Multi-Purpose Syntax Definition with SDF3

Eelco Visser

TU Delft

SEFM | ‘CWI, Amsterdam’ | September 16, 2020

UvA/CWI
1995

$\Rightarrow$

TU Delft
2020

SDF2

1995

A Family of Syntax Definition Formalisms

Eelco Visser

Programming Research Group, University of Amsterdam,
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
email: visser@fwi.uva.nl, http://adam.fwi.uva.nl/~visser/

Abstract. In this paper we design a syntax definition formalism as a family of formalisms. Starting with a small kernel, various features for syntax definition are designed orthogonally to each other. This provides a framework for constructing new formalisms by adapting and extending old ones. The formalism is developed with the algebraic specification formalism ASF+SDF. It provides the following features: lexical and context-free syntax, variables, disambiguation by priorities, regular expressions, character classes and modular definitions. New are the uniform treatment of lexical syntax, context-free syntax and variables, the treatment of regular expressions by normalization yielding abstract syntax without auxiliary sorts, regular expressions as result of productions and modules with hidden imports and renamings.

Key Words & Phrases: syntax definition formalism, language design, context-free grammar, context-free syntax, lexical syntax, priorities, regular expressions, formal language, parsing, abstract syntax, module, renaming, hidden imports

Note: Supported by the Dutch Organization for Scientific Research (NWO) under grant 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

1 Introduction

1.1 General

New programming, specification and special purpose languages are being developed continuously [C+94]. Syntax definition formalisms play a crucial role in the design and implementation of new languages. Syntax definition formalisms also play a role embedded in other languages: regular expressions in edit operations, macro definitions for macro preprocessors, user definable infix or distfix operators in programming languages, grammars as signatures in algebraic specification formalisms, and documents that contain a description of their own syntax.

The core of many syntax definition formalisms is formed by context-free grammars, which are widely used in computer science since their introduction by Chomsky in 1956 [Cho56]. A context-free grammar is a set of string rewrite rules of the form $\alpha \to \mathcal{A}$. A string w is member of the language described by a grammar $\mathcal{G}$ if it can be rewritten to the start symbol S, i.e., if there is a sequence $w = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_n = S$ and each step has the form $\alpha_i \beta_i \gamma_i \to \alpha_i \mathcal{B}_i \gamma_i$ where $\beta_i \to \mathcal{B}_i$ is a production in $\mathcal{G}$.

Despite, or maybe due to, the simplicity of this basic structure there has never emerged a standard formalism for syntax definition. The Backus Naur Form (BNF) [Bac59, N+60], originally developed for the definition of the syntax of Algol, is a commonly used notation for context-free grammars, but it does not have the status of a standard. Several standard notations for syntax definition have been proposed [Wir77, Wil82]. None of these has been convincing, instead a number of similar or overlapping formalisms exist.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

SDF3

2020

Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza[1] and Eelco Visser[2]

[1] Australian National University, Australia
[2] Delft University of Technology, The Netherlands

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permissive syntax, layout constraints, formatting templates, placeholder syntax, and modular composition. These features support the multi-purpose interpretation of syntax definitions, including derivation of type schemas for abstract syntax tree representations, scannerless generalized parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesization and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition · programming language · parsing.

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a *grammar formalism* in the tradition of Chomsky's context-free grammars [14] and the Backus-Naur Form [4]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASF+SDF [5], Stratego/XT [10], and Spoofax [38] language workbenches.

The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete and abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalized-LR (GLR) parsing algorithm [56,44]. Its programming environment, as part of the ASF+SDF MetaEnvironment [40], focused on live development of syntax definitions through

To appear in: F. S. de Boer and A. Cerone (Eds.). *Software Engineering and Formal Methods (SEFM 2020)*, LNCS, Springer, 2020.

SDF2

1995

SDF3

2020

SDF2

A Family of Syntax Definition Formalisms

Eelco Visser

Programming Research Group, University of Amsterdam,
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands
email: visser@fwi.uva.nl, http://adam.fwi.uva.nl/~visser/

Abstract. In this paper we design a syntax definition formalism as a family of formalisms. Starting with a small kernel, various features for syntax definition are designed orthogonally to each other. This provides a framework for constructing new formalisms by adapting and extending old ones. The formalism is developed with the algebraic specification formalism ASF+SDF. It provides the following features: lexical and context-free syntax, variables, disambiguation by priorities, regular expressions, character classes and modular definitions. New are the uniform treatment of lexical syntax, context-free syntax and variables, the treatment of regular expressions by normalization yielding abstract syntax without auxiliary sorts, regular expressions as result of productions and modules with hidden imports and renamings.

Key Words & Phrases: syntax definition formalism, language design, context-free grammar, context-free syntax, lexical syntax, priorities, regular expressions, formal language, parsing, abstract syntax, module, renaming, hidden imports

Note: Supported by the Dutch Organization for Scientific Research (NWO) under grant 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

1 Introduction

1.1 General

New programming, specification and special purpose languages are being developed continuously [C+94]. Syntax definition formalisms play a crucial role in the design and implementation of new languages. Syntax definition formalisms also play a role embedded in other languages: regular expressions in edit operations, macro definitions for macro preprocessors, user definable infix or distfix operators in programming languages, grammars as signatures in algebraic specification formalisms, and documents that contain a description of their own syntax.

The core of many syntax definition formalisms is formed by context-free grammars, which are widely used in computer science since their introduction by Chomsky in 1956 [Cho56]. A context-free grammar is a set of string rewrite rules of the form $\alpha \to \mathcal{A}$. A string w is member of the language described by a grammar $\mathcal{G}$ if it can be rewritten to the start symbol S, i.e., if there is a sequence $w = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_n = S$ and each step has the form $\alpha_i \beta_i \gamma_i \to \alpha_i \mathcal{B}_i \gamma_i$ where $\beta_i \to \mathcal{B}_i$ is a production in $\mathcal{G}$.

Despite, or maybe due to, the simplicity of this basic structure there has never emerged a standard formalism for syntax definition. The Backus Naur Form (BNF) [Bac59, N+60], originally developed for the definition of the syntax of Algol, is a commonly used notation for context-free grammars, but it does not have the status of a standard. Several standard notations for syntax definition have been proposed [Wir77, Wil82]. None of these has been convincing, instead a number of similar or overlapping formalisms exist.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

1995

SDF3

Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza[1] and Eelco Visser[2]

[1] Australian National University, Australia
[2] Delft University of Technology, The Netherlands

Abstract. SDF3 is a syntax definition formalism that extends plain context-free grammars with features such as constructor declarations, declarative disambiguation rules, character-level grammars, permissive syntax, layout constraints, formatting templates, placeholder syntax, and modular composition. These features support the multi-purpose interpretation of syntax definitions, including derivation of type schemas for abstract syntax tree representations, scannerless generalized parsing of the full class of context-free grammars, error recovery, layout-sensitive parsing, parenthesization and formatting, and syntactic completion. This paper gives a high level overview of SDF3 by means of examples and provides a guide to the literature for further details.

Keywords: Syntax definition · programming language · parsing.

1 Introduction

A syntax definition formalism is a formal language to describe the syntax of formal languages. At the core of a syntax definition formalism is a *grammar formalism* in the tradition of Chomsky's context-free grammars [14] and the Backus-Naur Form [4]. But syntax definition is concerned with more than just phrase structure, and encompasses all aspects of the syntax of languages.

In this paper, we give an overview of the syntax definition formalism SDF3 and its tool ecosystem that supports the multi-purpose interpretation of syntax definitions. The paper does not present any new technical contributions, but it is the first paper to give a (high-level) overview of all aspects of SDF3 and serves as a guide to the literature. SDF3 is the third generation in the SDF family of syntax definition formalisms, which were developed in the context of the ASF+SDF [5], Stratego/XT [10], and Spoofax [38] language workbenches.

The first SDF [23] supported modular composition of syntax definition, a direct correspondence between concrete and abstract syntax, and parsing with the full class of context-free grammars enabled by the Generalized-LR (GLR) parsing algorithm [56,44]. Its programming environment, as part of the ASF+SDF MetaEnvironment [40], focused on live development of syntax definitions through

To appear in: F. S. de Boer and A. Cerone (Eds.). *Software Engineering and Formal Methods (SEFM 2020)*, LNCS, Springer, 2020.

2020

History of SDF

SDF | 1989

The Syntax Definition Formalism SDF [1989]

- Heering, Hendriks, Klint, Rekers

Lexical Syntax + Context-free Syntax

- Separate scanner, parser
- Syntax definition $\simeq$ algebraic signature

Generalized LR Parsing

- Support full class of context-free grammars
- Lazy, incremental, modular scanner, parser generation

Modular Syntax Definition

ASF+SDF MetaEnvironment

SDF2 | 1995 - 2010

Scannerless Generalized LR (SGLR) Parsing [1997]

- Support character-level grammars
- Lexical disambiguation (follow restrictions, reject productions)

Disambiguation Filters for Associativity and Priority

- Shallow conflicts: Unsafe for prefix/postfix operators with low priority

A Family of Syntax Definition Formalism [1995]

- Transform high-level language to Kernel SDF

Language Composition

- Meta-programming with concrete object syntax [2002]
- Concrete object syntax [2004]

Spoofax Language Workbench [2010]

SDF3 | 2010 - 2020

Multi-Purpose Syntax Definition

- Many tools from single source

Templates

- Formatting instructions from syntax definition

Semantics of Associativity and Priority

- Safe and Complete Disambiguation, Deep conflicts
- Parenthesis insertion

Layout-Sensitive Syntax

- layout constraints, layout declarations

Spoofax 2

Impact

Education

- Compiler Construction
- Language Engineering Project

Research

- Syntax definition in Spoofax Language Workbench
- Meta-Language Design: NaBL, Statix, Stratego, FlowSpec, …
- DSLs: WebDSL, IceDust, PIE

Industry

- Oracle Labs: Graph Analytics
- Canon: Oil, CSX
- Philips/MasCot: Software Restructuring

Main SDF3 Contributors

Error Recovery

- Kats, De Jonge

Templates

- Vollebregt, Kats

Layout Constraints

- Erdweg

Layout Declarations

- Eduardo Amorim

Disambiguation

- Eduardo Amorim

Syntactic Completion

- Eduardo Amorim

JSGLR2 (in progress)

- Denkers, Sijm

SDF3 Implementation

- Eduardo Amorim

This Talk

Phrase Structure

– constructors

Formatting Templates

– syntactic completion

Declarative Disambiguation

– from unsafe to safe disambiguation

Layout Constraints/Declarations

– for layout-sensitive syntax

Take away: Multi-Purpose Interpretation

– *See paper for more*

Phrase Structure

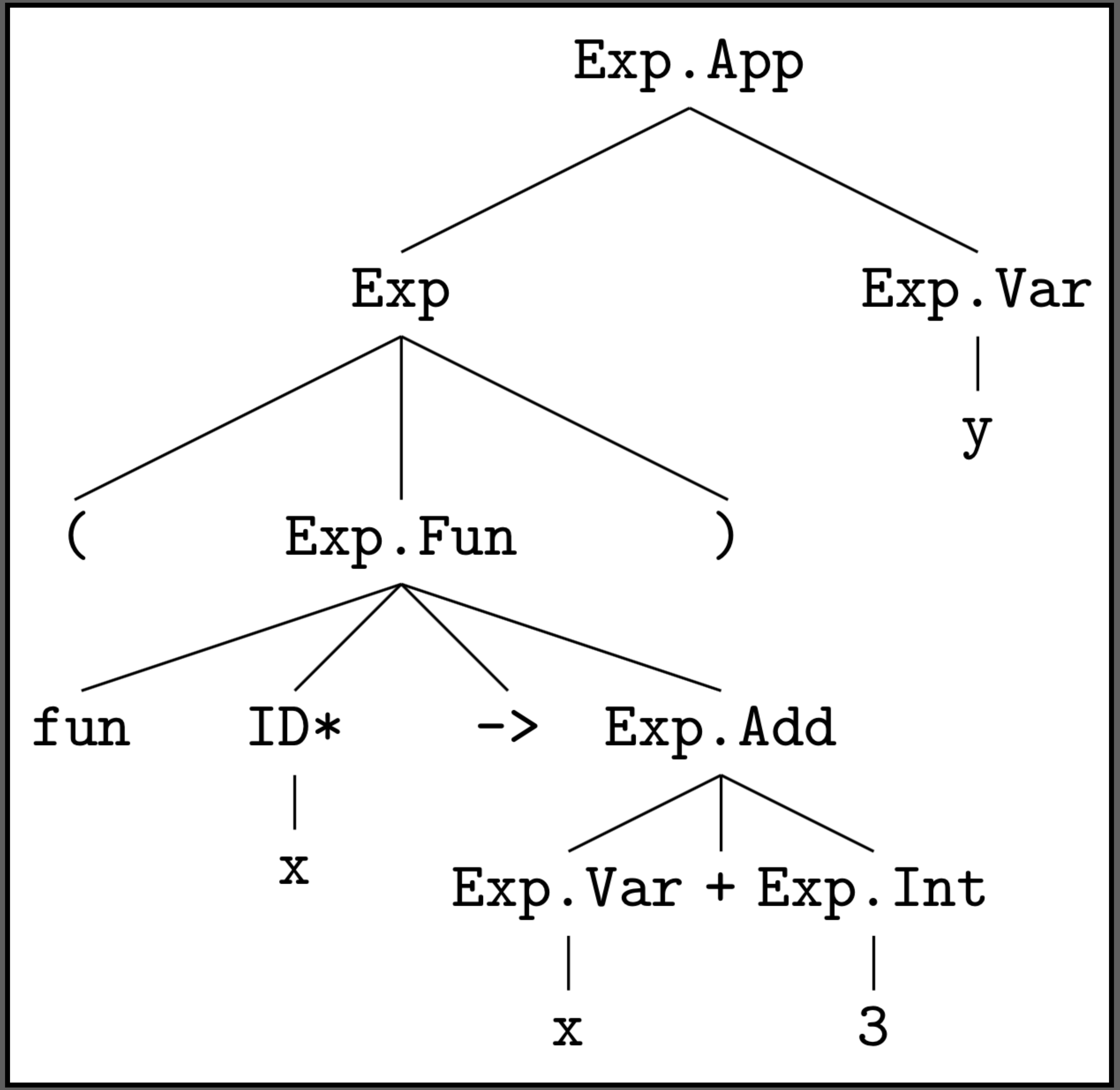
What is Syntax?

```
(fun x → x + 3) y
```

Syntax = Structure of Programs

```
context-free syntax
  Exp     = "(" Exp ")"

  Exp.Int = INT

  Exp.Var = ID

  Exp.Add = Exp "+" Exp

  Exp.Fun = "fun" ID* "→" Exp

  Exp.App = Exp Exp
```

```
(fun x → x + 3) y
```

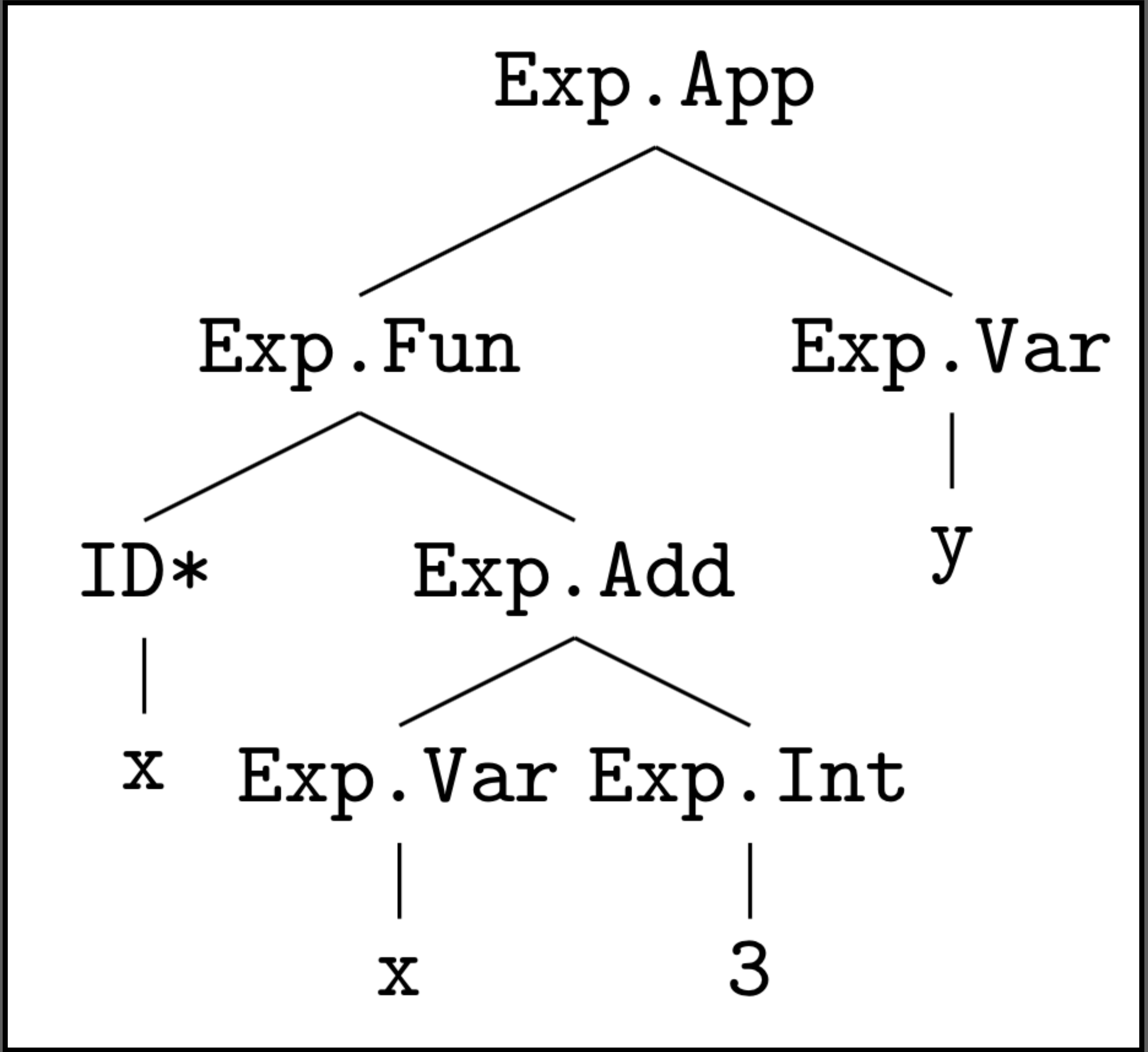


Kats, Visser, Wachsmuth: Pure and declarative syntax definition: paradise lost and regained. Onward 2010

Constructors ⟹ Abstract Syntax Tree

```
context-free syntax
  Exp     = "(" Exp ")" {bracket}

  Exp.Int = INT

  Exp.Var = ID

  Exp.Add = Exp "+" Exp

  Exp.Fun = "fun" ID* "→" Exp

  Exp.App = Exp Exp
```

```
(fun x → x + 3) y
```



=

```
App(
  Fun(
    ["x"]
  , Add(
      Var(
        "x"
      )
    , Int(
        "3"
      )
    )
  )
, Var(
    "y"
  )
)
```

Kats, Visser, Wachsmuth: Pure and declarative syntax definition: paradise lost and regained. Onward 2010

Abstract Syntax Terms

```
context-free syntax
  Exp     = "(" Exp ")" {bracket}

  Exp.Int = INT

  Exp.Var = ID

  Exp.Add = Exp "+" Exp

  Exp.Fun = "fun" ID* "->" Exp

  Exp.App = Exp Exp
```

```
(fun x -> x + 3) y
```

↓

```
App(
  Fun(["x"], Add(Var("x"), Int("3")))
, Var("y")
)
```

Kats, Visser, Wachsmuth: Pure and declarative syntax definition: paradise lost and regained. Onward 2010

Syntax Definition ≃ Algebraic Signature

```
context-free syntax
  Exp     = "(" Exp ")" {bracket}

  Exp.Int = INT

  Exp.Var = ID

  Exp.Add = Exp "+" Exp

  Exp.Fun = "fun" ID* "→" Exp

  Exp.App = Exp Exp
```

```
signature
  sorts INT ID Exp
  constructors
    Int : INT → Exp
    Var : ID → Exp
    Add : Exp * Exp → Exp
    Fun : List(ID) * Exp → Exp
    App : Exp * Exp → Exp
```

```
(fun x → x + 3) y
```

```
App(
  Fun(["x"], Add(Var("x"), Int("3")))
, Var("y")
)
```

Kats, Visser, Wachsmuth: Pure and declarative syntax definition: paradise lost and regained. Onward 2010

Parsing Declaratively

```
parse(yield(t)) = t
```

```
yield : ParseTree → String
parse : String → ParseTree
```

