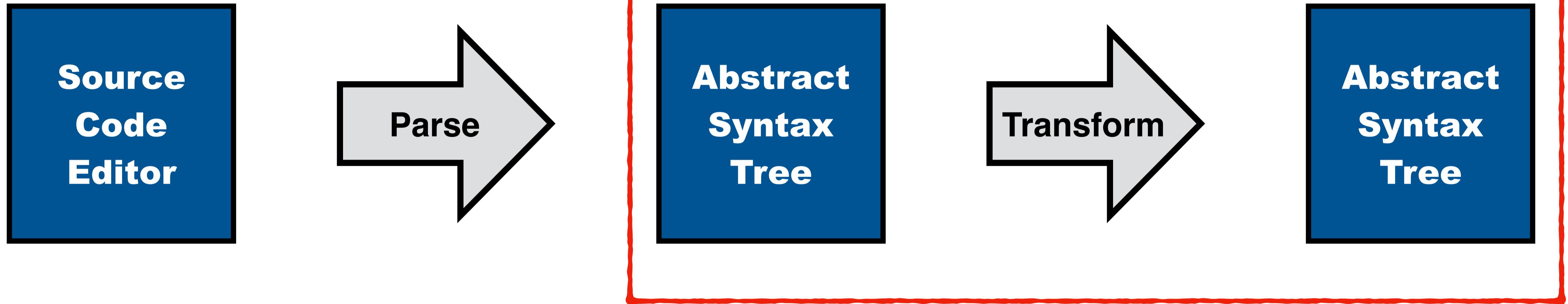


Declare Your Language

Chapter 4: Transformation by Strategic Term Rewriting

Eelco Visser

**IN4303 Compiler Construction
TU Delft
September 2017**



Define transformations on abstract syntax trees (terms) using rewrite rules

Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

Term rewrite rules define transformations on (abstract syntax) trees. Traditional rewrite systems apply rules exhaustively. This paper introduces programmable rewriting strategies to control the application of rules, the core of the design of the Stratego transformation language.

Note that the notation for contextual rules is no longer supported by Stratego. However, the technique to implement contextual rules still applies.

ICFP 1998

<https://doi.org/10.1145/291251.289425>

Building Program Optimizers with Rewriting Strategies*

Eelco Visser¹, Zine-el-Abidine Benaissa¹, Andrew Tolmach^{1,2}
Pacific Software Research Center

¹ Dept. of Comp. Science and Engineering, Oregon Graduate Institute, P.O. Box 91000, Portland, Oregon 97291-1000, USA

² Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, Oregon 97207 USA

visser@acm.org, benaissa@cse.ogi.edu, apt@cs.pdx.edu

Abstract

We describe a language for defining term rewriting strategies, and its application to the production of program optimizers. Valid transformations on program terms can be described by a set of rewrite rules; rewriting strategies are used to describe when and how the various rules should be applied in order to obtain the desired optimization effects. Separating rules from strategies in this fashion makes it easier to reason about the behavior of the optimizer as a whole, compared to traditional monolithic optimizer implementations. We illustrate the expressiveness of our language by using it to describe a simple optimizer for an ML-like intermediate representation.

The basic strategy language uses operators such as sequential composition, choice, and recursion to build transformers from a set of labeled unconditional rewrite rules. We also define an extended language in which the side-conditions and contextual rules that arise in realistic optimizer specifications can themselves be expressed as strategy-driven rewrites. We show that the features of the basic and extended languages can be expressed by breaking down the rewrite rules into their primitive building blocks, namely matching and building terms in variable binding environments. This gives us a low-level core language which has a clear semantics, can be implemented straightforwardly and can itself be optimized. The current implementation generates C code from a strategy specification.

1 Introduction

Compiler components such as parsers, pretty-printers and code generators are routinely produced using program generators. The component is specified in a high-level language from which the program generator produces its implementation. Program optimizers are difficult labor-intensive components that are usually still developed manually, despite many attempts at producing optimizer generators (e.g., [19, 12, 28, 25, 18, 11]).

*This work was supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP '98 Baltimore, MD USA
© 1998 ACM 1-58113-024-4/98/0009...\$5.00

A program optimizer transforms the source code of a program into a program that has the same meaning, but is more efficient. On the level of specification and documentation, optimizers are often presented as a set of correctness-preserving *rewrite rules* that transform code fragments into equivalent more efficient code fragments (e.g., see Table 5). This is particularly attractive for functional language compilers (e.g., [3, 4, 24]) that operate via successive small transformations, and don't rely on analyses requiring significant auxiliary data structures. The paradigm provided by conventional rewrite engines is to compute the normal form of a program with respect to a set of rewrite rules. However, optimizers are usually not implemented in this way. Instead, an algorithm is produced that implements a *strategy* for applying the optimization rules. Such a strategy contains meta-knowledge about the set of rewrite rules and the programming language they are applied to in order to (1) control the application of rules; (2) guarantee termination of optimization; (3) make optimization more efficient.

Such an ad-hoc implementation of a rewriting system has several drawbacks, even when implemented in a language with good support for pattern matching, such as ML or Haskell. First of all, the transformation rules are embedded in the code of the optimizer, making them hard to understand, to maintain, and to reuse individual rules in other transformations. Secondly, the strategy is not specified at the same level of abstraction as the transformation rules, making it hard to reason about the correctness of the optimizer even if the individual rules are correct. Finally, the host language has no awareness of the transformation domain underlying the implementation and can therefore not use this domain knowledge to optimize the optimizer itself.

It would be desirable to apply term rewriting technology directly to produce program optimizers. However, the standard approach to rewriting is to provide a fixed strategy (e.g., innermost or outermost) for normalizing a term with respect to a set of user-defined rewrite rules. This is not satisfactory when—as is usually the case for optimizers—the rewrite rules are neither confluent nor terminating. A common work-around is to encode a strategy into the rules themselves, e.g., by using an explicit function symbol that controls where rewrites are allowed. But this approach has the same disadvantages as the ad-hoc implementation of rewriting described above: the rules are hard to read, and the strategies are still expressed at a low level of abstraction.

In this paper we argue that a better solution is to use explicit specification of *rewriting strategies*. We show how

Stratego/XT combines SDF2 and Stratego into toolset for program transformation.

This paper gives a high-level overview of the concepts.

The StrategoXT.jar is still part of the Spoofax distribution.

Lecture Notes in Computer Science 2003

https://doi.org/10.1007/978-3-540-25935-0_13

Program Transformation with Stratego/XT Rules, Strategies, Tools, and Systems in Stratego/XT 0.9

Eelco Visser

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80089 3508 TB, Utrecht, The Netherlands
visser@acm.org
<http://www.stratego-language.org>

Abstract. Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation systems including parsing and pretty-printing. The framework addresses the entire range of the development process; from the specification of transformations to their composition into transformation systems. This chapter gives an overview of the main ingredients involved in the composition of transformation systems with Stratego/XT, where we distinguish the abstraction levels of rules, strategies, tools, and systems.

1 Introduction

Program transformation, the automatic manipulation of source programs, emerged in the context of compilation for the implementation of components such as optimizers [28]. While compilers are rather specialized tools developed by few, transformation systems are becoming widespread. In the paradigm of generative programming [13], the generation of programs from specifications forms a key part of the software engineering process. In refactoring [21], transformations are used to restructure a program in order to improve its design. Other applications of program transformation include migration and reverse engineering. The common goal of these transformations is to increase programmer productivity by automating programming tasks.

With the advent of XML, transformation techniques are spreading beyond the area of programming language processing, making transformation a necessary operation in any scenario where structured data play a role. Techniques from program transformation are applicable in document processing. In turn, applications such as Active Server Pages (ASP) for the generation of web-pages in dynamic HTML has inspired the creation of program generators such as Jstraca [31], where code templates specified in the concrete syntax of the object language are instantiated with application data.

Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation

Spoofax combines SDF2 and Stratego into a language workbench, i.e. an IDE for creating language definition from which IDEs for the defined languages can be generated.

A distinctive feature of Spoofax is live language development, which supports developing a language definition and programs in the defined language in the same IDE instance.

Spoofax was developed for Eclipse, which is still the main development platform. However, Spoofax Core is now independent of any IDE.

OOPSLA 2010

<https://doi.org/10.1145/1932682.1869497>

The Spoofax Language Workbench

Rules for Declarative Specification of Languages and IDEs

Lennart C. L. Kats

Delft University of Technology

l.c.l.kats@tudelft.nl

Eelco Visser

Delft University of Technology

visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this paper we describe the architecture of Spoofax and introduce idioms for high-level specifications of language semantics using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [38, 47]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [38].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10... \$10.00

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [42] revolutionized the IDE landscape [17] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a “build” button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the *syntax* of the language. (2) Semantic *analysis* to validate DSL programs according to some set of constraints. (3) *Transformations* manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A *code generator* that emits executable code. (5) Integration of the language into an *IDE*.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parsers from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [3, 10, 12, 20, 35] and frameworks [39, 57] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development such as IMP [7, 8] and TMF [56], simplify the implementation of IDE services. Other tools, such as the Synthesizer

Note that since the publication of this paper, we have introduced more declarative approaches to name and type analysis, which will be the topic of the next lectures.

OOPSLA 2010

<https://doi.org/10.1145/1932682.1869497>

The Spoofax Language Workbench

Rules for Declarative Specification of Languages and IDEs

Lennart C. L. Kats

Delft University of Technology

l.c.l.kats@tudelft.nl

Eelco Visser

Delft University of Technology

visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this paper we describe the architecture of Spoofax and introduce idioms for high-level specifications of language semantics using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [38, 47]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [38].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10... \$10.00

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [42] revolutionized the IDE landscape [17] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a “build” button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the *syntax* of the language. (2) Semantic *analysis* to validate DSL programs according to some set of constraints. (3) *Transformations* manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A *code generator* that emits executable code. (5) Integration of the language into an *IDE*.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parsers from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [3, 10, 12, 20, 35] and frameworks [39, 57] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development such as IMP [7, 8] and TMF [56], simplify the implementation of IDE services. Other tools, such as the Synthesizer

Documentation for Stratego at the [metaborg.org](http://www.metaborg.org) website.

The screenshot shows a documentation page for the Spoofax Language Workbench. The left sidebar contains navigation links for the workbench, examples, publications, tutorials (including sections on installing, creating projects, using the API, and getting support), and a reference manual (sections on language definition, abstract syntax, syntax definition, static semantics, and transformation with Stratego). The main content area is titled "Transformation with Stratego" and discusses the use of Stratego for defining transformations on abstract syntax trees. It includes a bulleted list of related links: "Stratego Tutorial/Reference (Old)", "The Stratego Library", and "Concrete Syntax in Stratego Transformations". Navigation buttons for "Previous" and "Next" are also visible.

Docs » Transformation with Stratego [Edit on GitHub](#)

Transformation with Stratego

Parsing a program text results in an abstract syntax tree. Stratego is a language for defining transformations on such trees. Stratego provides a term notation to construct and deconstruct trees and uses *term rewriting* to define transformations. Instead of applying all rewrite rules to all sub-terms, Stratego supports programmable *rewriting strategies* that control the application of rewrite rules.

- [Stratego Tutorial/Reference \(Old\)](#)
- [The Stratego Library](#)
- [Concrete Syntax in Stratego Transformations](#)

[Previous](#) [Next](#)

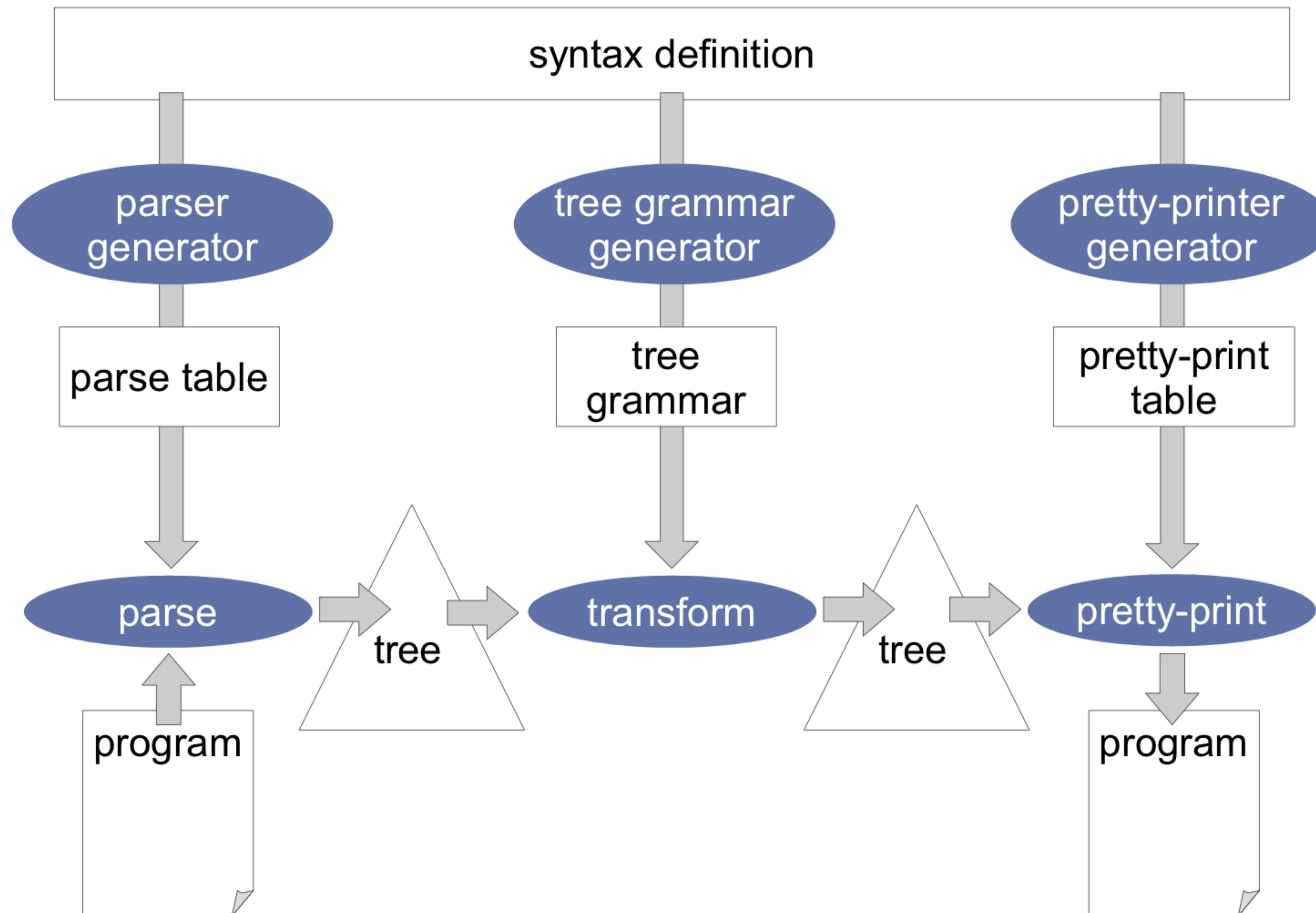
© Copyright 2016-2017, MetaBorg. Revision `ce45afc9`.
Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).

<http://www.metaborg.org/en/latest/source/langdev/meta/lang/stratego/index.html>

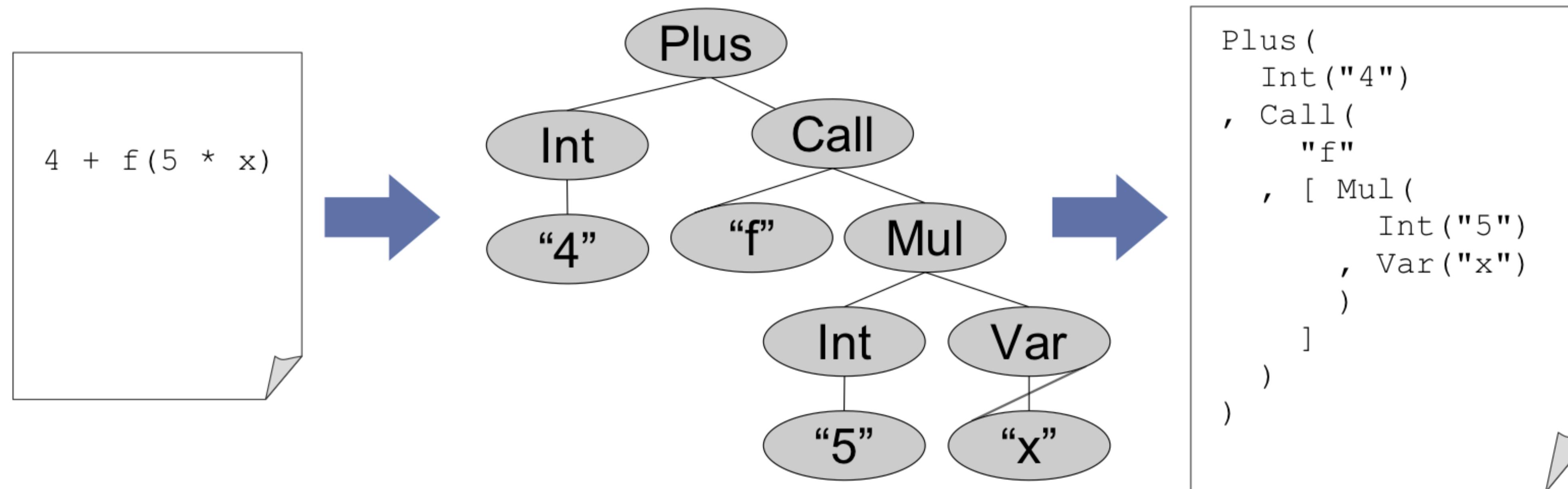
Transformation Architecture

This presentation uses the Stratego Shell for explaining the behavior of Stratego programs. The Stratego Shell is currently not supported by Spoofax

