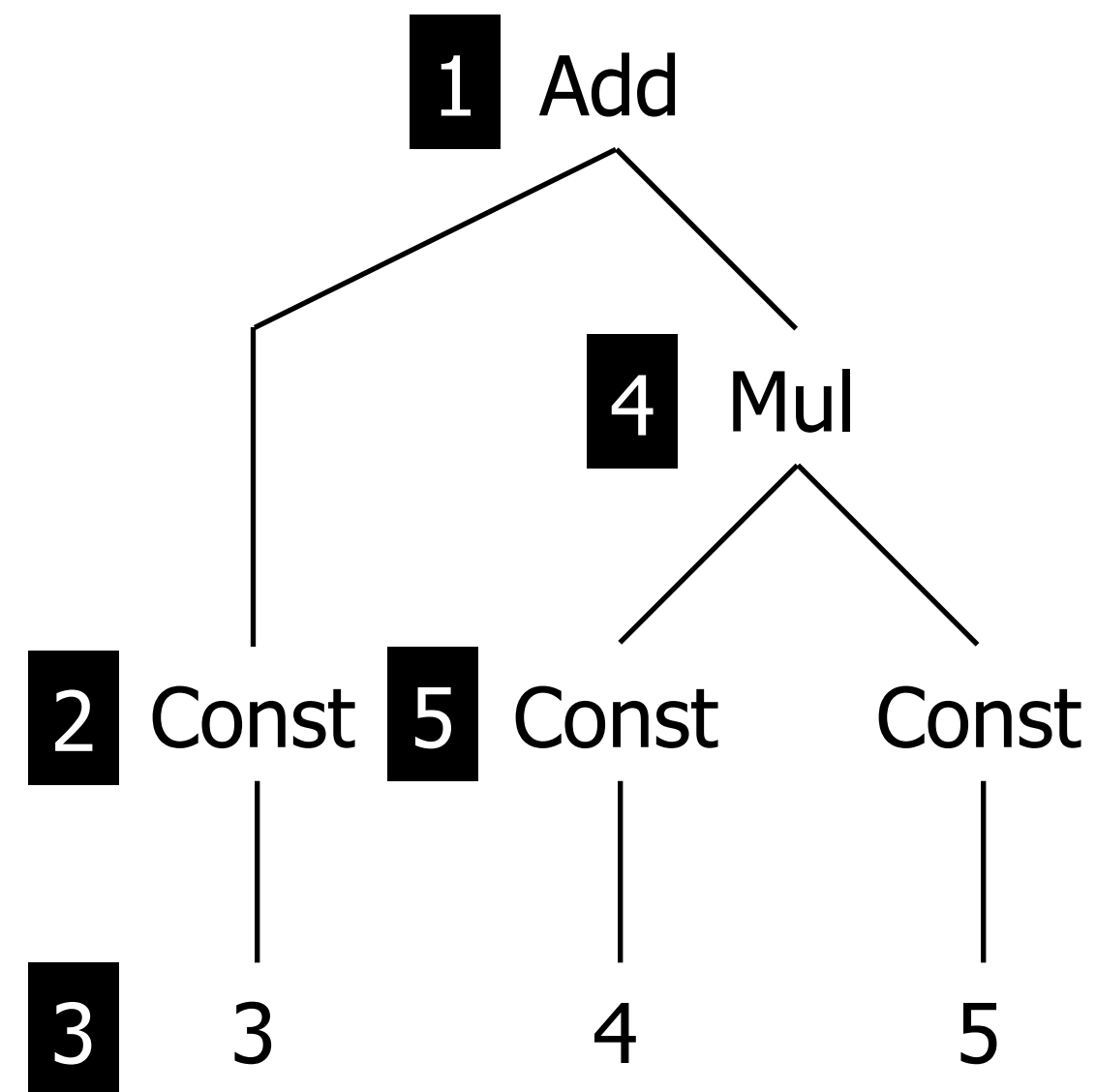
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

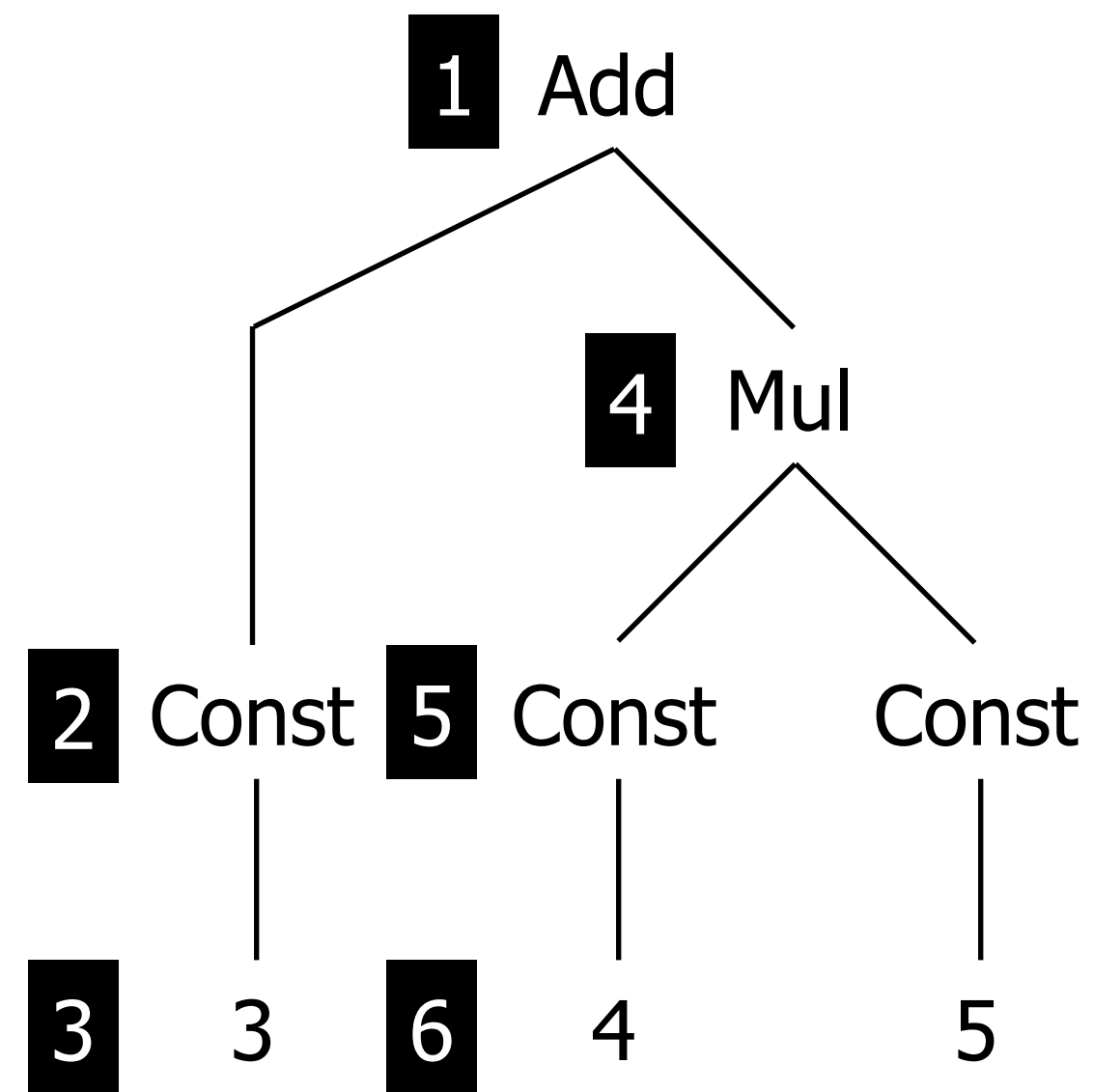
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

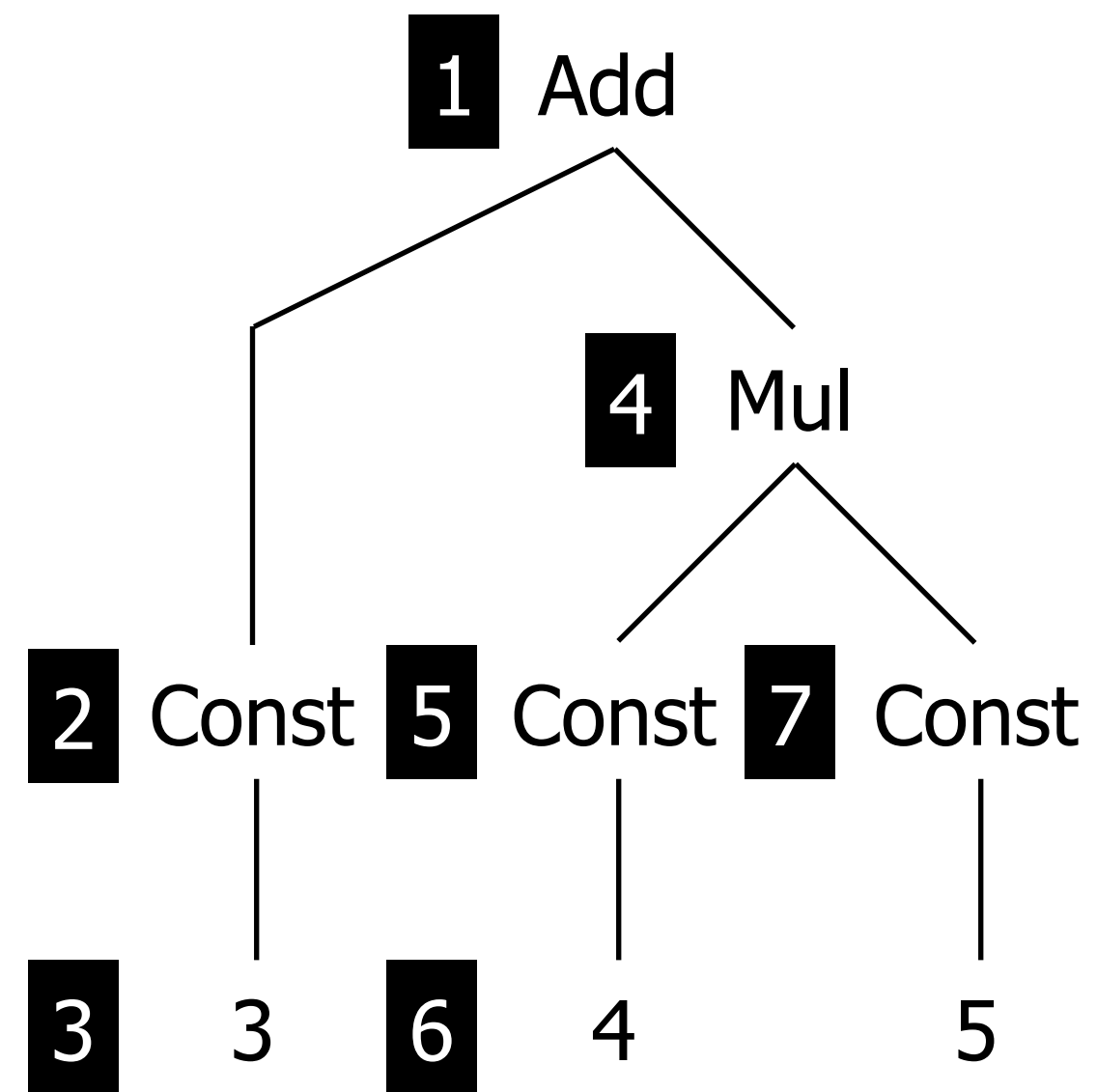
```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



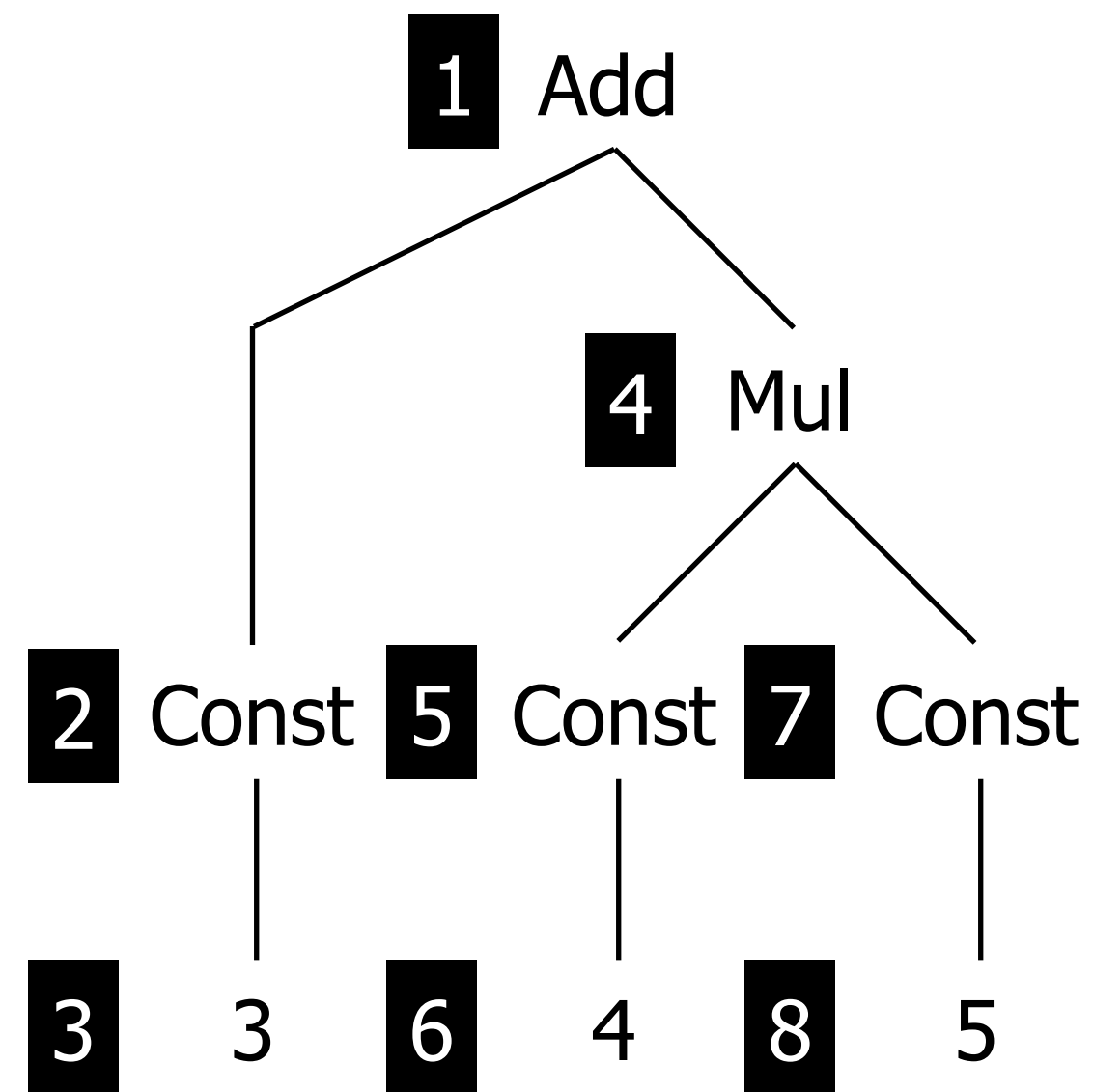
# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)  
switch: Mul(e1, e2) -> Mul(e2, e1)  
bottomup(s) = all(bottomup(s)) ; s  
bottomup(try(switch))
```