Language Designers focus on Structure of Programs

Formatting Templates

parse = (implode ; format)$^{-1}$

```
context-free syntax
  Exp     = <(<Exp>)> {bracket}

  Exp.Int = INT

  Exp.Var = ID

  Exp.Add = <<Exp> + <Exp>>

  Exp.Fun = [fun [ID*] → [Exp]]

  Exp.App = <<Exp> <Exp>>

  Exp.Let = <
    let <{Bnd "\n\n"}*>
     in <Exp>
  >

  Bnd.Bnd = <<ID> = <Exp>>
```

```
let
  inc = fun x → x + 1
 in
  inc 3
```

parse; implode

```
Let(
  [ Bnd(
      "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
    )
  ]
, App(Var("inc"), Int("3"))
)
```

format

```
let inc = fun x → x + 1
in inc 3
```

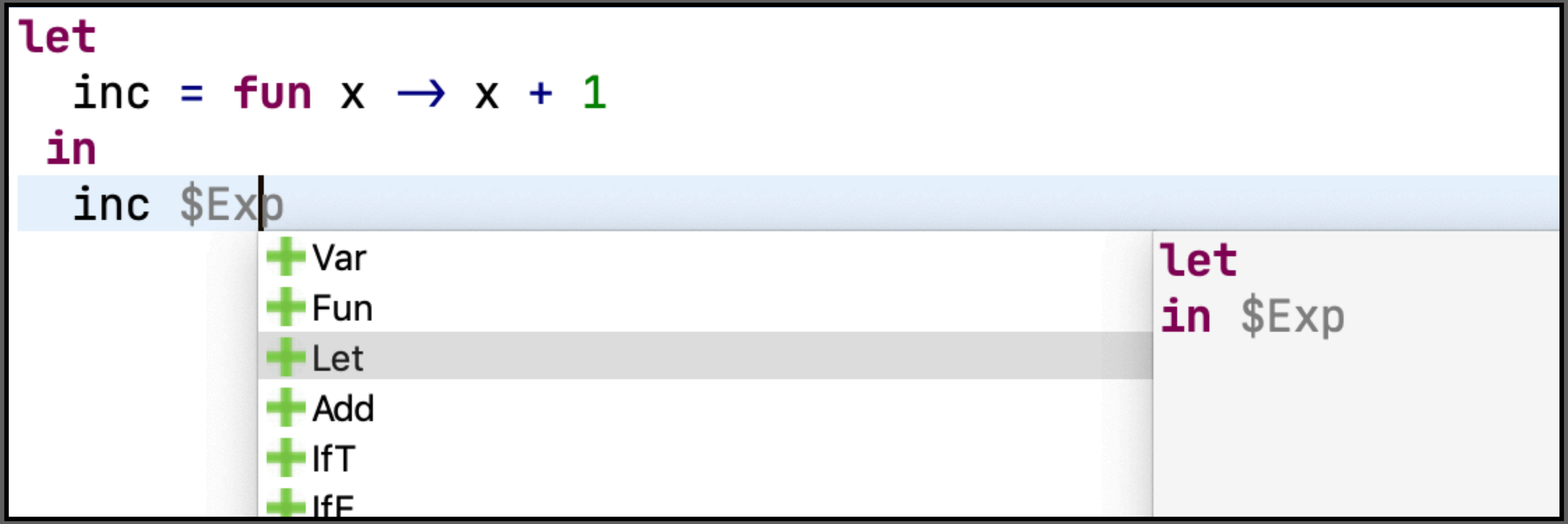
Vollebregt, Kats, Visser: Declarative specification of template-based textual editors. LDTA 2012

Parsing + Formatting Declaratively

```
implode(parse(format(t))) = t
```

```
format  : AST → String
implode : ParseTree → AST
parse   : String → ParseTree
```

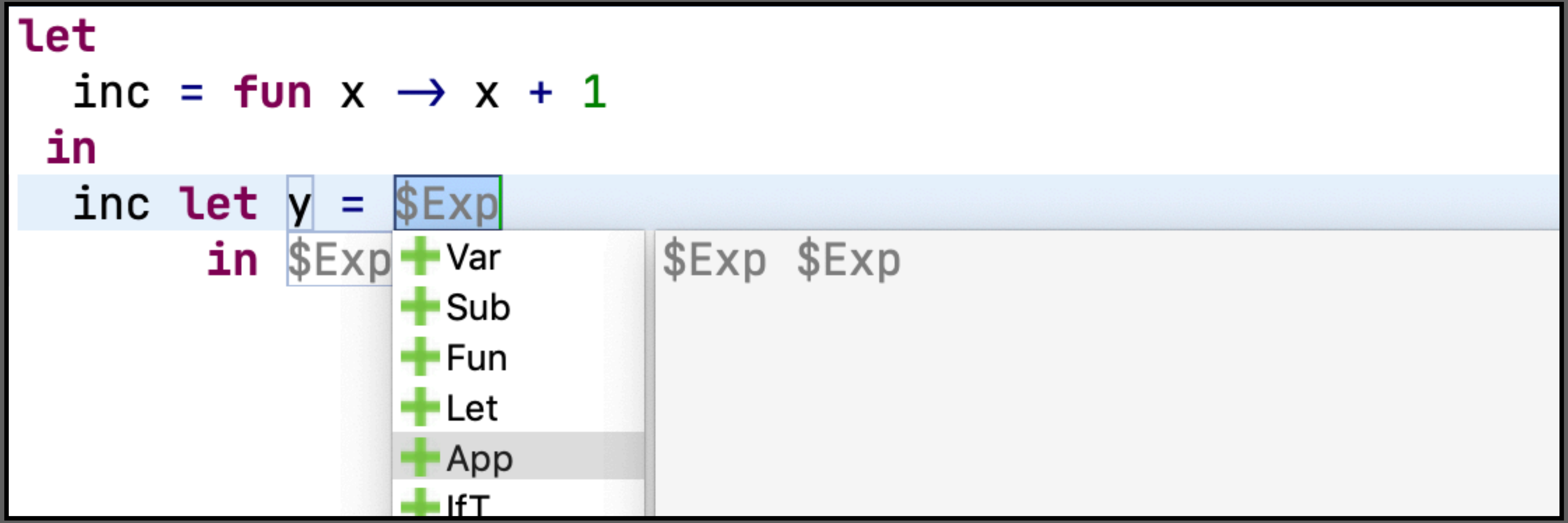
Syntactic Completion = Rewriting Incomplete Programs



Explicit incompleteness: extend language with placeholders

Completion: rewrite placeholders

Templates:Formatting proposals



Soundness:
Only syntactically correct proposals

Completeness:
Reach all programs

De Souza Amorim, Erdweg, Wachsmuth Visser. Principled syntactic code completion using placeholders. SLE 2016

Notation = Formatting

How does structure map to text?

Declarative Disambiguation

Ambiguous Grammar

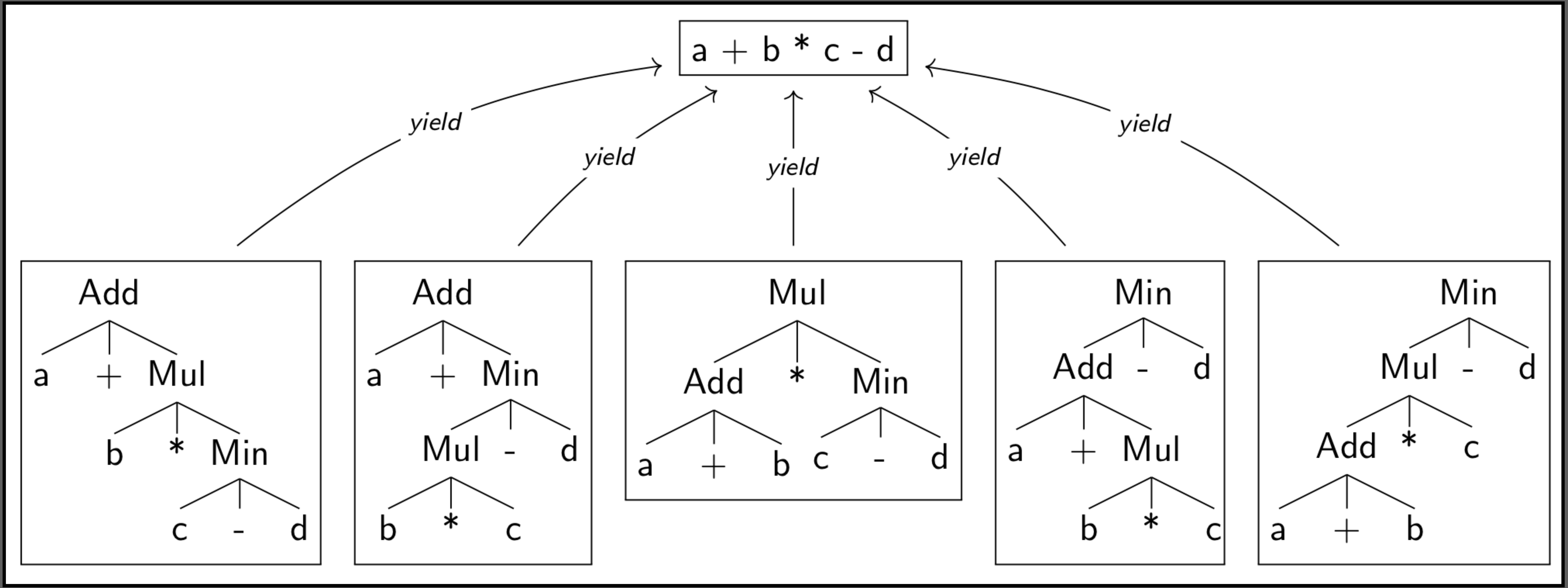
```
context-free syntax
  Exp.Int = INT
  Exp.Var = ID
  Exp.Min = <<Exp> - <Exp>>
  Exp.Add = <<Exp> + <Exp>>
  Exp.Mul = <<Exp> * <Exp>>
```

Ambiguous Grammar

```
context-free syntax
  Exp.Int = INT
  Exp.Var = ID
  Exp.Min = <<Exp> - <Exp>>
  Exp.Add = <<Exp> + <Exp>>
  Exp.Mul = <<Exp> * <Exp>>
```

Ambiguous Sentence has Multiple Parse Trees

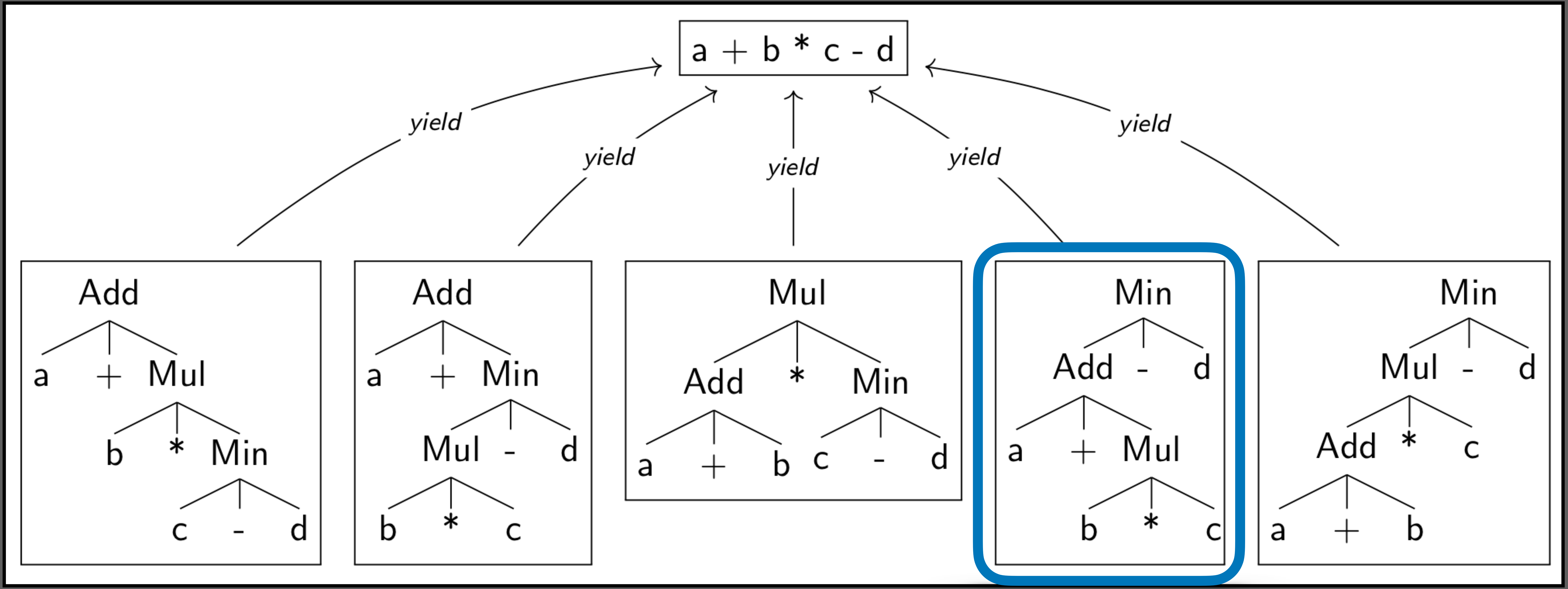
```
context-free syntax
  Exp.Int = INT
  Exp.Var = ID
  Exp.Min = <<Exp> - <Exp>>
  Exp.Add = <<Exp> + <Exp>>
  Exp.Mul = <<Exp> * <Exp>>
```



Disambiguation with Associativity and Priority Rules

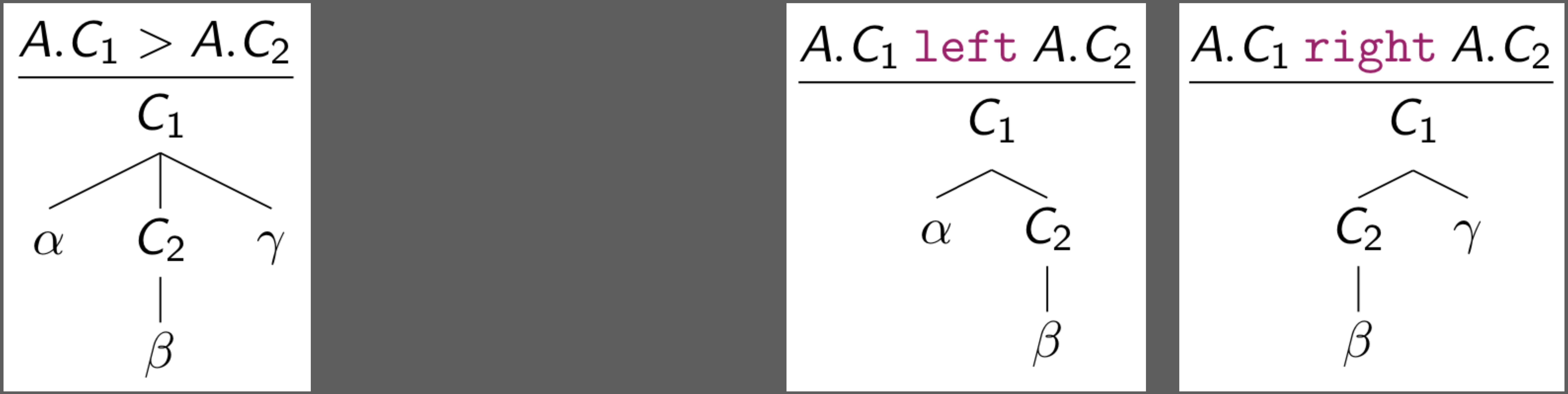
```
context-free syntax
  Exp.Int = INT
  Exp.Var = ID
  Exp.Min = <<Exp> - <Exp>>
  Exp.Add = <<Exp> + <Exp>>
  Exp.Mul = <<Exp> * <Exp>>
```

```
context-free syntax
  Exp.Min = <<Exp> - <Exp>> {left}
  Exp.Add = <<Exp> + <Exp>> {left}
  Exp.Mul = <<Exp> * <Exp>> {left}
context-free priorities
  Exp.Mul > {left: Exp.Min Exp.Add}
```



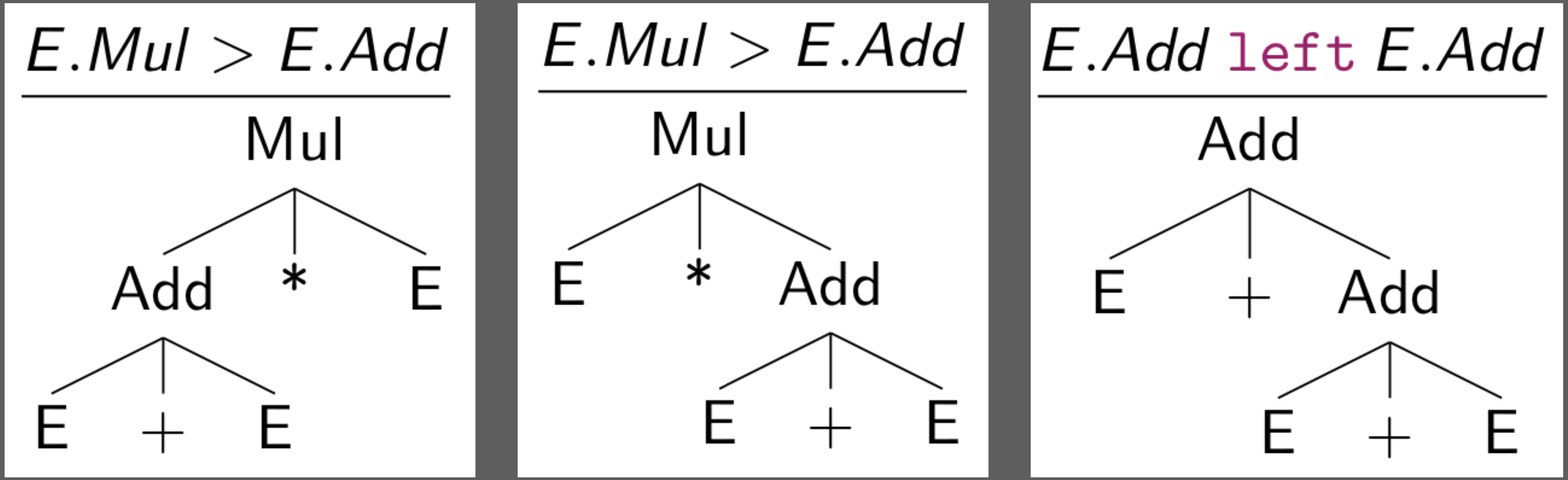
Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules



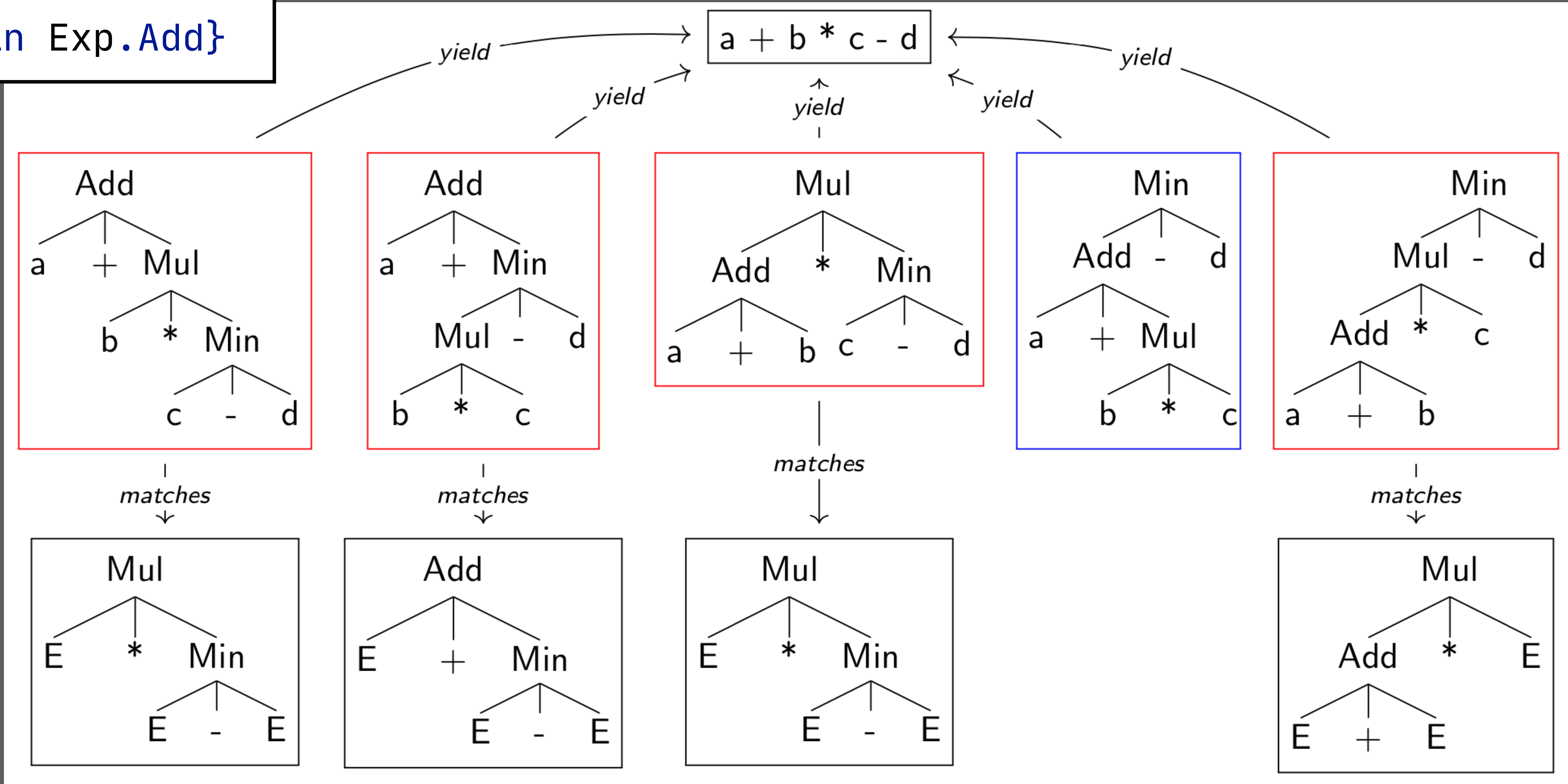
Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances



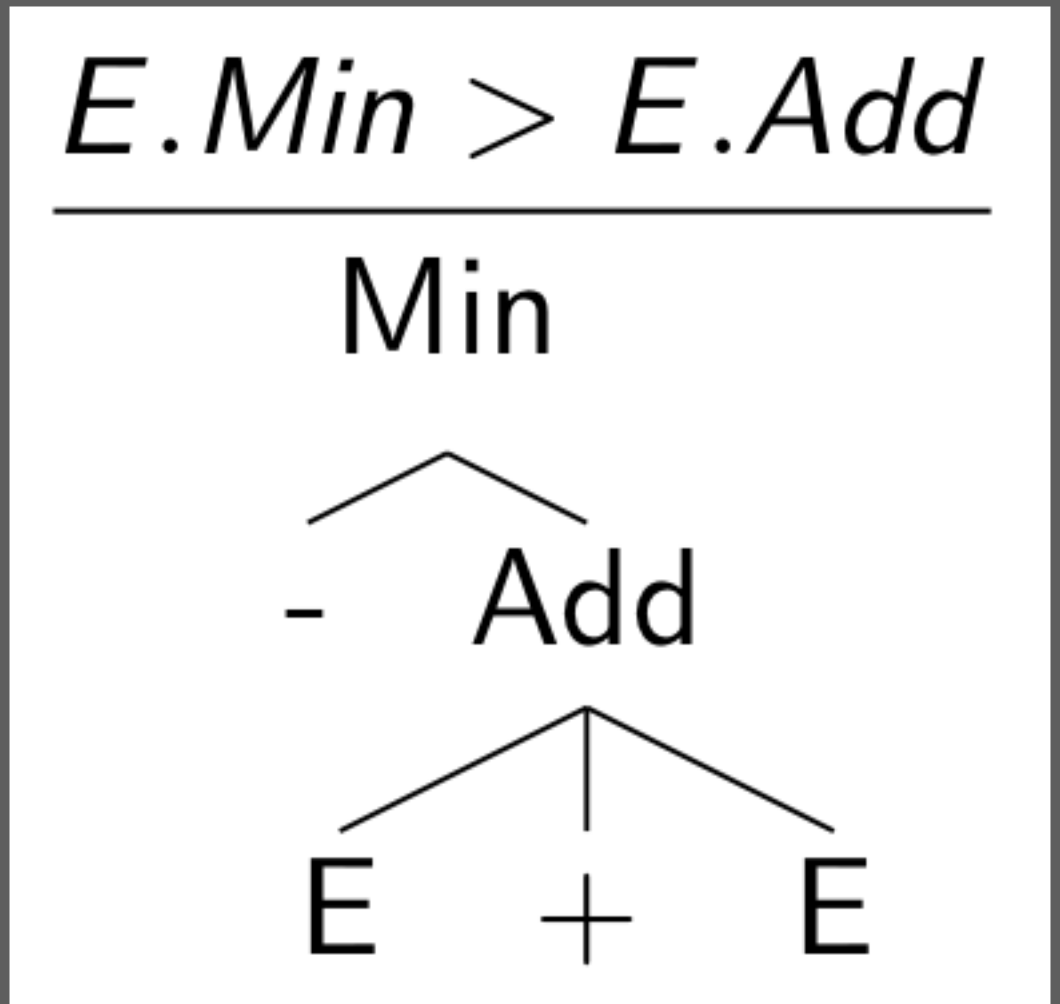
Disambiguation by Subtree Exclusion

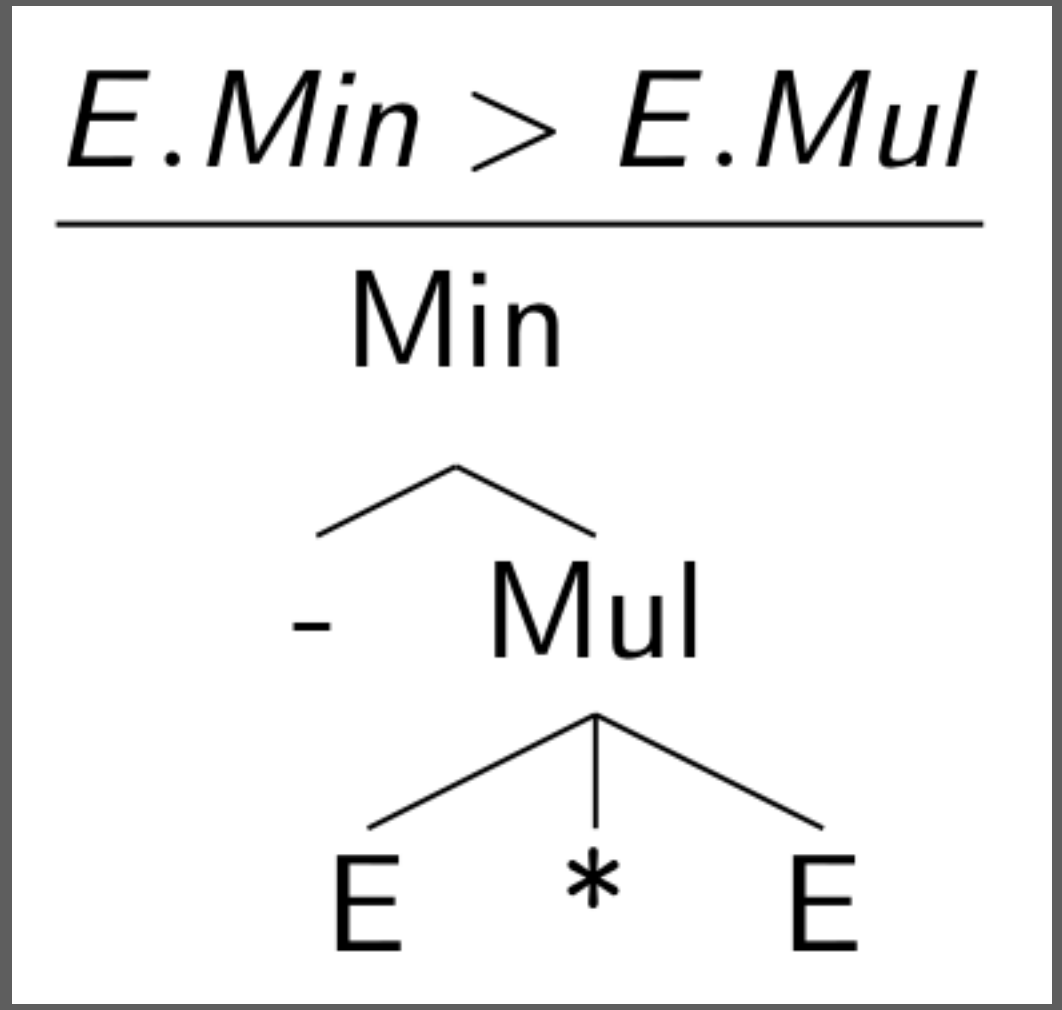
```
context-free syntax
  Exp.Min = <<Exp> - <Exp>> {left}
  Exp.Add = <<Exp> + <Exp>> {left}
  Exp.Mul = <<Exp> * <Exp>> {left}
context-free priorities
  Exp.Mul > {left: Exp.Min Exp.Add}
```