Architecture of Stratego/XT



Programs as Terms



Trees are represented as terms in the ATerm format

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))])))
```

ATerm Format

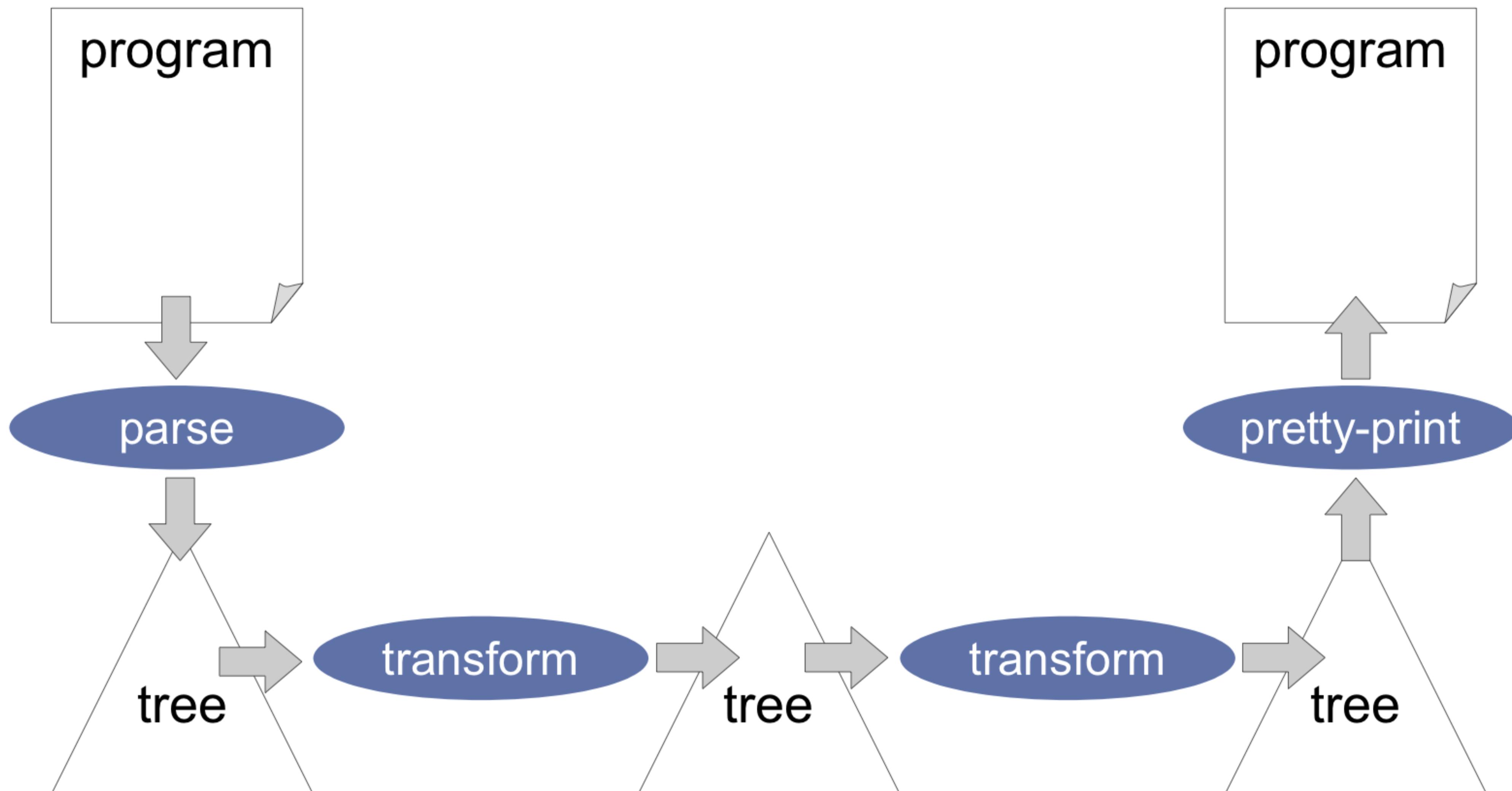
Application	Void(), Call(t , t)
List	[], [t , t , t]
Tuple	(t , t), (t , t , t)
Integer	25
Real	38.87
String	"Hello world"
Annotated term	$t\{t, t, t\}$

- Exchange of structured data
- Efficiency through maximal sharing
- Binary encoding

Structured Data: comparable to XML

Stratego: internal is external representation

How to Realize Program Transformations?



Conventional Term Rewriting

- Rewrite system = set of rewrite rules
- Redex = reducible expression
- Normalization = exhaustive application of rules to term
- (Stop when no more redices found)
- Strategy = algorithm used to search for redices
- Strategy given by engine

Strategic Term Rewriting

- Select rules to use in a specific transformation
- Select strategy to apply
- Define your own strategy if necessary
- Combine strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

<current term>

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

```
<current term>  
stratego> <strategy expression>  
<transformed term>
```

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

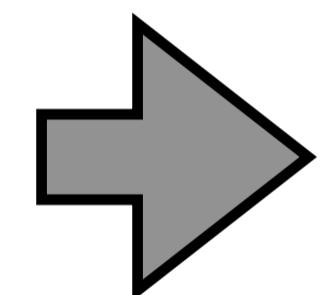
Stratego Shell: An Interactive Interpreter for Stratego

```
<current term>  
stratego <strategy expression>  
<transformed term>  
stratego <strategy expression>  
command failed
```

Terms

Parsing: From Text to Terms

```
let function fact(n : int) : int =
    if n < 1 then 1 else (n * fact(n - 1))
in fact(10)
end
```



```
Let(
  FunDec(
    "fact",
    [FArg("n", Tp(Tid("int"))),
     Tp(Tid("int"))],
    If(
      Lt(Var("n"), Int("1")),
      Int("1"),
      Seq(
        Times(
          Var("n"),
          Call(
            Var("fact"),
            [Minus(Var("n"), Int("1"))]
          )
        )
      )
    )
  ),
  [Call(Var("fact"), [Int("10")])]
```

Syntax of Terms

```
module Terms

sorts Cons Term

lexical syntax

Cons = [a-zA-Z][a-zA-Z0-9]*
context-free syntax

Term.App = <<Cons>(<{Term ",,"}*>)>
```

```
Zero()

Succ(Zero())

Cons(A(), Cons(B(), Nil()))
```

Syntax of Terms

```
module Terms

sorts Cons Term

lexical syntax

Cons = [a-zA-Z][a-zA-Z0-9]*
context-free syntax

Term.App    = <<Cons>(<{Term ","}*>)|
Term.List   = <[<{Term ","}*>]>
Term.Tuple  = <(<{Term ","}*>)|
```

```
Zero()

Succ(Zero())

[A(), B()]
```

Syntax of Terms

```
module ATerms

sorts Cons Term

lexical syntax
  Cons      = [a-zA-Z][a-zA-Z0-9]*
  Cons      = String
  Int       = [0-9]+
  String    = "\"" StringChar* "\""
  StringChar = ~["\\n"]
  StringChar = "\\" ["\\"]

context-free syntax
  Term.Str  = <<String>>
  Term.Int   = <<Int>>
  Term.App   = <<Cons><{Term ","}*>>
  Term.List  = <[<{Term ","}*]>
  Term.Tuple = <(<{Term ","}*>)>
```

```
0
1
[A(), B()]
Var("x\\")
Let(
  [ Decl("x", Int()), Decl("y", Bool())
  ]
, Eq(Var("x"), Int(0))
)
```

Syntax of A(nnotated) Terms

```
module ATerms

sorts Cons Term

lexical syntax
  Cons = [a-zA-Z][a-zA-Z0-9]*
  // more lexical syntax omitted

context-free syntax

Term.Anno      = <<PreTerm>{<{Term ","}*}>>
Term           = <<PreTerm>>

PreTerm.Str    = <<String>>
PreTerm.Int    = <<Int>>
PreTerm.App    = <<Cons>(<{Term ","}*}>>)
PreTerm.List   = <[<{Term ","}*]>>
PreTerm.Tuple  = <(<{Term ","}*}>>
```

Var("x"){Type(IntT())}

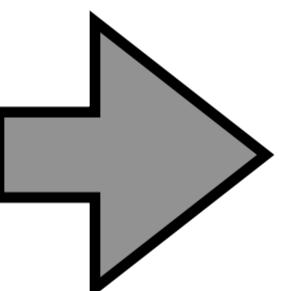
Signatures

Signatures

Signature declares argument and return types of term constructors

context-free syntax

```
Exp.Uminus = [- [Exp]]  
Exp.Power = [[Exp] ** [Exp]]  
Exp.Times = [[Exp] * [Exp]]  
Exp.Divide = [[Exp] / [Exp]]  
Exp.Plus = [[Exp] + [Exp]]  
Exp.Minus = [[Exp] - [Exp]]  
Exp.CPlus = [[Exp] +i [Exp]]  
Exp.CMinus = [[Exp] -i [Exp]]  
Exp.Eq = [[Exp] = [Exp]]  
Exp.Neq = [[Exp] <> [Exp]]  
Exp.Gt = [[Exp] > [Exp]]  
Exp.Lt = [[Exp] < [Exp]]  
Exp.Geq = [[Exp] >= [Exp]]  
Exp.Leq = [[Exp] <= [Exp]]  
Exp.True = <true>  
Exp.False = <false>  
Exp.And = [[Exp] & [Exp]]  
Exp.Or = [[Exp] | [Exp]]
```



signature
constructors

Uminus	: Exp \rightarrow Exp
Power	: Exp * Exp \rightarrow Exp
Times	: Exp * Exp \rightarrow Exp
Divide	: Exp * Exp \rightarrow Exp
Plus	: Exp * Exp \rightarrow Exp
Minus	: Exp * Exp \rightarrow Exp
CPlus	: Exp * Exp \rightarrow Exp
CMinus	: Exp * Exp \rightarrow Exp
Eq	: Exp * Exp \rightarrow Exp
Neq	: Exp * Exp \rightarrow Exp
Gt	: Exp * Exp \rightarrow Exp
Lt	: Exp * Exp \rightarrow Exp
Geq	: Exp * Exp \rightarrow Exp
Leq	: Exp * Exp \rightarrow Exp
True	: Exp
False	: Exp
And	: Exp * Exp \rightarrow Exp
Or	: Exp * Exp \rightarrow Exp

Signature is automatically
generated from SDF3 productions

Stratego compiler only checks
arity of constructor applications

Rewrite Rules

Desugaring

```
if x then  
  printint(x)
```

```
if x then  
  printint(x)  
else  
  ()
```

```
IfThen(  
  Var("x")  
, Call(  
  "printint"  
, [Var("x")])  
)
```

pattern matching

```
IfThenElse(  
  Var("x")  
, Call(  
  "printint"  
, [Var("x")])  
, NoVal())  
)
```

pattern instantiation

```
desugar: IfThen(e1, e2) -> IfThenElse(e1, e2, NoVal())
```

More Desugaring

signature
constructors

PLUS: BinOp

MINUS: BinOp

MUL: BinOp

DIV: BinOp

EQ: RelOp

NE: RelOp

LE: RelOp

LT: RelOp

Bop: BinOp * Expr * Expr -> Expr

Rop: RelOp * Expr * Expr -> Expr

desugar: Uminus(e) -> Bop(MINUS(), Int("0"), e)

desugar: Plus(e1, e2) -> Bop(PLUS(), e1, e2)

desugar: Minus(e1, e2) -> Bop(MINUS(), e1, e2)

desugar: Times(e1, e2) -> Bop(MUL(), e1, e2)

desugar: Divide(e1, e2) -> Bop(DIV(), e1, e2)

desugar: Eq(e1, e2) -> Rop(EQ(), e1, e2)

desugar: Neq(e1, e2) -> Rop(NE(), e1, e2)

desugar: Leq(e1, e2) -> Rop(LE(), e1, e2)

desugar: Lt(e1, e2) -> Rop(LT(), e1, e2)

desugar: Geq(e1, e2) -> Rop(LE(), e2, e1)

desugar: Gt(e1, e2) -> Rop(LT(), e2, e1)

desugar: And(e1, e2) -> IfThenElse(e1, e2, Int("0"))

desugar: Or(e1, e2) -> IfThenElse(e1, Int("1"), e2)

Constant Folding

```
x := 21 + 21 + x
```

```
x := 42 + x
```

```
Assign(  
  Var("x")  
, Plus(  
  Plus(  
    Int("21")  
, Int("21"))  
, Var("x"))  
)
```

```
Assign(  
  Var("x")  
, Plus(  
  Int("42")  
, Var("x"))  
)
```

```
eval: Bop(PLUS(), Int(i1), Int(i2)) -> Int(i3)  
where <addS> (i1, i2) => i3
```

More Constant Folding

```
eval: Bop(PLUS(), Int(i1), Int(i2)) -> Int(<addS> (i1, i2))

eval: Bop(MINUS(), Int(i1), Int(i2)) -> Int(<subS> (i1, i2))

eval: Bop(MUL(), Int(i1), Int(i2)) -> Int(<mulS> (i1, i2))

eval: Bop(DIV(), Int(i1), Int(i2)) -> Int(<divS> (i1, i2))

eval: Rop(EQ(), Int(i), Int(i)) -> Int("1")
eval: Rop(EQ(), Int(i1), Int(i2)) -> Int("0") where not( <eq> (i1, i2) )

eval: Rop(NE(), Int(i), Int(i)) -> Int("0")
eval: Rop(NE(), Int(i1), Int(i2)) -> Int("1") where not( <eq> (i1, i2) )

eval: Rop(LT(), Int(i1), Int(i2)) -> Int("1") where <ltS> (i1, i2)
eval: Rop(LT(), Int(i1), Int(i2)) -> Int("0") where not( <ltS> (i1, i2) )

eval: Rop(LE(), Int(i1), Int(i2)) -> Int("1") where <leqS> (i1, i2)
eval: Rop(LE(), Int(i1), Int(i2)) -> Int("0") where not( <leqS> (i1, i2))
```

Term Rewriting (Bottom-up)

Building and Matching Terms

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"))
```

Match pattern

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

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

Match pattern

- Syntax: $?p$
- Match current term (t) against pattern p
- Succeed if there is a substitution σ 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 σ 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 σ 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"
```

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: $\mathbf{!}p[<s>]$
- Strategy application in pattern to current subterm
- Equivalent to: $\{x : \mathbf{where}(s \Rightarrow x) ; \mathbf{!}p[x]\}$
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: $?p[<s>]$
- Strategy application in pattern match
- Equivalent to: $\{x: ?p[x]; <s>x\}$
for some fresh variable x

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

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

Naming and Composing Strategies

Reuse of transformation requires definitions

1. Naming strategy expressions
2. Named rewrite rules
3. Reusing rewrite rules through modules

Simple strategy definition and call

- Syntax: $f = s$
- Name strategy expression s
- Syntax: f
- Invoke (call) named strategy f

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs = {e1,e2:(Plus(e1,e2) -> Plus(e2,e1))}
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

Named rewrite rules (sugar)

- Syntax: $f : p_1 \rightarrow p_2 \text{ where } s$
- Name rewrite rule $p_1 \rightarrow p_2 \text{ where } s$
- Equivalent to: $f = \{x_1, \dots, x_n : (p_1 \rightarrow p_2 \text{ where } s)\}$
(with x_1, \dots, x_n the variables in p_1 , p_2 , and s)

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

Example: Inverting If Not Equal

```
if(x != y)
    doSomething();
else
    doSomethingElse();
```

⇒

```
if(x == y)
    doSomethingElse();
else
    doSomething();
```

InvertIfNot :

```
If(NotEq(e1, e2), stm1, stm2) ->
If(Eq(e1, e2), stm2, stm1)
```

Modules with Reusable Transformation Rules

```
module Simplification-Rules
rules
  PlusAssoc :
    Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))

  EvalIf :
    If(Lit(Bool(True())), stm1, stm2) -> stm1

  EvalIf :
    If(Lit(Bool(False())), stm1, stm2) -> stm2

  IntroduceBraces :
    If(e, stm) -> If(e, Block([stm]))
    where <not(?Block(_))> stm
```

```
stratego> import Simplification-Rules
```

Rules define one-step transformations

Program transformations require many one-step transformations and selection of rules

1. Choice
2. Identity, Failure, and Negation
3. Parameterized and Recursive Definitions

Composing Strategies

Deterministic choice (left choice)

- Syntax: $s_1 \leftarrow s_2$
- First apply s_1 , if that fails apply s_2
- Note: local backtracking

```
PlusAssoc :
```

```
  Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))
```

```
EvalPlus :
```

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

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

```
stratego> PlusAssoc
```

```
command failed
```

```
stratego> PlusAssoc <- EvalPlus
```

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

Composing Strategies

Guarded choice

- Syntax: $s_1 < s_2 + s_3$
- First apply s_1 if that succeeds apply s_2 to the result
else apply s_3 to the original term
- Do not backtrack to s_3 if s_2 fails!

Motivation

- $s_1 <+ s_2$ always backtracks to s_2 if s_1 fails
- $(s_1 ; s_2) <+ s_3 \not\equiv s_1 < s_2 + s_3$
- commit to branch if test succeeds, even if that branch fails

```
test1 < transf1
+ test2 < transf2
+ transf3
```

Composing Strategies

Guarded choice

- Syntax: $s_1 < s_2 + s_3$
- First apply s_1 if that succeeds apply s_2 to the result
else apply s_3 to the original term
- Do not backtrack to s_3 if s_2 fails!

Motivation

- $s_1 <+ s_2$ always backtracks to s_2 if s_1 fails
- $(s_1 ; s_2) <+ s_3 \not\equiv s_1 < s_2 + s_3$
- commit to branch if test succeeds, even if that branch fails