# Traversal: Bottomup

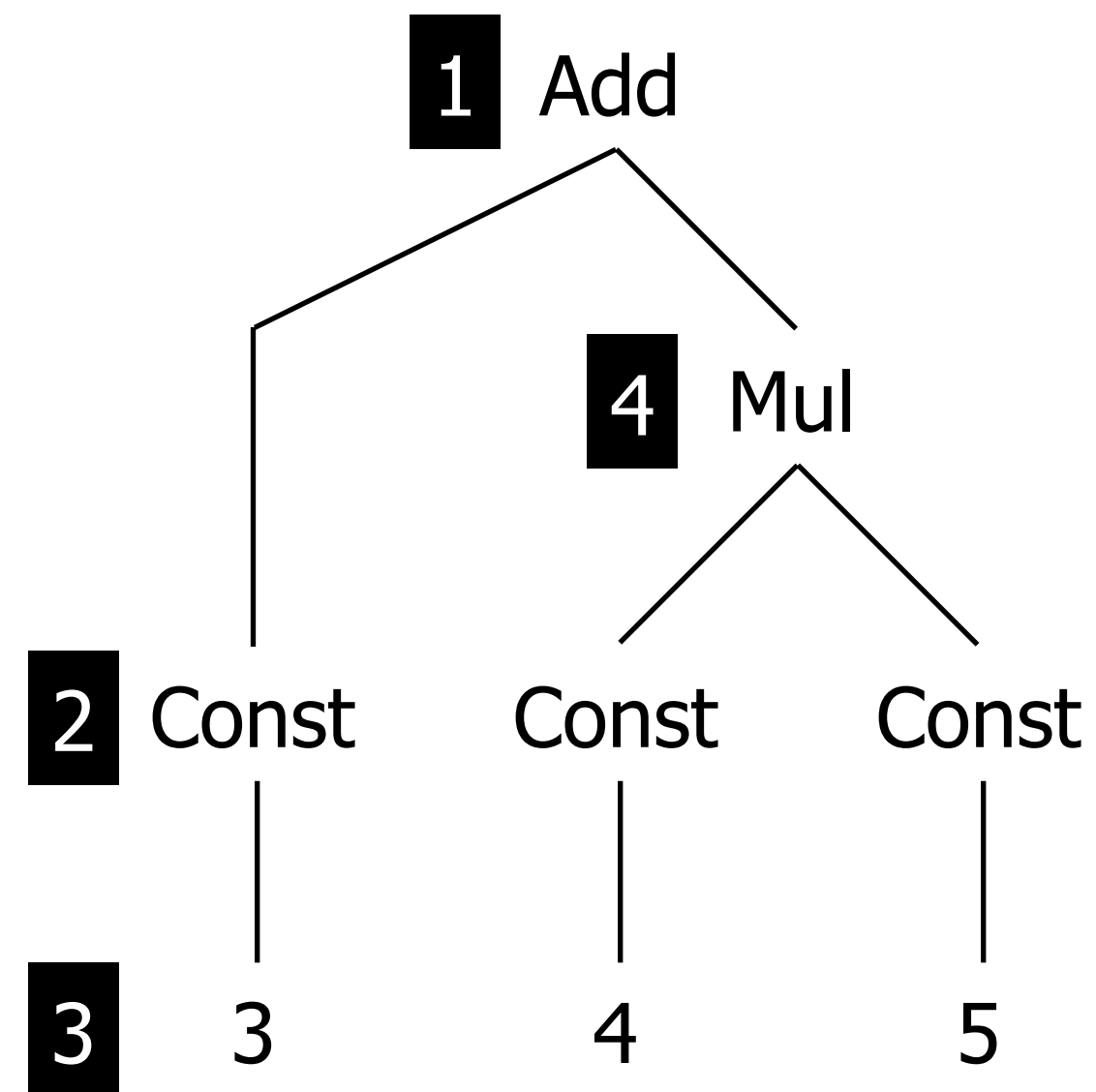
```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```

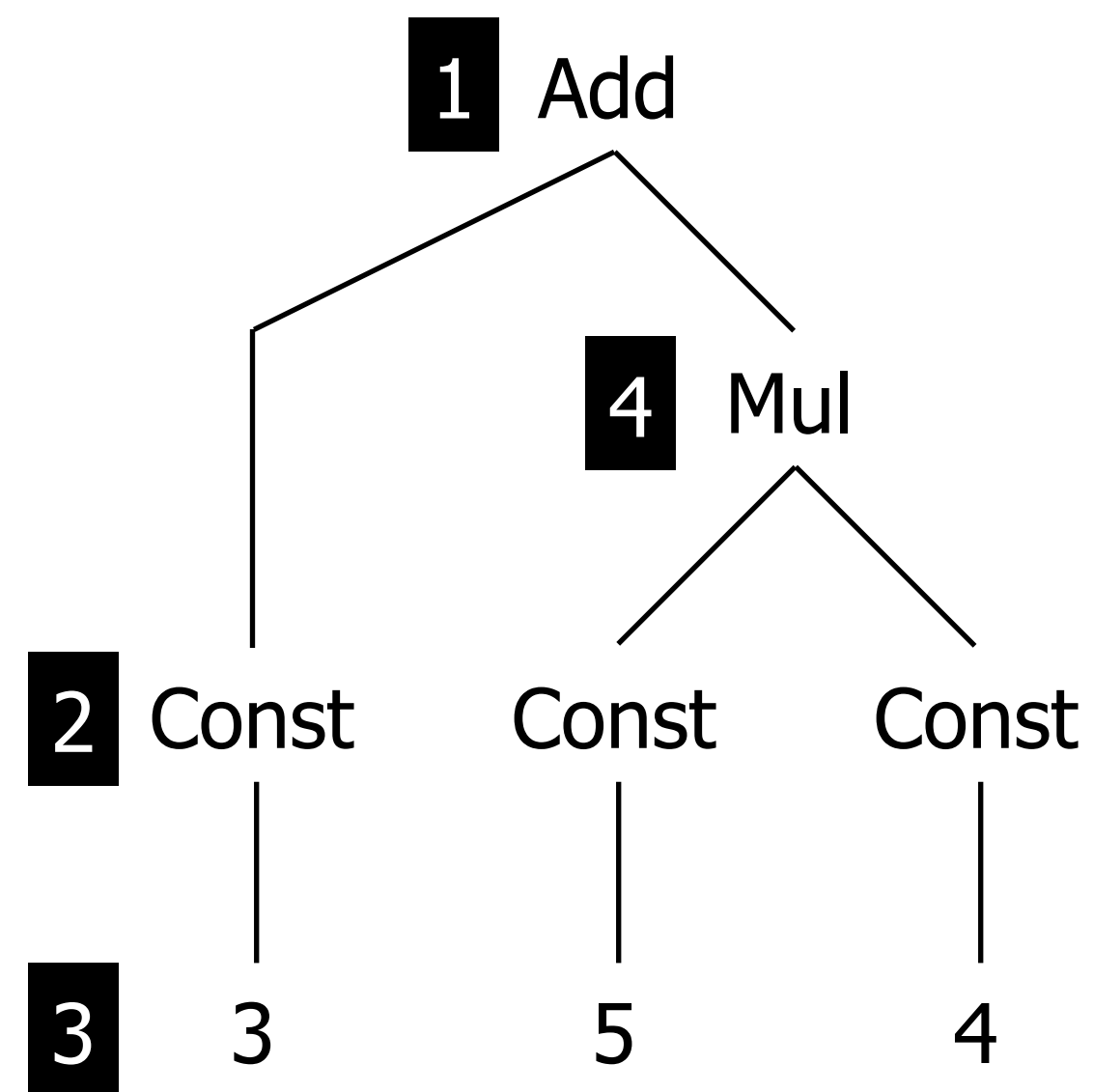




# Traversal: Bottomup

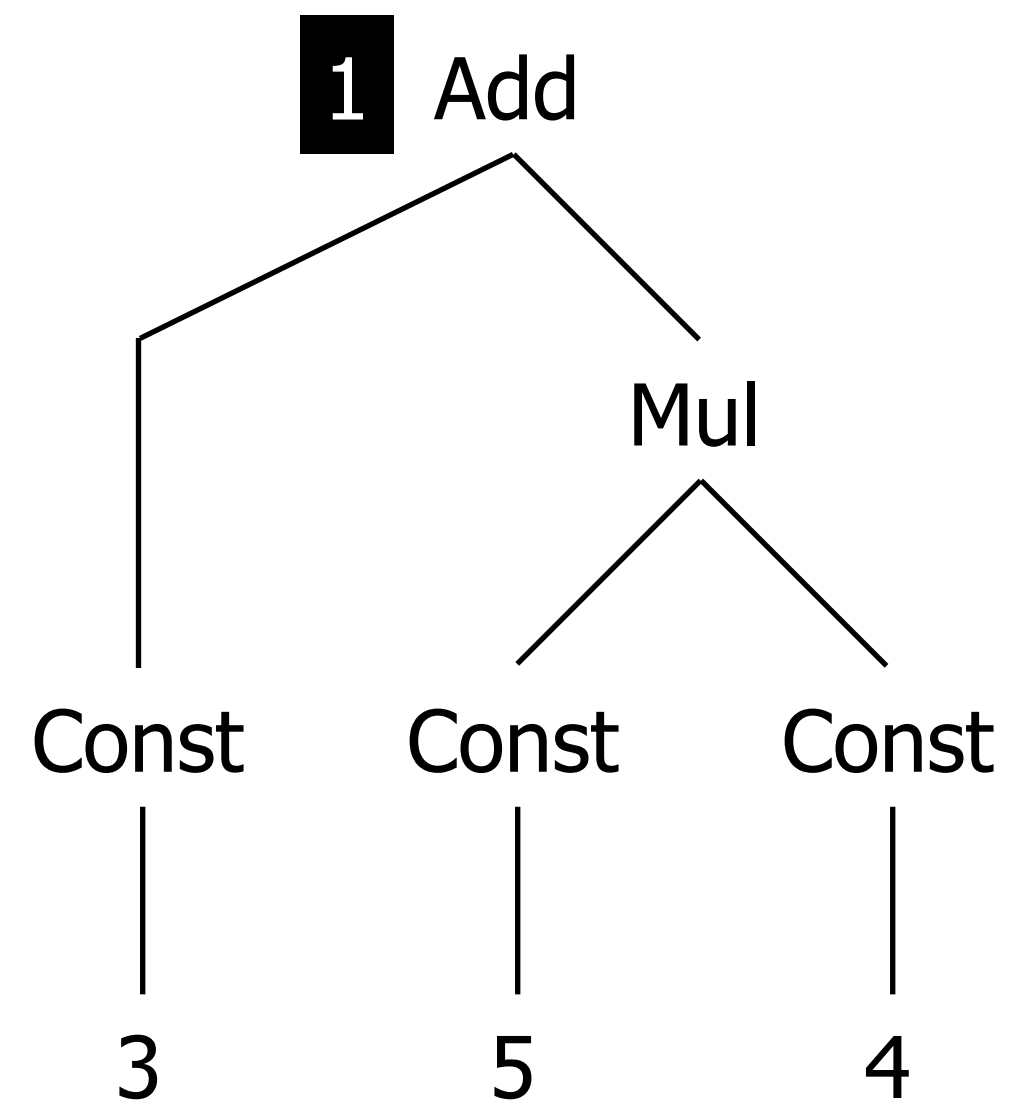
```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)

bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



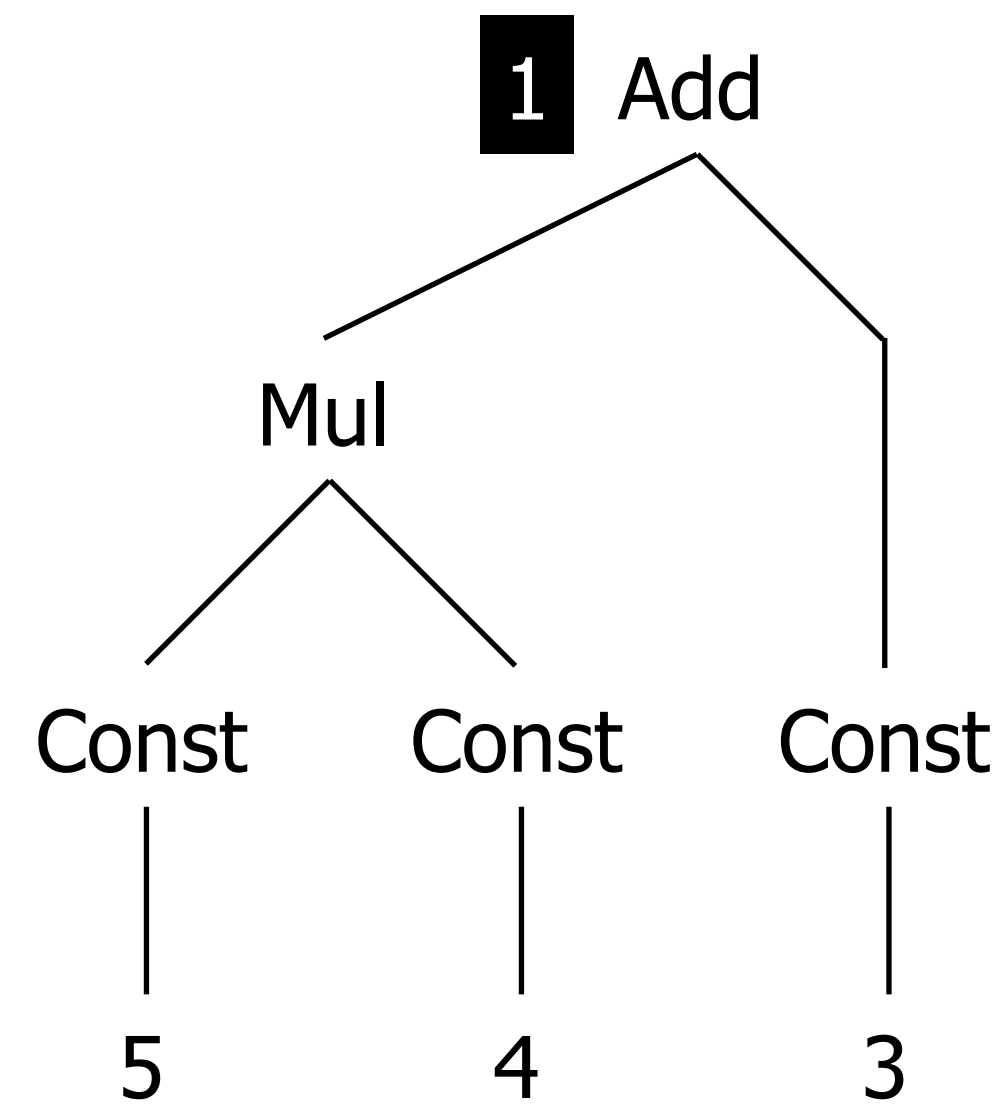
# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Traversal: Bottomup

```
switch: Add(e1, e2) -> Add(e2, e1)
switch: Mul(e1, e2) -> Mul(e2, e1)
bottomup(s) = all(bottomup(s)) ; s
bottomup(try(switch))
```



# Generic Traversal: Desugaring

## Example: Desugaring Expressions

```
DefAnd      : And(e1, e2) -> If(e1, e2, Int("0"))
DefPlus     : Plus(e1, e2) -> BinOp(PLUS(), e1, e2)
DesugarExp  = DefAnd <+ DefPlus <+ ...
desugar     = topdown(try(DesugarExp))
```

```
IfThen(
  And(Var("a"), Var("b")),
  Plus(Var("c"), Int("3")))
stratego> desugar
IfThen(
  If(Var("a"), Var("b"), Int("0")),
  BinOp(PLUS, Var("c"), Int("3")))
```

# Generic Traversal: Fixed-Point Traversal

## Fixed-point traversal

```
innermost(s) = bottomup(try(s; innermost(s)))
```

## Normalization

```
dnf = innermost(DAOL <+ DAOR <+ DN <+ DMA <+ DMO)  
cnf = innermost(DOAL <+ DOAR <+ DN <+ DMA <+ DMO)
```

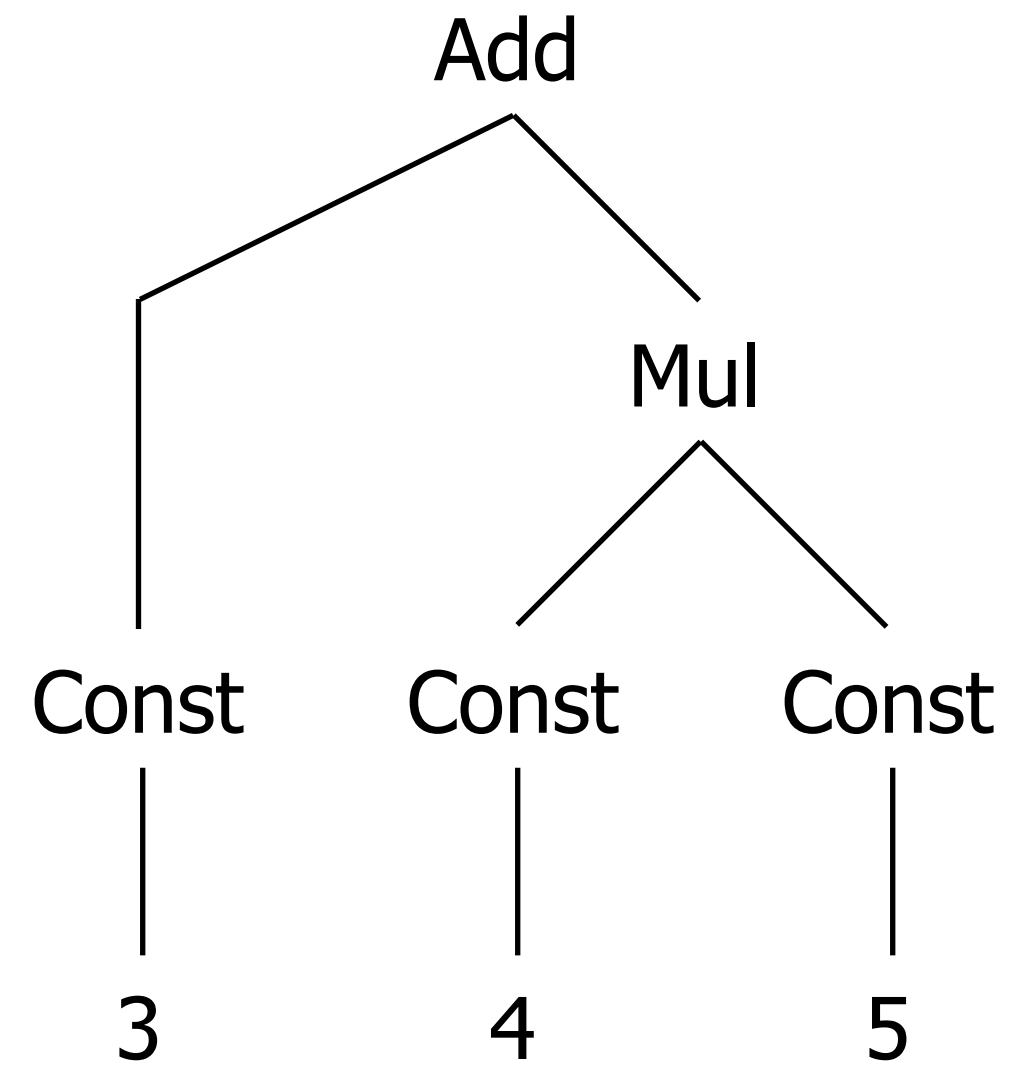
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



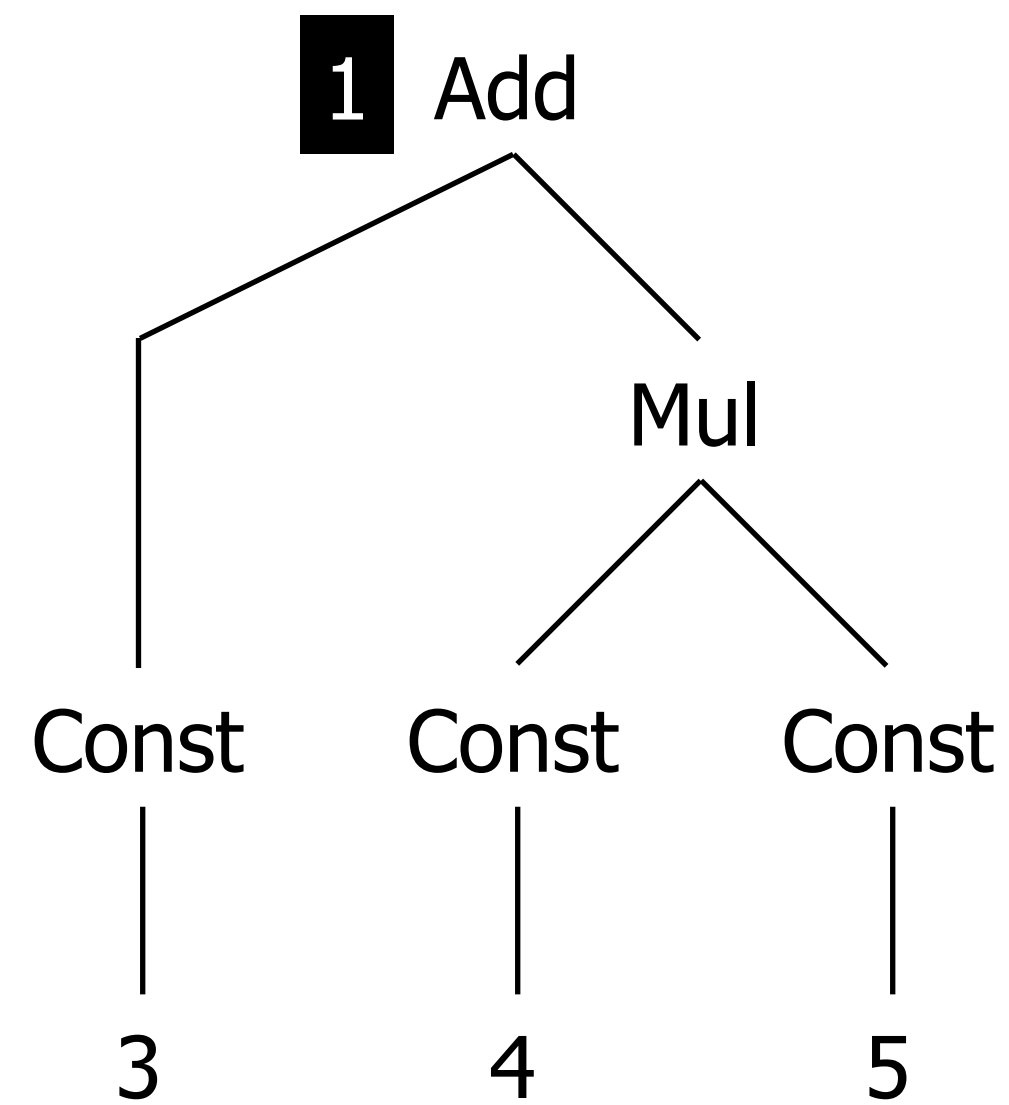
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



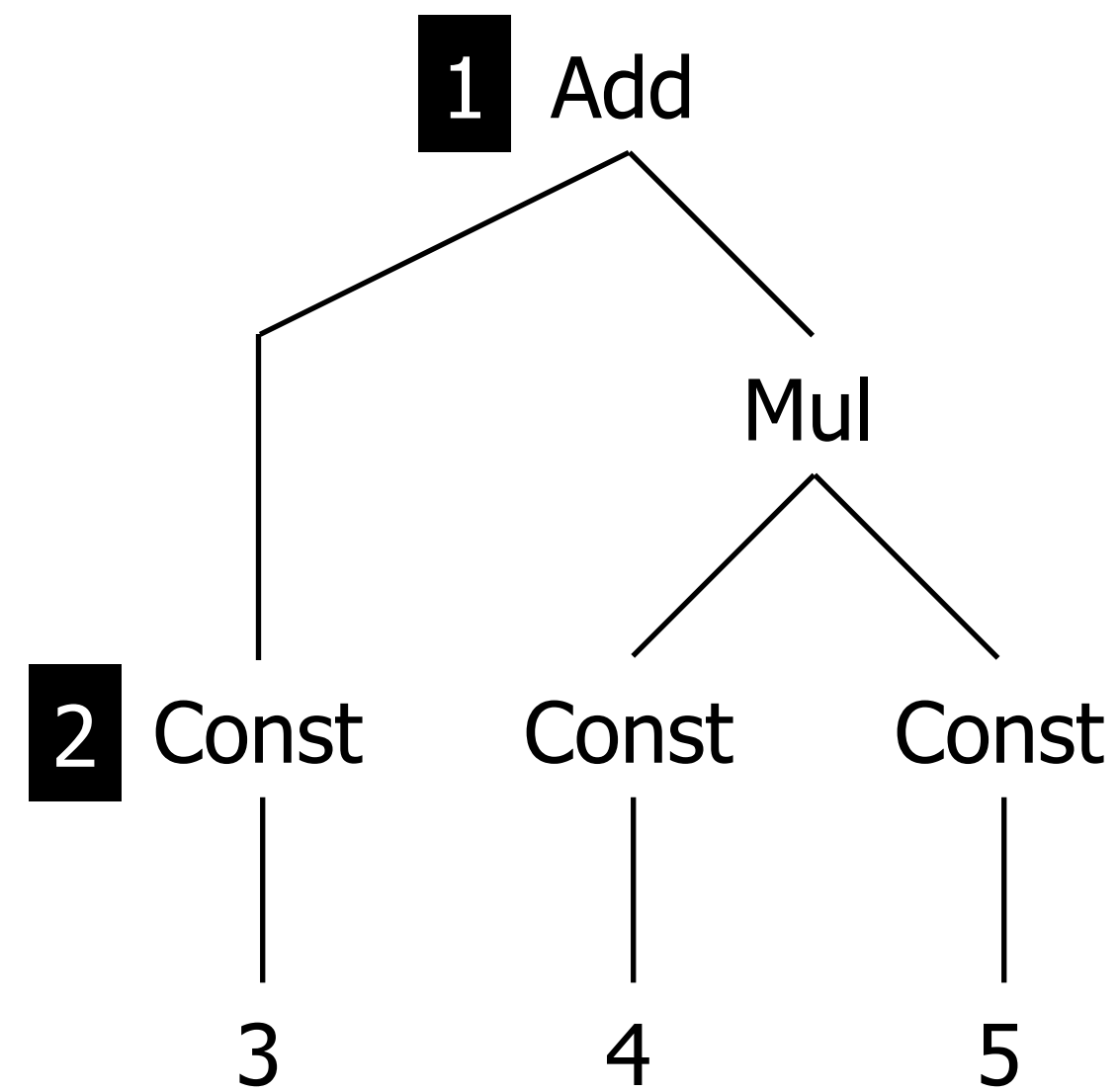
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```