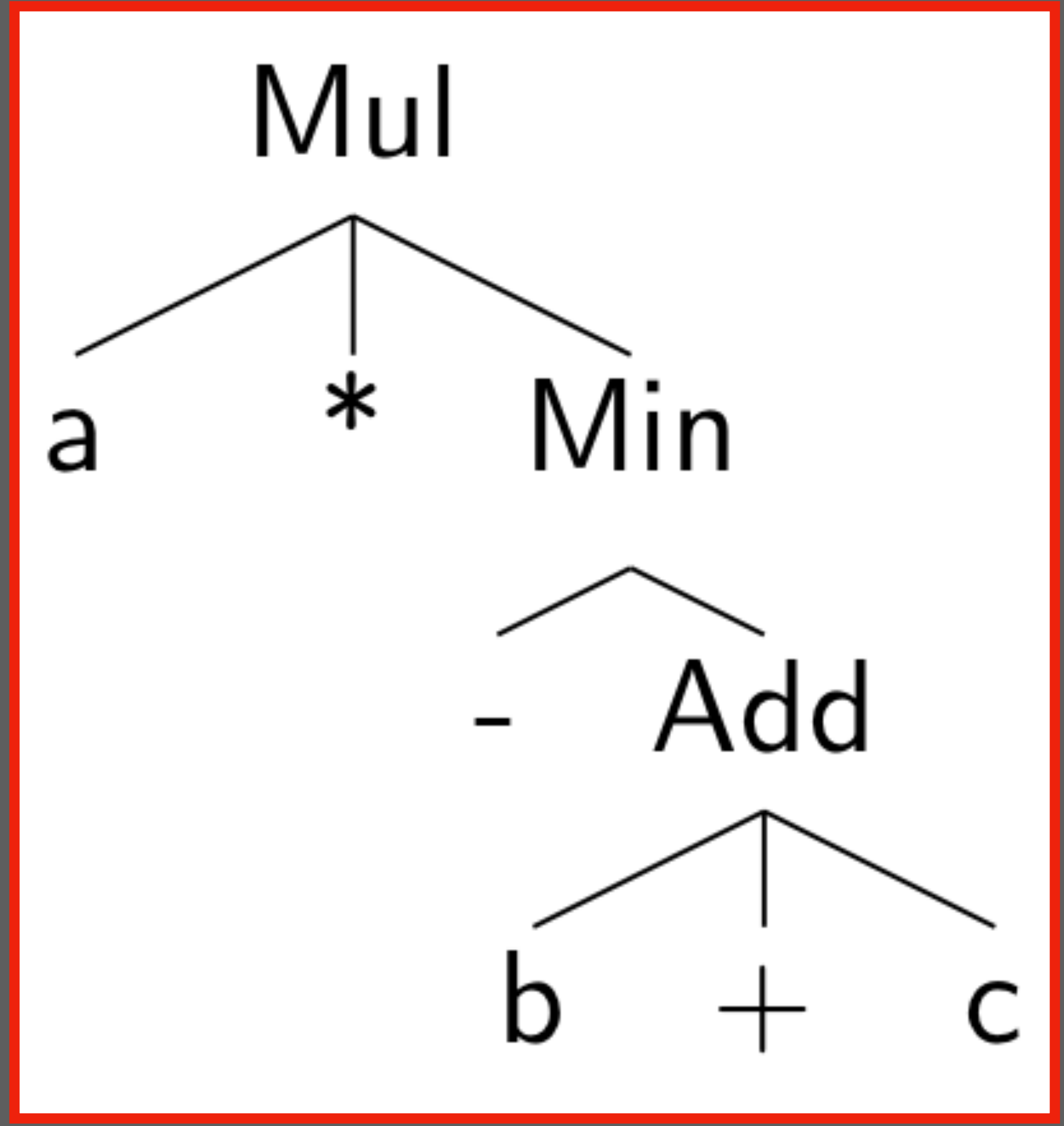
Safe for High Priority Prefix Operators

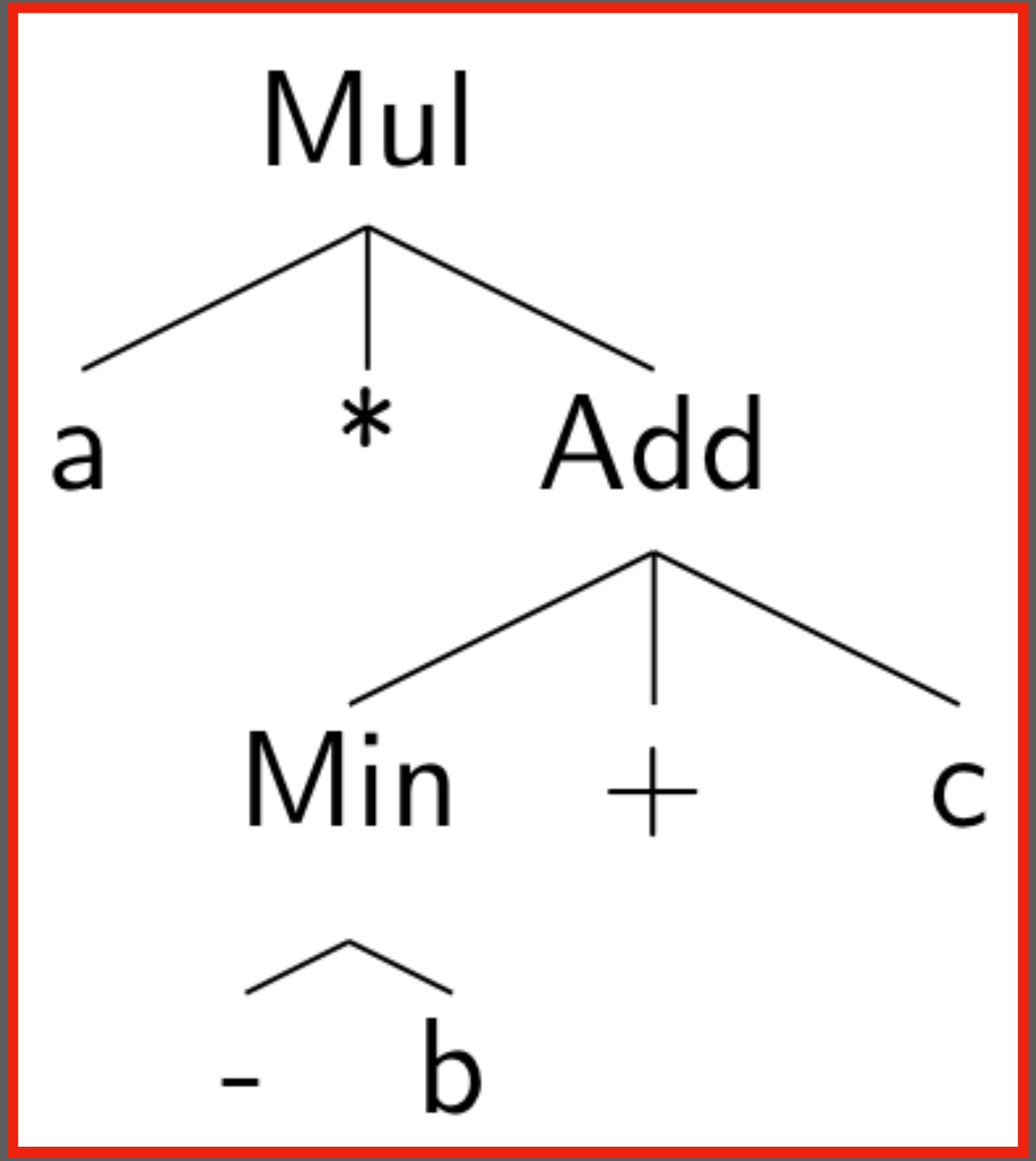
Conflict Patterns

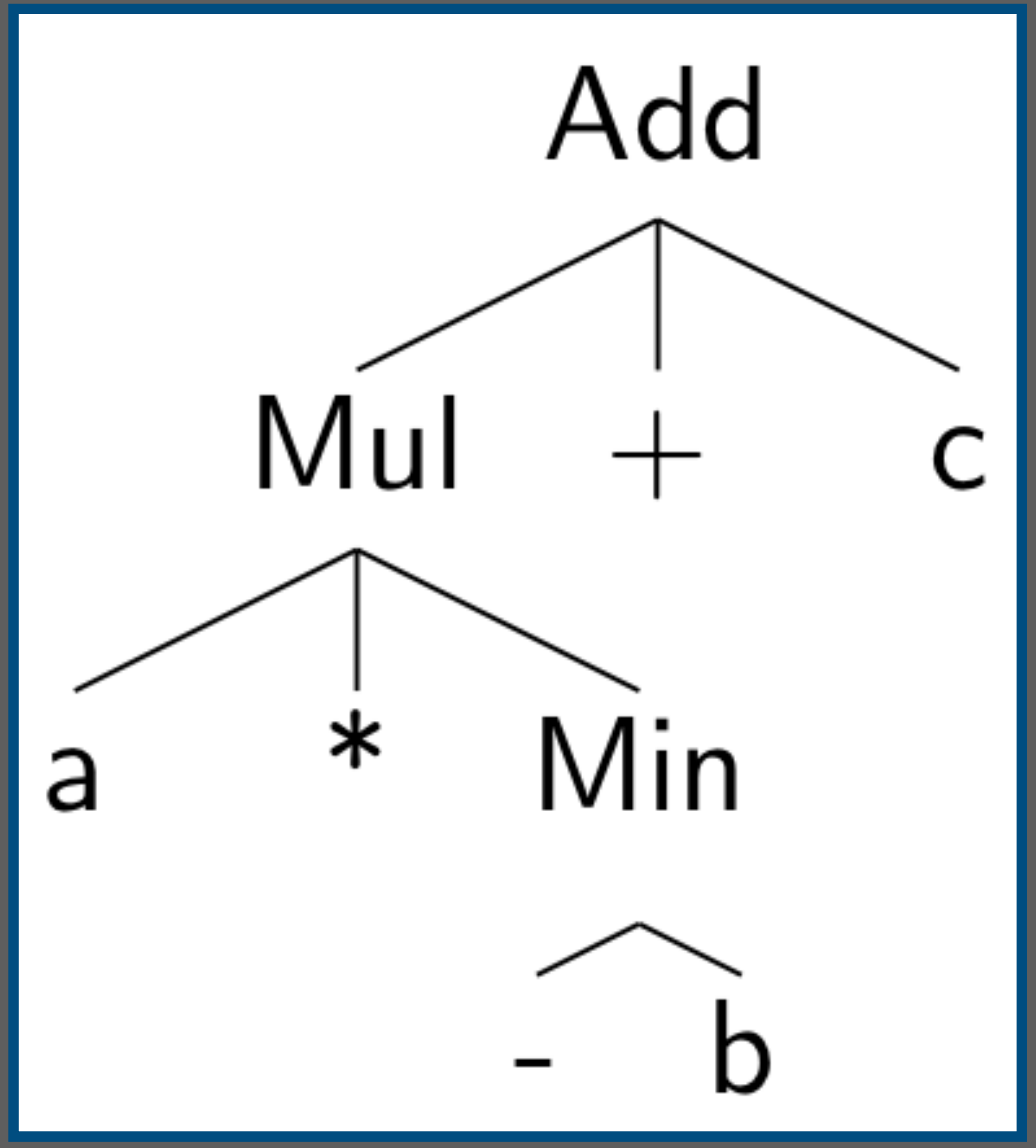




Trees



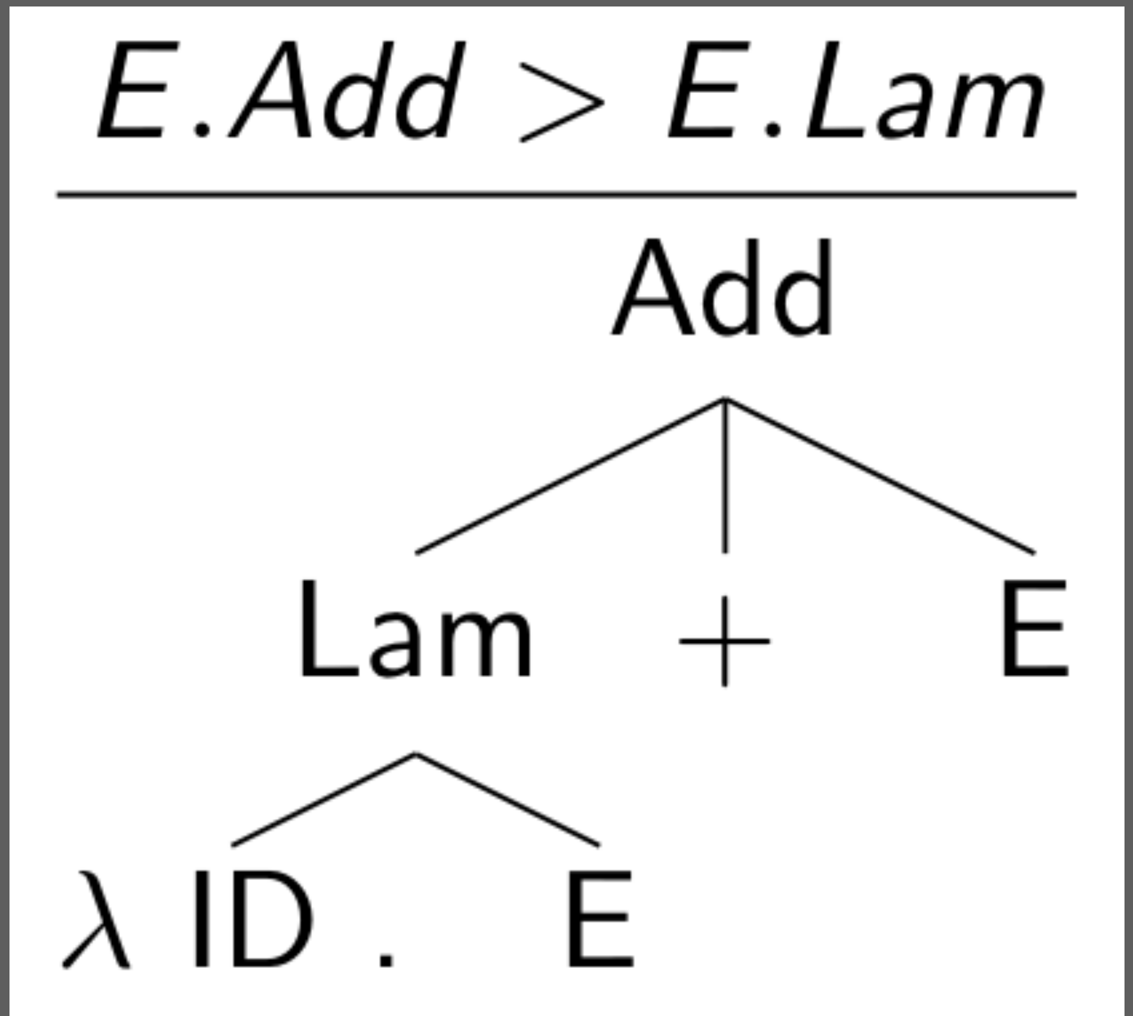


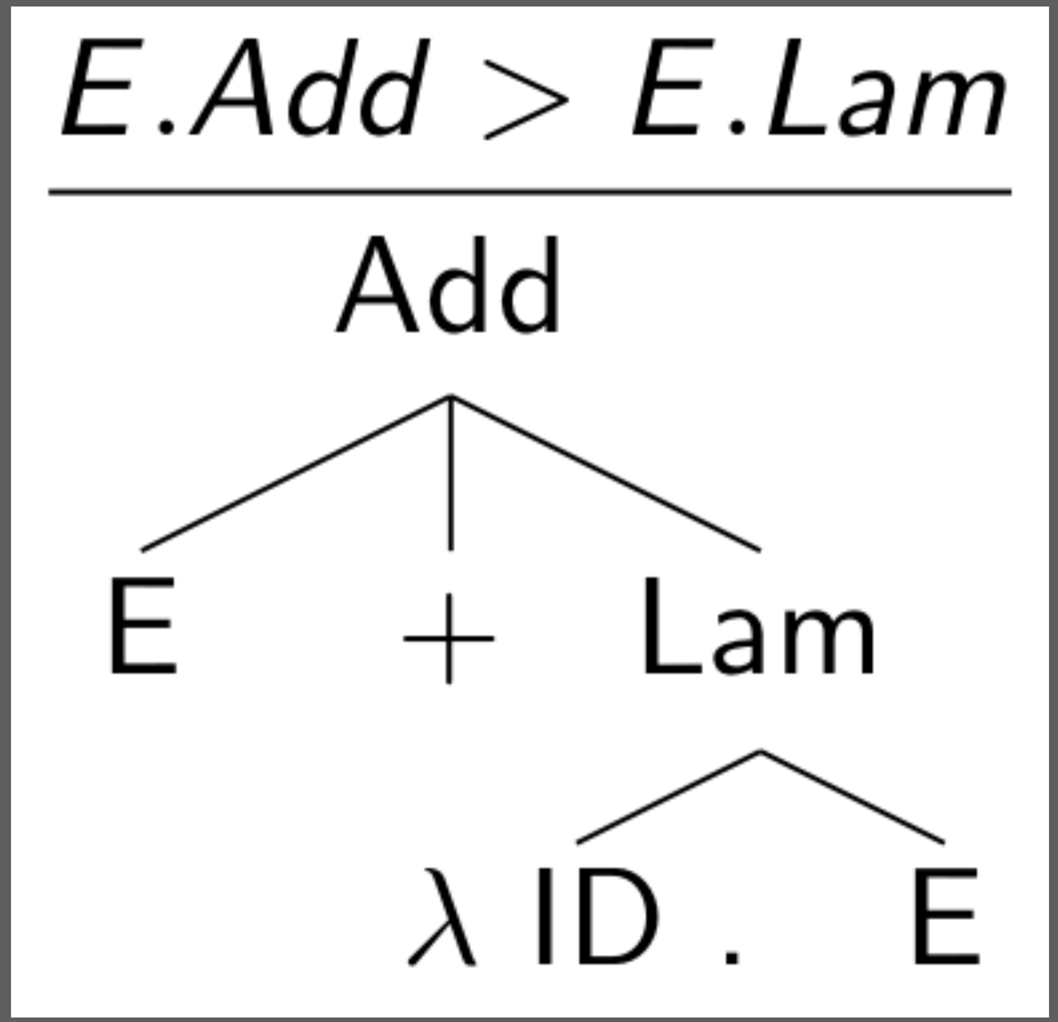


Unsafe for Low Priority Prefix Operators [SDF2]

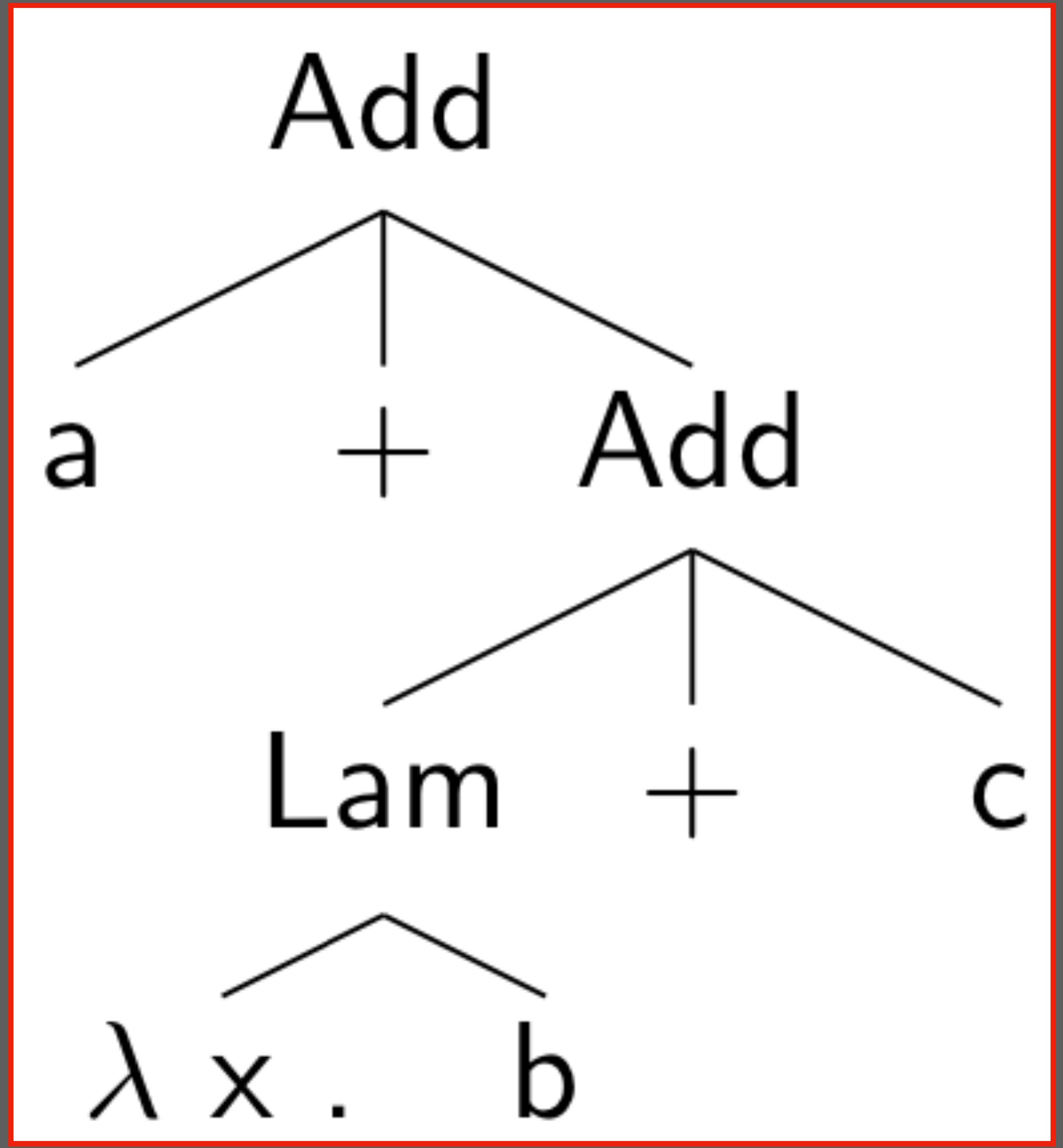
Afroozeh, van den Brand, Johnstone, Scott, Vinju: Safe specification of operator precedence rules. SLE 2013

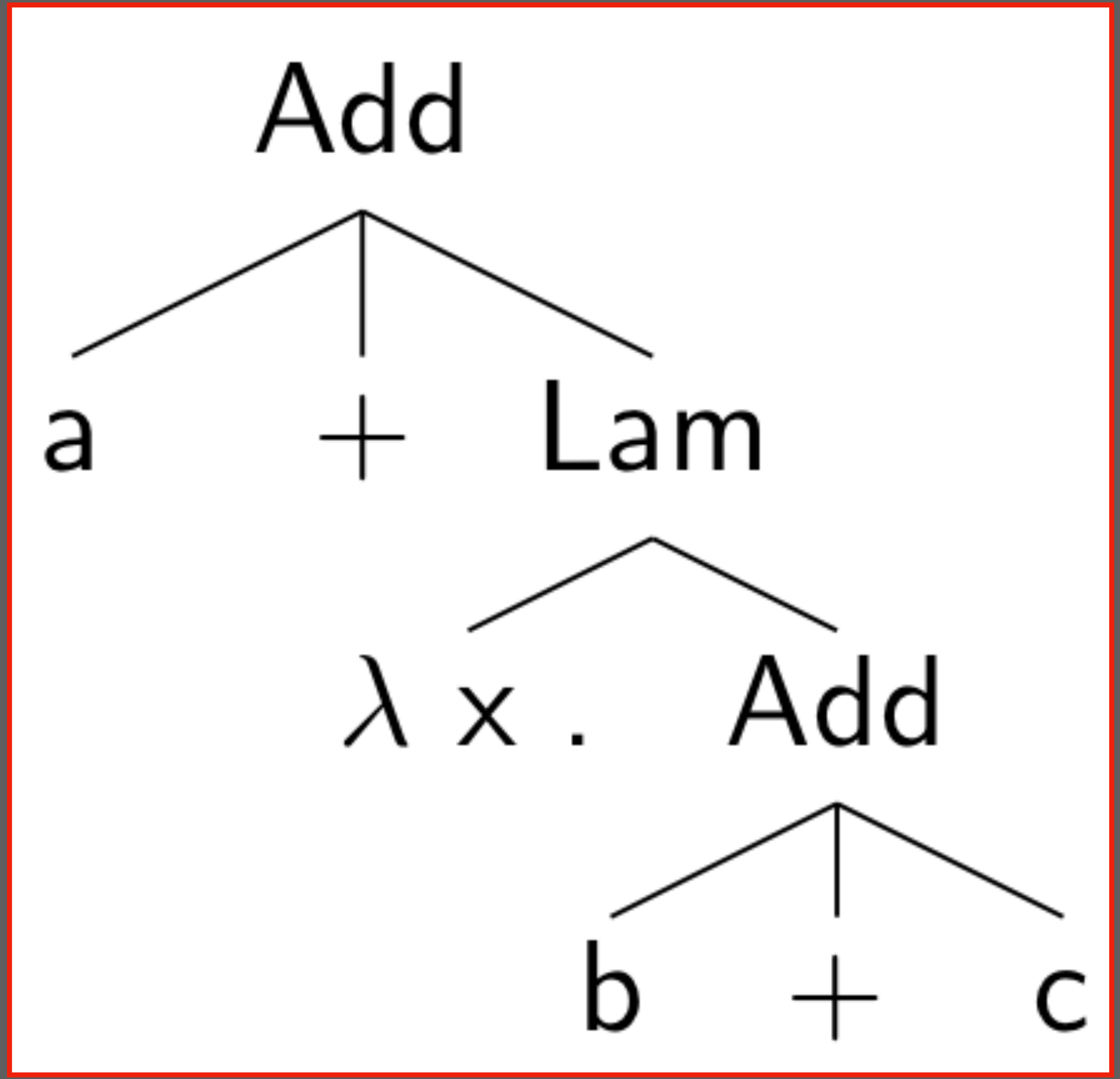
Conflict Patterns

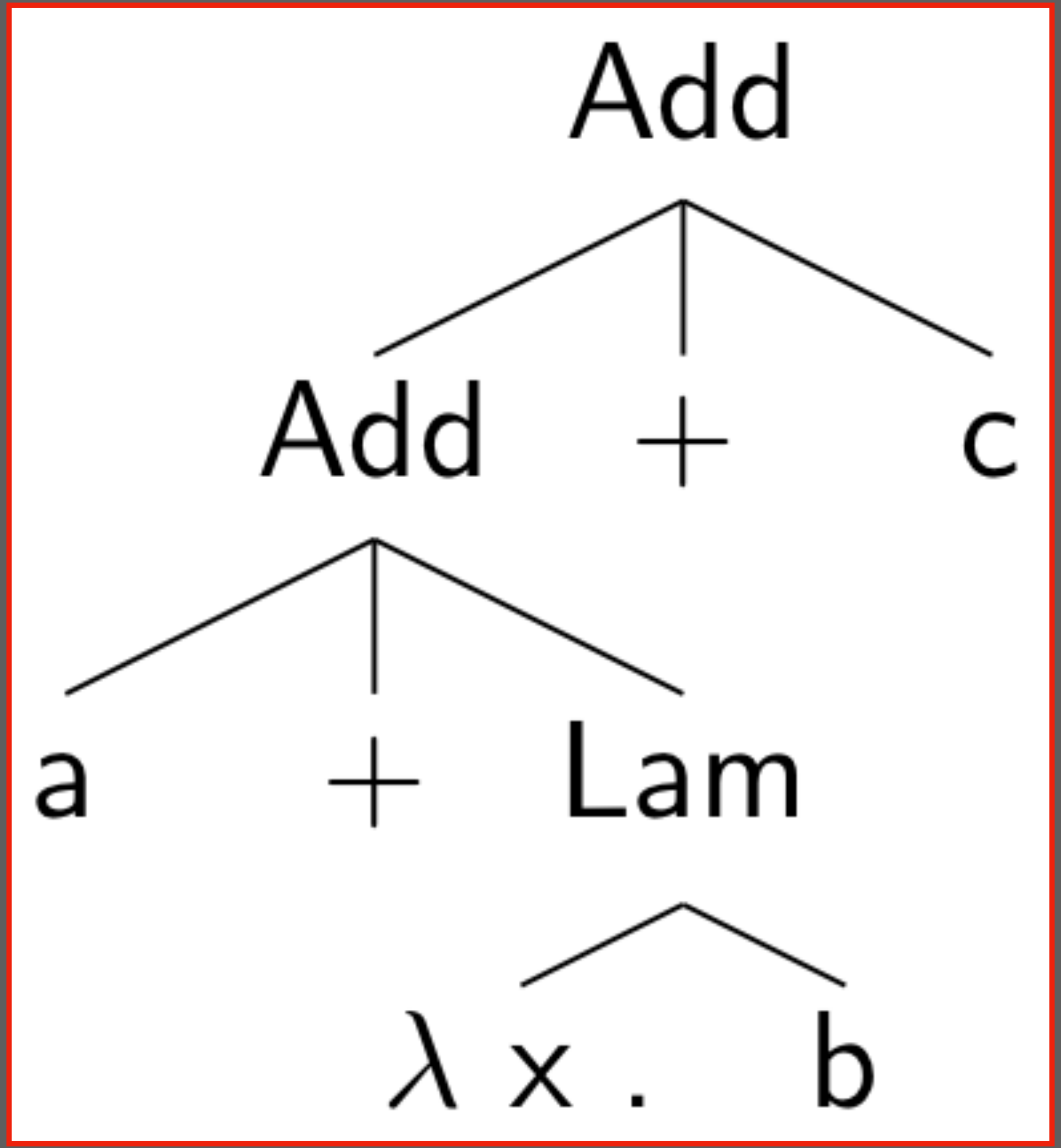




Trees

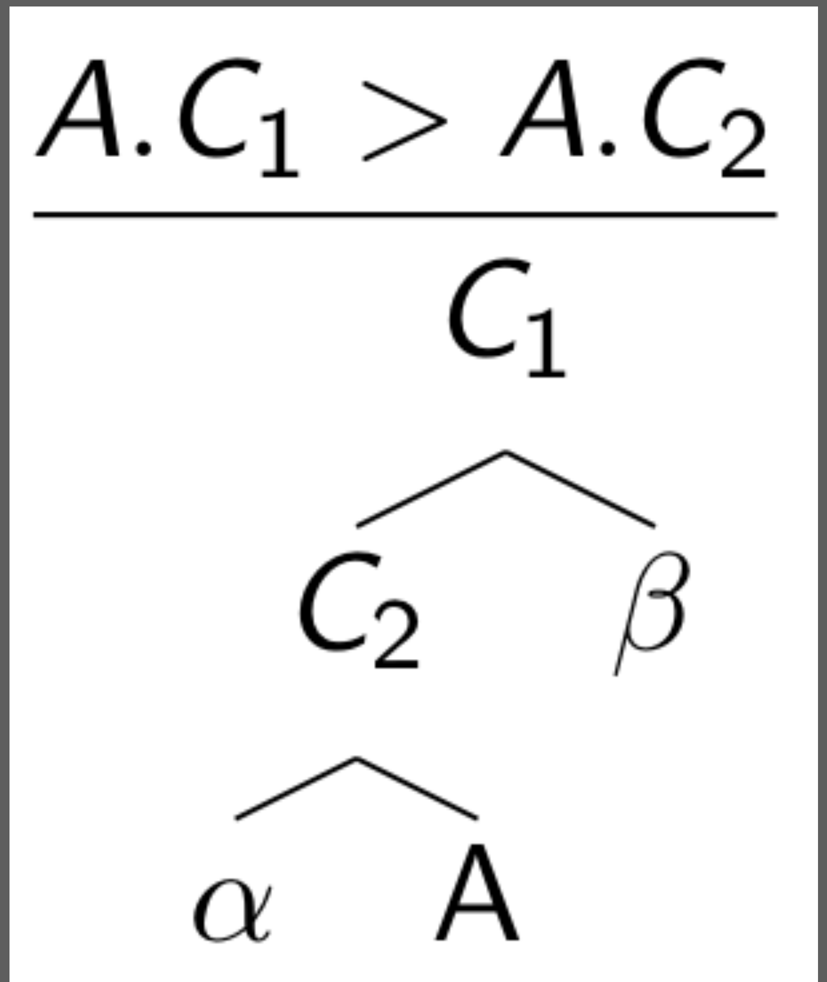




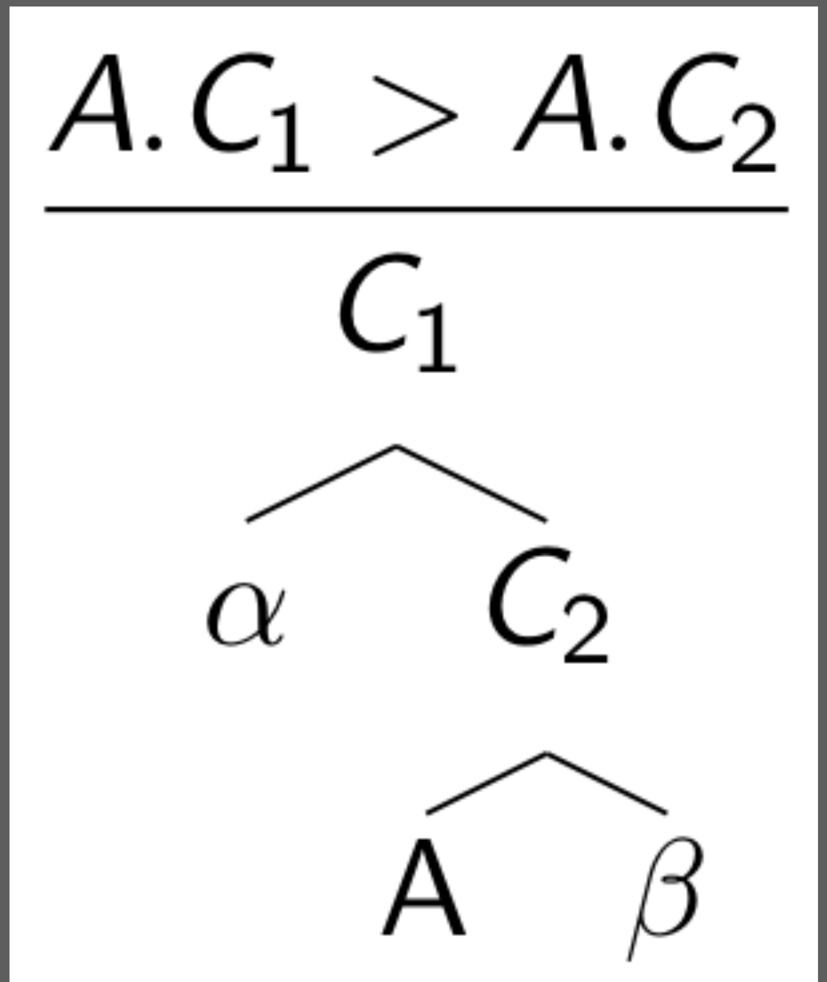


Safe Subtree Exclusion Rules [SDF3 (2019)]