```
test1 < transf1
+ test2 < transf2
+ transf3
```

If then else (sugar)

- Syntax: if s_1 then s_2 else s_3 end
- Equivalent to: where(s_1) $< s_2 + s_3$

Composing Strategies

Identity

- Syntax: id
- Always succeed
- Some laws
 - $\text{id} ; s \equiv s$
 - $s ; \text{id} \equiv s$
 - $\text{id} \<+ s \equiv \text{id}$
 - $s \<+ \text{id} \not\equiv s$
 - $s_1 < \text{id} + s_2 \equiv s_1 \<+ s_2$

Failure

- Syntax: fail
- Always fail
- Some laws
 - $\text{fail} \<+ s \equiv s$
 - $s \<+ \text{fail} \equiv s$
 - $\text{fail} ; s \equiv \text{fail}$
 - $s ; \text{fail} \not\equiv \text{fail}$

Composing Strategies

Identity

- Syntax: id
- Always succeed
- Some laws
 - $\text{id} ; s \equiv s$
 - $s ; \text{id} \equiv s$
 - $\text{id} \<+ s \equiv \text{id}$
 - $s \<+ \text{id} \not\equiv s$
 - $s_1 < \text{id} + s_2 \equiv s_1 \<+ s_2$

Failure

- Syntax: fail
- Always fail
- Some laws
 - $\text{fail} \<+ s \equiv s$
 - $s \<+ \text{fail} \equiv s$
 - $\text{fail} ; s \equiv \text{fail}$
 - $s ; \text{fail} \not\equiv \text{fail}$

Negation (sugar)

- Syntax: $\text{not}(s)$
- Fail if s succeeds, succeed if s fails
- Equivalent to: $s < \text{fail} + \text{id}$

Parameterized and recursive definitions

- Syntax: $f(x_1, \dots, x_n | y_1, \dots, y_m) = s$
- Strategy definition parameterized with strategies (x_1, \dots, x_n) and terms (y_1, \dots, y_m)
- Note: definitions may be recursive

Parameterizing Strategies

Parameterized and recursive definitions

- Syntax: $f(x_1, \dots, x_n | y_1, \dots, y_m) = s$
- Strategy definition parameterized with strategies (x_1, \dots, x_n) and terms (y_1, \dots, y_m)
- Note: definitions may be recursive

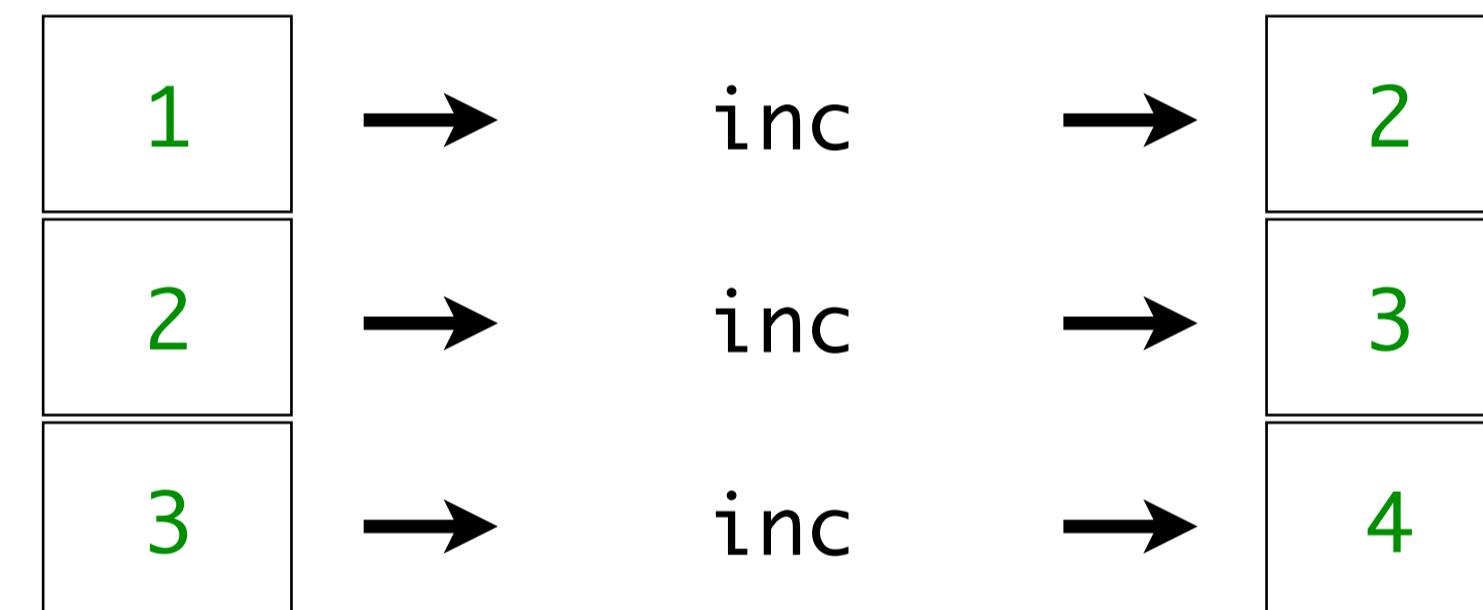
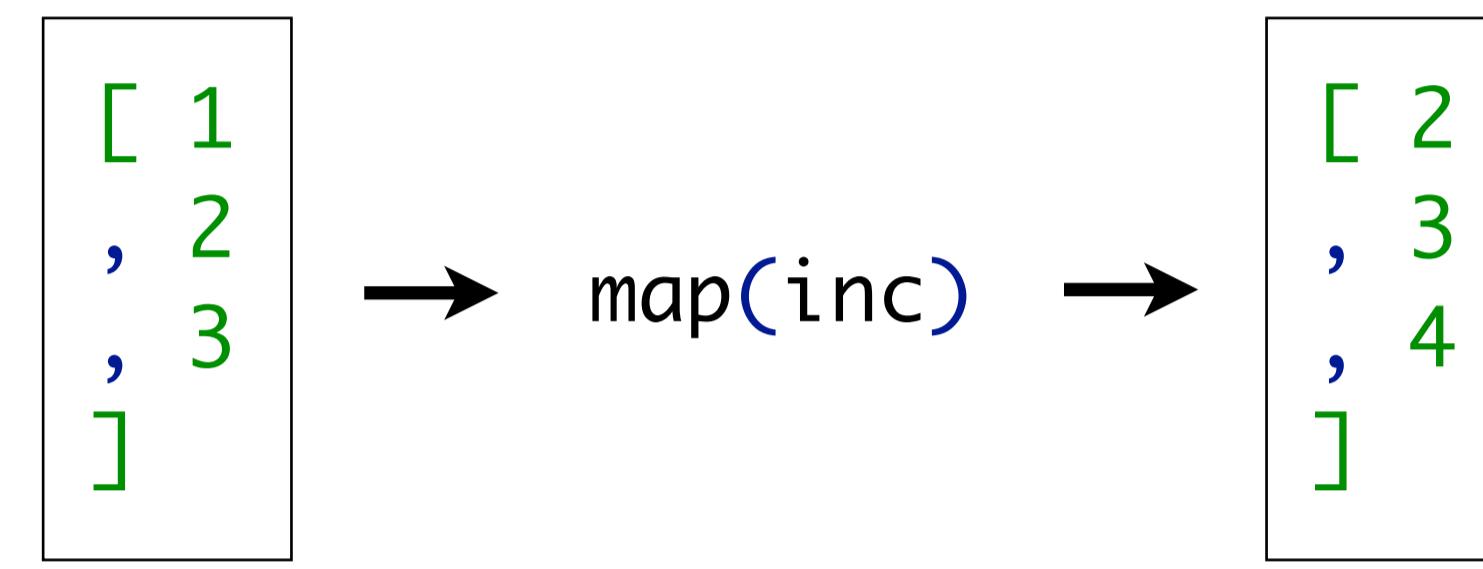
```
try(s)      = s <+ id
```

```
repeat(s)   = try(s; repeat(s))
```

```
while(c, s) = if c then s; while(c, s) end
```

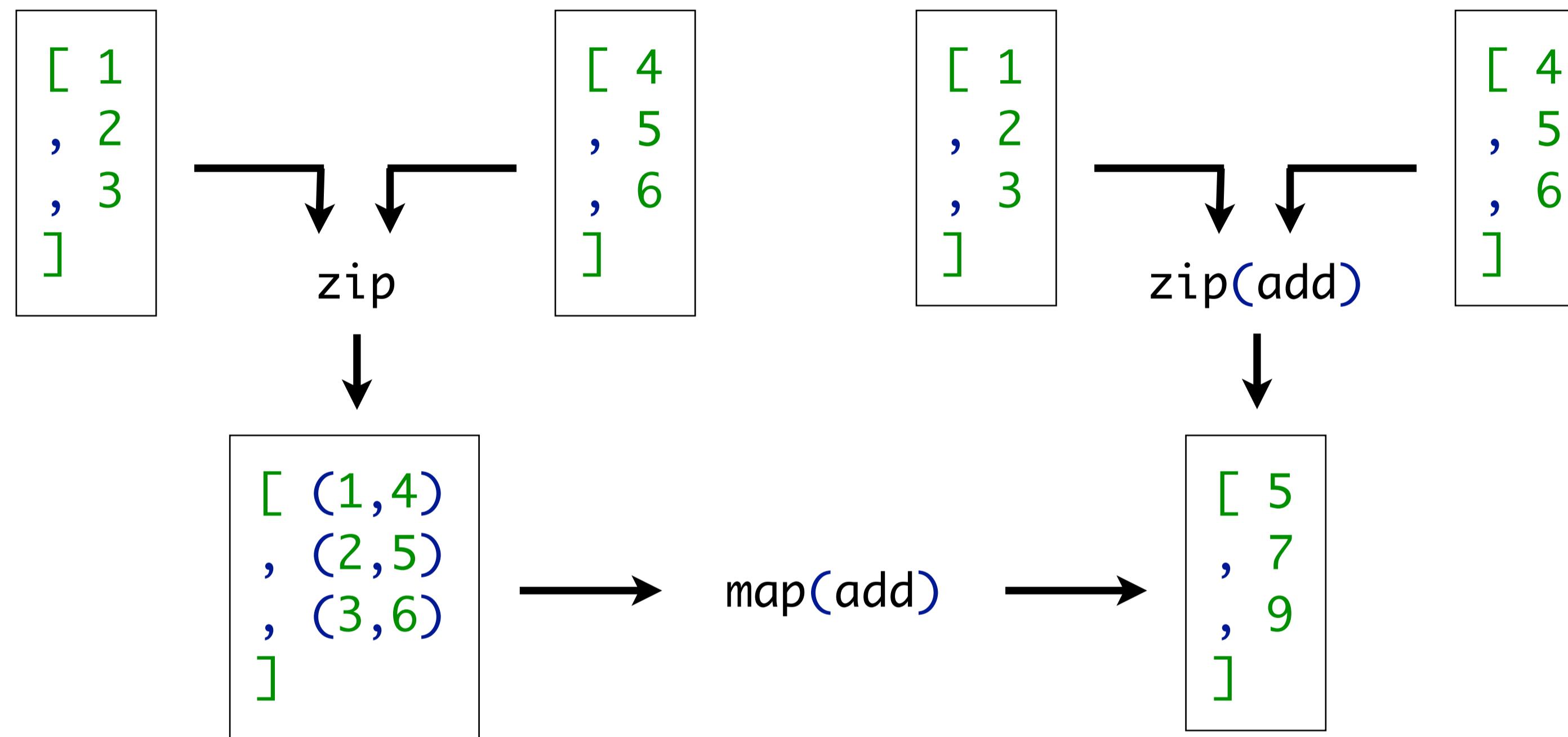
```
do-while(s, c) = s; if c then do-while(s, c) end
```

Parameterized Rewrite Rules: Map



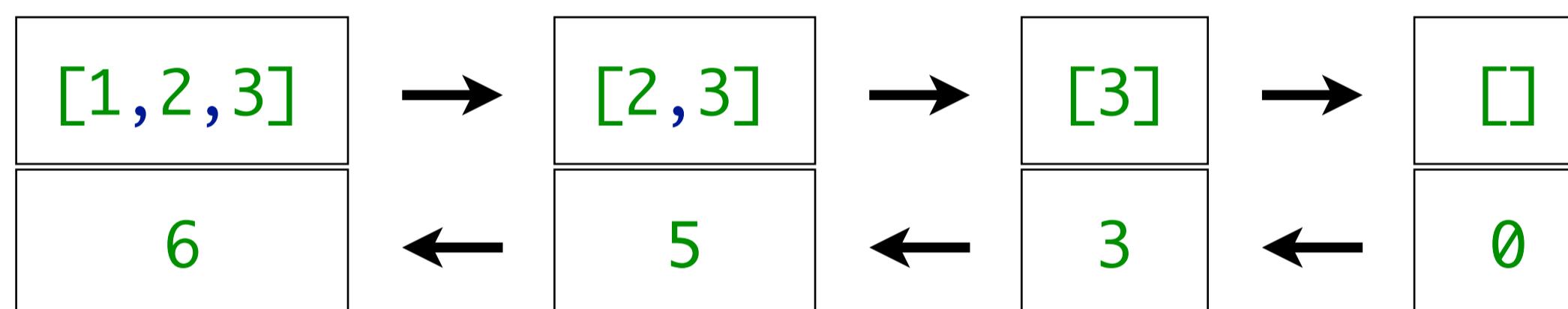
```
map(s): []    -> []
map(s): [x|xs] -> [<s> x | <map(s)> xs]
```

Parameterized Rewrite Rules: Zip



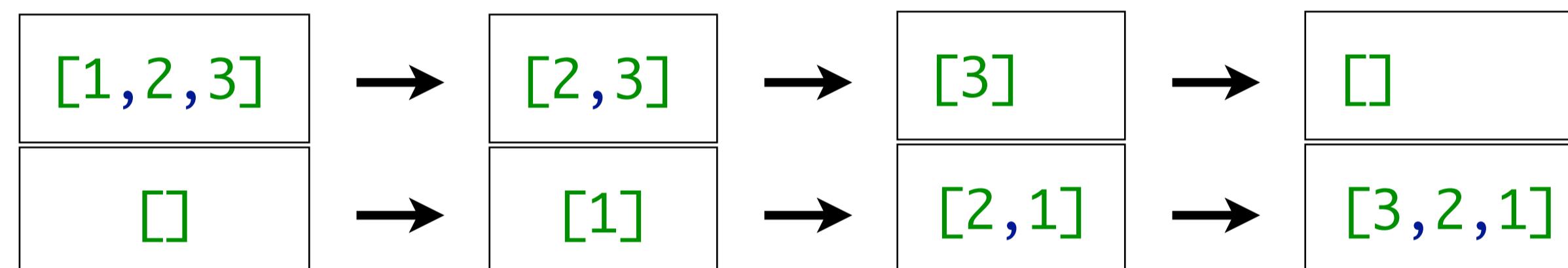
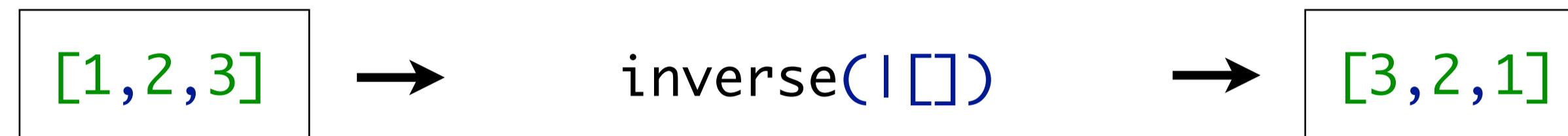
```
zip(s): ([] , [])      -> []
zip(s): ([x|xs], [y|ys]) -> [<s> (x,y) | <zip(s)> (xs,ys)]
zip = zip(id)
```

Parameterized Rewrite Rules: Fold



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

Parameterized Rewrite Rules: Inverse



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

Local Definitions

- Syntax: `let $f_1 = s_1 \dots f_n = s_n$ in s end`
- Strategy s with local strategies $f_1 \dots f_n$

Local Strategies

Local Definitions

- Syntax: `let $f_1 = s_1 \dots f_n = s_n$ in s end`
- Strategy s with local strategies $f_1 \dots f_n$

```
[1, 2, 3, 4]
stratego> let tl = ?[_|<id>] in tl; tl end
[3, 4]
stratego> tail
operator tail/(0,0) not defined
```

Local Strategies

Local Definitions

- Syntax: `let $f_1 = s_1 \dots f_n = s_n$ in s end`
- Strategy s with local strategies $f_1 \dots f_n$

```
[1, 2, 3, 4]
stratego> let tl = ?[_|<id>] in tl; tl end
[3, 4]
stratego> tail
operator tail/(0,0) not defined
```

Local Recursive Strategy

- Syntax: `rec $f(s)$`
- Equivalent to `let $f = s$ in f end`

Local Strategies

Local Definitions

- Syntax: `let $f_1 = s_1 \dots f_n = s_n$ in s end`
- Strategy s with local strategies $f_1 \dots f_n$

```
[1, 2, 3, 4]
stratego> let tl = ?[_|<id>] in tl; tl end
[3, 4]
stratego> tail
operator tail/(0,0) not defined
```

Local Recursive Strategy

- Syntax: `rec $f(s)$`
- Equivalent to `let $f = s$ in f end`

```
[1, 2, 3]
stratego> rec x(?[<id>] <+ ?[_| <x>])
3
```

Traversal Strategies

Term Rewriting

- apply set of rewrite rules exhaustively

Advantages

- First-order terms describe abstract syntax
- Rewrite rules express basic transformation rules
(operationalizations of the algebraic laws of the language.)
- Rules specified separately from strategy

Limitations

- Rewrite systems for programming languages often non-terminating and/or non-confluent
- In general: do not apply all rules at the same time or apply all rules under all circumstances

Term Rewriting for Program Transformation

signature

 sorts Prop

 constructors

 False : Prop

 True : Prop

 Atom : String -> Prop

 Not : Prop -> Prop

 And : Prop * Prop -> Prop

 Or : Prop * Prop -> Prop

rules

DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))

DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))

DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))

DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))

DN : Not(Not(x)) -> x

DMA : Not(And(x, y)) -> Or(Not(x), Not(y))

DMO : Not(Or(x, y)) -> And(Not(x), Not(y))

Term Rewriting for Program Transformation

signature

 sorts Prop

 constructors

 False : Prop

 True : Prop

 Atom : String -> Prop

 Not : Prop -> Prop

 And : Prop * Prop -> Prop

 Or : Prop * Prop -> Prop

rules

DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))

DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))

DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))

DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))

DN : Not(Not(x)) -> x

DMA : Not(And(x, y)) -> Or(Not(x), Not(y))

DMO : Not(Or(x, y)) -> And(Not(x), Not(y))

This is a non-terminating rewrite system

Encoding Control with Recursive Rewrite Rules

Common solution

- Introduce additional constructors that achieve normalization under a restricted set of rules
- Replace a ‘pure’ rewrite rule

$$p_1 \rightarrow p_2$$

with a functionalized rewrite rule:

$$f : p_1 \rightarrow p'_2$$

applying f recursively in the right-hand side

- Normalize terms $f(t)$ with respect to these rules
- The function now controls where rules are applied

Recursive Rewrite Rules: Disjunctive Normal Form

```
dnf  : True          -> True
dnf  : False         -> False
dnf  : Atom(x)       -> Atom(x)
dnf  : Not(x)        -> <not>(<dnf>x)
dnf  : And(x,y)      -> <and>(<dnf>x,<dnf>y)
dnf  : Or(x,y)       -> Or(<dnf>x,<dnf>y)
```

```
and1 : (Or(x,y),z) -> Or(<and>(x,z),<and>(y,z))
and2 : (z,Or(x,y)) -> Or(<and>(z,x),<and>(z,y))
and3 : (x,y)         -> And(x,y)
and   = and1 <+ and2 <+ and3
```

```
not1 : Not(x)        -> x
not2 : And(x,y)      -> Or(<not>(x),<not>(y))
not3 : Or(x,y)        -> <and>(<not>(x),<not>(y))
not4 : x              -> Not(x)
not   = not1 <+ not2 <+ not3 <+ not4
```

Functional encoding has two main problems

Overhead due to explicit specification of traversal

- A traversal rule needs to be defined for each constructor in the signature and for each transformation.

Separation of rules and strategy is lost

- Rules and strategy are completely *intertwined*
- Intertwining makes it more difficult to *understand* the transformation
- Intertwining makes it impossible to *reuse* the rules in a different transformation.

Language Complexity

Traversal overhead and reuse of rules is important, considering the complexity of real programming languages:

language	# constructors
Tiger	65
C	140
Java 5	325
COBOL	300–1200

Requirements

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

Programmable Rewriting Strategies

- Select rules to be applied in specific transformation
- Select strategy to control their application
- Define your own strategy if necessary
- Combine strategies

Idioms

- Cascading transformations
- One-pass traversal
- Staged transformation
- Local transformation

Strategic Idioms

Rules for rewriting proposition formulae

signature

 sorts Prop

 constructors

 False : Prop

 True : Prop

 Atom : String -> Prop

 Not : Prop -> Prop

 And : Prop * Prop -> Prop

 Or : Prop * Prop -> Prop

rules

DAOL : And(Or(x, y), z) -> Or(And(x, z), And(y, z))

DAOR : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))

DOAL : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))

DOAR : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))

DN : Not(Not(x)) -> x

DMA : Not(And(x, y)) -> Or(Not(x), Not(y))

DMO : Not(Or(x, y)) -> And(Not(x), Not(y))

Strategic Idioms: Cascading Transformation

Cascading Transformations

- Apply small, independent transformations in combination
- Accumulative effect of small rewrites

```
simplify = innermost(R1 <+ ... <+ Rn)
```

disjunctive normal form

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

conjunctive normal form

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

Strategic Idioms: One-Pass Traversal

One-pass Traversal

- Apply rules in a single traversal over a program tree

```
simplify1 = downup(repeat(R1 <+ ... <+ Rn))
simplify2 = bottomup(repeat(R1 <+ ... <+ Rn))
```

constant folding

```
Eval : And(True, e) -> e
Eval : And(False, e) -> False
Eval : ...
```

```
eval = bottomup(try(Eval))
```

Strategic Idioms: One-Pass Traversal

Example: Desugarings

```
DefN  : Not(x)      -> Impl(x, False)
DefI  : Impl(x, y)  -> Or(Not(x), y)
DefE  : Eq(x, y)    -> And(Impl(x, y), Impl(y, x))
Def01 : Or(x, y)    -> Impl(Not(x), y)
Def02 : Or(x, y)    -> Not(And(Not(x), Not(y)))
DefA1 : And(x, y)   -> Not(Or(Not(x), Not(y)))
DefA2 : And(x, y)   -> Not(Impl(x, Not(y)))
IDefI : Or(Not(x), y) -> Impl(x, y)
IDefE : And(Impl(x, y), Impl(y, x)) -> Eq(x, y)

desugar = topdown(try(DefI <+ DefE))

impl-nf  = topdown(repeat(DefN <+ DefA2 <+ Def01 <+ DefE))
```

Strategic Idioms: Staged Transformation

Staged Transformation

- Transformations are not applied to a subject term all at once, but rather in stages
- In each stage, only rules from some particular subset of the entire set of available rules are applied.