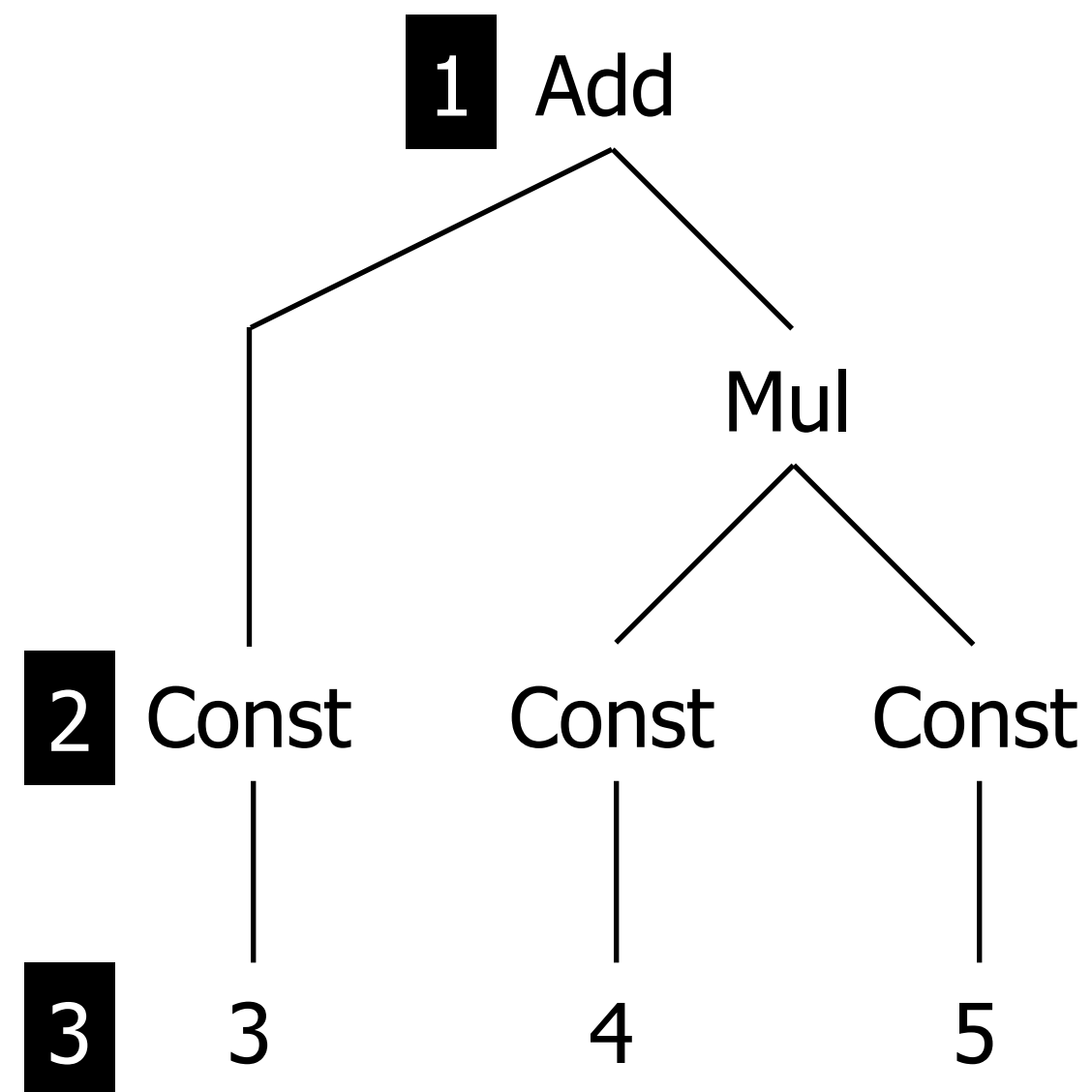
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



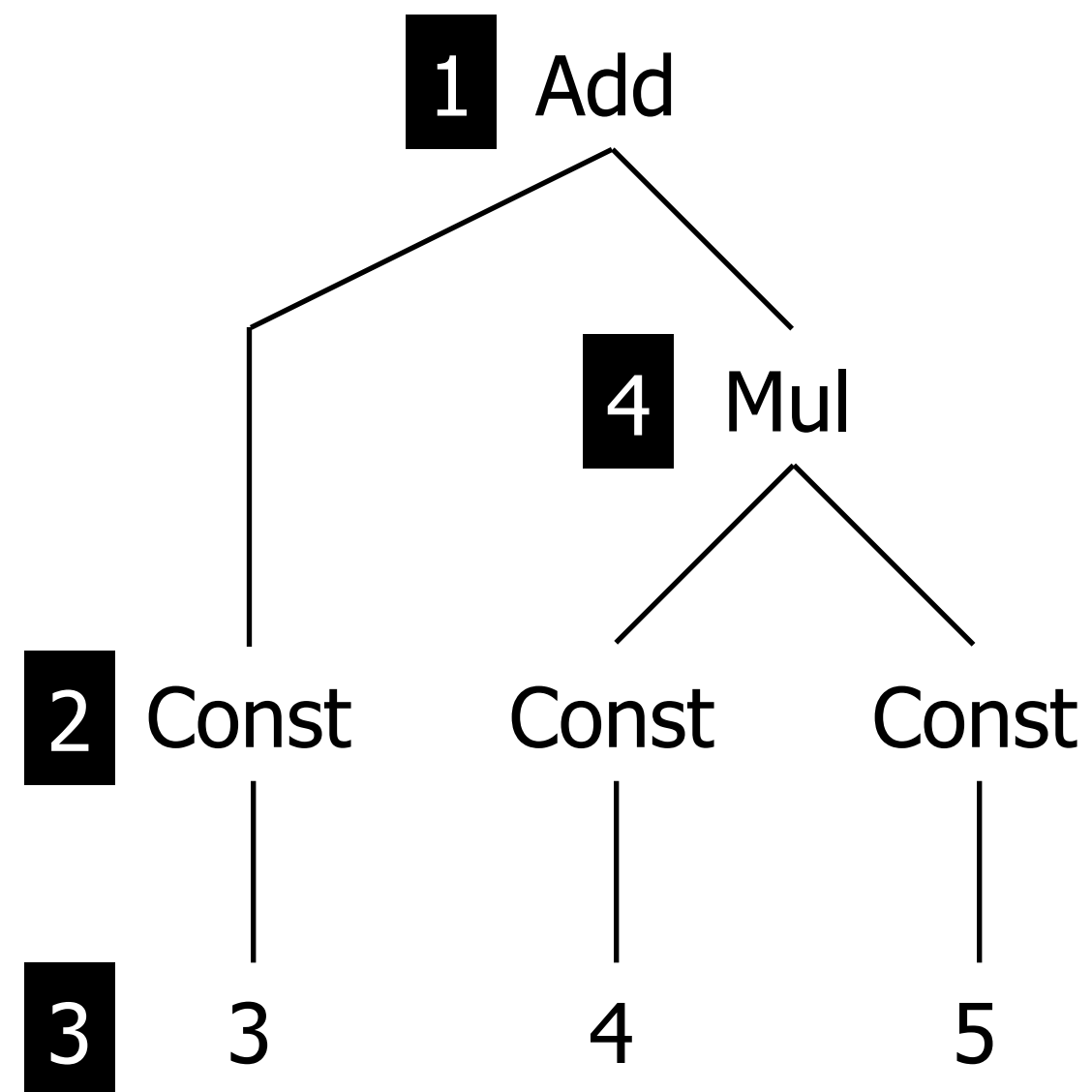
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



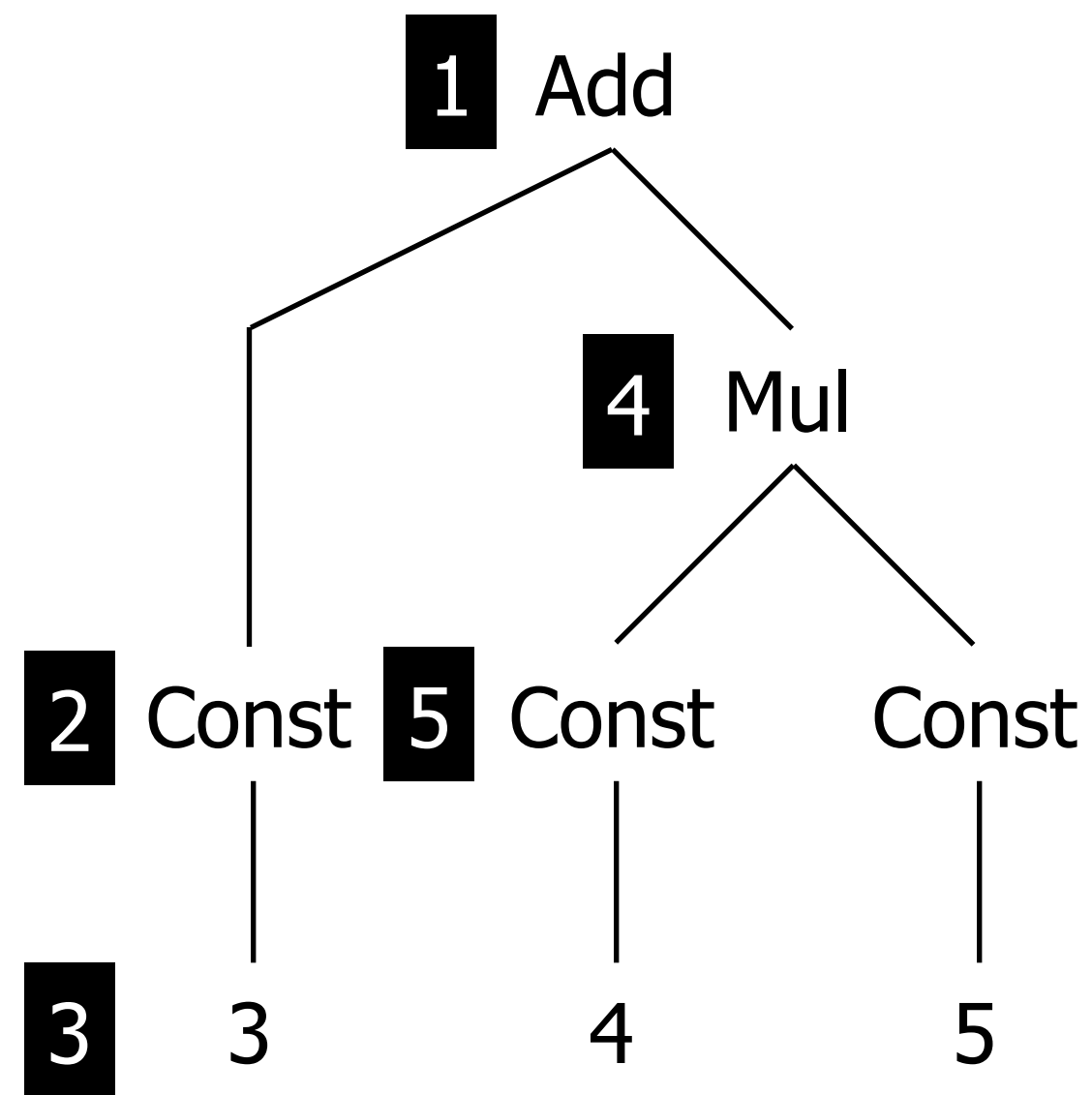
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