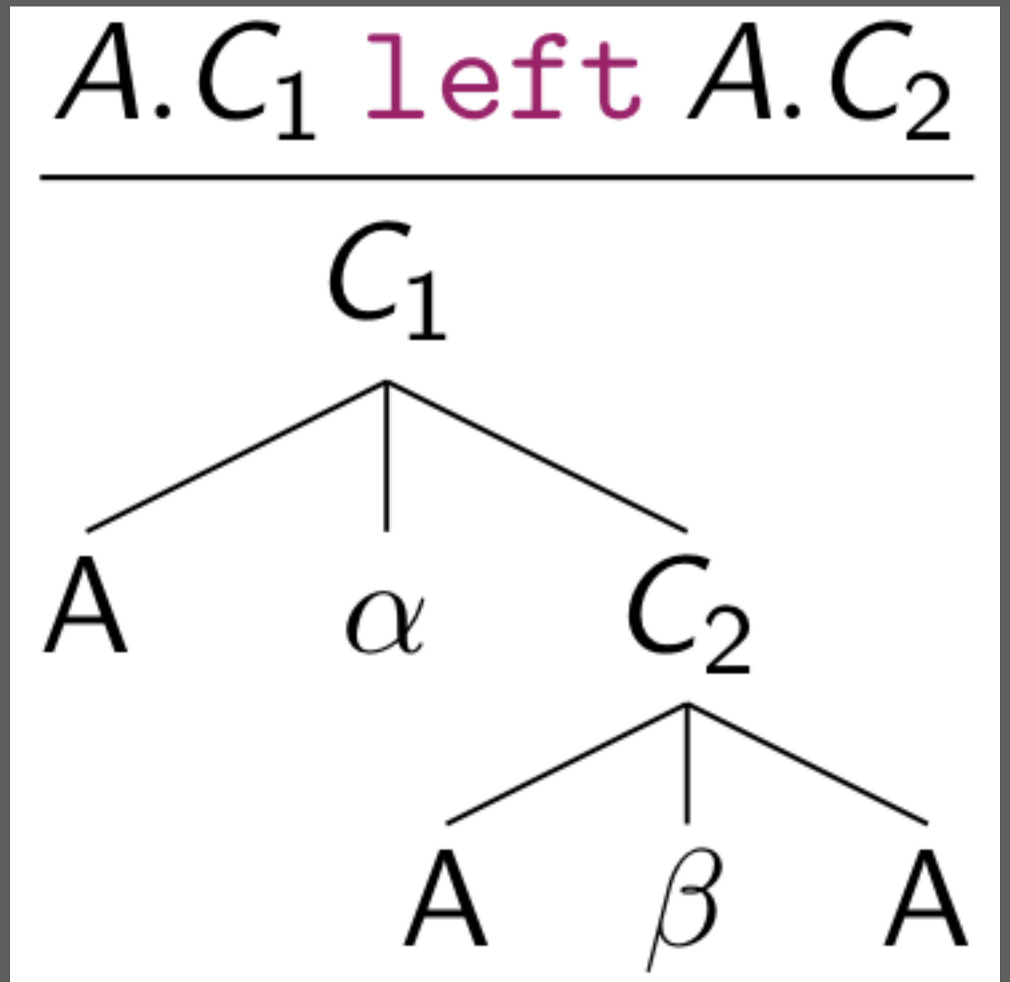
Rules

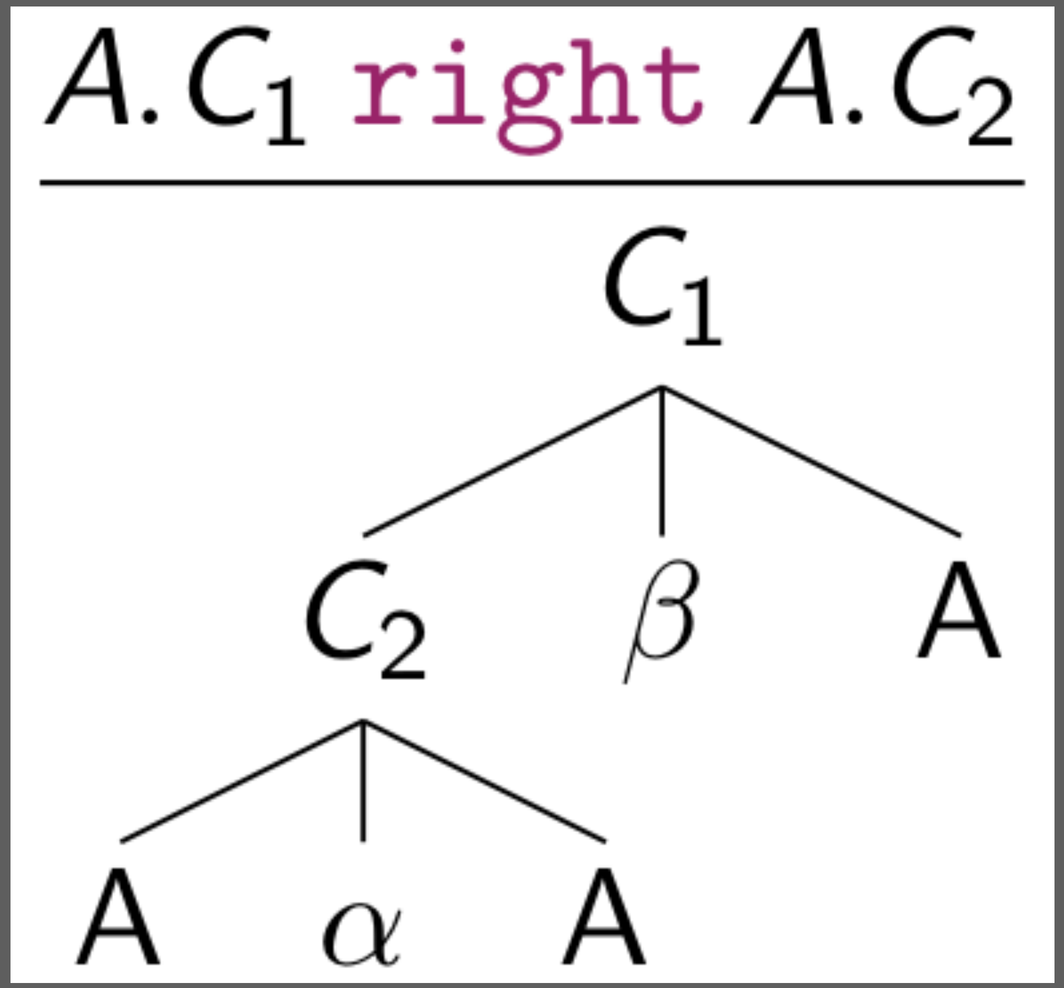


Right Recursive in
Left Recursive Position



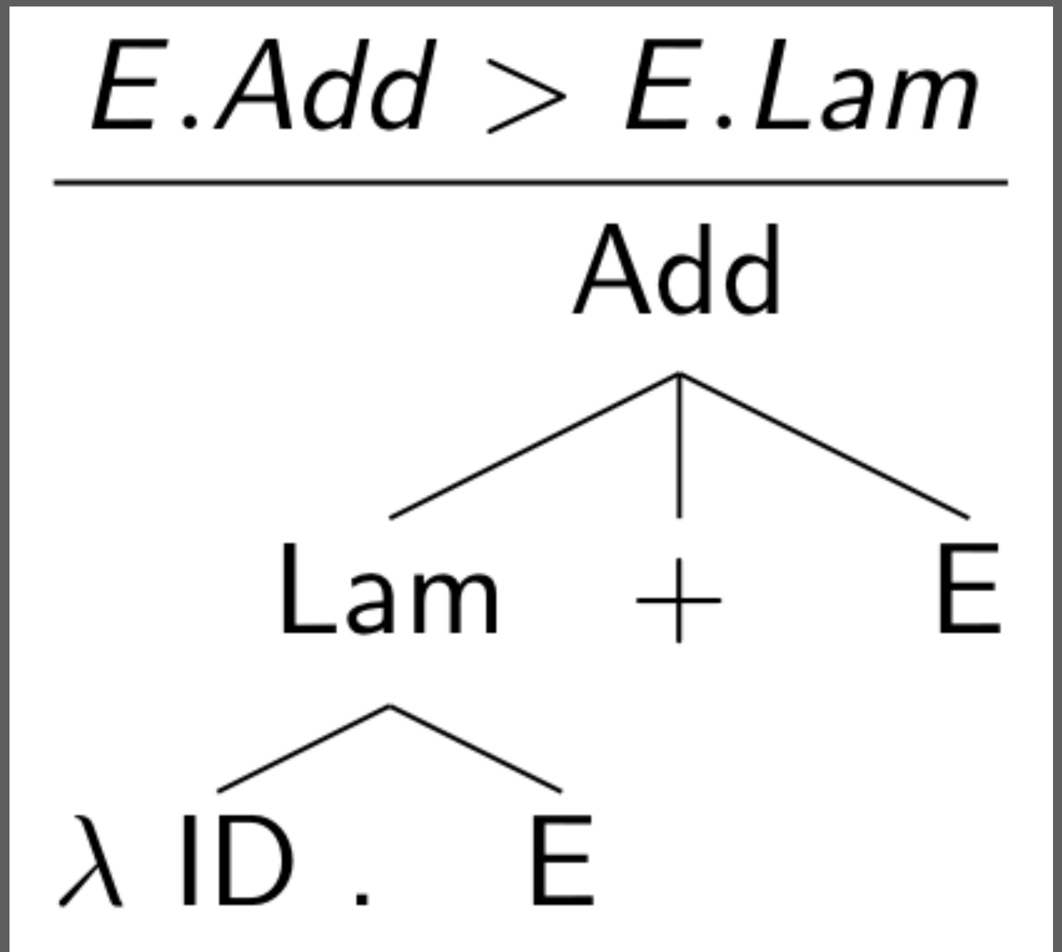
Left Recursive in
Right Recursive Position



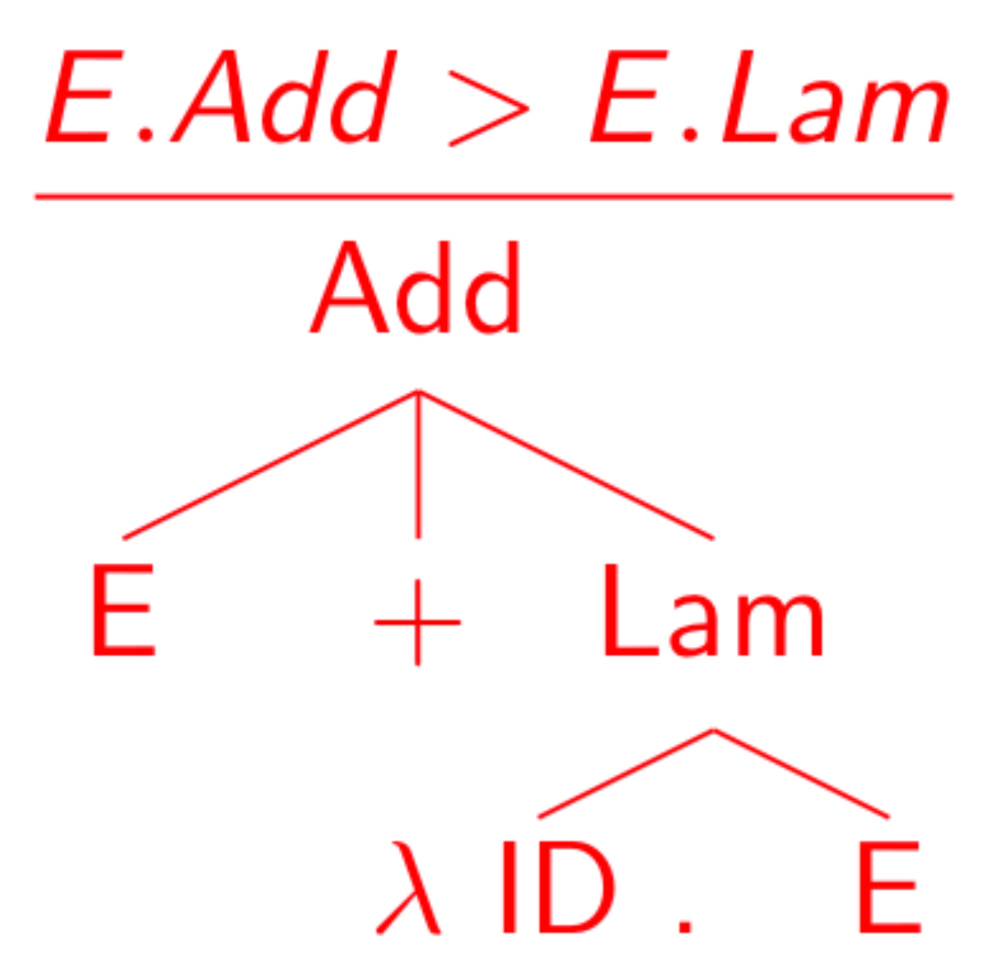


Associativity

Conflict Patterns



conflict pattern:
\ right recursive

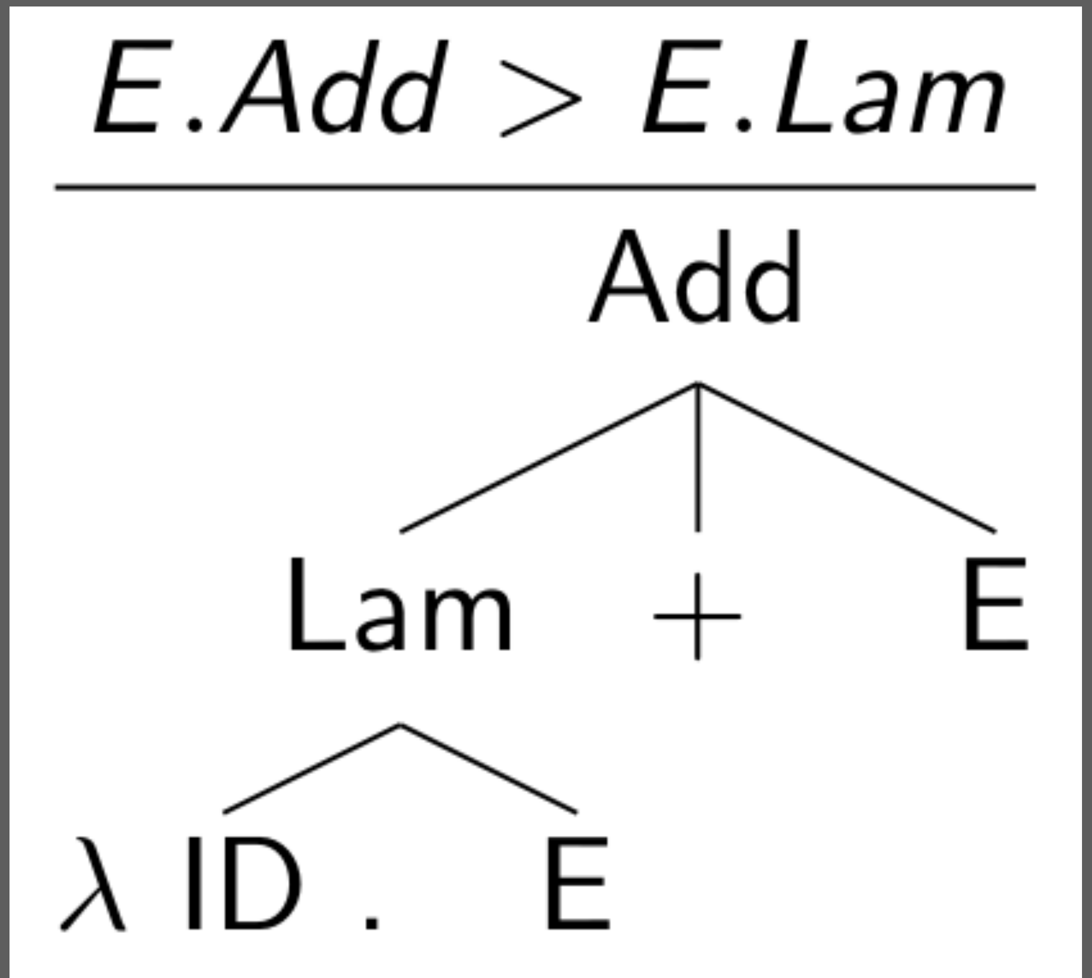


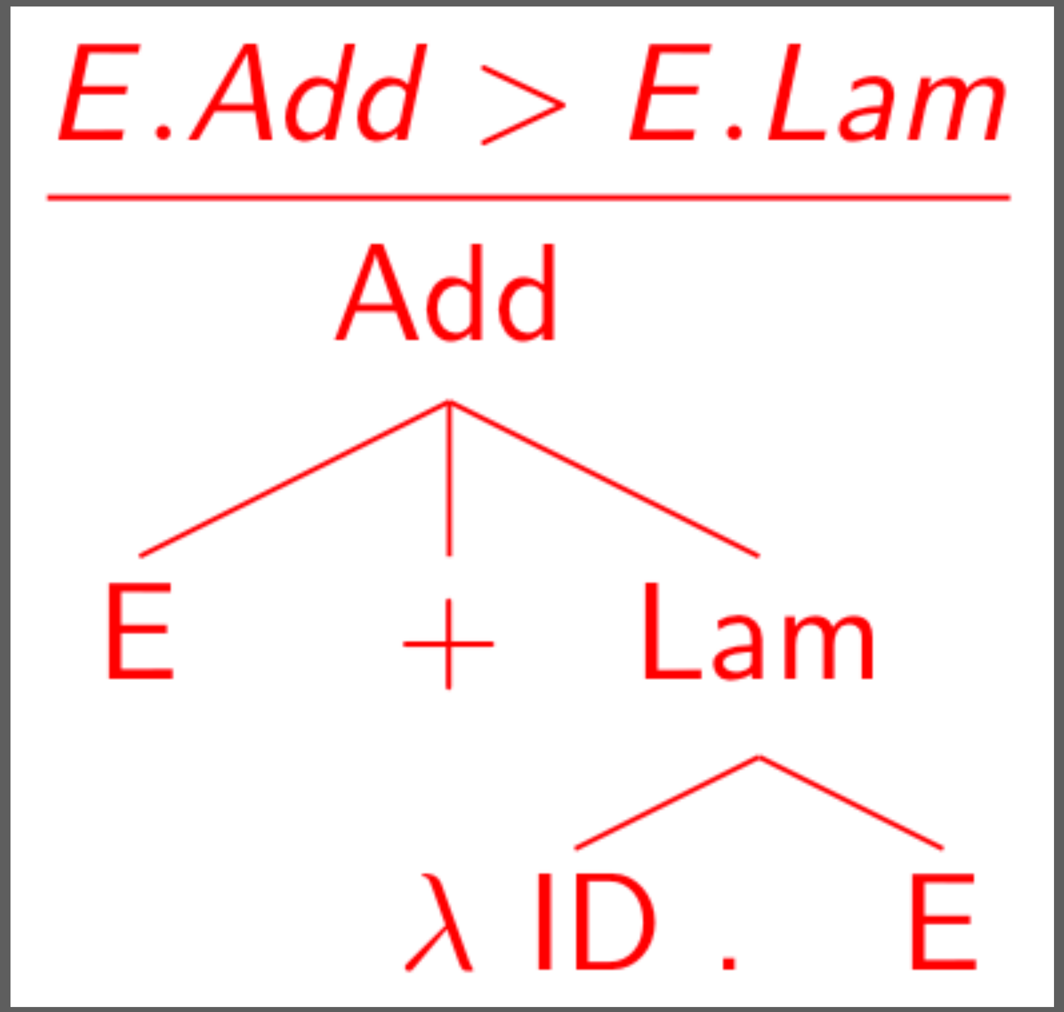
Amorim, Visser: A direct semantics of declarative disambiguation rules. (Under revision)

not a conflict pattern:
\ not left recursive

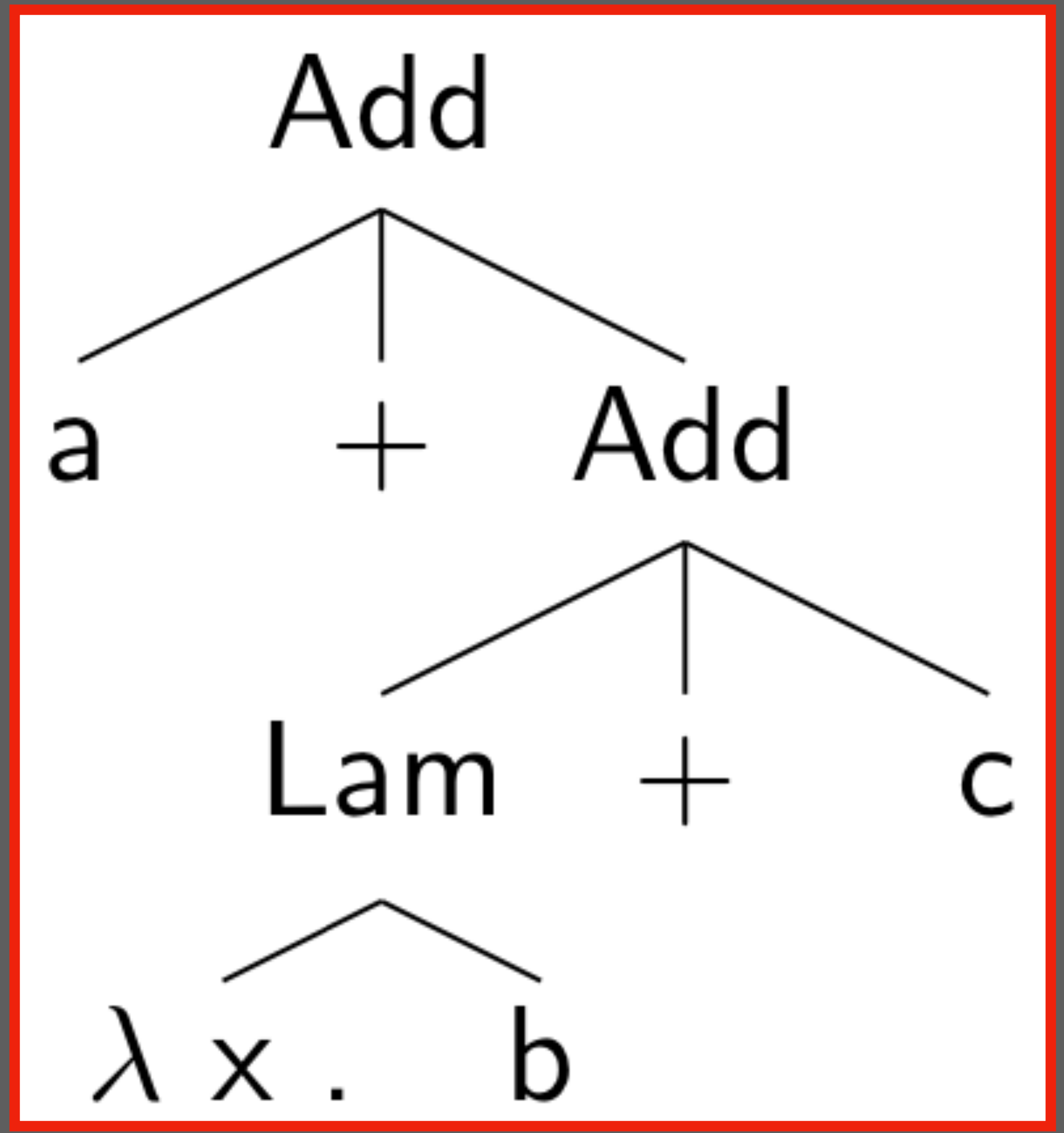
Shallow Interpretation: Safe for Low Priority Prefix Operators