```
simplify =  
    innermost(A1 <+ ... <+ Ak)  
    ; innermost(B1 <+ ... <+ Bl)  
    ; ...  
    ; innermost(C1 <+ ... <+ Cm)
```

Strategic Idioms: Local Transformation

Local transformation

- Apply rules only to selected parts of the subject program

```
transformation =  
  alltd(  
    trigger-transformation  
    ; innermost(A1 <+ ... <+ An)  
  )
```

Traversal Combinators

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))
```

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))
```

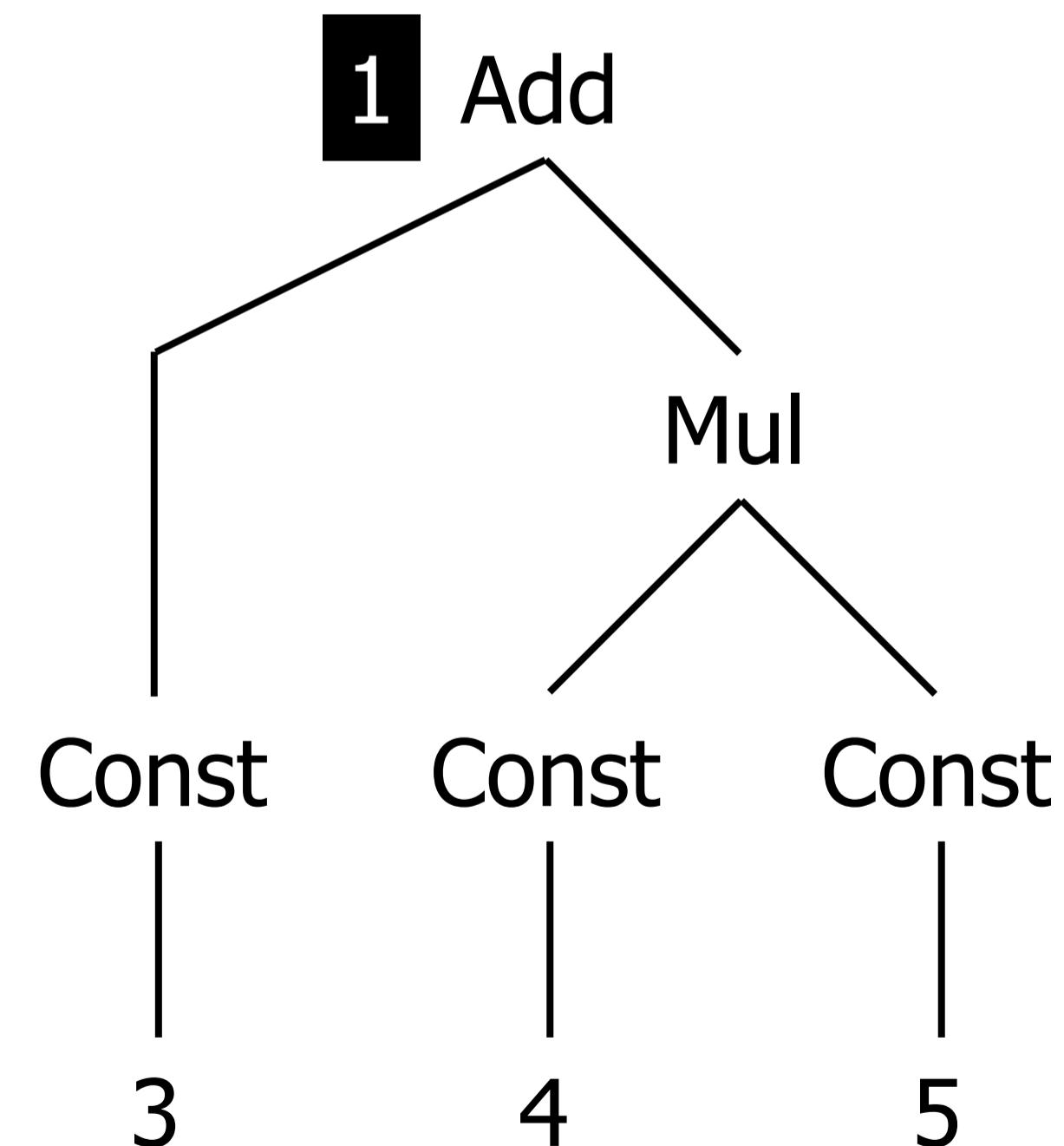
```
const-fold =
bottomup(try(EvalBinOp <+ EvalCall <+ EvalIf))
```

Traversal: Topdown

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

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