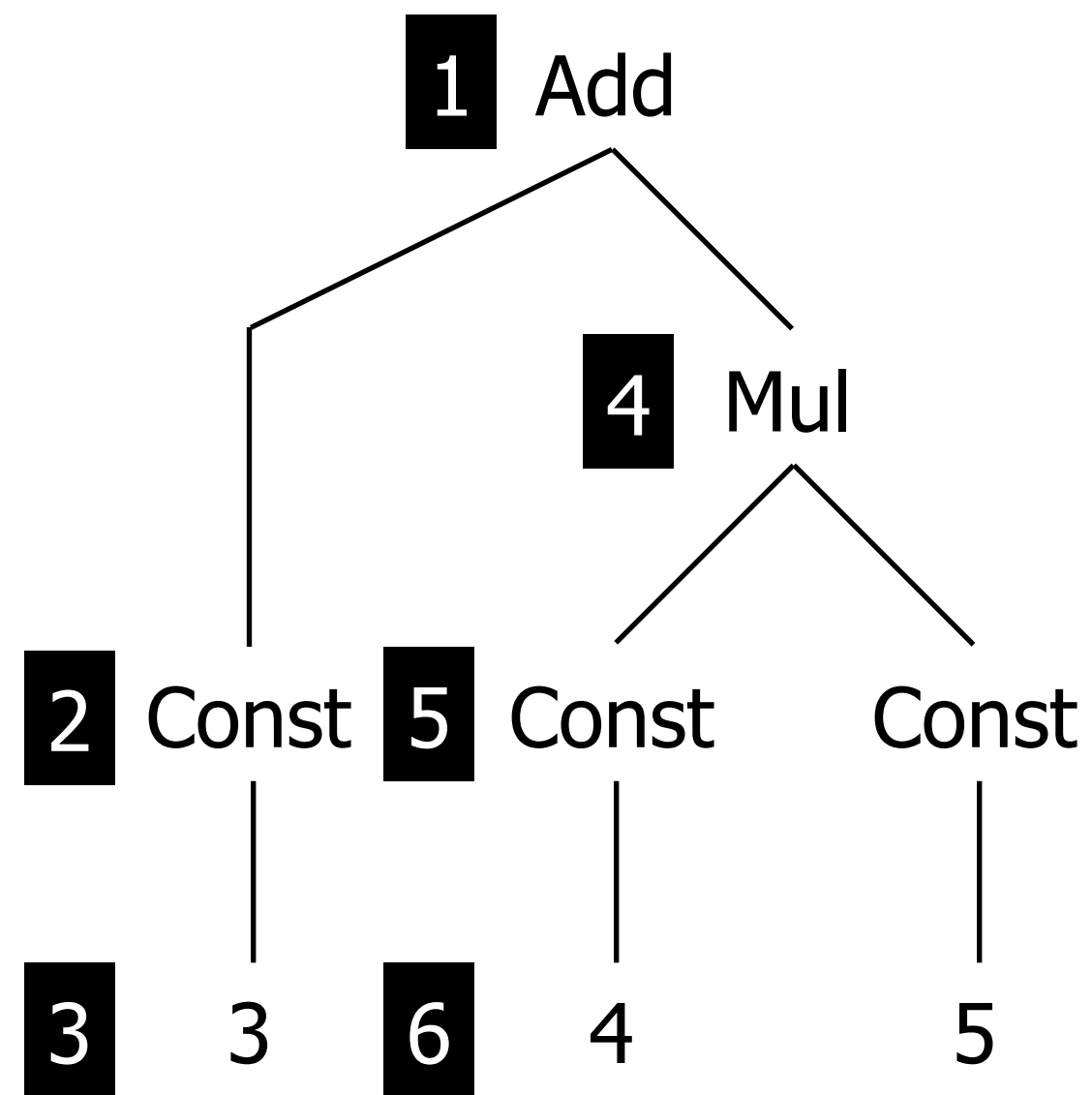
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



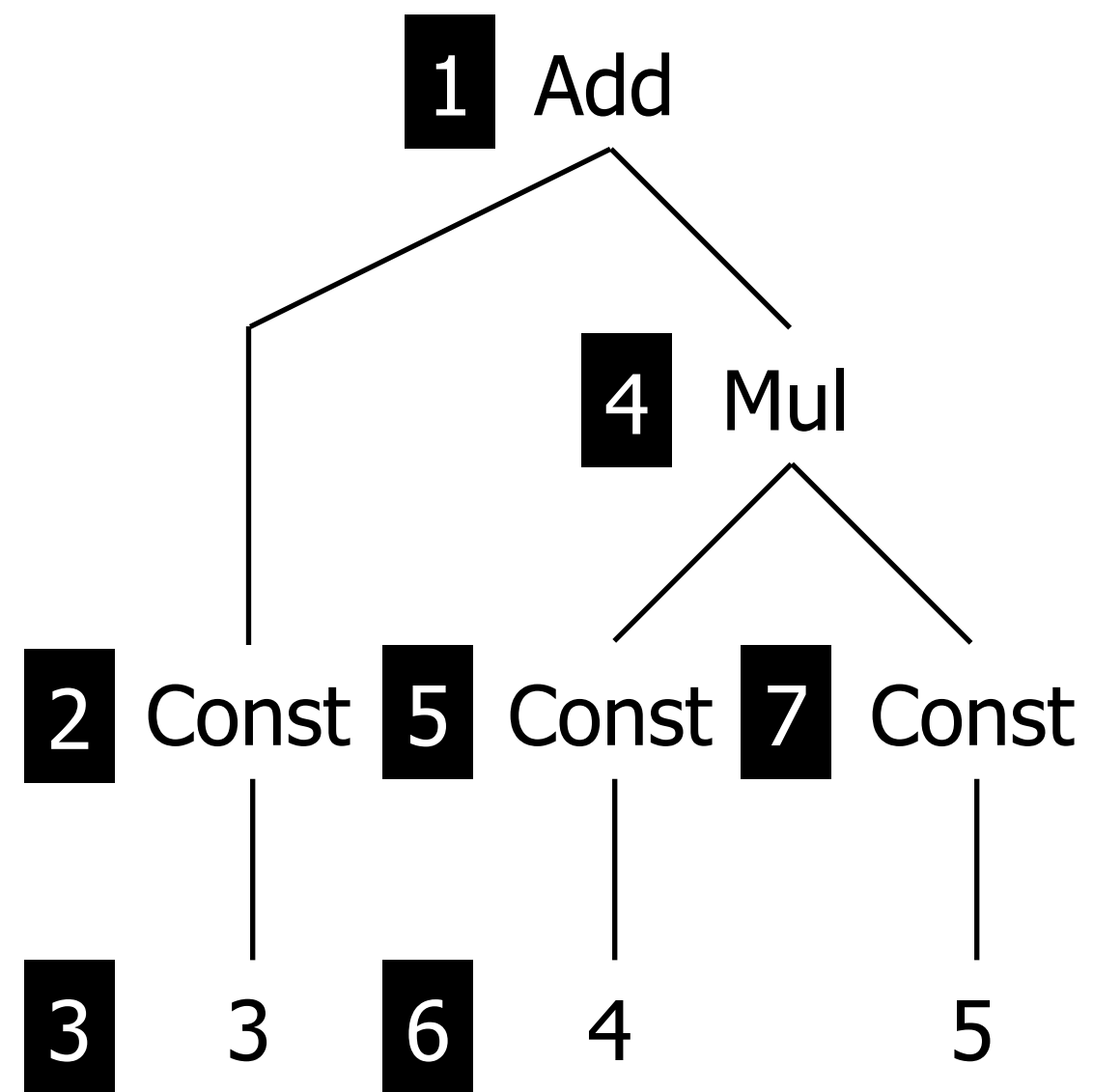
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



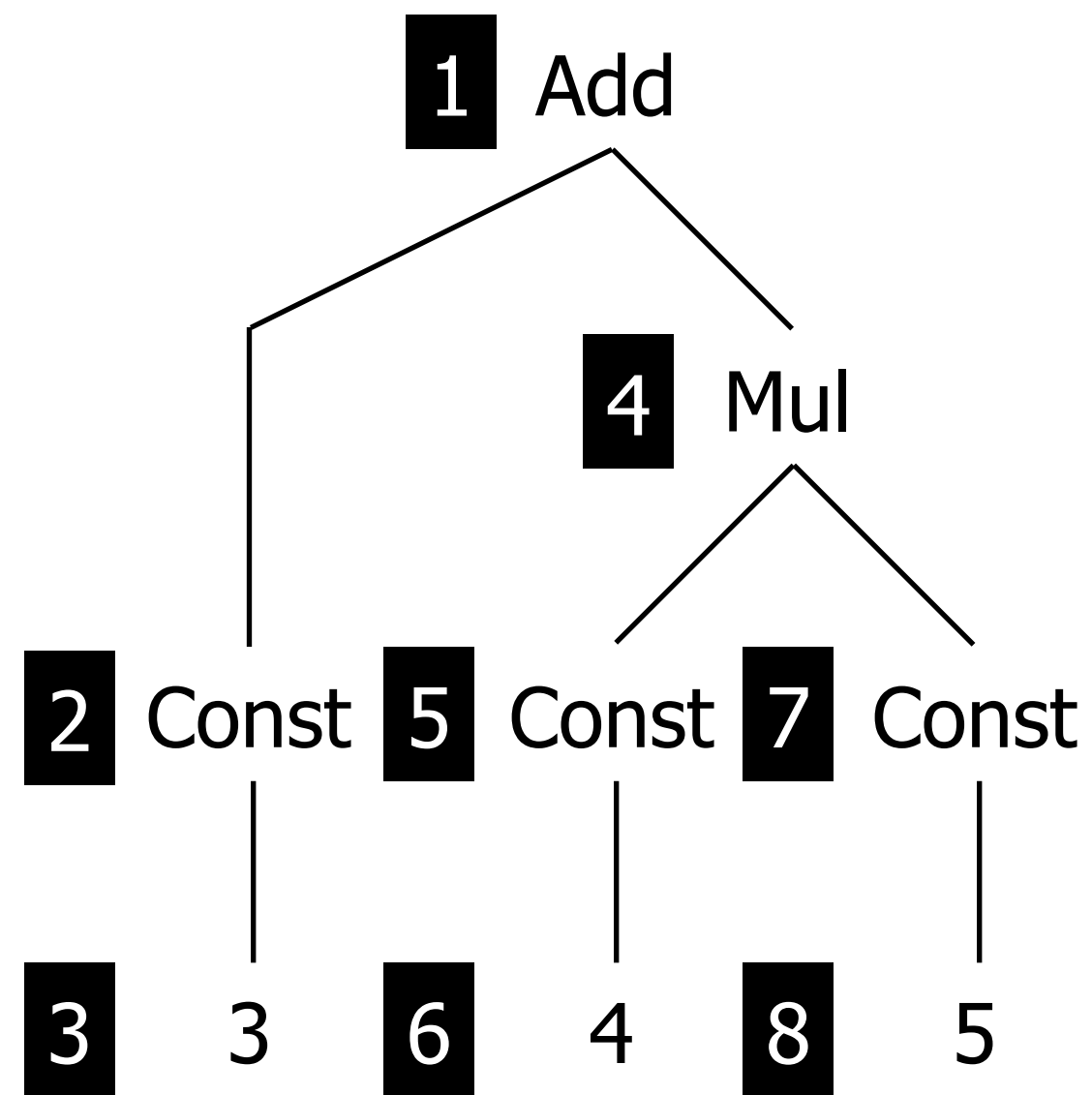
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



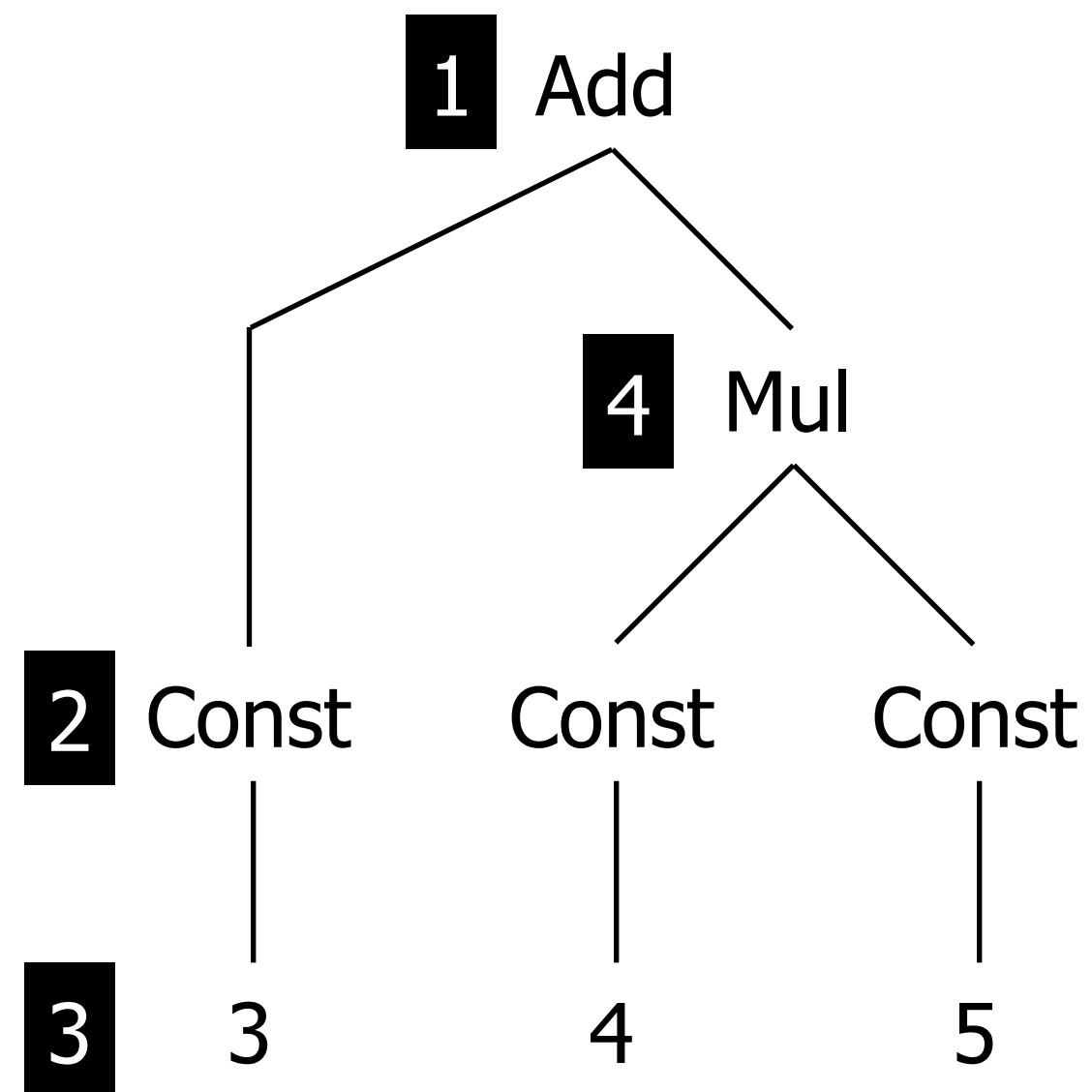
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