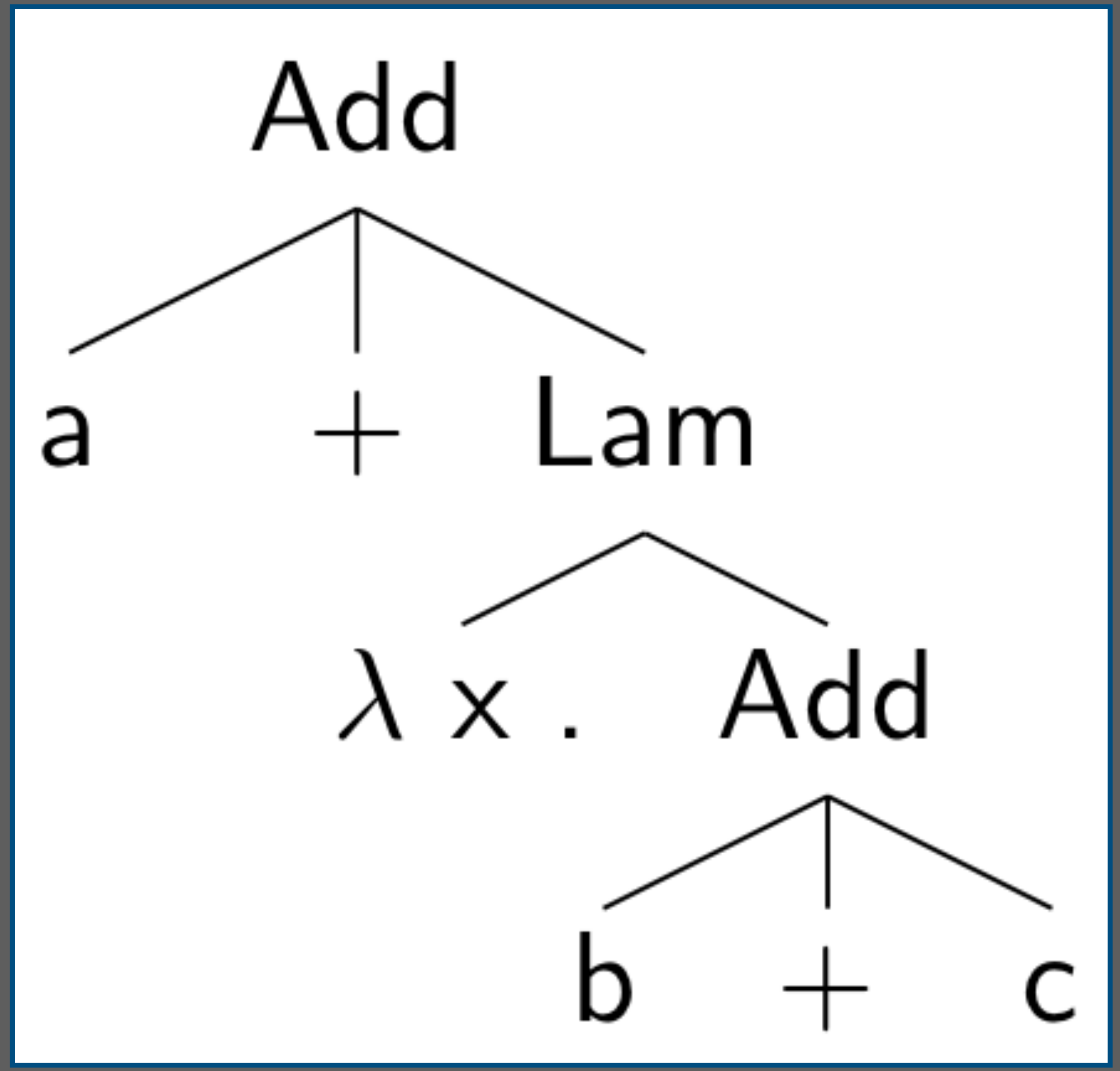
Conflict Patterns

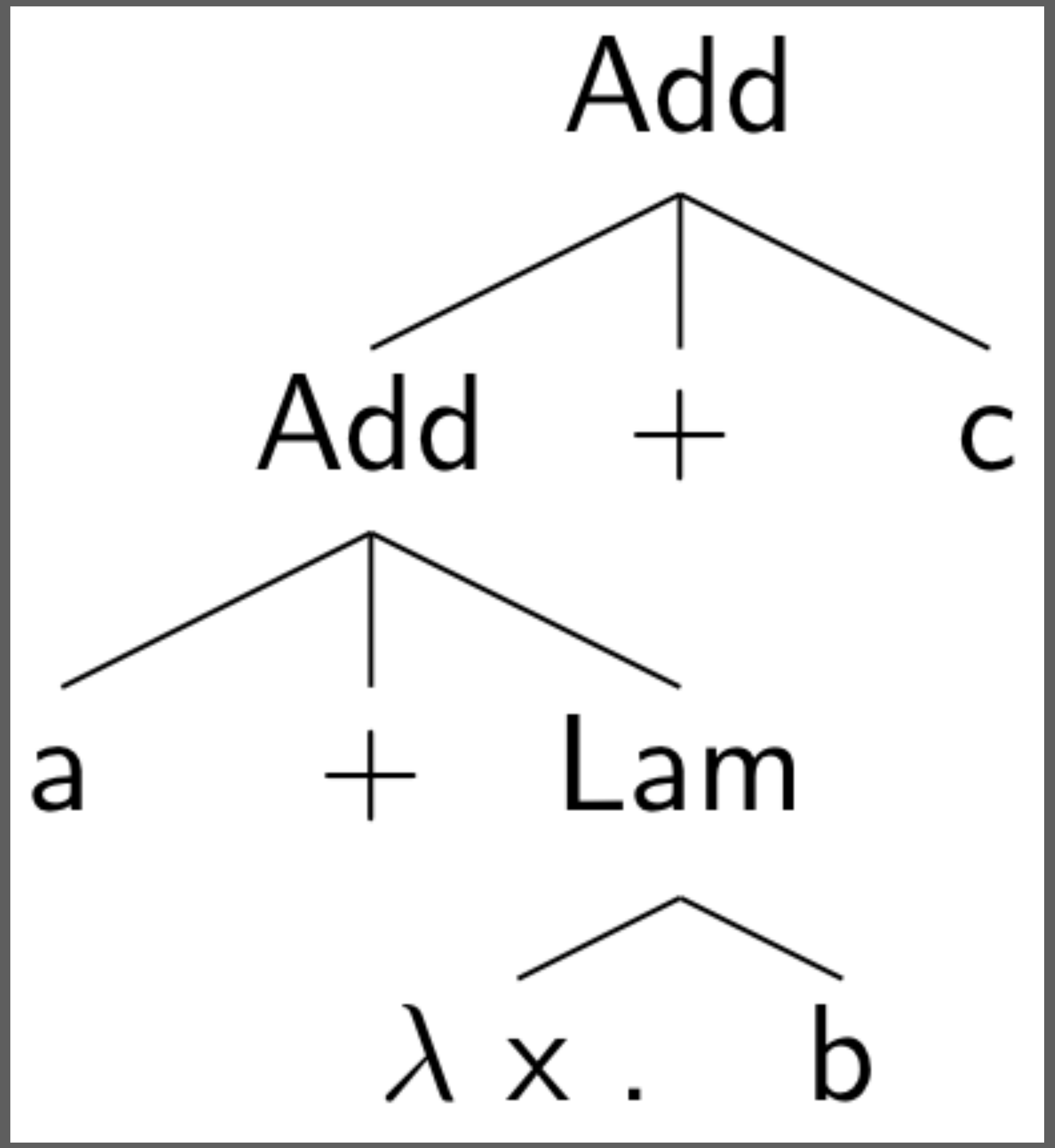




Trees

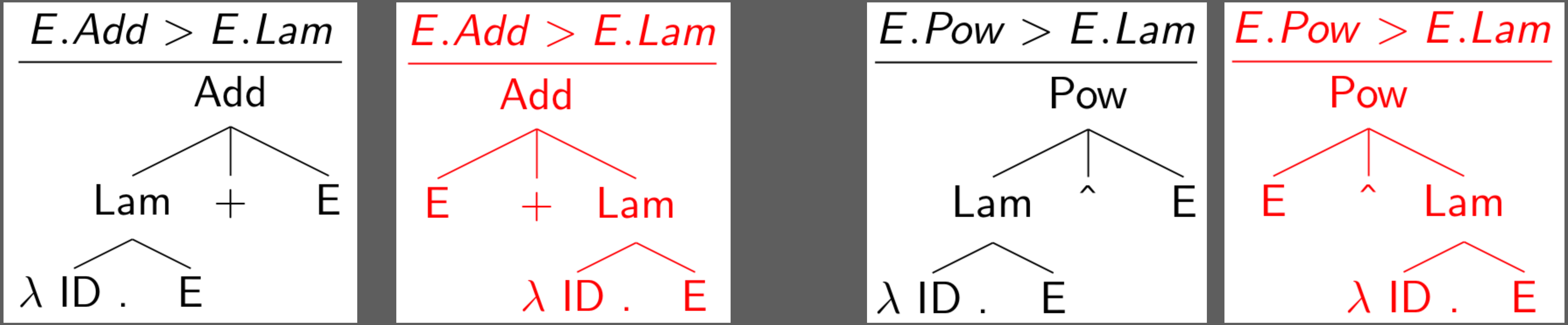






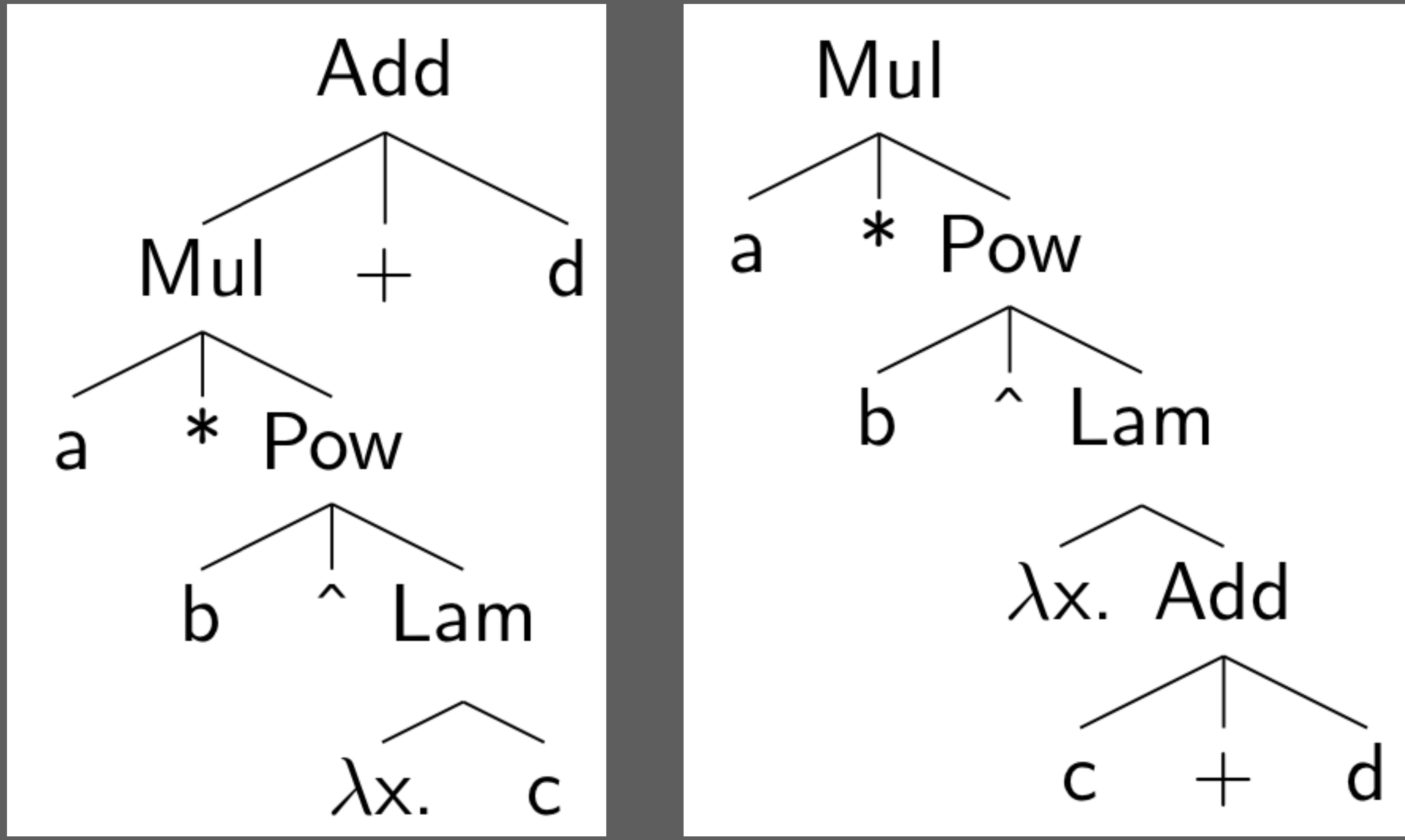
Shallow Interpretation: Incomplete for Low Priority Prefix Operators