```
topdown(switch)
```



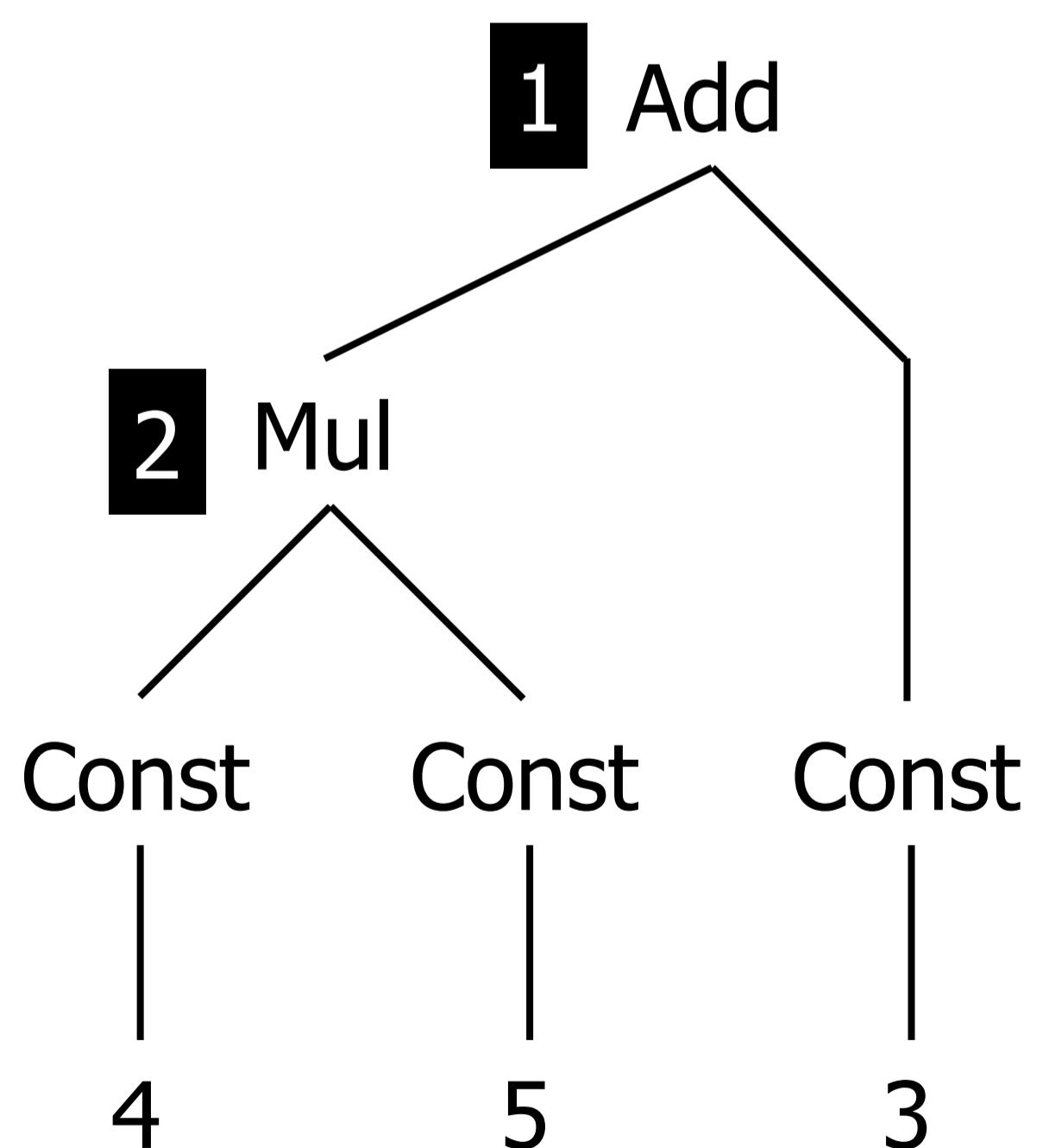
Traversal: Topdown

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

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

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

topdown(switch)



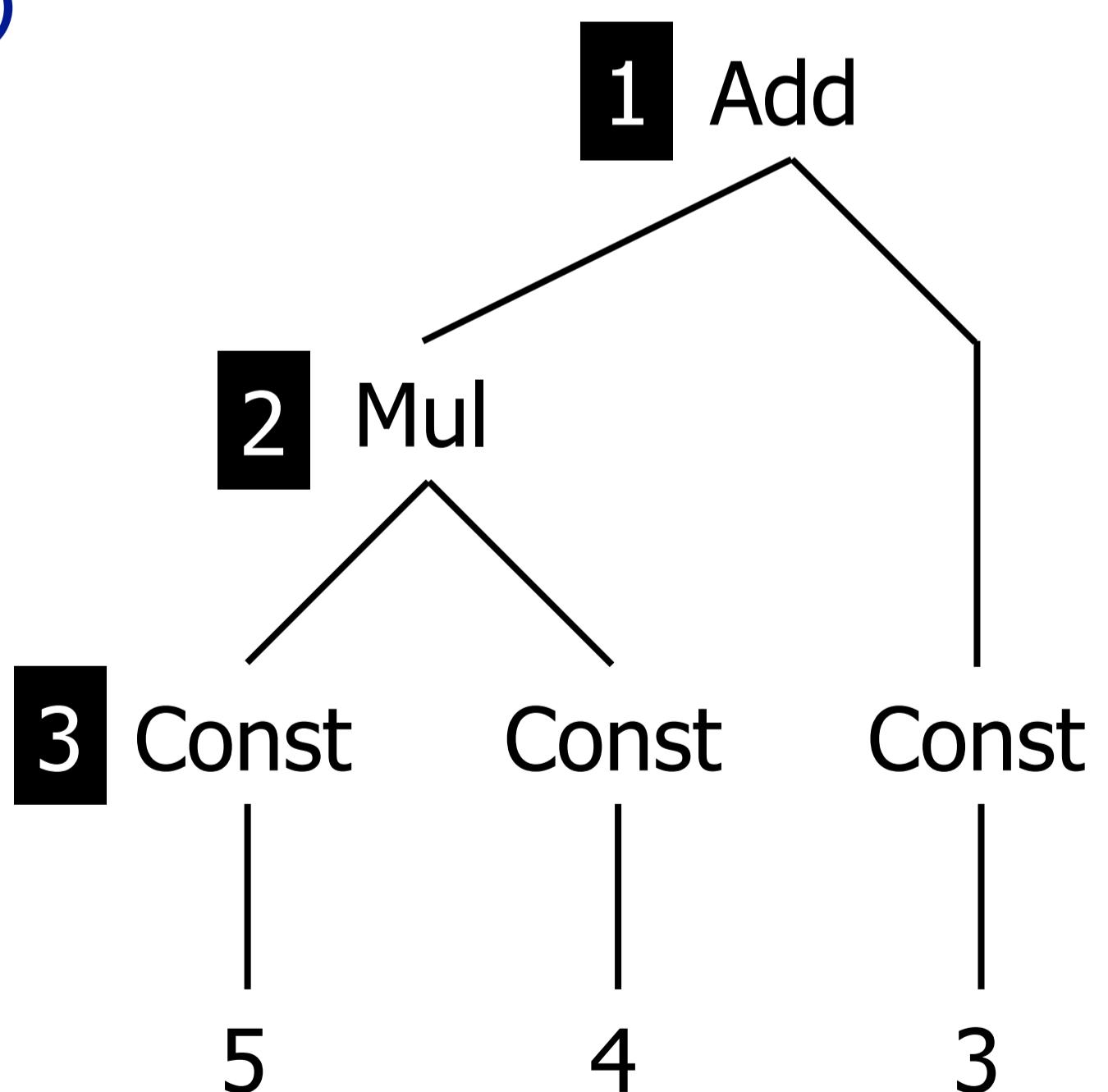
Traversal: Topdown

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

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

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

topdown(switch)

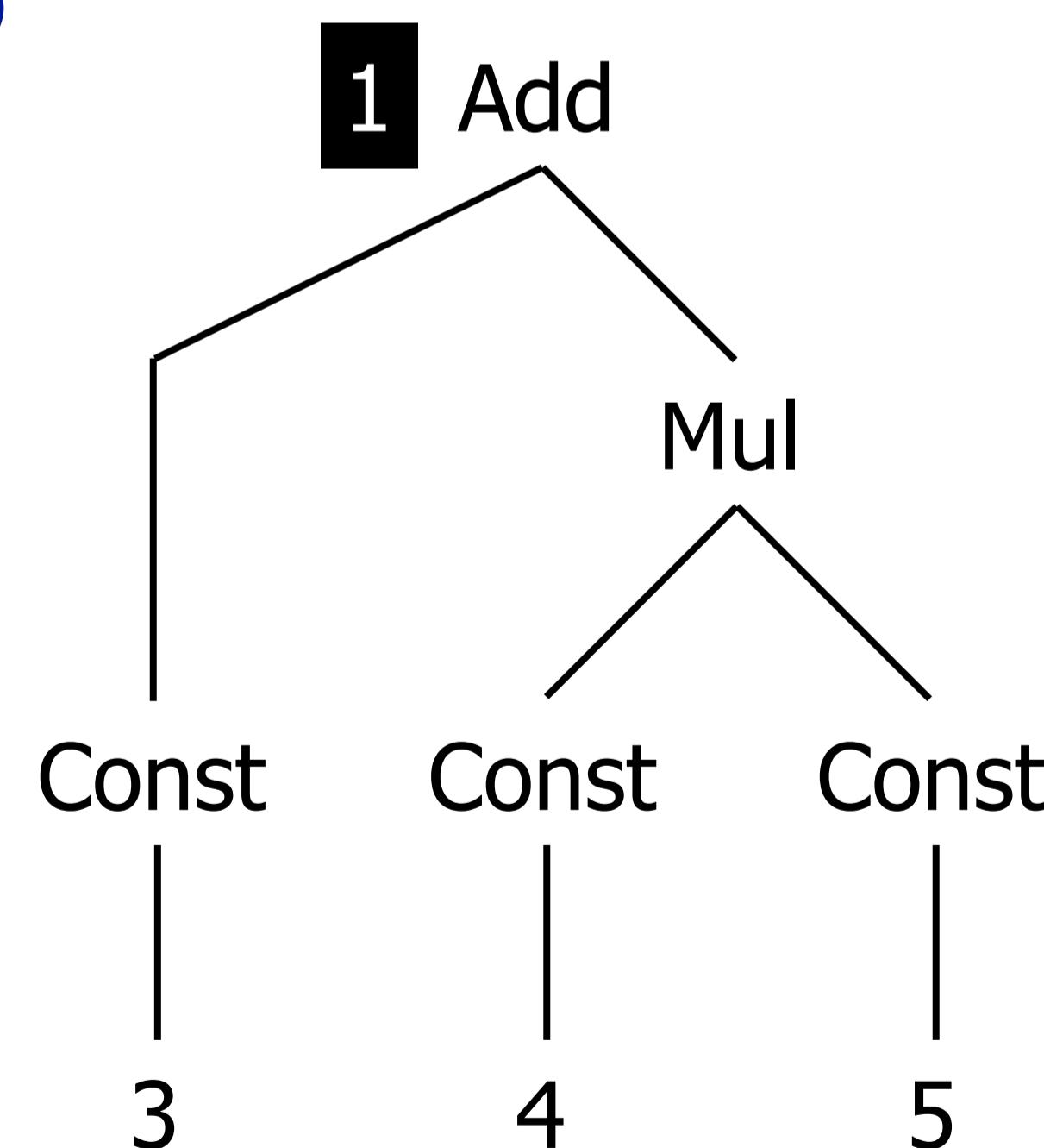


Traversal: Topdown/Try

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

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

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

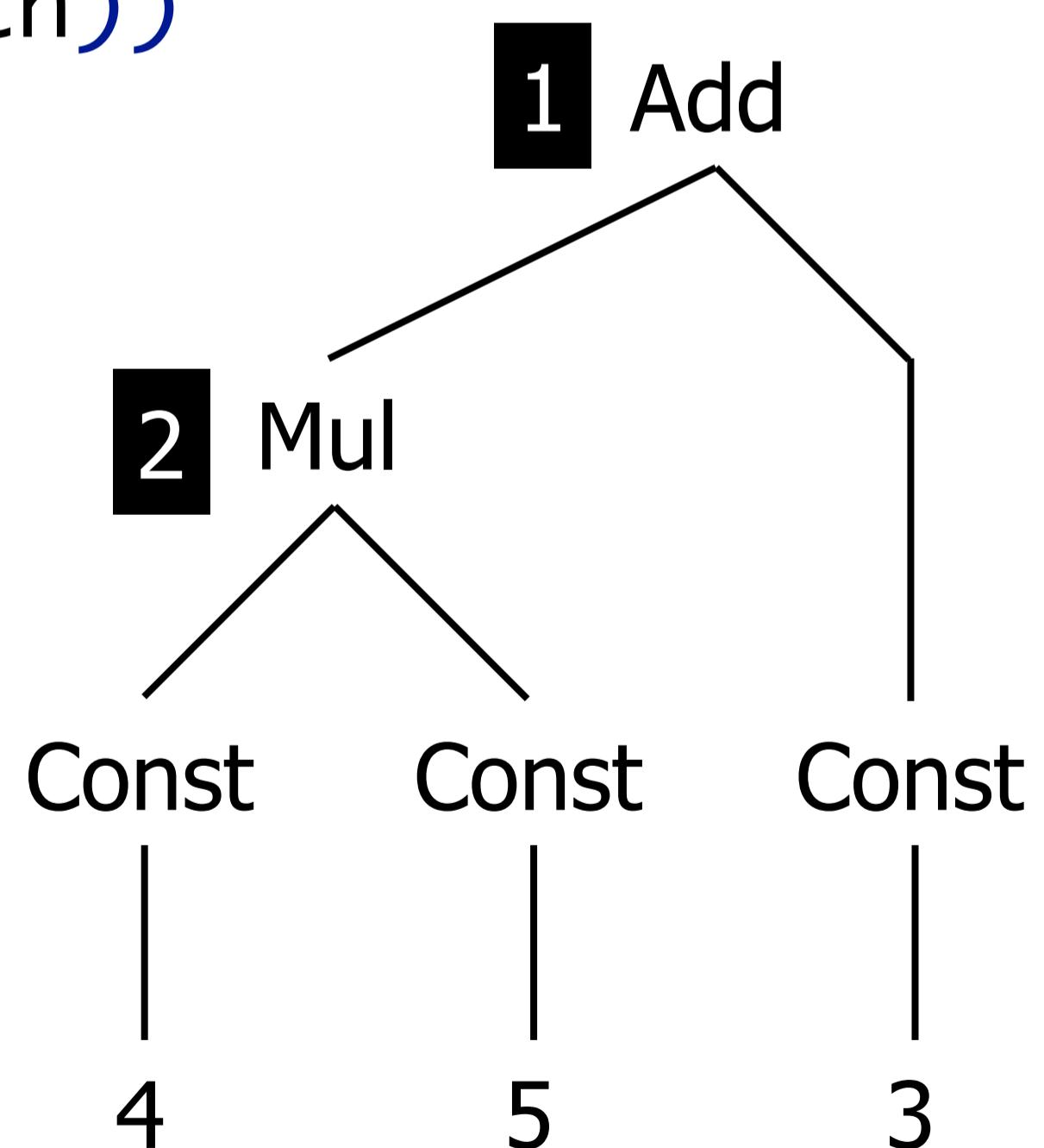


Traversal: Topdown/Try

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

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

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



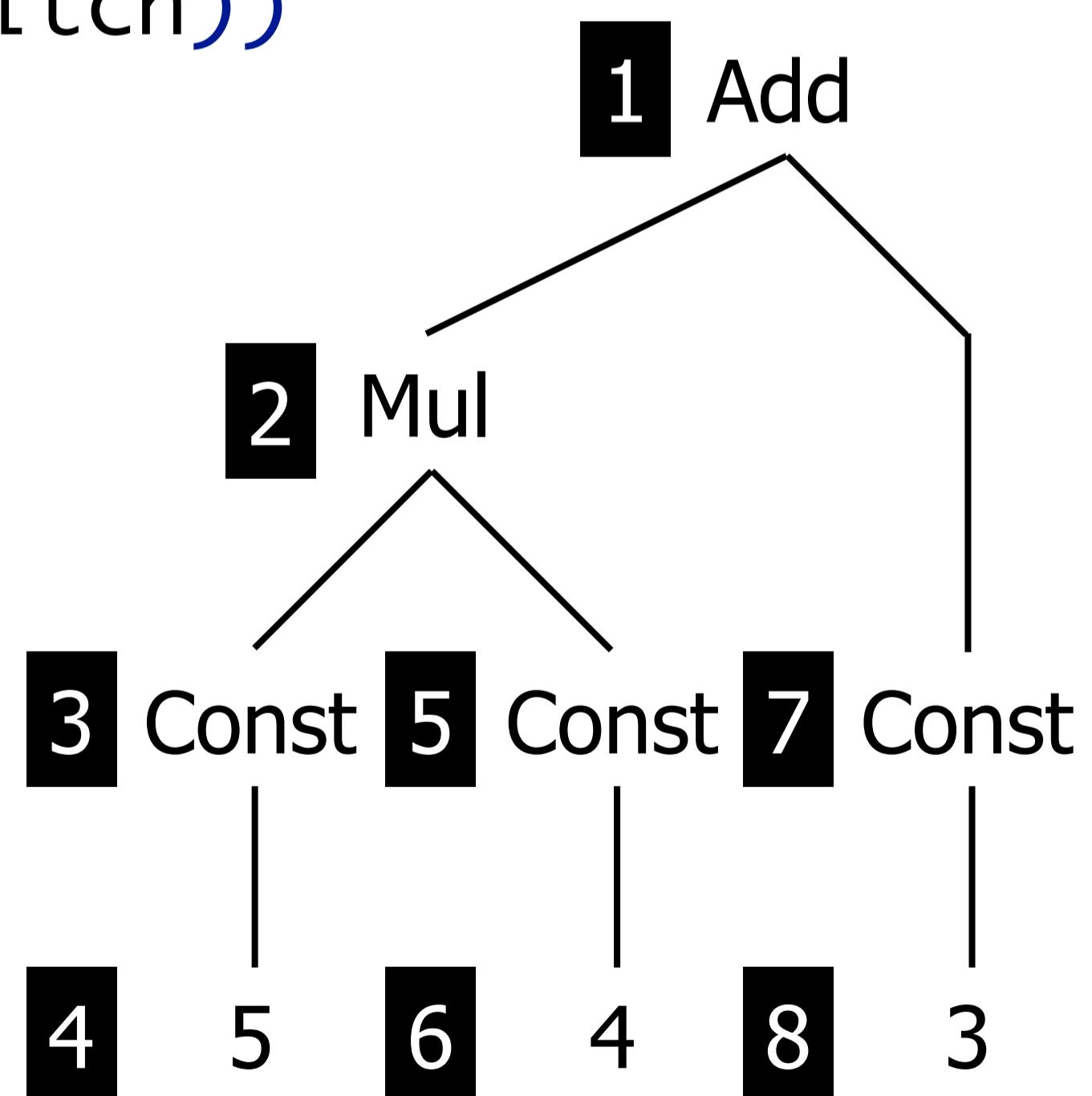
Traversal: Topdown/Try

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

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

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

topdown(try(switch))



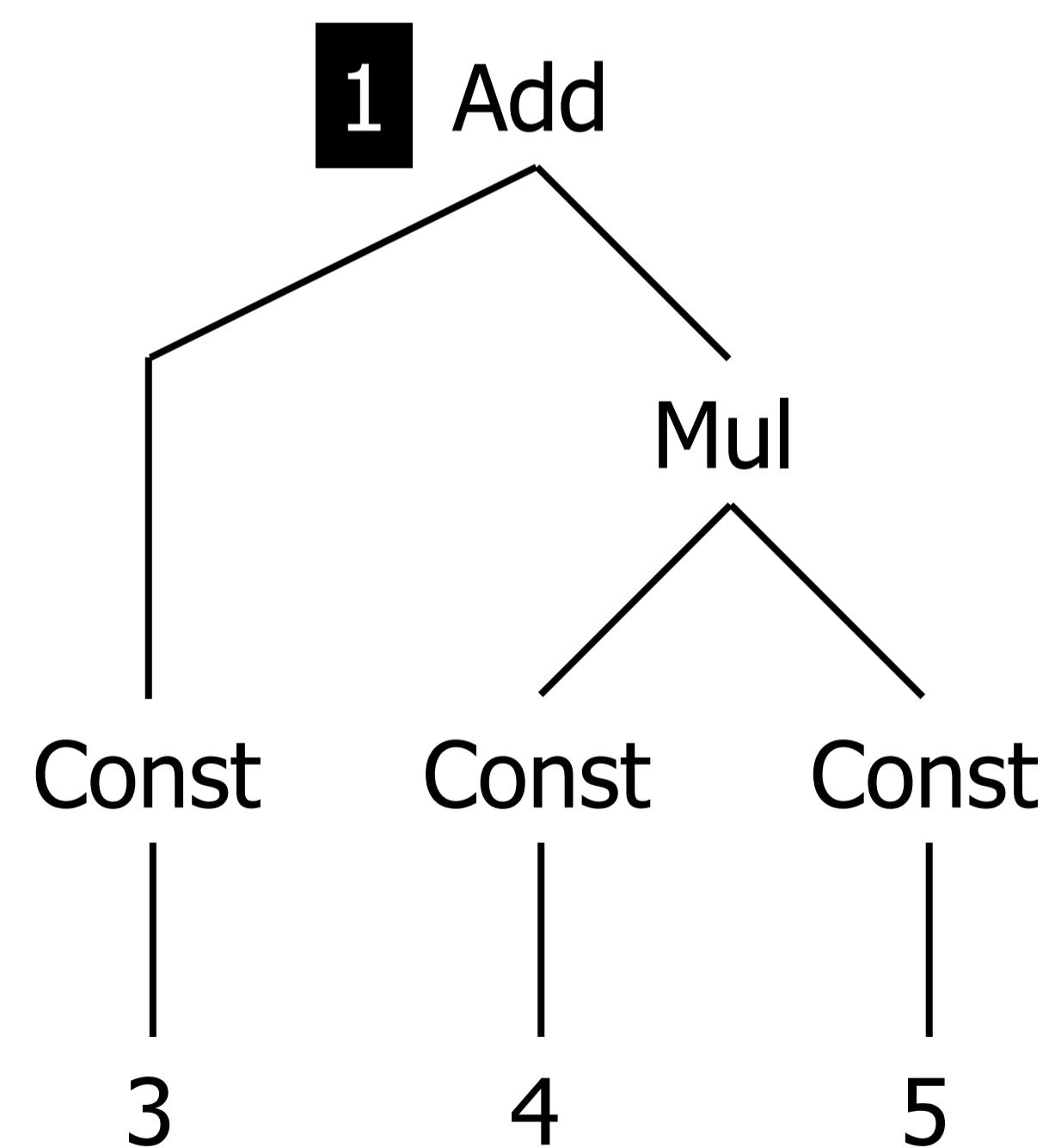
Traversal: Alltd

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

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

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

alltd(switch)



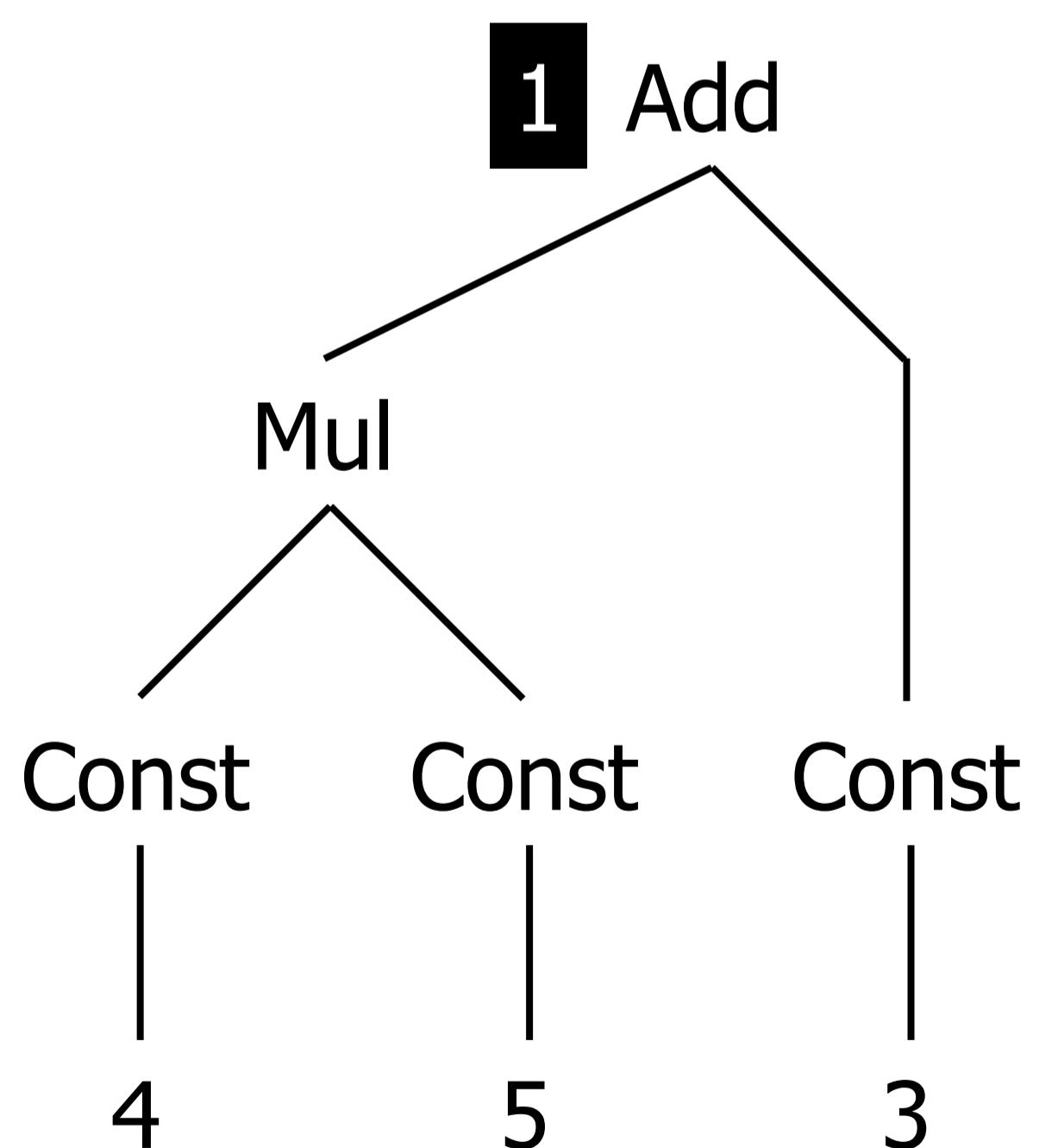
Traversal: Alltd

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

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

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

alltd(switch)

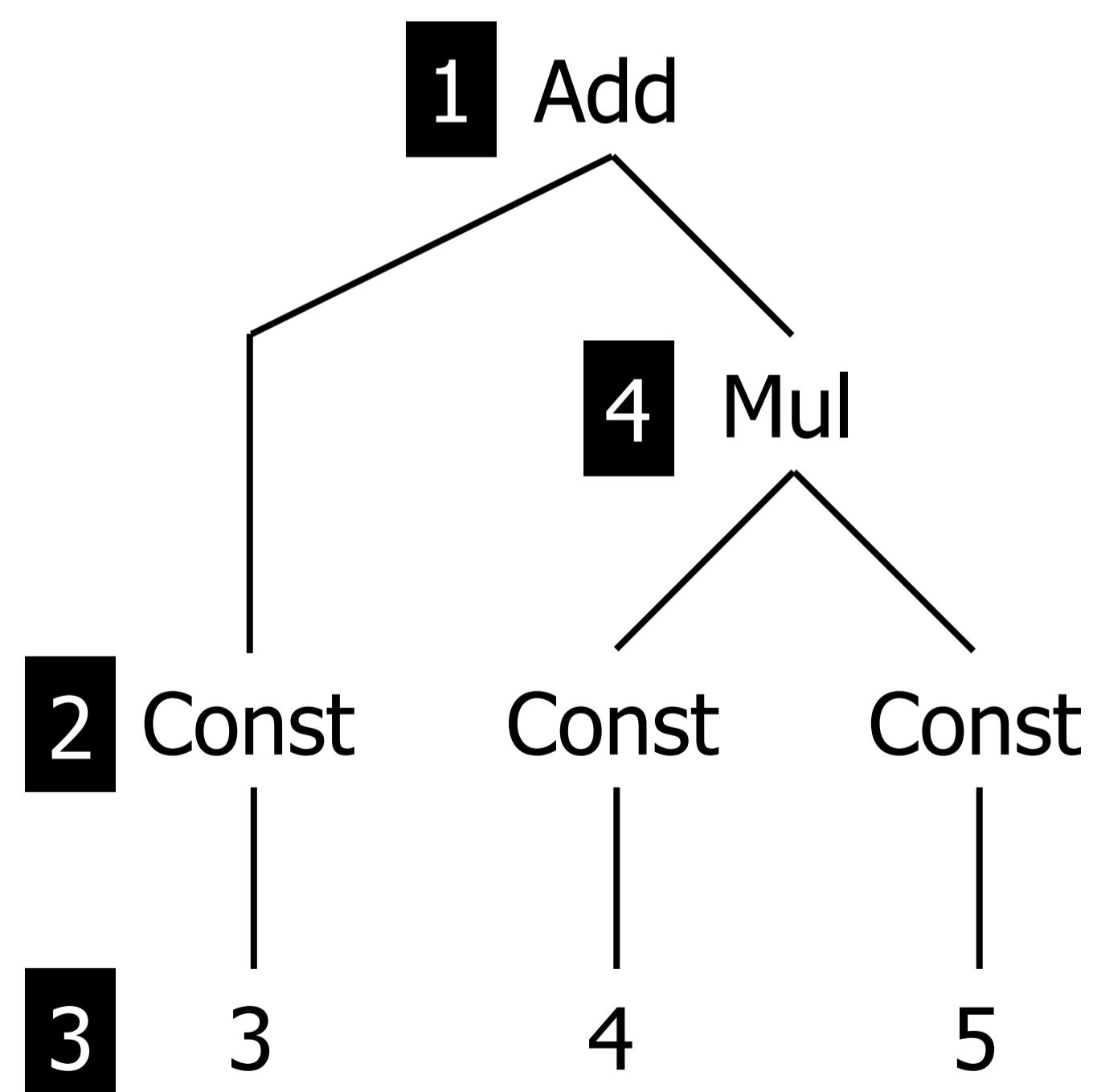


Traversal: Alltd

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

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

`alltd(switch)`

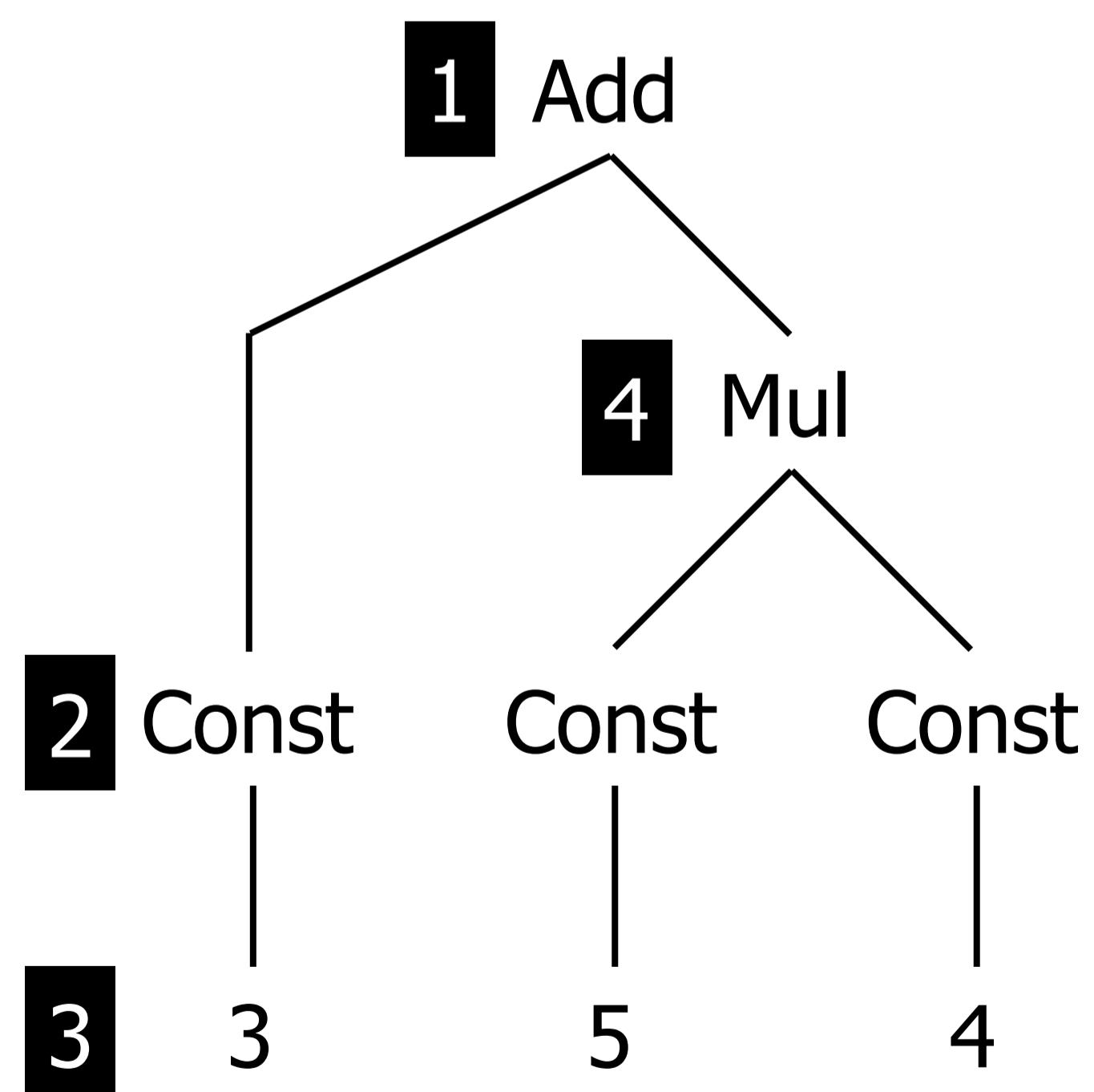


Traversal: Alltd

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

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

`alltd(switch)`

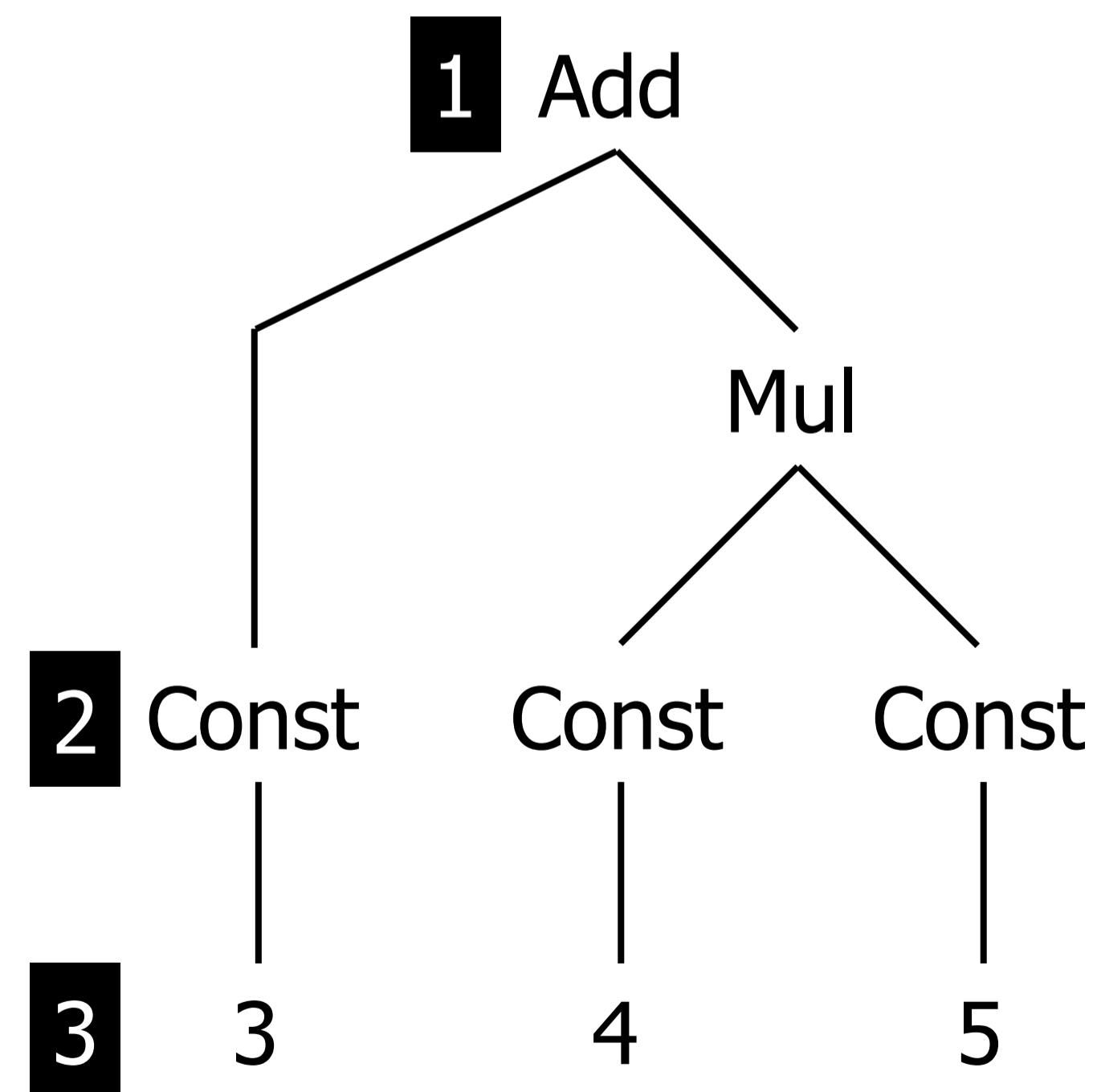


Traversal: Bottomup

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

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