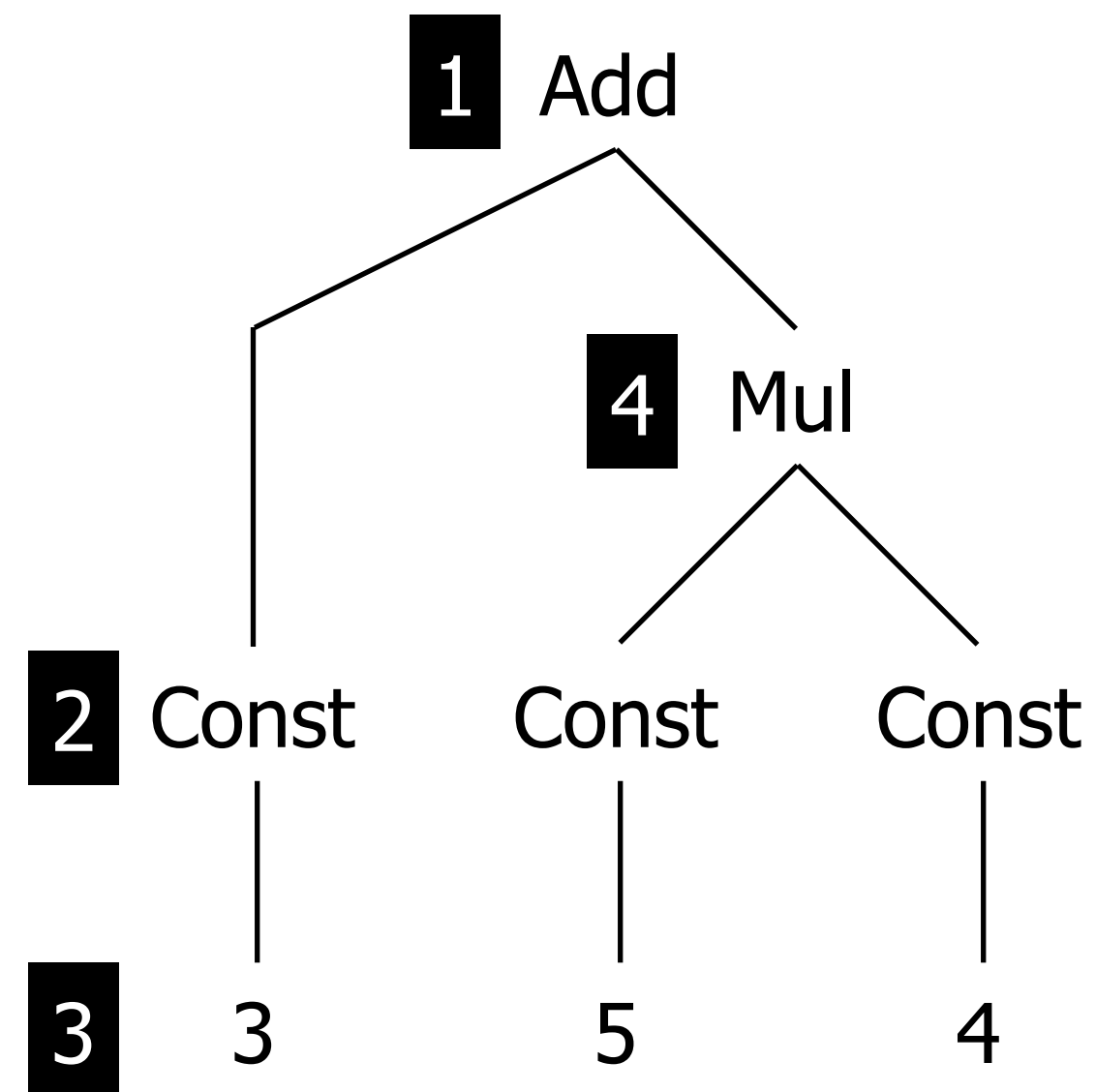
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```





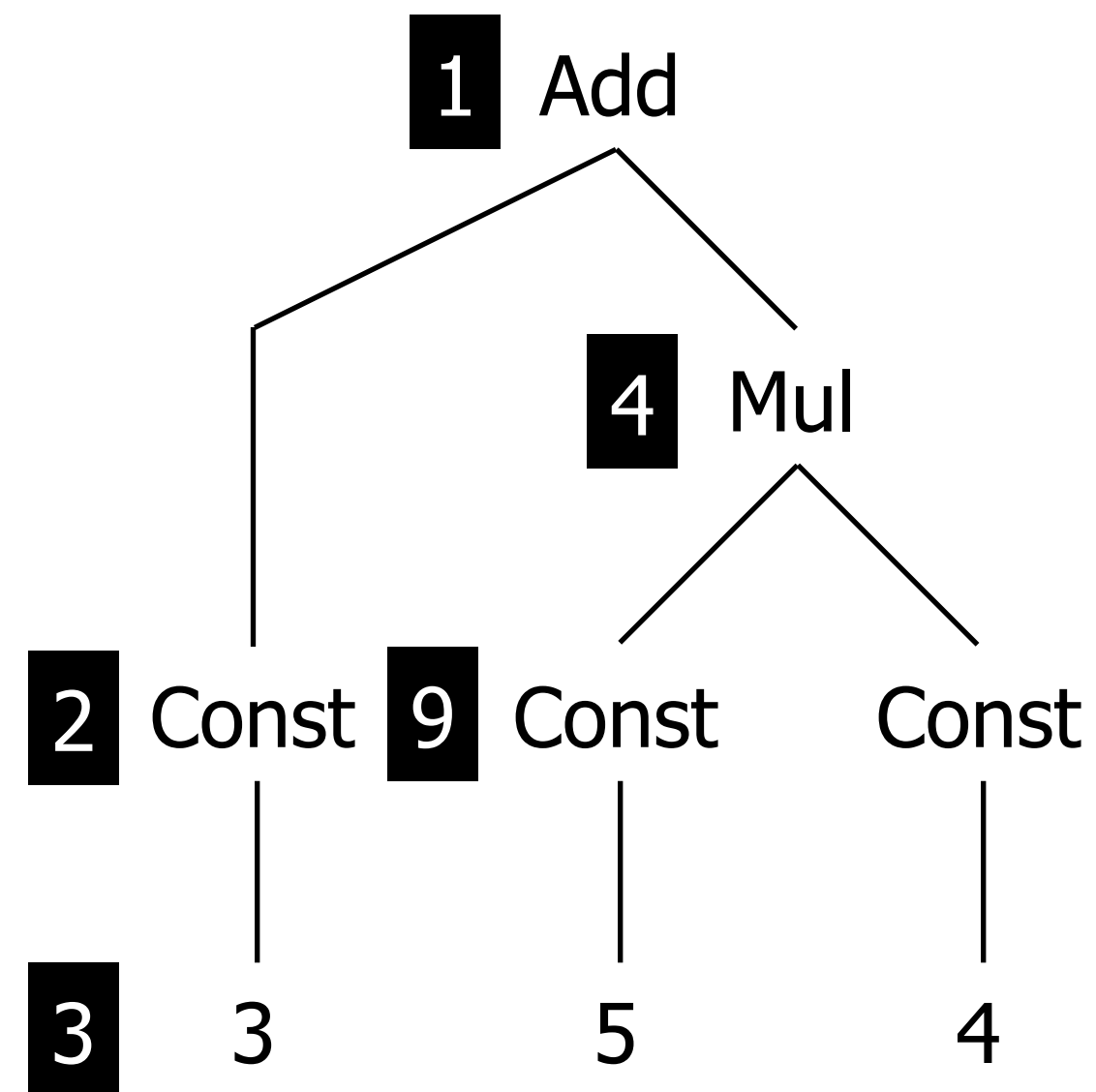
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



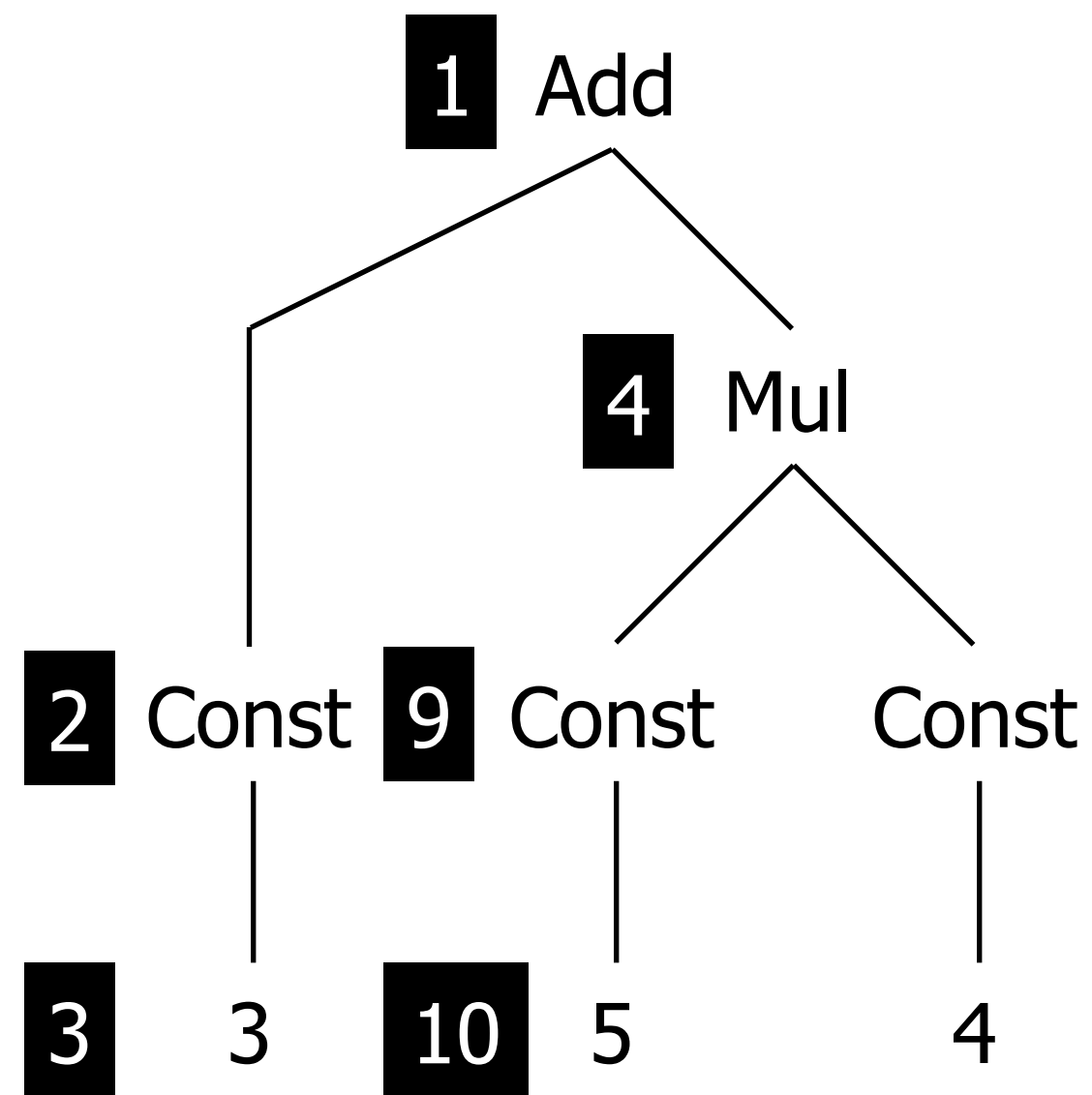
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



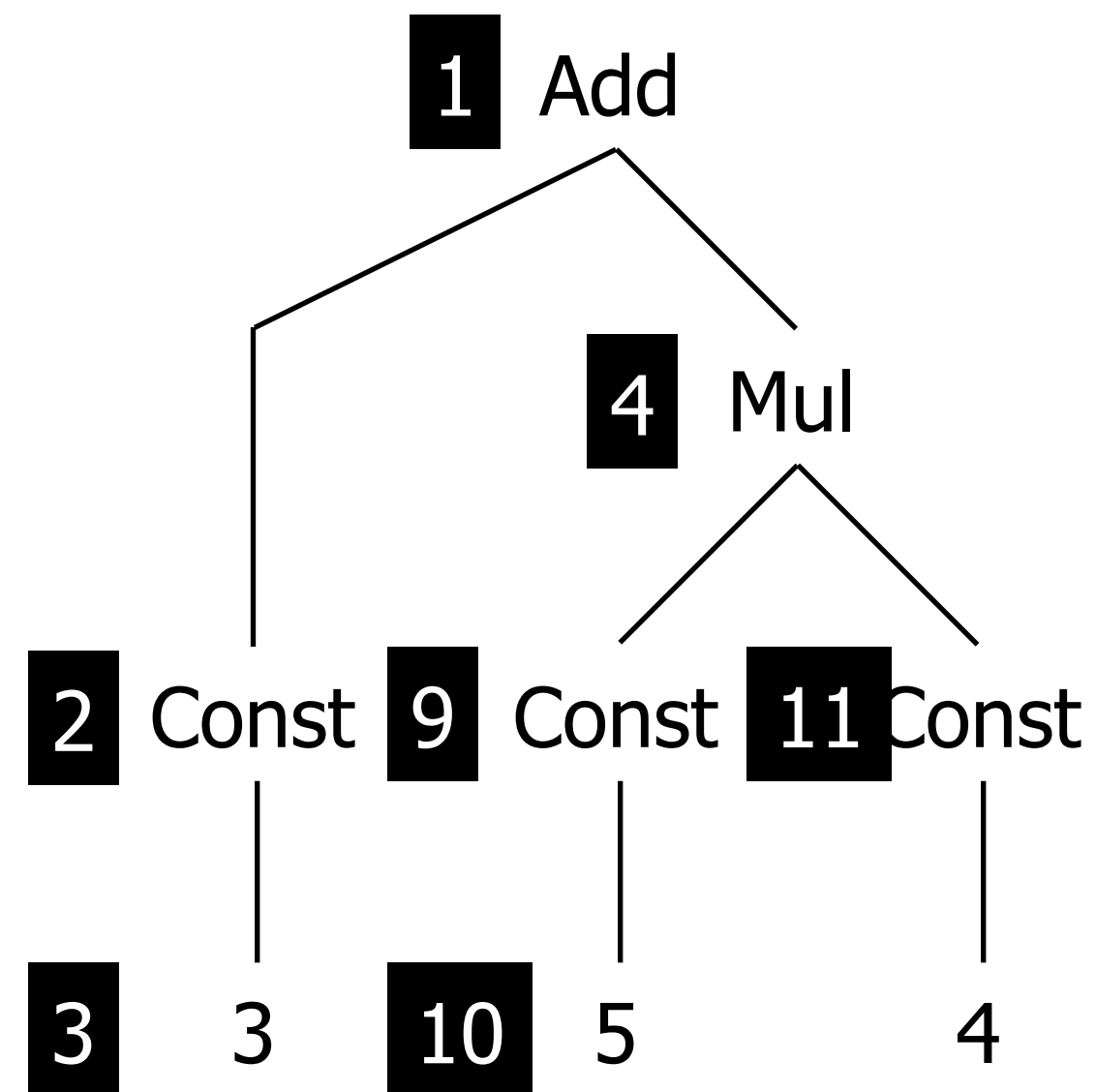
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