Conflict Patterns

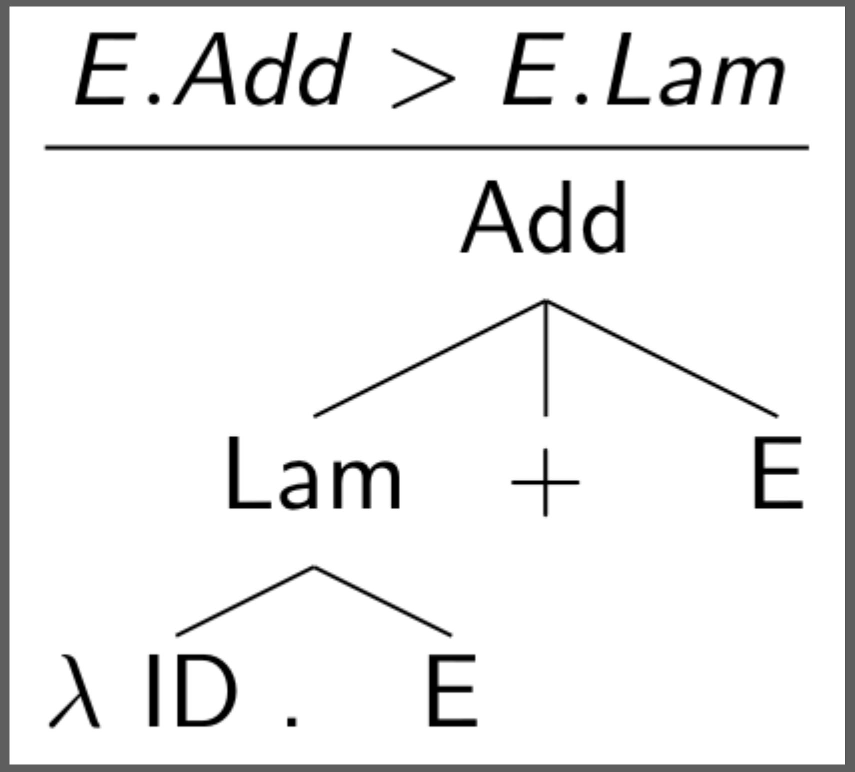


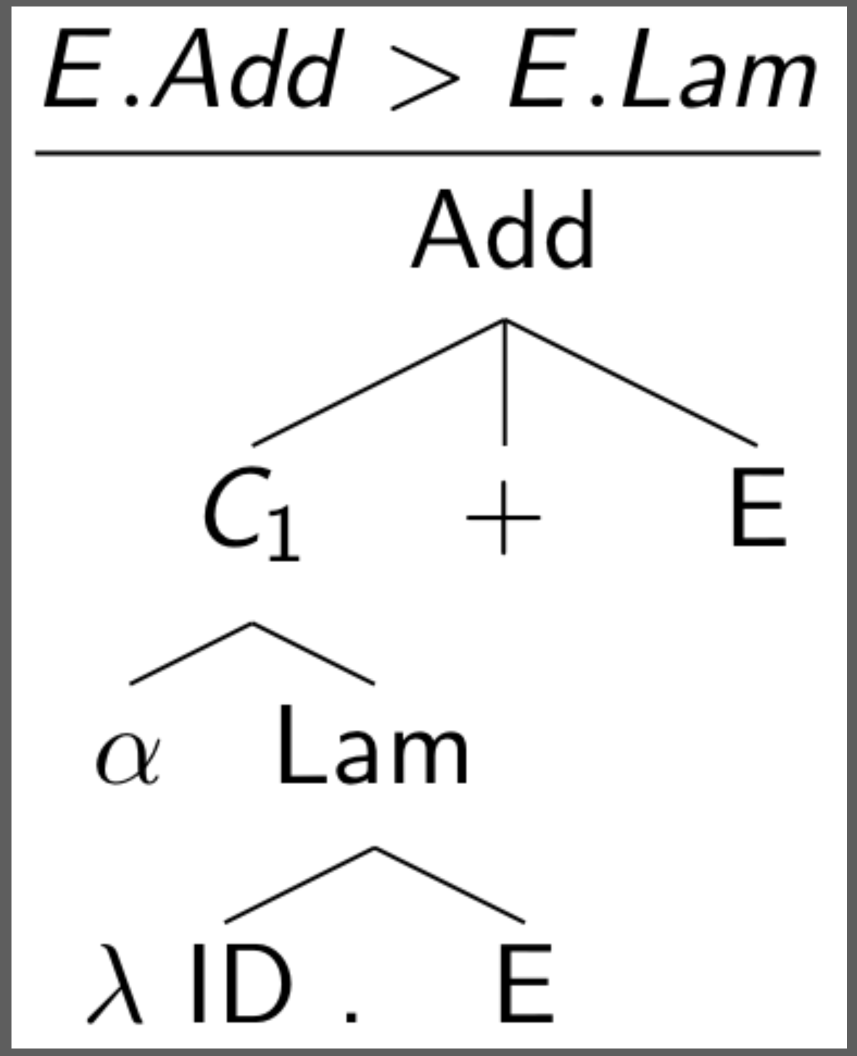
Trees

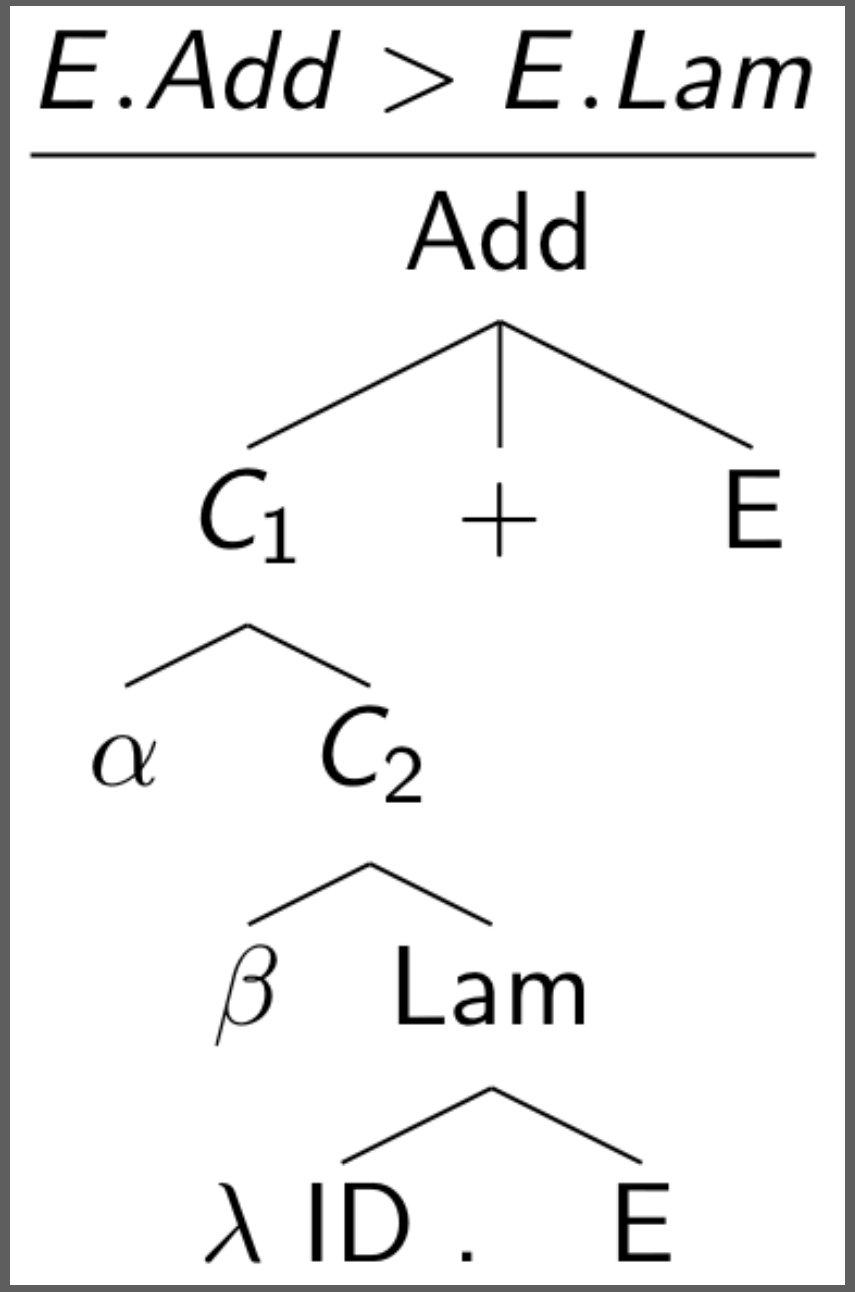
a * b ^ λ x. c + d



Deep Priority Conflicts: Match Subpattern in Right-Most Subtree

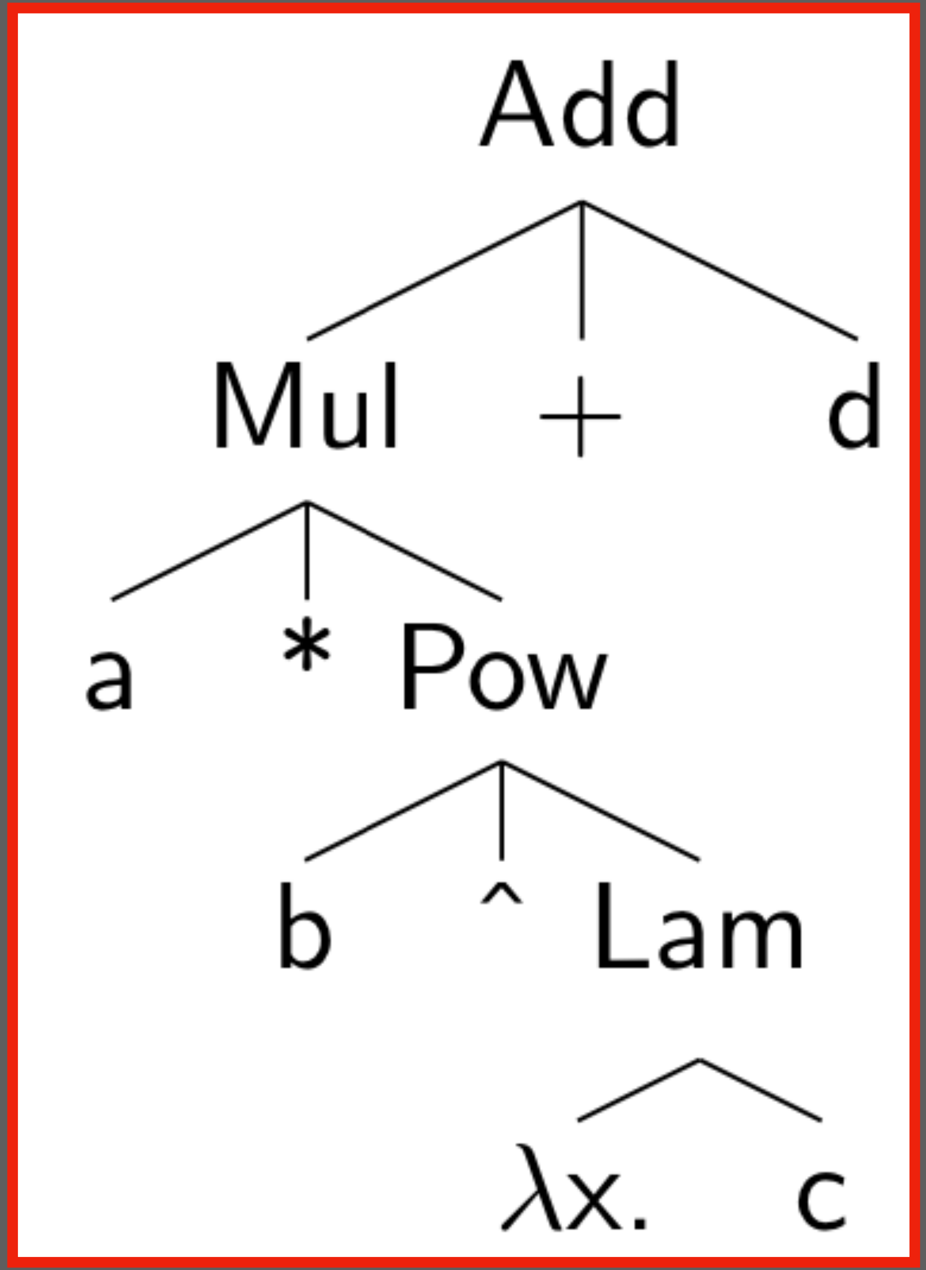


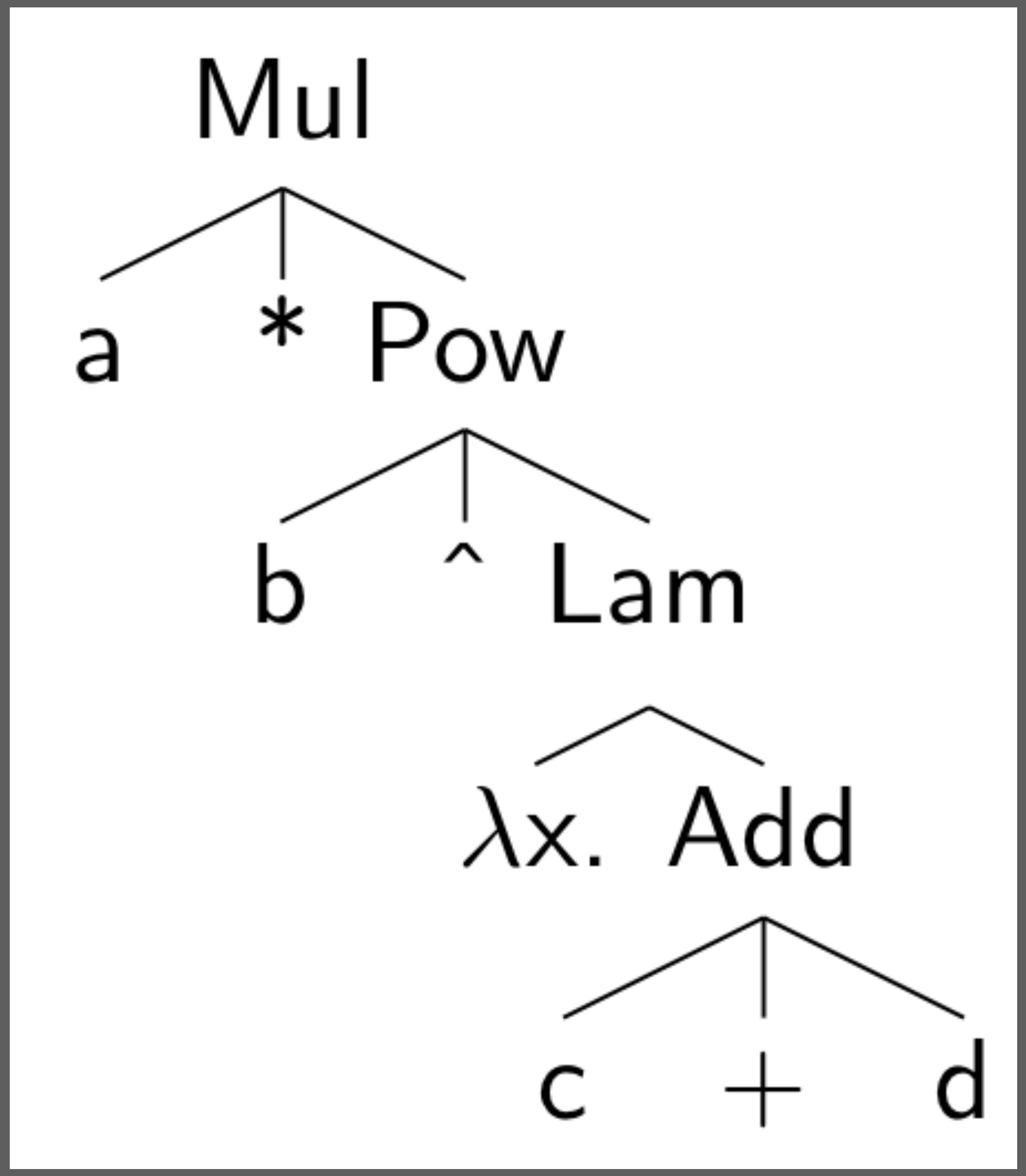




…

Amorim, Visser: A direct semantics of declarative disambiguation rules. (Under revision)

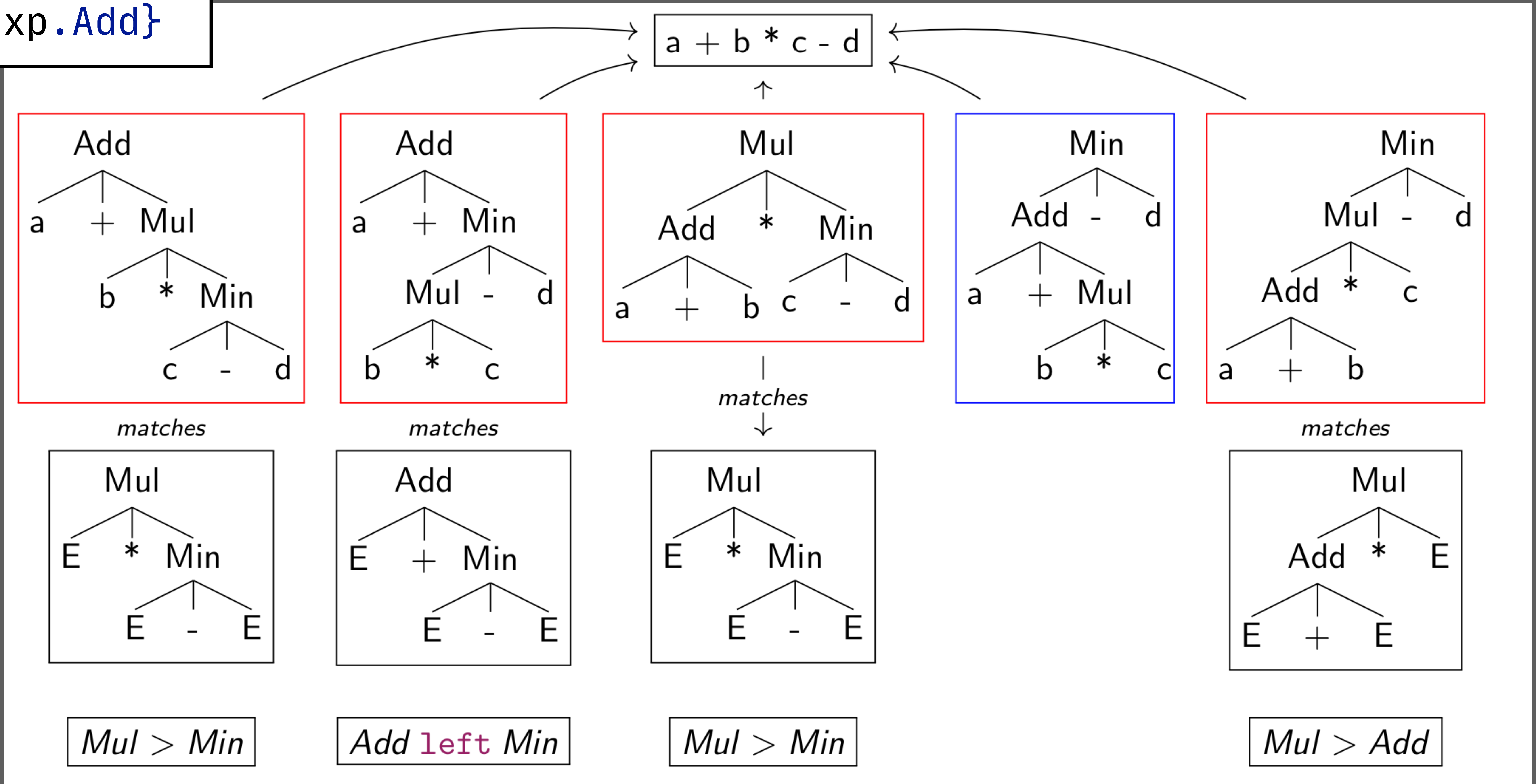




Infinite set of conflict patterns

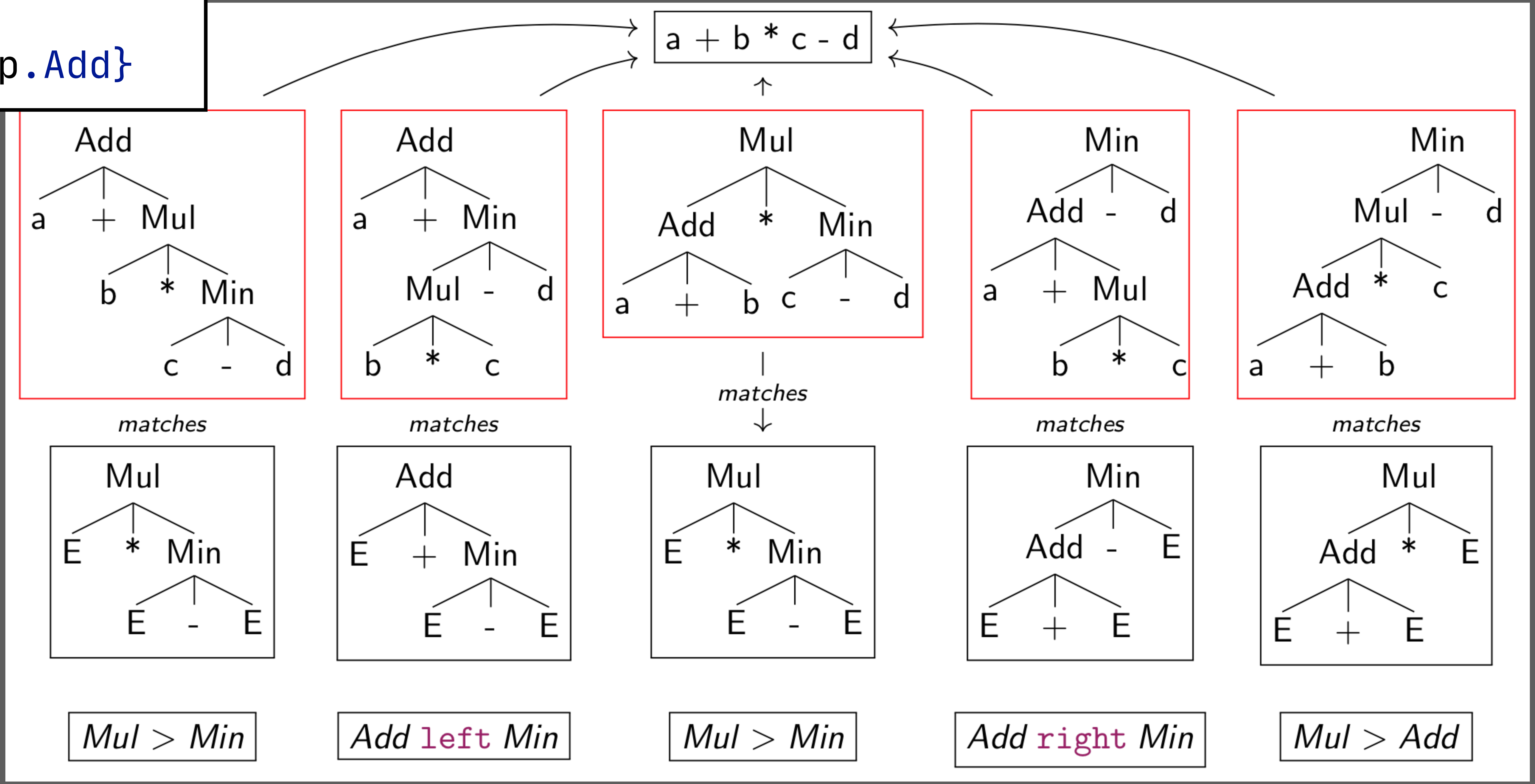
Safe and Complete Disambiguation Rules

```
context-free syntax
  Exp.Min = <<Exp> - <Exp>> {left}
  Exp.Add = <<Exp> + <Exp>> {left}
  Exp.Mul = <<Exp> * <Exp>> {left}
context-free priorities
  Exp.Mul > {left: Exp.Min Exp.Add}
```



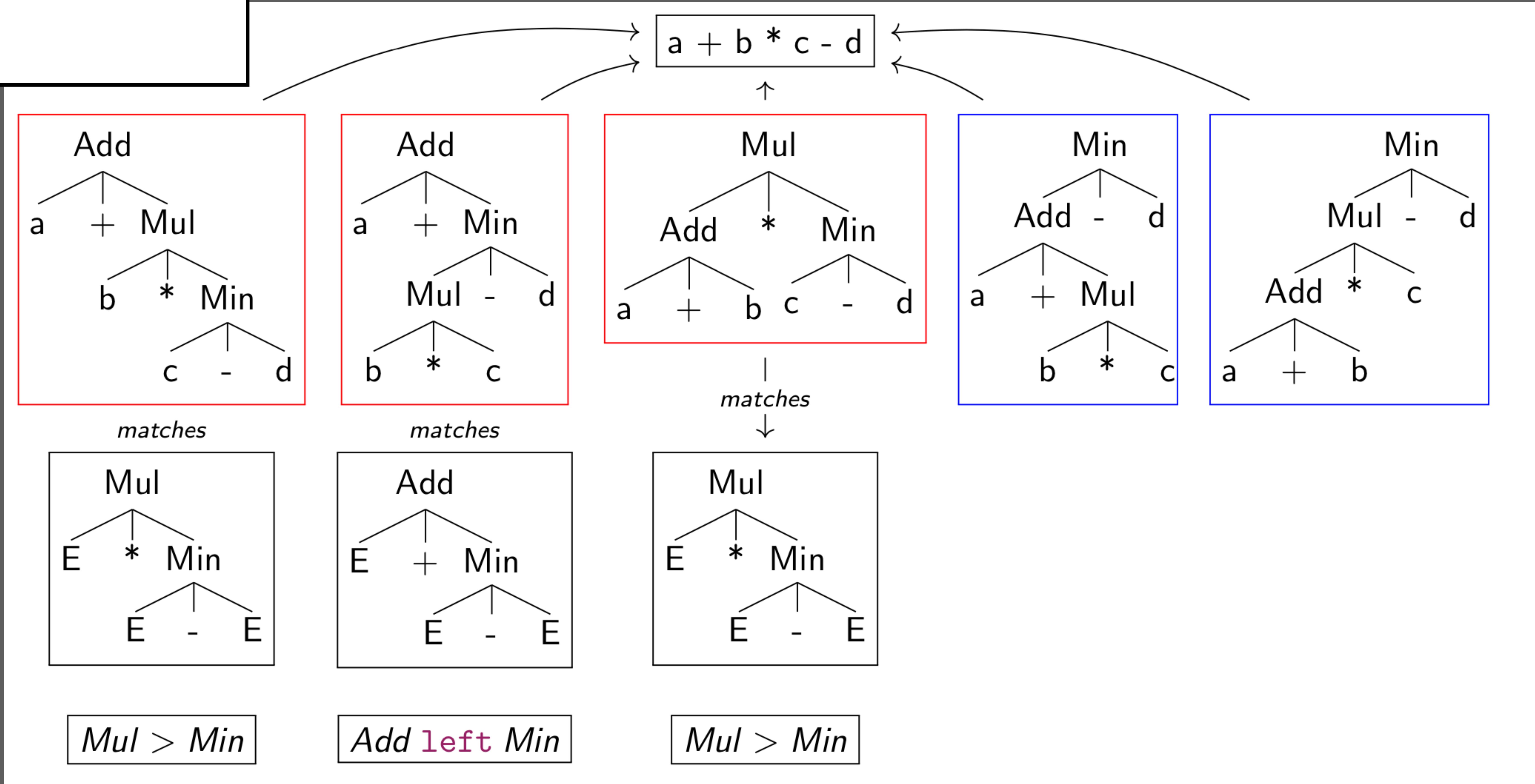
Unsafe: Too Many Disambiguation Rules

```
context-free syntax
  Exp.Min = <<Exp> - <Exp>> {left}
  Exp.Add = <<Exp> + <Exp>> {left}
  Exp.Mul = <<Exp> * <Exp>> {left}
context-free priorities
  Exp.Mul
  > {left, right: Exp.Min Exp.Add}
```



Incomplete: Too Few Disambiguation Rules

```
context-free syntax
  Exp.Min = <<Exp> - <Exp>> {left}
  Exp.Add = <<Exp> + <Exp>> {left}
  Exp.Mul = <<Exp> * <Exp>> {left}
context-free priorities
  {left: Exp.Min Exp.Add}
```



Semantics of Associativity and Priority

What is the semantics of associativity and priority rules?