```
bottomup(switch)
```

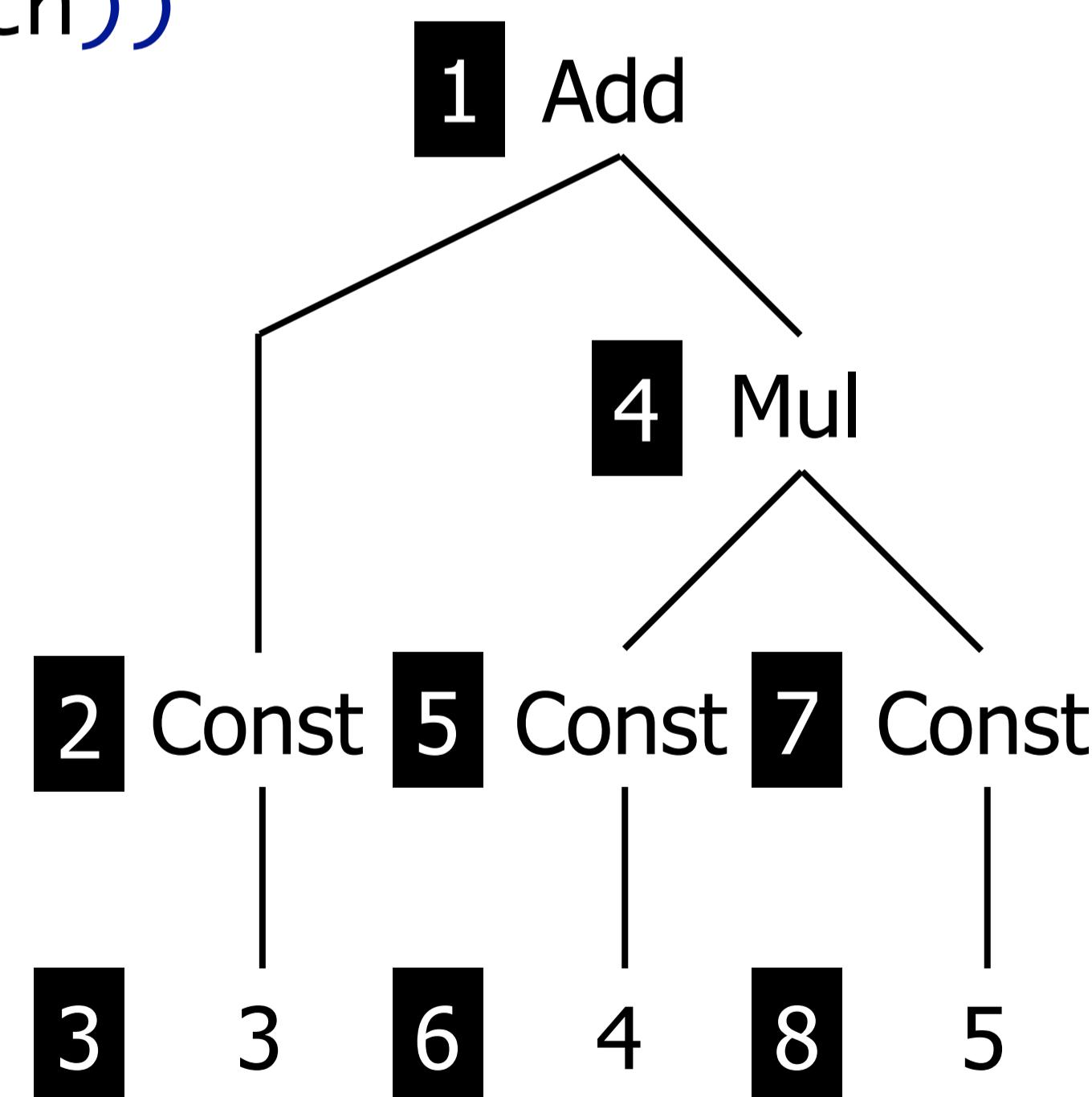


Traversal: Bottomup

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

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

```
bottomup(try(switch))
```

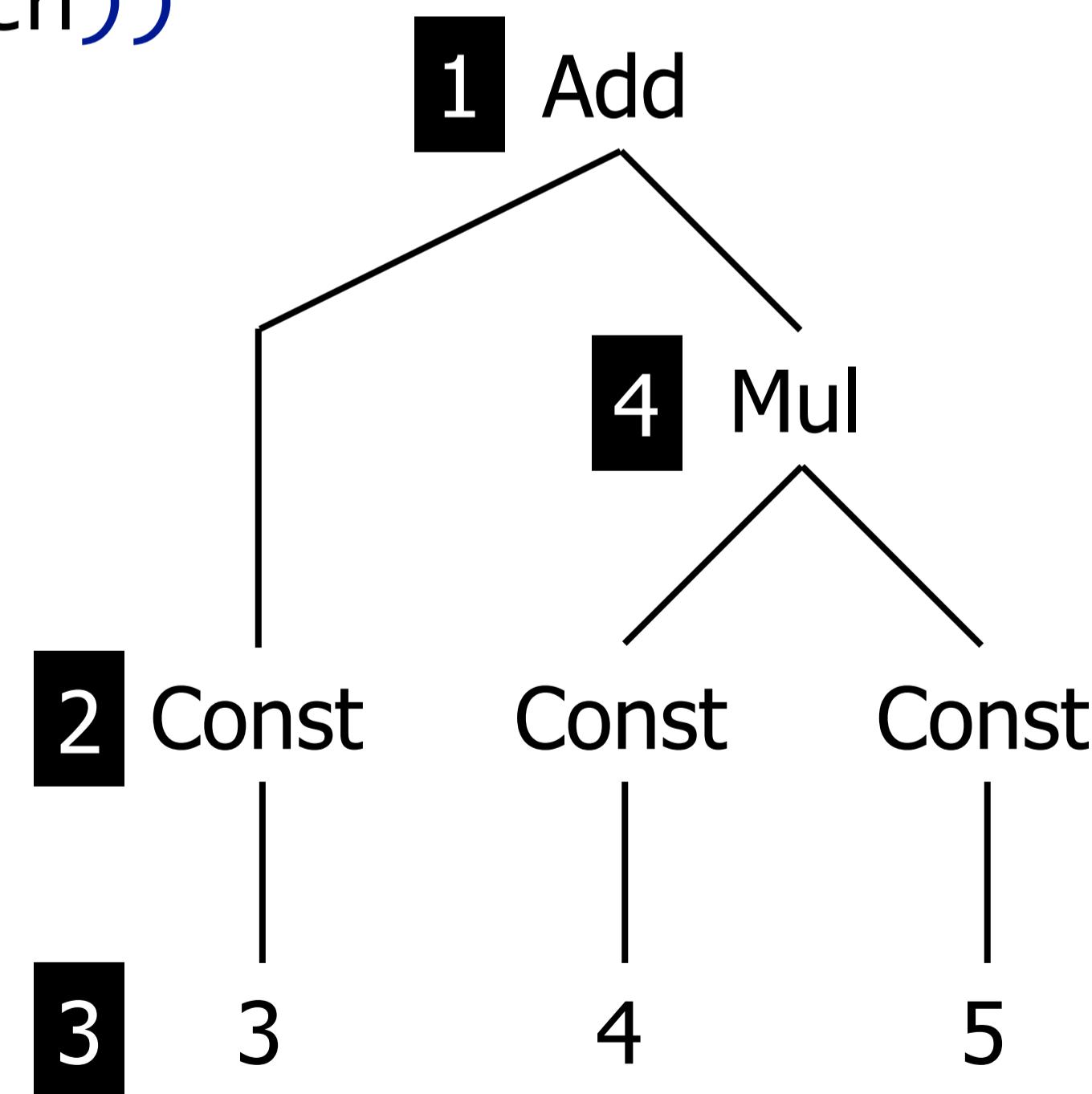


Traversal: Bottomup

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

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

```
bottomup(try(switch))
```

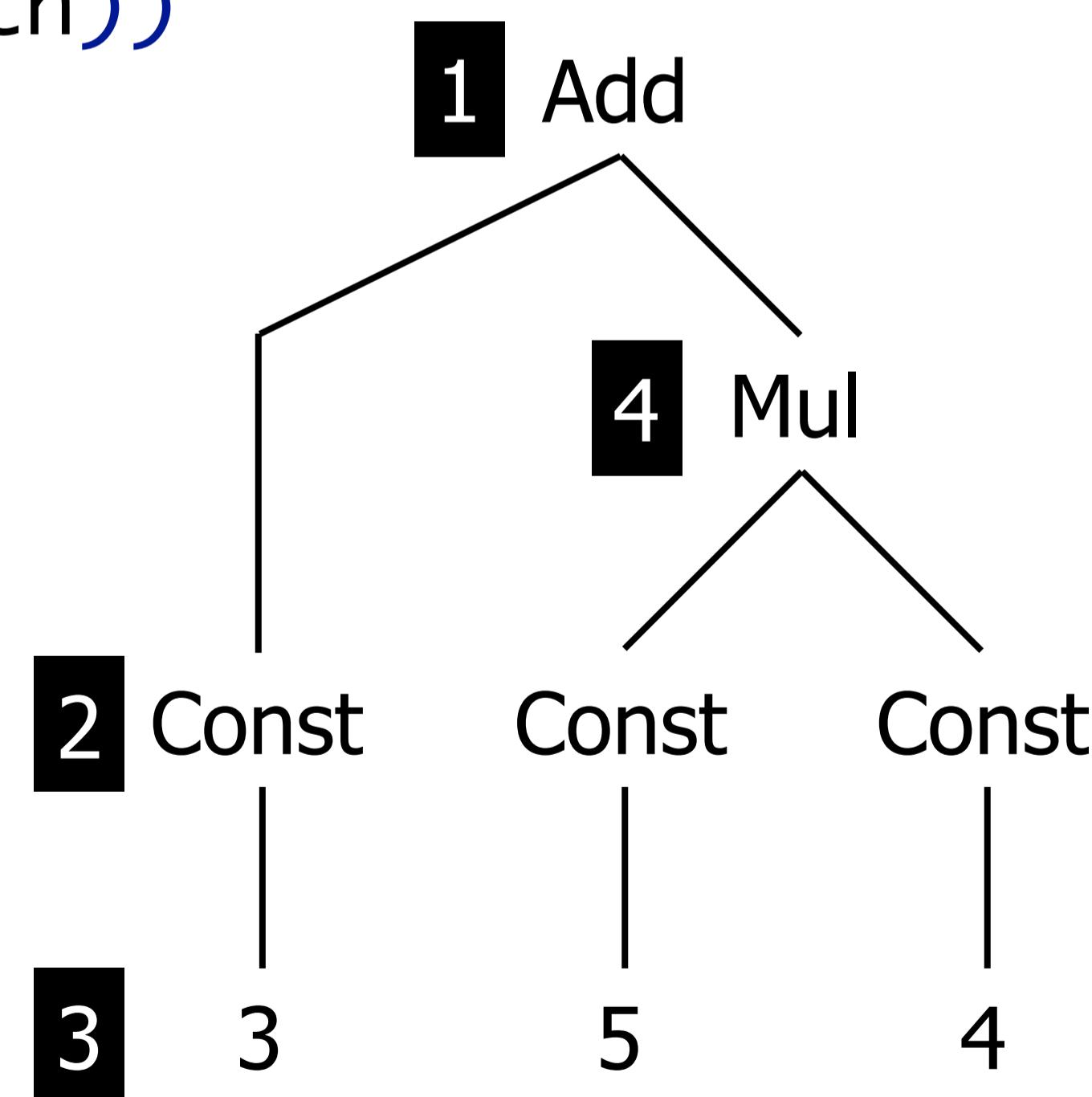


Traversal: Bottomup

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

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