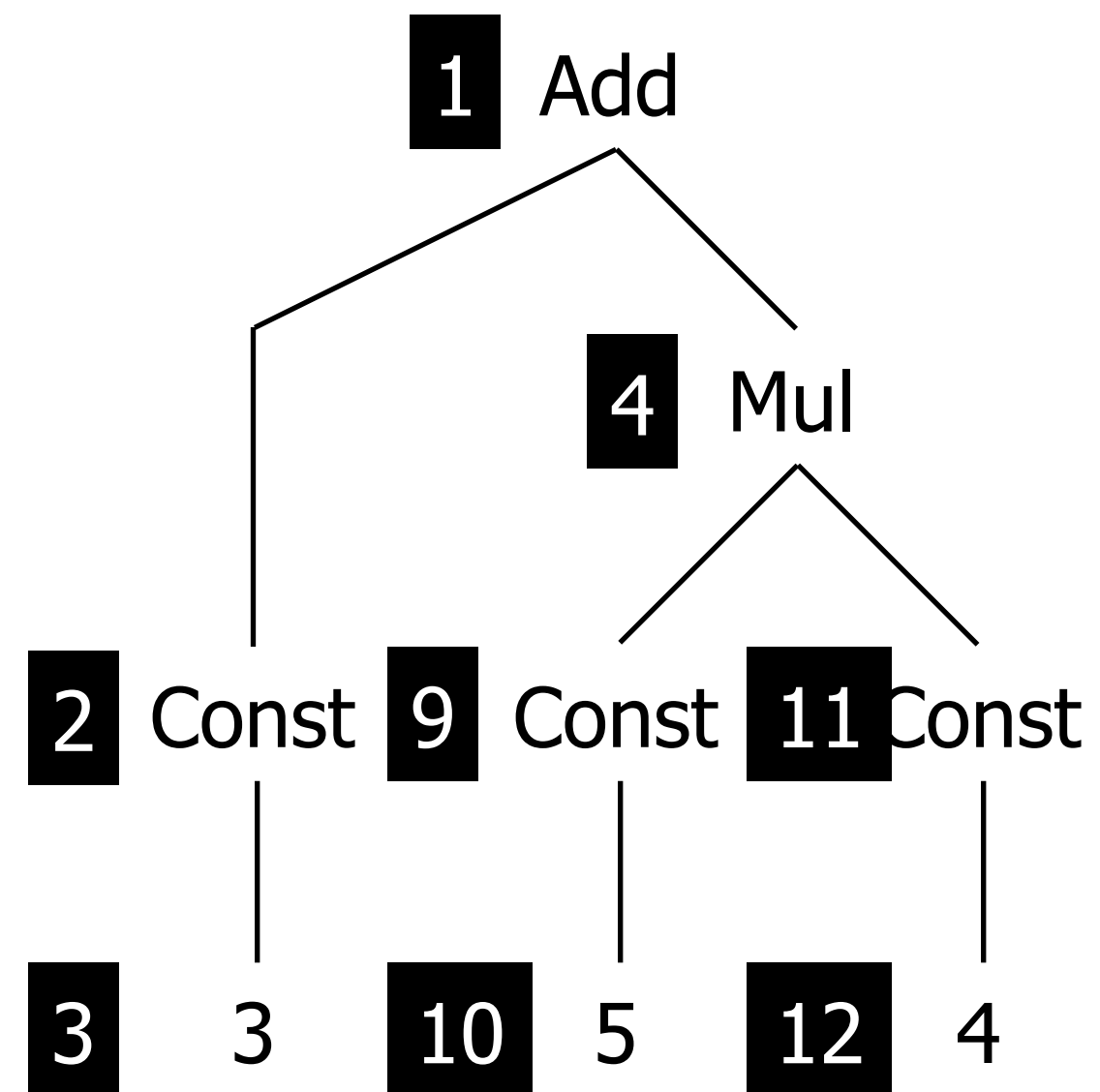
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



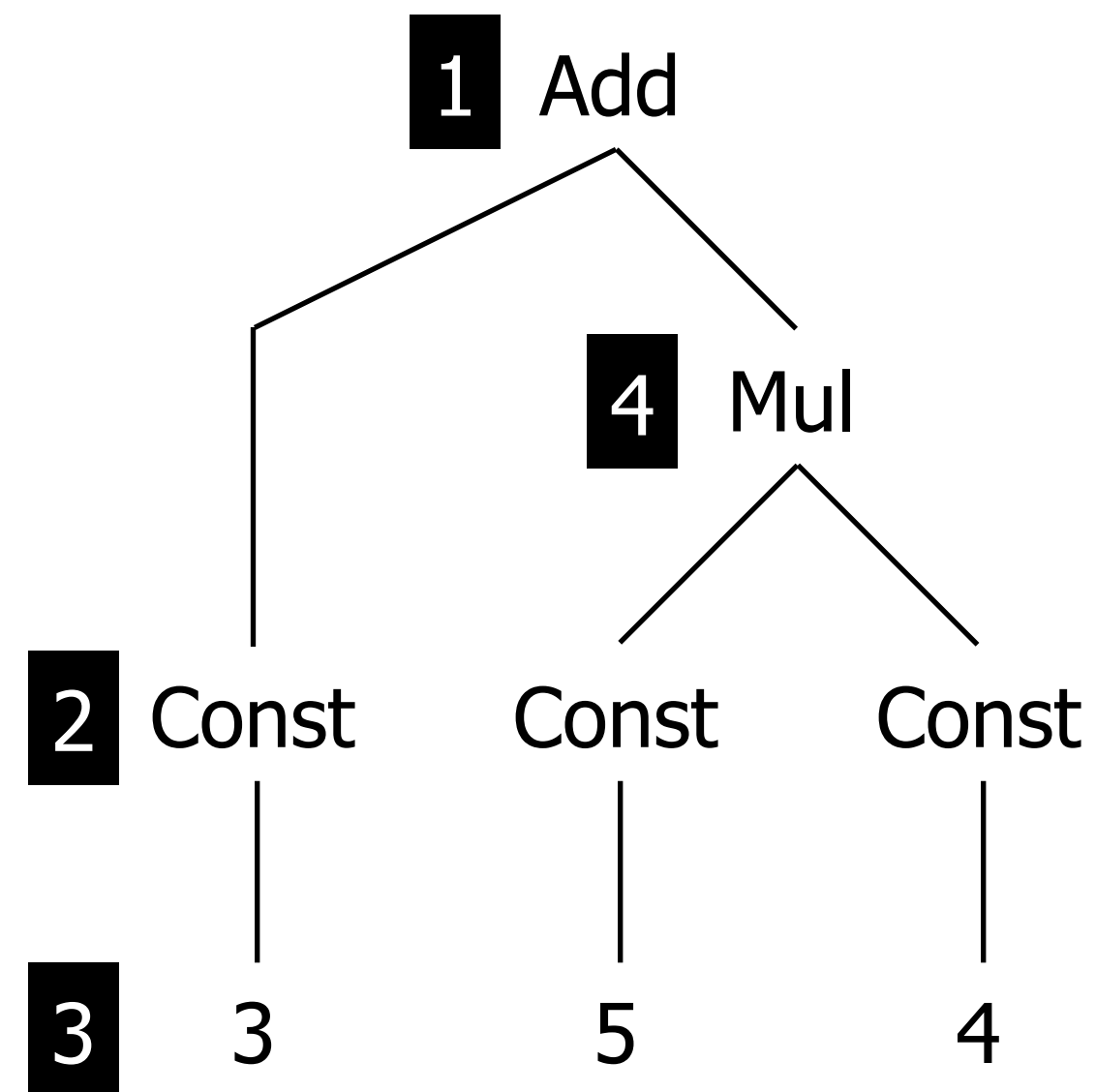
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



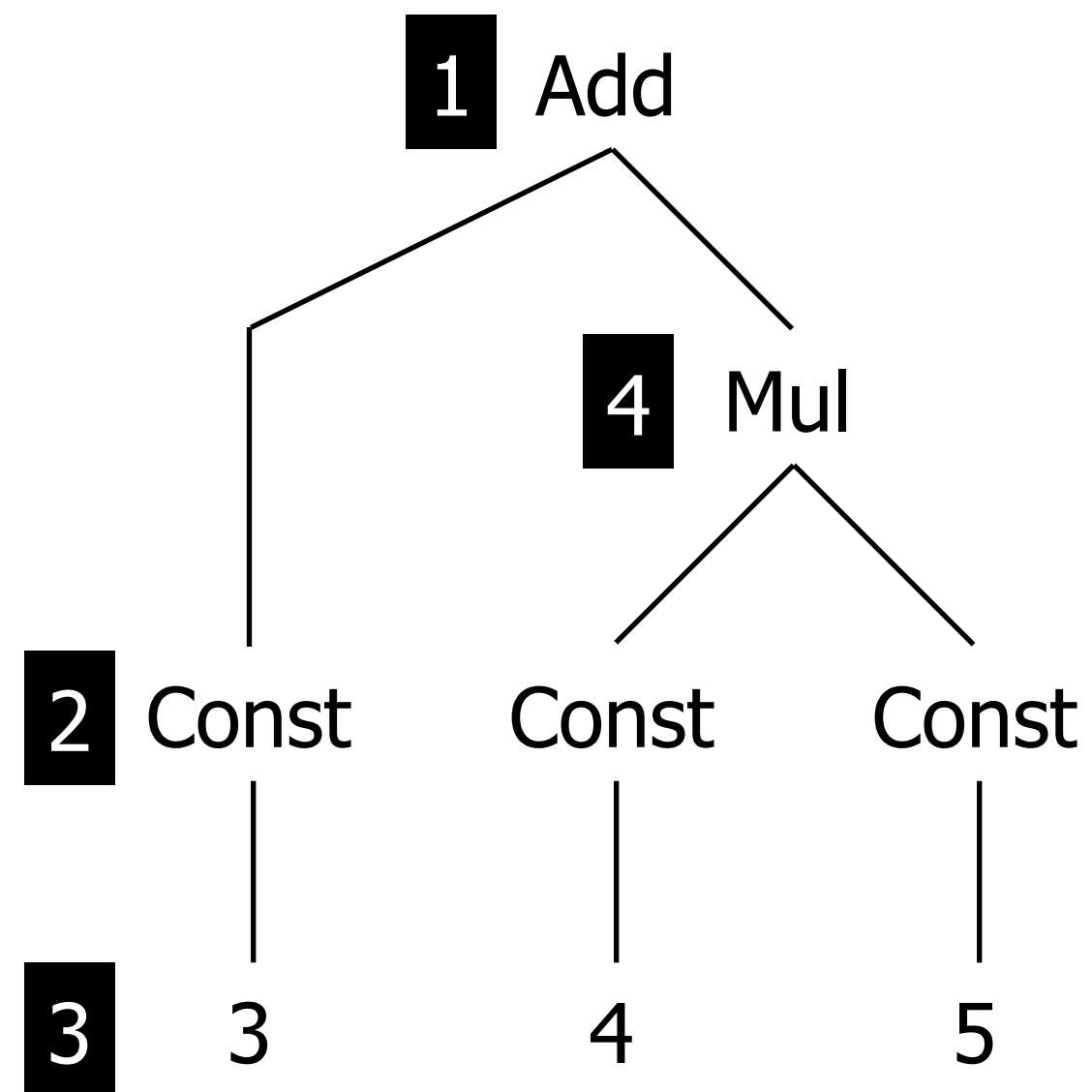
# Traversal: Innermost

```
switch: Add(e1, e2) -> Add(e2, e1)
```

```
switch: Mul(e1, e2) -> Mul(e2, e1)
```

```
innermost(s) = bottomup(try(s ; innermost(s)))
```

```
innermost(switch)
```



# Generic Traversal: One

## Visiting One Subterms

- Syntax: `one(s)`
- Apply strategy `s` to exactly one direct sub-terms

```
Plus(Int("14"), Int("3"))  
stratego> one(!Var("a"))  
Plus(Var("a"), Int("3"))
```

# Generic Traversal: One

## Visiting One Subterms

- Syntax: `one(s)`
- Apply strategy `s` to exactly one direct sub-terms

```
Plus(Int("14"), Int("3"))  
stratego> one(!Var("a"))  
Plus(Var("a"), Int("3"))
```

```
oncetd(s) = s <+ one(oncetd(s))  
oncebu(s) = one(oncebu(s)) <+ s  
spinetd(s) = s; try(one(spinetd(s)))  
spinebu(s) = try(one(spinebu(s))); s
```



# Generic Traversal: One

## Visiting One Subterms

- Syntax: `one(s)`
- Apply strategy `s` to exactly one direct sub-terms

```
Plus(Int("14"), Int("3"))  
stratego> one(!Var("a"))  
Plus(Var("a"), Int("3"))
```

```
oncetd(s) = s <+ one(oncetd(s))  
oncebu(s) = one(oncebu(s)) <+ s  
spinetd(s) = s; try(one(spinetd(s)))  
spinebu(s) = try(one(spinebu(s))); s
```

```
contains(|t) = oncetd(?t)
```

# Generic Traversal: One

## Visiting One Subterms

- Syntax: `one(s)`
- Apply strategy `s` to exactly one direct sub-terms

```
Plus(Int("14"), Int("3"))  
stratego> one(!Var("a"))  
Plus(Var("a"), Int("3"))
```

```
oncetd(s) = s <+ one(oncetd(s))  
oncebu(s) = one(oncebu(s)) <+ s  
spinetd(s) = s; try(one(spinetd(s)))  
spinebu(s) = try(one(spinebu(s))); s
```

```
contains(|t) = oncetd(?t)
```

```
reduce(s) = repeat(rec x(one(x) + s))  
outermost(s) = repeat(oncetd(s))  
innermostI(s) = repeat(oncebu(s))
```

## Visiting some subterms (but at least one)

- Syntax: `some(s)`
- Apply strategy `s` to as many direct subterms as possible, and at least one

```
Plus(Int("14"), Int("3"))  
stratego> some(?Int(_); !Var("a"))  
Plus(Var("a"), Var("a"))
```

## One-pass traversals

```
sometd(s) = s <+ some(sometd(s))  
somebu(s) = some(somebu(s)) <+ s
```

## Fixed-point traversal

```
reduce-par(s) = repeat(rec x(some(x) + s))
```

## Summary

- Tangling of rules and strategy (traversal) considered harmful
- Separate traversal from rules
- One-level traversal primitives allow wide range of traversals

# Type-Unifying Transformations

# Type Preserving vs Type Unifying

## Transformations are type preserving

- Structural transformation
- Types stay the same
- Application: transformation
- Examples: simplification, optimization, ...

## Collections are type unifying

- Terms of different types mapped onto one type
- Application: analysis
- Examples: free variables, uncaught exceptions, call-graph

# Example Problems

`term-size`

- Count the number of nodes in a term

`occurrences`

- Count number of occurrences of a subterm in a term

`collect-vars`

- Collect all variables in expression

`free-vars`

- Collect all *free* variables in expression

`collect-uncaught-exceptions`

- Collect all *uncaught* exceptions in a method

# List Implementation: Size (Number of Nodes)

Replacing Nil by s1 and Cons by s2

```
foldr(s1, s2) =  
  []; s1 <+ \ [y|ys] -> <s2>(y, <foldr(s1, s2)> ys) \
```

Add the elements of a list of integers

```
sum = foldr(!0, add)
```

Fold and apply f to the elements of the list

```
foldr(s1, s2, f) =  
  []; s1 <+ \ [y|ys] -> <s2>(<f>y, <foldr(s1, s2, f)>ys) \
```

Length of a list

```
length = foldr(!0, add, !1)
```



# List Implementation: Number of Occurrences

Number of occurrences in a list

```
list-occurrences(s) = foldr(!0, add, s < !1 + !0)
```

Number of local variables in a list

```
list-occurrences(?ExprName(_))
```

# List Implementation: Collect Terms

Filter elements in a list for which `s` succeeds

```
filter(s) = [] + [s | filter(s)] <+ ?[_|<filter(s)>]
```

Collect local variables in a list

```
filter(ExprName(id))
```

Collect local variables in first list, exclude elements in second list

```
(filter(ExprName(id)), id); diff
```

# Folding Expressions

Generalize folding of lists to arbitrary terms

Example: Java expressions

```
fold-exp(plus, minus, assign, cond, ...) =  
  rec f(  
    \ Plus(e1, e2) -> <plus>(<f>e1, <f>e2) \  
  + \ Minus(e1, e2) -> <minus>(<f>e1, <f>e2) \  
  + \ Assign(lhs, e) -> <assign>(<f>lhs, <f>e) \  
  + \ Cond(e1, e2, e3) -> <cond>(<f>e1, <f>e2, <f>e3) \  
  + ...  
  )
```

# Term-Size with Fold

```
term-size =  
  fold-exp(MinusSize, PlusSize, AssignSize, ...)  
  
MinusSize :  
  Minus(e1, e2) -> <add> (1, <add> (e1, e2))  
  
PlusSize :  
  Plus(e1, e2) -> <add> (1, <add> (e1, e2))  
  
AssignSize :  
  Assign(lhs, e) -> <add> (1, <add> (lhs, e))  
  
// etc.
```

## Definition of fold

- One parameter for each constructor
- Define traversal for each constructor

## Instantiation of fold

- One rule for each constructor
- Default behaviour not generically specified

# Defining Fold with Generic Traversal

## Fold is bottomup traversal:

```
fold-exp(s) =  
  bottomup(s)
```

```
term-size =  
  fold-exp(MinusSize <+ PlusSize <+ AssignSize <+ ...)
```

## Definition of fold

- Recursive application to subterms defined generically
- One parameter: rules combined with choice

**Instantiation: default behaviour not generically specified**

# Generic Term Deconstruction (1)

## Specific definitions

MinusSize :

$$\text{Minus}(e1, e2) \rightarrow \langle \text{add} \rangle (1, \langle \text{add} \rangle (e1, e2))$$

AssignSize :

$$\text{Assign}(lhs, e) \rightarrow \langle \text{add} \rangle (1, \langle \text{add} \rangle (lhs, e))$$

## Generic definition

CSize :

$$C(e1, e2, \dots) \rightarrow \langle \text{add} \rangle (1, \langle \text{add} \rangle (e1, \langle \text{add} \rangle (e2, \dots)))$$

**Requires generic decomposition of constructor application**

# Generic Term Deconstruction (2)

## Generic Term Deconstruction

- Syntax:  $?p_1\#(p_2)$
- Semantics: when applied to a term  $c(t_1, \dots, t_n)$  matches
  - "c" against  $p_1$
  - $[t_1, \dots, t_n]$  against  $p_2$
- Decompose constructor application into its constructor name and list of direct subterms

```
Plus(Lit(Deci("1")), ExprName(Id("x")))
stratego> ?c#(xs)
stratego> :binding c
variable c bound to "Plus"
stratego> :binding xs
variable xs bound to [Lit(Deci("1")), ExprName(Id("x"))]
```



## Definition of Crush

```
crush(nul, sum, s) :  
  _#(xs) -> <foldr(nul, sum, s)> xs
```

## Applications of Crush

```
node-size =
```

```
term-size =
```

```
om-occurrences(s) =
```

```
occurrences(s) =
```

## Definition of Crush

```
crush(nul, sum, s) :  
  _#(xs) -> <foldr(nul, sum, s)> xs
```

## Applications of Crush

```
node-size =  
  crush(!0, add, !1)
```

```
term-size =
```

```
om-occurrences(s) =
```

```
occurrences(s) =
```

## Definition of Crush

```
crush(nul, sum, s) :  
  _#(xs) -> <foldr(nul, sum, s)> xs
```

## Applications of Crush

```
node-size =  
  crush(!0, add, !1)  
  
term-size =  
  crush(!1, add, term-size)  
  
om-occurrences(s) =  
  
occurrences(s) =
```

## Definition of Crush

```
crush(nul, sum, s) :  
  _#(xs) -> <foldr(nul, sum, s)> xs
```

## Applications of Crush

```
node-size =  
  crush(!0, add, !1)  
  
term-size =  
  crush(!1, add, term-size)  
  
om-occurrences(s) =  
  if s then !1 else crush(!0, add, om-occurrences(s)) end  
  
occurrences(s) =
```

## Definition of Crush

```
crush(nul, sum, s) :  
  _#(xs) -> <foldr(nul, sum, s)> xs
```

## Applications of Crush

```
node-size =  
  crush(!0, add, !1)  
  
term-size =  
  crush(!1, add, term-size)  
  
om-occurrences(s) =  
  if s then !1 else crush(!0, add, om-occurrences(s)) end  
  
occurrences(s) =  
  <add> (<if s then !1 else !0 end>,  
        <crush(!0, add, occurrences(s))>)
```

# McCabe's cyclomatic complexity

```
public class Metric {  
    public int foo() {  
        if(1 > 2)  
            return 0;  
        else  
            if(3 < 4)  
                return 1;  
            else  
                return 2;  
        if(5 > 6)  
            return 3;  
    }  
  
    public int bar() {  
        for(int i=0; i<5; i++) {}  
    }  
}
```

# McCabe's cyclomatic complexity

- Computes the number of decision points in a function.
- Measure of minimum number of execution paths.
- Each control flow construct introduces another possible path.

```
cyclomatic-complexity =  
  occurrences(is-control-flow)  
; inc
```

```
is-control-flow =  
  ?If( _, _ )  
  <+ ?If( _, _, _ )  
  <+ ?While( _, _ )  
  <+ ?For( _, _, _ , _ )  
  <+ ?SwitchGroup( _, _ )
```

# NPATH complexity

```
public class Metric {  
    public int foo() {  
        if(1 > 2)  
            return 0;  
        else  
            if(3 < 4)  
                return 1;  
            else  
                return 2;  
        if(5 > 6)  
            return 3;  
    }  
  
    public int bar() {  
        for(int i=0; i<5; i++) {}  
    }  
}
```



## Complexity Analysis Algorithm (improved)

- Number of acyclic execution paths (not just nodes)
- Want to take into account the nesting of the control flow statements.
- Cost of a given control flow construct depends on its nesting level.

# NPATH complexity: Implementation

```
npath-complexity =
  rec rec (
    ?Block(<map(rec)>)
    ; foldr(!1, mul)
  <+ {extra:
    is-control-flow
    ; where(extra := <AddPaths <+ !0>)
    ; crush(!0, add, rec)
    ; <add> (<id>, extra)
  }
  <+ is-BlockStm ; !1
  <+ crush(!0, add, rec)
  )
```

AddPaths: If(\_, \_) -> 1

AddPaths: While(\_, \_) -> 1

AddPaths: For(\_, \_, \_, \_) -> 1

# Collect

Collect all (outermost) sub-terms for which `s` succeeds

```
collect(s) =
```

Collect all sub-terms for which `s` succeeds

```
collect-all(s) =
```

Collect all local variables in an expression

```
get-exprnames = collect(ExprName(id))
```

# Collect

Collect all (outermost) sub-terms for which `s` succeeds

```
collect(s) =  
  ! [<s>] <+ crush(! [], union, collect(s))
```

Collect all sub-terms for which `s` succeeds

```
collect-all(s) =
```

Collect all local variables in an expression

```
get-exprnames = collect(ExprName(id))
```

Collect all (outermost) sub-terms for which  $s$  succeeds

```
collect(s) =  
  ! [<s>] <+ crush(! [], union, collect(s))
```

Collect all sub-terms for which  $s$  succeeds

```
collect-all(s) =  
  ! [<s> | <crush(! [], union, collect-all(s))>]  
  <+ crush(! [], union, collect-all(s))
```

Collect all local variables in an expression

```
get-exprnames = collect(ExprName(id))
```

# Uncaught Exceptions (1)

## Collect all uncaught exceptions

- Collect thrown exceptions
- Remove caught exceptions

## Example

```
void thrower() throws
    IOException, Exception, NullPointerException { }

void g() throws Exception {
    try { thrower(); }
    catch(IOException e) {}
}
```

Uncaught exceptions:

# Uncaught Exceptions (1)

## Collect all uncaught exceptions

- Collect thrown exceptions
- Remove caught exceptions

## Example

```
void thrower() throws
    IOException, Exception, NullPointerException { }

void g() throws Exception {
    try { thrower(); }
    catch(IOException e) {}
}
```

Uncaught exceptions: {NullPointerException, Exception}

# Uncaught Exceptions (2)

## Algorithm

- Recurse over the method definitions.
- Consider control constructs that deal with exceptions:
  - Method invocation and `throw` add uncaught exceptions.
  - `Try/catch` will remove uncaught exceptions.



# Uncaught Exceptions (2)

## Algorithm

- Recurse over the method definitions.
- Consider control constructs that deal with exceptions:
  - Method invocation and `throw` add uncaught exceptions.
  - `Try/catch` will remove uncaught exceptions.

```
collect-uncaught-exceptions =  
  rec rec (  
    ThrownExceptions(rec)  
    <+ crush(![], union, rec)  
  )
```

# Uncaught Exceptions (3)

## Handling throw

```
ThrownExceptions(rec):  
  Throw(e) -> <union> ([<type-attr> e], children)  
  where  
    children := <rec> e
```

## Handling method invocation

# Uncaught Exceptions (3)

## Handling throw

```
ThrownExceptions(rec):  
  Throw(e) -> <union> ([<type-attr> e], children)  
  where  
    children := <rec> e
```

## Handling method invocation

```
ThrownExceptions(rec):  
  e@Invoke(o, args) -> <union> (this, children)  
  where  
    children := <rec> (o, args)  
    ; <compile-time-declaration-attr> e  
    ; lookup-method  
    ; this := <get-declared-exception-types>
```

# Uncaught Exceptions (4)

## Handling try/catch

```
ThrownExceptions(rec):  
  try@Try(body, catches) ->  
    <union> (uncaught, <rec> catches)  
  where  
    uncaught := <rec; remove-all-caught(|try)> body
```

## Summary

Generic term construction and deconstruction support the definition of generic analysis and generic translation problems

## Next

Context-sensitive transformation problems

- bound variable renaming
- function/method inlining
- data-flow transformation
- interpretation

Solution: dynamic definition of rewrite rules

Except where otherwise noted, this work is licensed under