```
bottomup(try(switch))
```

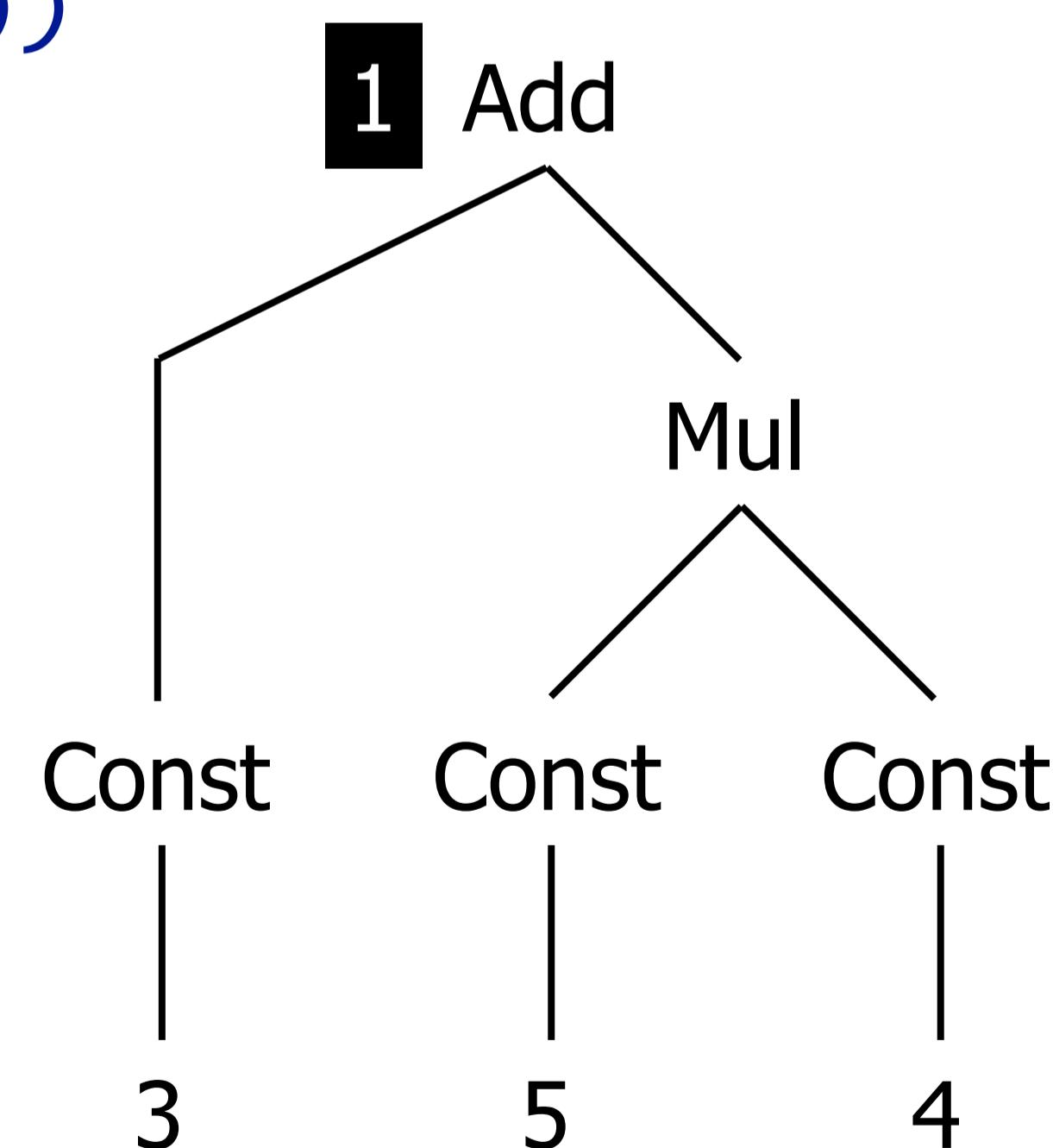


Traversal: Bottomup

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

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

```
bottomup(try(switch))
```

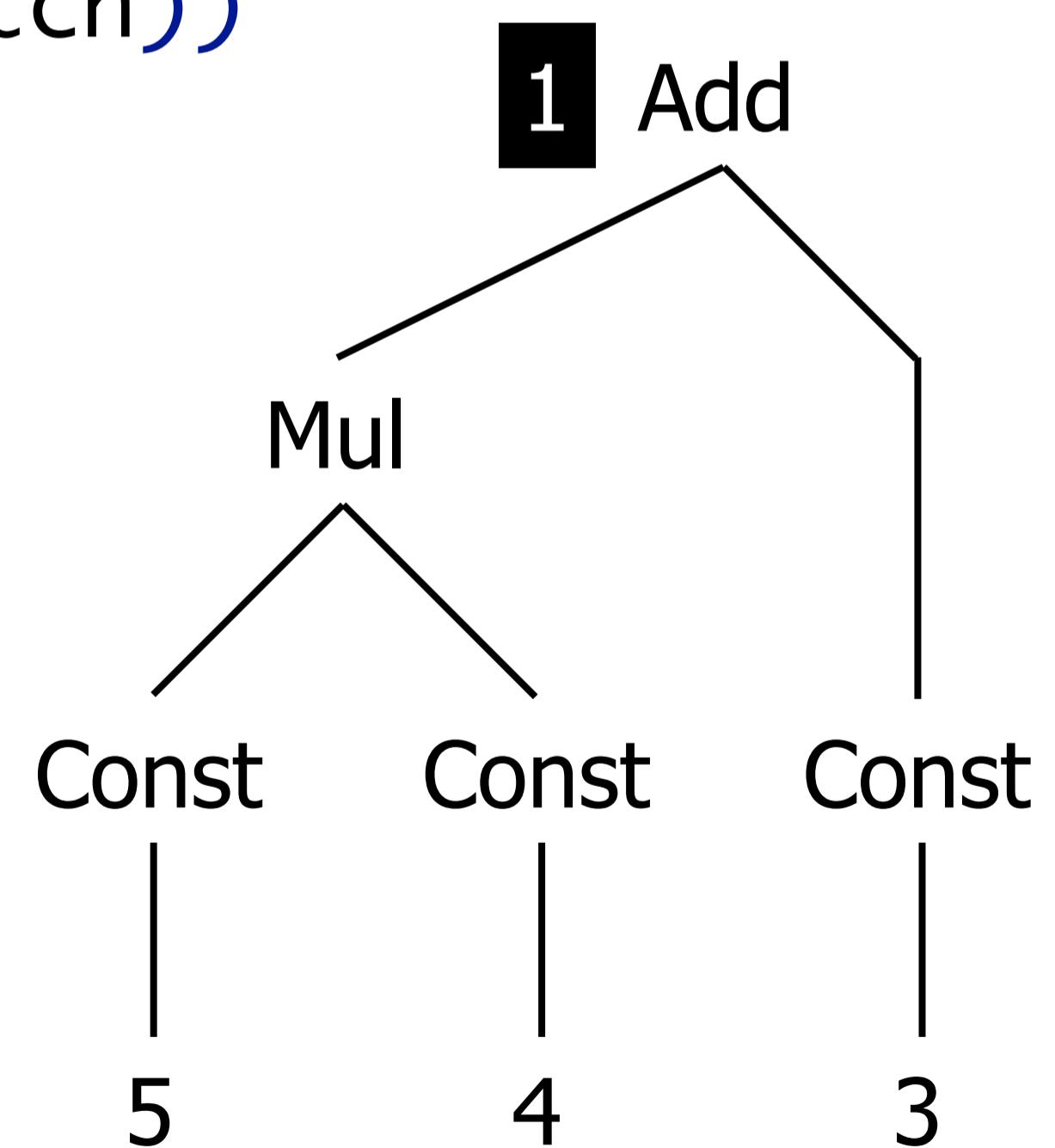


Traversal: Bottomup

```
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)
```

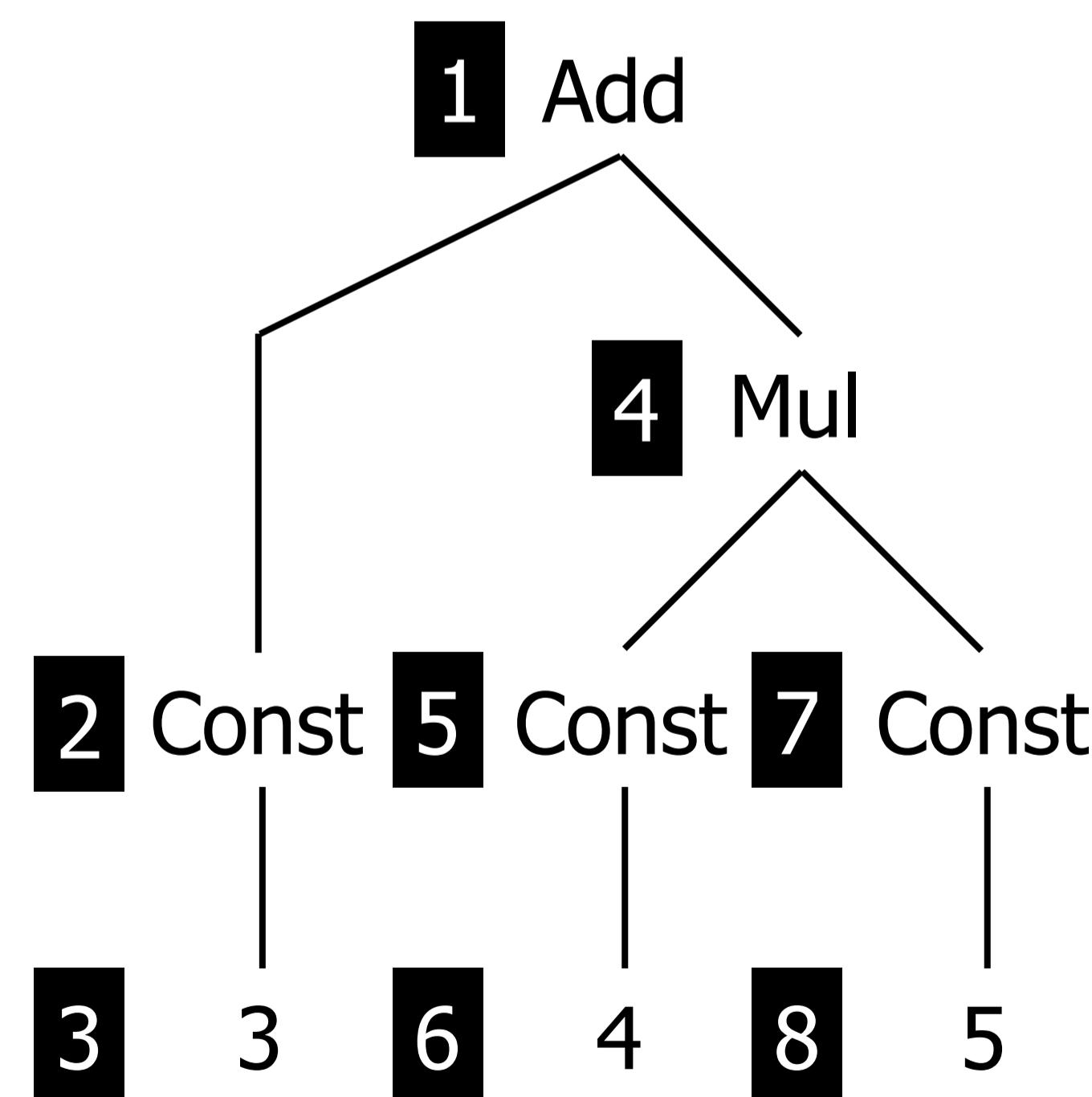
Traversal: Innermost

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

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

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

innermost(switch)



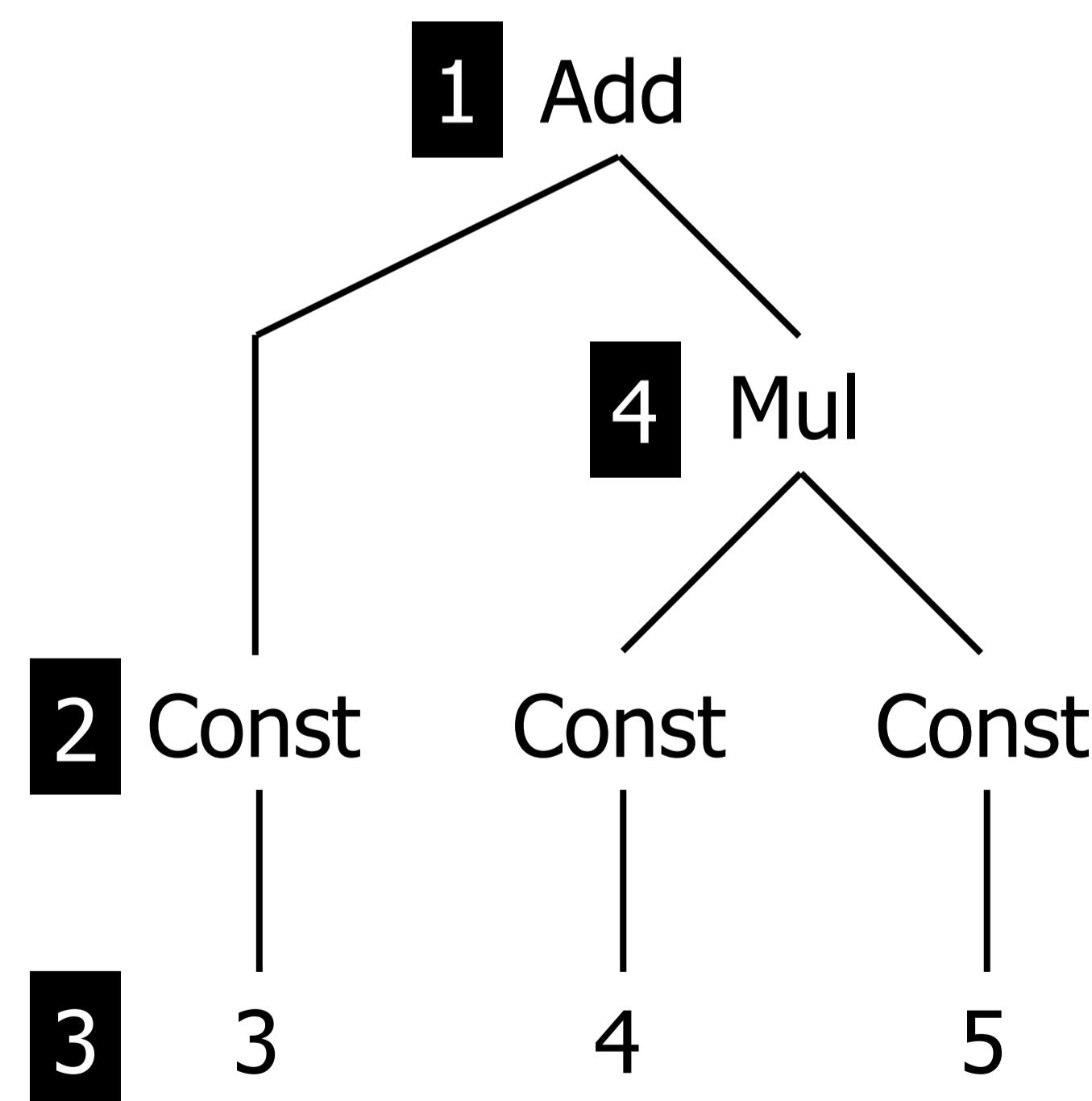
Traversal: Innermost

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

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

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

innermost(switch)



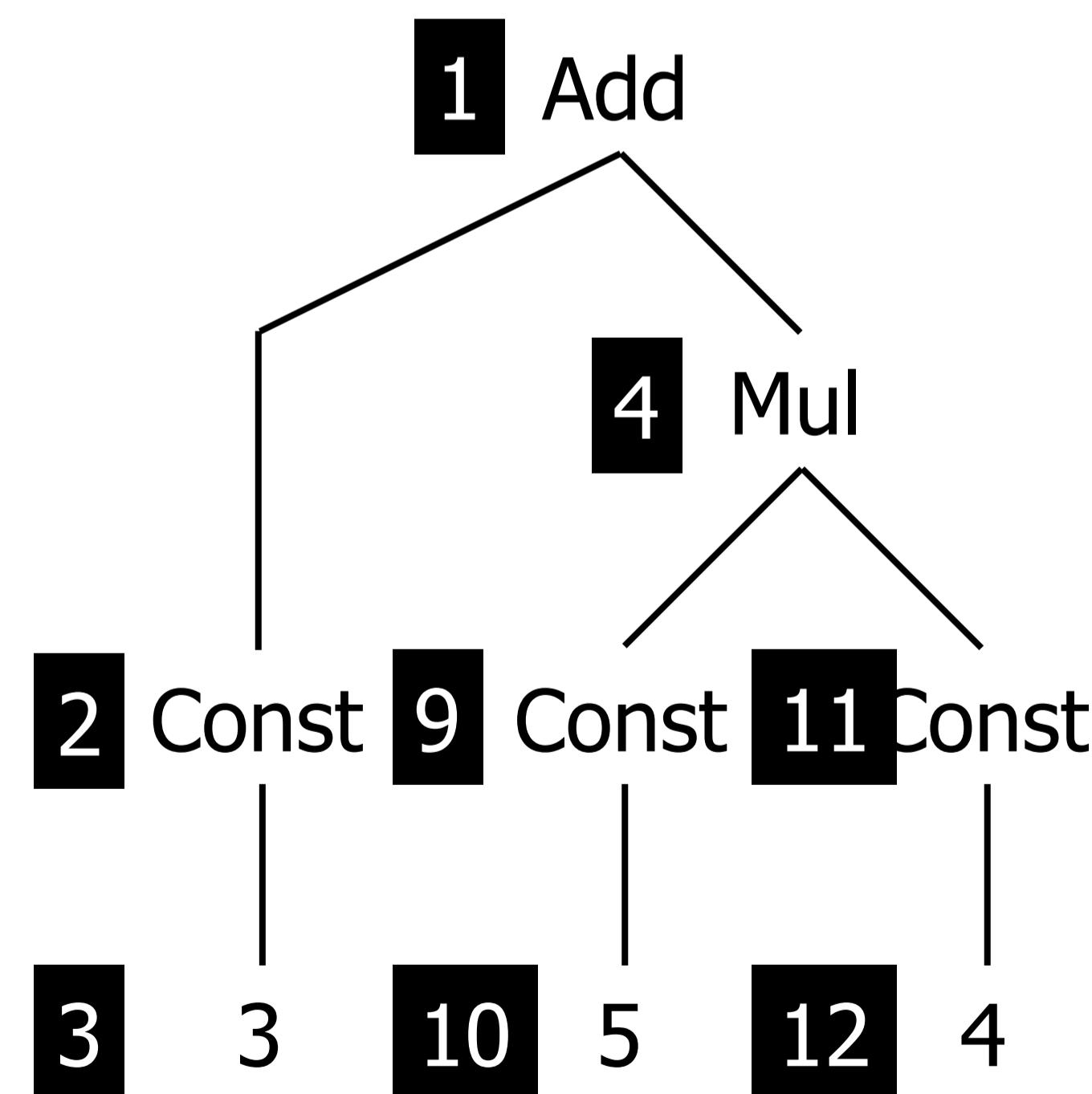
Traversal: Innermost

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

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

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

innermost(switch)



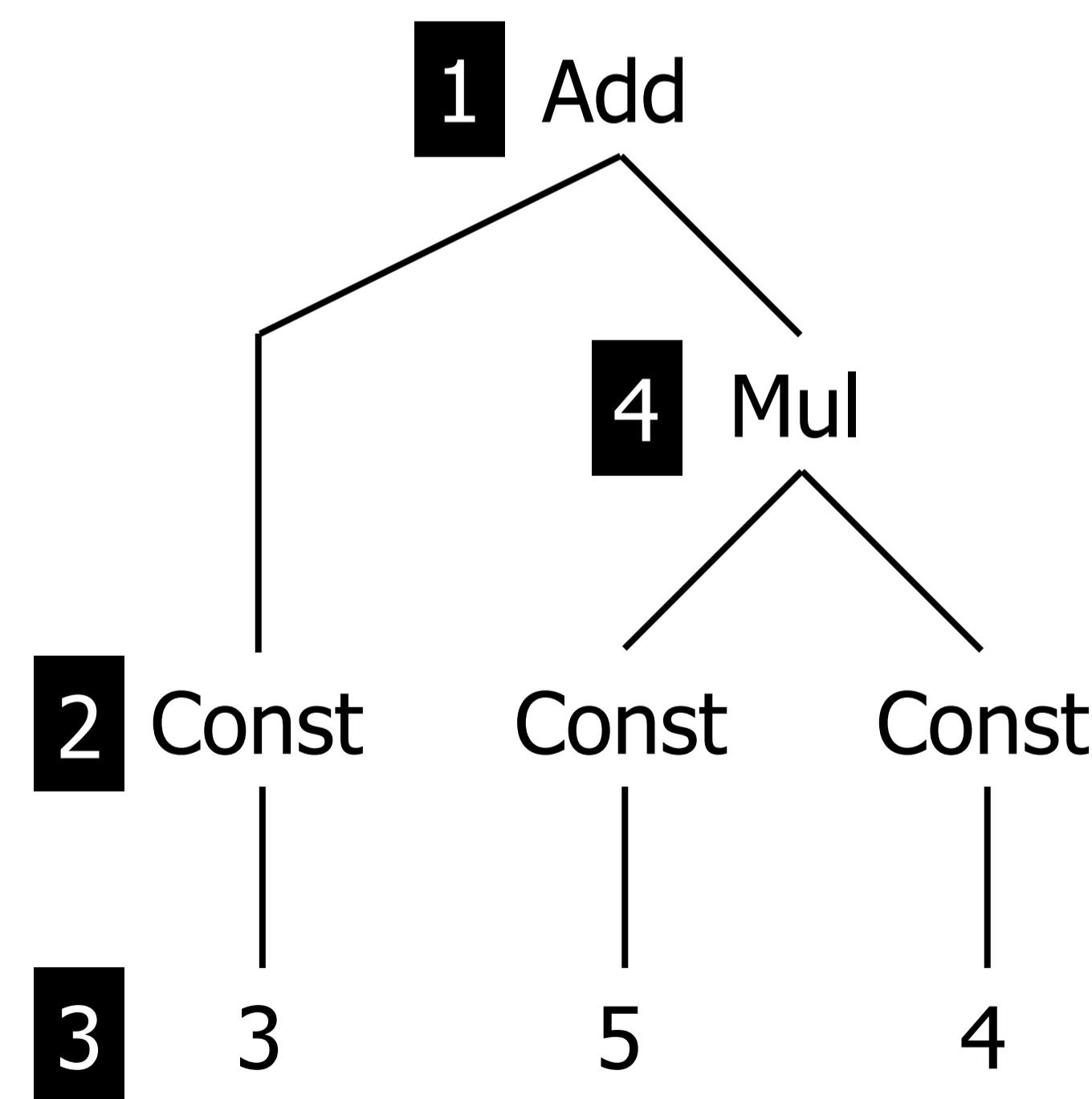
Traversal: Innermost

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

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

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

innermost(switch)



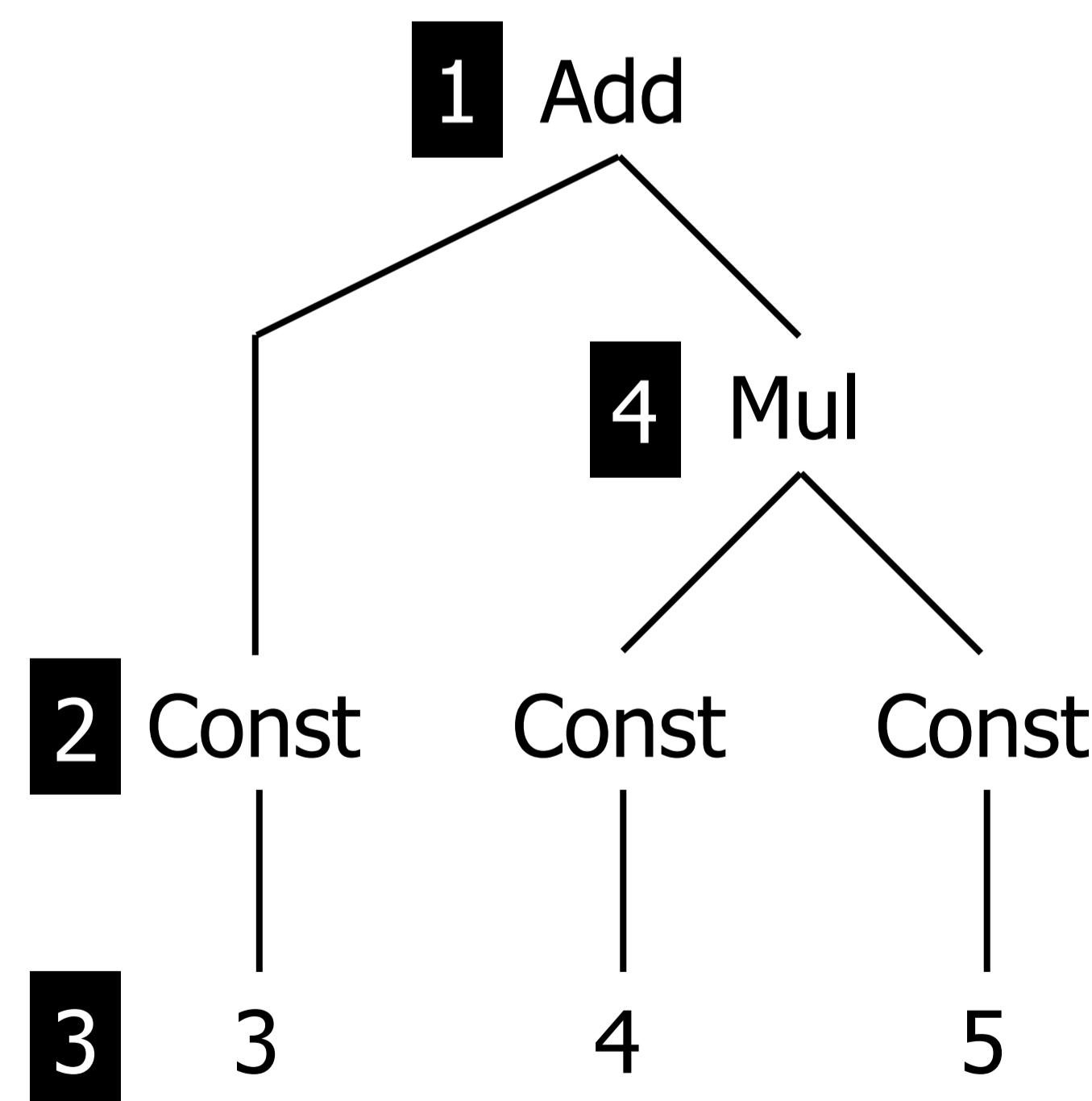
Traversal: Innermost

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

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

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

innermost(switch)



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 :

$\text{Minus}(e_1, e_2) \rightarrow \langle \text{add} \rangle (1, \langle \text{add} \rangle (e_1, e_2))$

AssignSize :

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

Generic definition

CSize :

$C(e_1, e_2, \dots) \rightarrow \langle \text{add} \rangle (1, \langle \text{add} \rangle (e_1, \langle \text{add} \rangle (e_2, \dots)))$

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

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}

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

Spoofax Builders

Spoofax Builders

```
rules

build :
  (node, _, ast, path, project-path) -> result
with
  <some-transformation> ast => result
```

Builder gets AST as parameter

Tiger: AST Builder

```
module Syntax

menus

    menu: "Syntax" (openeditor)

        action: "Format"          = editor-format (source)
        action: "Show parsed AST" = debug-show-aterm (source)
```

editor/syntax.esv

trans/tiger.str

```
rules // Debugging

debug-show-aterm:
    (node, _, _, path, project-path) -> (filename, result)
    with
        filename := <guarantee-extension(|"aterm")> path
    ; result   := node
```

Builder gets AST as parameter

Application: Formatting

Tiger: Formatting Builder

```
module Syntax
```

```
menus
```

```
menu: "Syntax" (openeditor)
```

```
action: "Format" = editor-format (source)
```

```
action: "Show parsed AST" = debug-show-aterm (source)
```

editor/syntax.esv

trans/pp.str

trans/pp.str

```
rules
```

```
editor-format:
```

```
(node, _, ast, path, project-path) -> (filename, result)
```

```
with
```

```
  ext      := <get-extension> path
```

```
; filename := <guarantee-extension(1$[pp.[ext]])> path
```

```
; result   := <pp-debug> node
```

rules

```
pp-Tiger-string =  
  parenthesize-Tiger  
  ; prettyprint-Tiger-start-symbols  
  ; !V([], <id>)  
  ; box2text-string(1120)
```

```
pp-debug :
```

```
  ast -> result
```

```
  with
```

```
    result := <pp-Tiger-string> ast
```

```
    <+ ...
```

```
; result := ""
```

Tiger: Parenthesize

```
module pp/Tiger-parenthesize

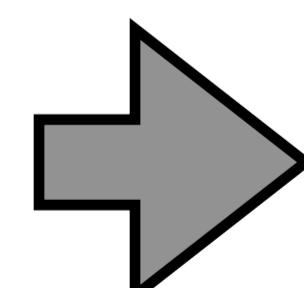
imports
    libstratego-lib
    signatures/-

strategies

parenthesize-Tiger =
    innermost(TigerParenthesize)
```

```
context-free priorities

Exp.And
> Exp.Or
> Exp.Array
> Exp.Assign
> ...
```



```
rules
TigerParenthesize :
    Or(t_0, t_1) -> Or(t_0, Parenthetical(t_1))
    where <(?For(_, _, _, _)
        + ?While(_, _)
        + ?IfThen(_, _)
        + ?If(_, _, _)
        + ?Assign(_, _)
        + ?Array(_, _, _)
        + ?Or(_, _)
        + fail)> t_1
```

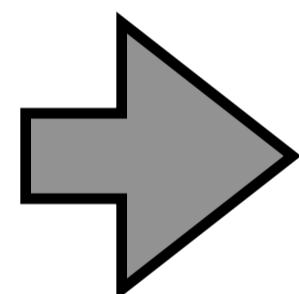
```
TigerParenthesize :
    And(t_0, t_1) -> And(Parenthetical(t_0), t_1)
    where <(?For(_, _, _, _)
        + ?While(_, _)
        + ?IfThen(_, _)
        + ?If(_, _, _)
        + ?Assign(_, _)
        + ?Array(_, _, _)
        + ?Or(_, _)
        + fail)> t_0
```

pp/Tiger-parenthesize.str

Tiger: Pretty-Print Rules

context-free syntax

```
Exp.If = <  
    if <Exp> then  
        <Exp>  
    else  
        <Exp>  
>
```



syntax/Control-Flow.sdf3

rules

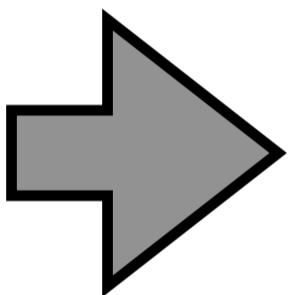
```
prettyprint-Tiger-Exp :  
    If(t1__, t2__, t3__) -> [ HC  
        [SOpt(HS(), "0")]  
        , [ S("if ")  
            , t1__'  
            , S(" then")  
        ]  
    )  
    , t2__'  
    , HC  
        [SOpt(HS(), "0")]  
        , [S("else")]  
    )  
    , t3__'  
]  
  
with t1__' := <pp-one-Z(prettyprint-Tiger-Exp)  
            <+ pp-one-Z(prettyprint-completion-aux)> t1__  
with t2__' := <pp-indent(I"2")> [  
    <pp-one-Z(prettyprint-Tiger-Exp)  
    <+ pp-one-Z(prettyprint-completion-aux)> t2__ ]  
with t3__' := <pp-indent(I"2")> [  
    <pp-one-Z(prettyprint-Tiger-Exp)  
    <+ pp-one-Z(prettyprint-completion-aux)> t3__ ]
```

pp/Control-Flow-pp.str

Application: Desugaring

Tiger: Desugaring

```
function printboard() = (
  for i := 0 to N-1 do (
    for j := 0 to N-1 do
      print(if col[i]=j then " 0" else " .");
      print("\n")
    );
    print("\n")
)
```



```
function printboard() = (
  let
    var i : int := 0
  in
    while i < N - 1 do (
      let
        var j : int := 0
      in
        while j < N - 1 do (
          print(if col[i] = j then
                " 0"
              else
                " .");
          j := j + 1
        );
        print("\n")
      );
      i := i + 1
    );
    print("\n")
)
```

Expressing for in terms of while++

Tiger: Desugaring Builder

```
module Transformation
```

```
menus
```

```
menu: "Transform" (openeditor) (realtime)
```

```
action: "Desugar"      = editor-desugar (source)
```

```
action: "Desugar AST" = editor-desugar-ast (source)
```

editor/Transformation.esv

```
rules // Desugaring
```

```
editor-desugar :
```

```
(node, _, _, path, project-path) -> (filename, result)
```

```
with
```

```
filename := <guarantee-extension(I"des.tig")> path
```

```
; result   := <desugar-all; pp-Tiger-string>node
```

```
editor-desugar-ast :
```

```
(node, _, _, path, project-path) -> (filename, result)
```

```
with
```

```
filename := <guarantee-extension(I"aterm")> path
```

```
; result   := <desugar-all>node
```

trans/tiger.str

Tiger: Desugaring Transformation

```
module desugar
imports signatures/Tiger-sig
imports ...
strategies
  desugar-all = topdown(try(desugar))
rules
  desugar :
    For(
      Var(i)
    , e1
    , e2
    , e_body
    ) ->
    Let(
      [VarDec(i, Tid("int"), e1)]
    , [ While(
        Lt(Var(i), e2)
      , Seq(
        [ e_body
        , Assign(Var(i), Plus(Var(i), Int("1")))]
      )
    )
  ]
```

Application: Outline View

Tiger: Outline View

The screenshot shows the Tiger IDE interface with two main windows: the code editor and the outline view.

Code Editor (queens.tig):

```
1 /* A program to solve the 8-queens problem */
2
3 let
4     var N := 8
5
6     type intArray = array of int
7
8     var row := intArray [ N ] of 0
9     var col := intArray [ N ] of 0
10    var diag1 := intArray [N+N-1] of 0
11    var diag2 := intArray [N+N-1] of 0
12
13    function printboard() =
14        for i := 0 to N-1 do (
15            for j := 0 to N-1 do
16                print(if col[i]=j then " Q" else ".");
17                print("\n")
18            );
19            print("\n")
20        )
21
22    function try(c:int) =
23        if c=N then
24            printboard()
25        else
26            for r := 0 to N-1 do
27                if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0 then (
28                    row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
29                    col[c]:=r;
30                    try(c+1);
31                    row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0
32                )
33            )
34    in try(0)
35 end
```

Outline View:

- let
 - var N
 - type intArray
 - var row
 - var col
 - var diag1
 - var diag2
 - fun printboard
 - for i
 - for j
 - print()
 - if
 - print()
 - print()
 - fun try
 - if
 - try()

Tiger: Outline View Builder

module Syntax

views

```
outline view: editor-outline (source)
  expand to level: 3
```

trans/outline.str

editor/syntax.esv

module outline

imports

```
signatures/Tiger-sig
signatures/Types-sig
signatures/Records-sig
signatures/Variables-sig
signatures/Functions-sig
signatures/Bindings-sig
libsfoofax/editor/outline
pp
```

rules

editor-outline:

```
(_, _, ast, path, project-path) -> outline
```

where

```
outline := <simple-label-outline(to-outline-label)> ast
```

Tiger: Outline View Rules

rules

```
to-outline-label :  
  Let(_, _) -> ${[let]}
```

```
to-outline-label :  
  TypeDec(name, _) -> ${[type [name]]}
```

```
to-outline-label :  
  FunDec(name, _, t, _) -> ${[fun [name] : [<pp-tiger-type> t]]}
```

```
to-outline-label :  
  ProcDec(name, _, _) -> ${[fun [name]]}
```

```
to-outline-label :  
  VarDecNoType(name, _) -> ${[var [name]]}
```

```
to-outline-label :  
  VarDec(name, _, _) -> ${[var [name]]}
```

trans/outline.str

rules

```
to-outline-label :  
  Call(f, _) -> ${[f]()}]
```

```
to-outline-label :  
  If(_, _, _) -> ${[if]}
```

```
to-outline-label :  
  For(Var(name), _, _, _) -> ${[for [name]]}
```

// etc.

Outline View: Generic Strategy

```
module libspoofax/editor/outline
imports ...
signature
constructors
  Node : Label * Children -> Node
rules

  simple-label-outline(s1) =
    collect-om(to-outline-node(s1, fail), conc)

  custom-label-outline(s1, s2) =
    collect-om(origin-track-forced(s2) <+ to-outline-node(s1, s2), conc)

  to-outline-node(s1, s2):
    term -> Node(label, children)
    where
      random := <next-random>;
      label := <origin-track-forced(s1; term-to-outline-label)> term;
      children := <get-arguments; custom-label-outline(s1, s2)> term

  term-to-outline-label =
    is-string
    <+ ?term{a}; origin-text; ?label; !label{a}
    <+ write-to-string // fallback
```

Term structure for outline nodes

Language-independent strategy for collecting outline nodes

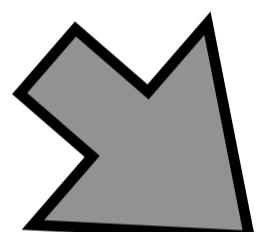
Application: Completion

Tiger: Completion Rules

context-free syntax

```
Exp.If = <  
  if <Exp> then  
    <Exp>  
  else  
    <Exp>  
>
```

syntax/Control-Flow.sdf3



rules

```
suggest-completions(|completions):  
  Exp-Plhdr() -> <add-completions(  
    | ( "If"  
      , If(  
        <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()  
        , <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()  
        , <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()  
      )  
    )  
  ); fail> completions
```

completion/Control-Flow-cp.str

Except where otherwise noted, this work is licensed under