- Subtree exclusion: (deep) tree patterns that are forbidden

Is a set of disambiguation rules safe?

- At most one rule for each pair of productions

Is a set of disambiguation rules complete?

- At least one rule for each pair of productions

Correctness guaranteed by language definition

- *Manual disambiguation by transformation of grammars is non-trivial*
- *Proof of safety and completeness is non-trivial*

Parenthesize = Disambiguate^{-1} (Insert Necessary Parentheses)

```
context-free syntax
  Exp     = <(<Exp>)> {bracket}

  Exp.Int = INT
  Exp.Var = ID
  Exp.Add = <<Exp> + <Exp>> {left}

  Exp.Let = <let <Bnd*> in <Exp>>
  Bnd.Bnd = <<ID> = <Exp>>
context-free priorities
  Exp.Add > Exp.Let
```

```
(a + (let x = b in c)) + d
```

```
Add(
  Add(
    Var("a")
  , Let([Bnd("x", Var("b"))], Var("c"))
  )
, Var("d")
)
```

```
a + (let
  x = b
in
  c) + d
```

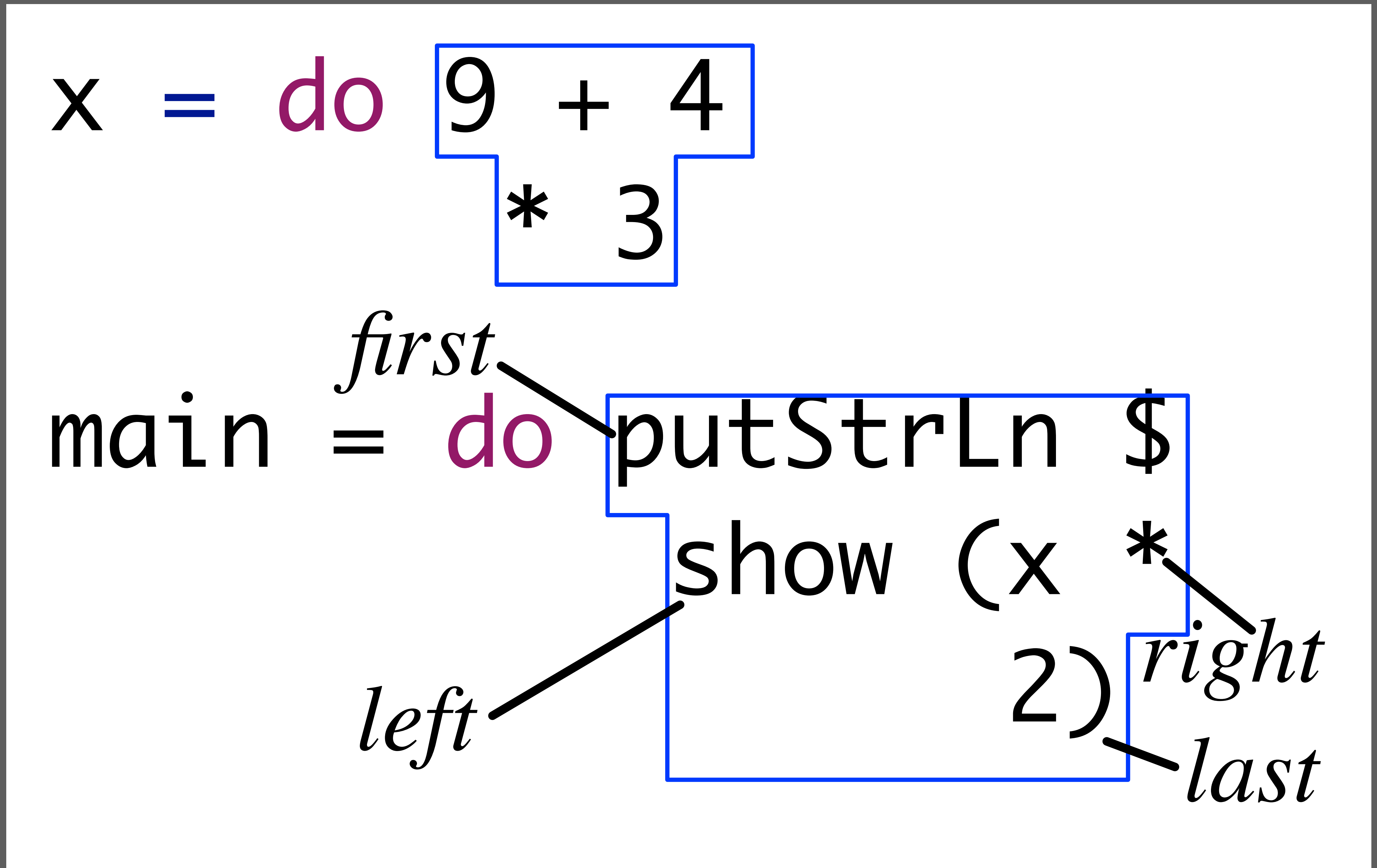
Layout-Sensitive Syntax

Layout-Sensitive Languages

```
if x ≠ y:
  if x > 0:
    y = x
else:
  y = -x
```

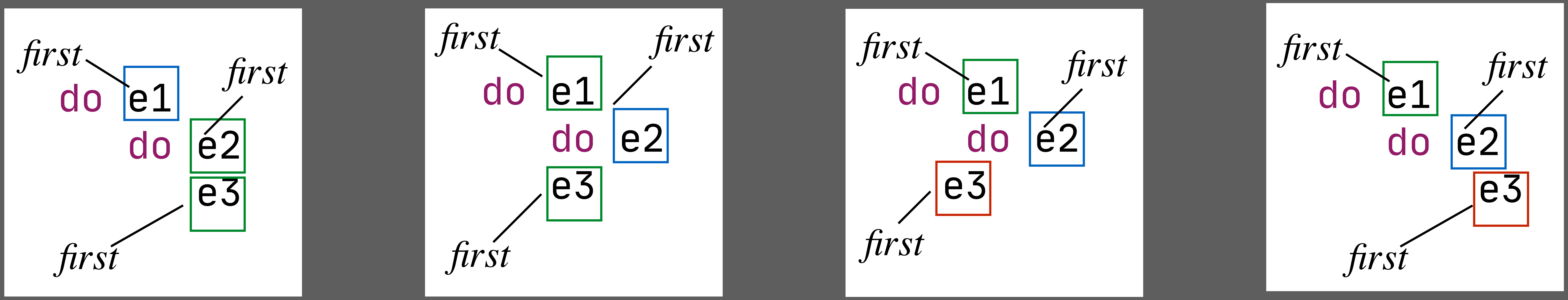
```
guessValue x = do
  putStrLn "Enter your guess:"
  guess ← getLine
  case compare (read guess) x of
    EQ → putStrLn "You won!"
    _  → do putStrLn "Keep guessing."
            guessValue x
```

Token Selectors Identify Two-Dimensional Structure



Alignment with Layout Constraints

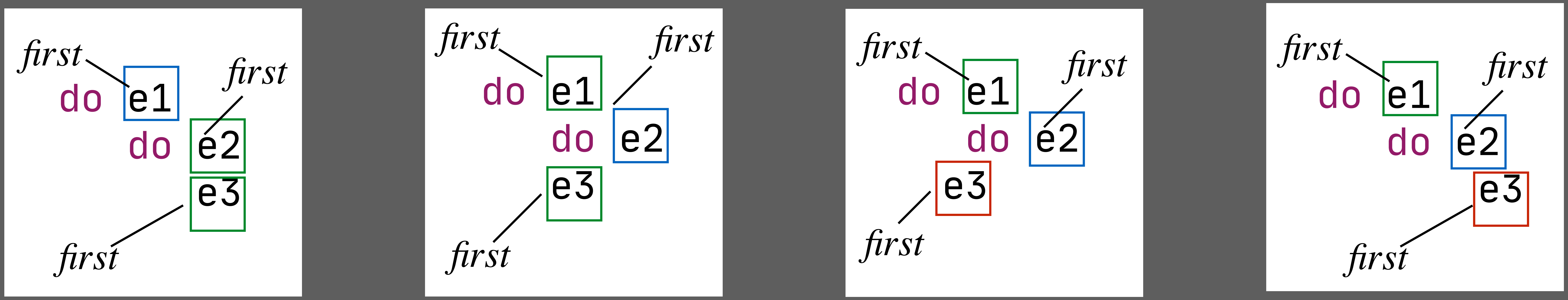
```
context-free syntax
  Exp.Do      = "do" ExpList
  ExpList.Cns = Exp
  ExpList.Lst = ExpList Exp {layout(1.first.col == 2.first.col)}
  Exp.Id      = ID
```



Erdweg, Rendel, Kästner, and Ostermann. Layout-Sensitive Generalized Parsing. SLE 2012

Alignment Declaration

```
context-free syntax
  Exp.Do      = "do" ExpList
  ExpList.Cns = Exp
  ExpList.Lst = exps:ExpList exp:Exp {layout(align exps exp)}
  Exp.Id      = ID
```



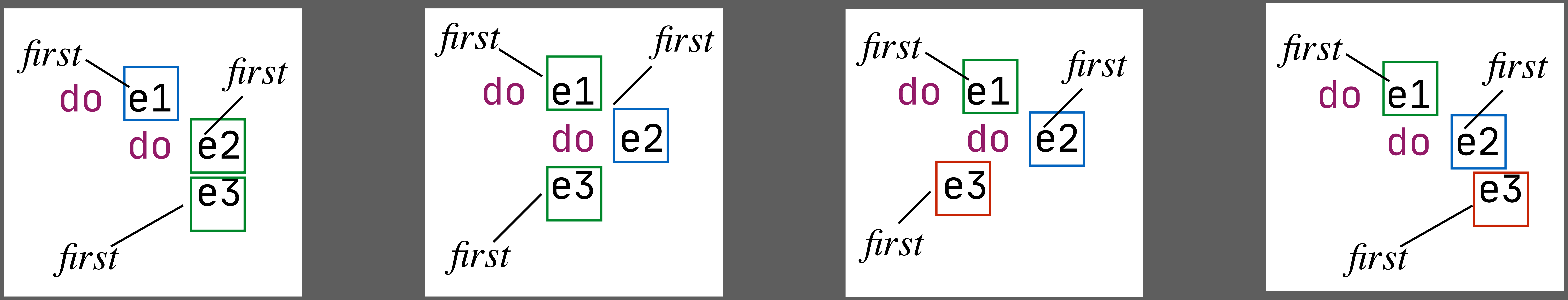
Semantics

$$\frac{x.first.col == y.first.col}{align\ x\ y}$$

Amorim, Steindorfer, Erdweg, Visser: Declarative specification of indentation rules. SLE 2018

List Alignment Declaration

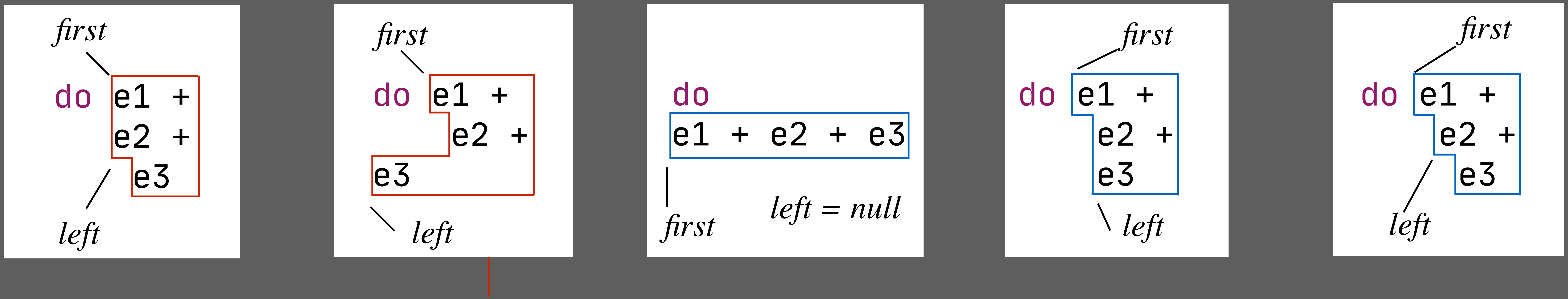
```
context-free syntax
  Exp.Do  = "do" exps:Exp+ {layout(align-list exps)}
  Exp.Id  = ID
  Exp+    = Exp+ Exp // normalized
  Exp+    = Exp      // productions
```



Semantics

$$\frac{\texttt{A+ = A+ A layout(1.first.col == 2.first.col)}}{\texttt{align-list x}}$$

Offside Rule

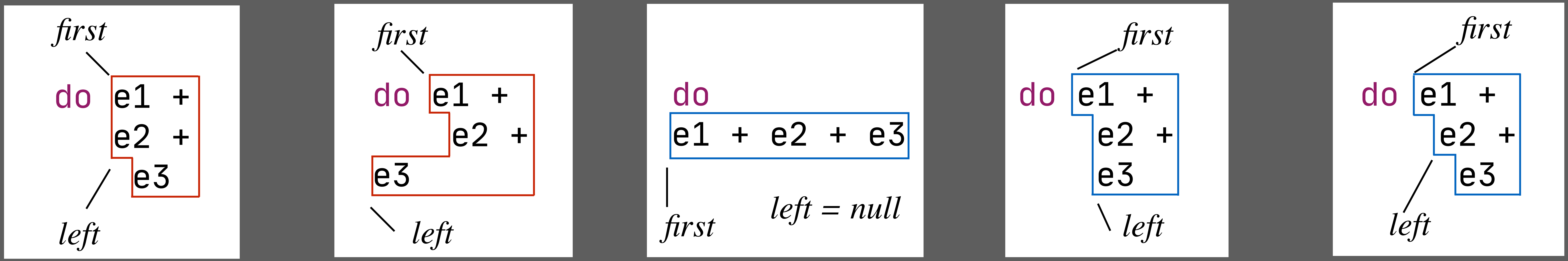


"The offside rule prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure."

Erdweg et. al.. Layout-Sensitive Generalized Parsing. In SLE'12.

Offside with Layout Constraints

```
context-free syntax
  Exp.Do  = "do" Exp {layout(2.left.col > 2.first.col)}
  Exp.Add = Exp "+" Exp {left}
  Exp.Id  = ID
```

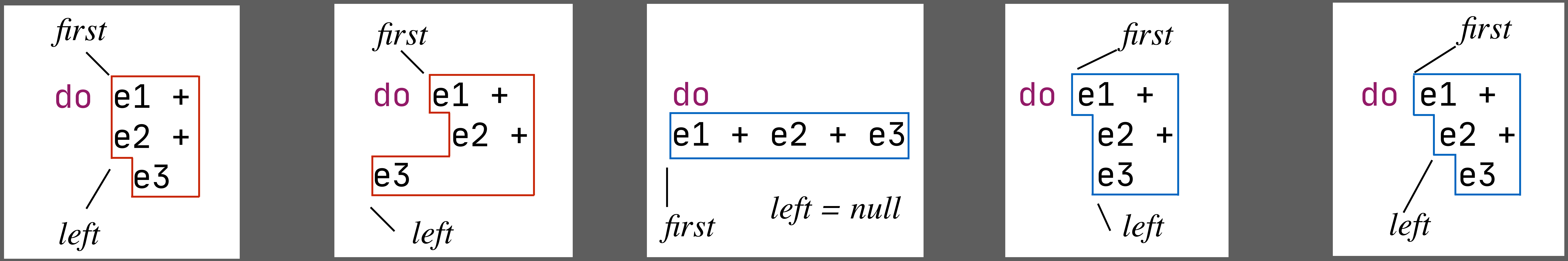


“The offside rule prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure.”

Erdweg et. al.. Layout-Sensitive Generalized Parsing. In SLE’12.

Offside

```
context-free syntax
  Exp.Do  = "do" exp:Exp {layout(offside exp)}
  Exp.Add = Exp "+" Exp {left}
  Exp.Id  = ID
```

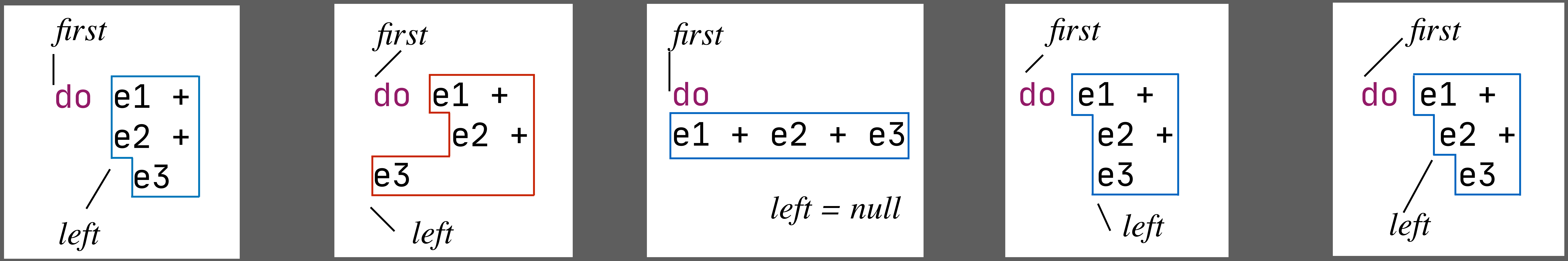


Semantics

$$\frac{\texttt{x.left.col > x.first.col}}{\texttt{offside x}}$$

Relative Offside

```
context-free syntax
  Exp.Do  = "do" exp:Exp {layout(offside "do" exp)}
  Exp.Add = Exp "+" Exp {left}
  Exp.Id  = ID
```

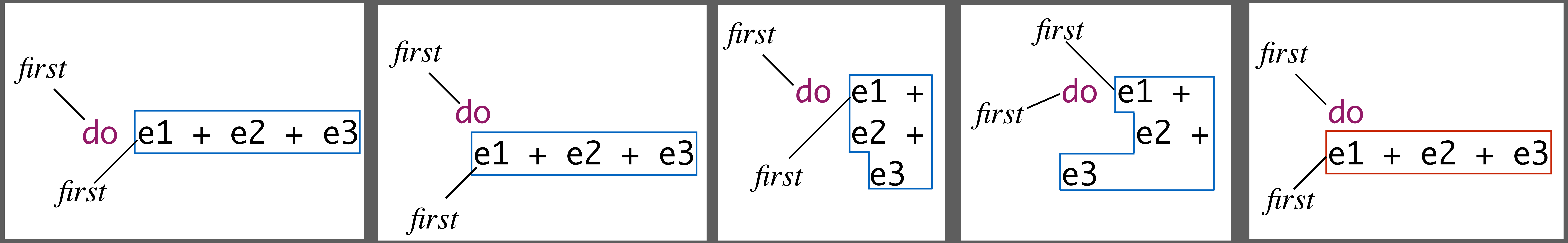


Semantics

$$\frac{\texttt{y.left.col > x.first.col}}{\texttt{offside x y}}$$

Indentation

```
context-free syntax
  Exp.Do  = "do" exp:Exp {layout(indent "do" exp)}
  Exp.Add = Exp "+" Exp {left}
  Exp.Id  = ID
```

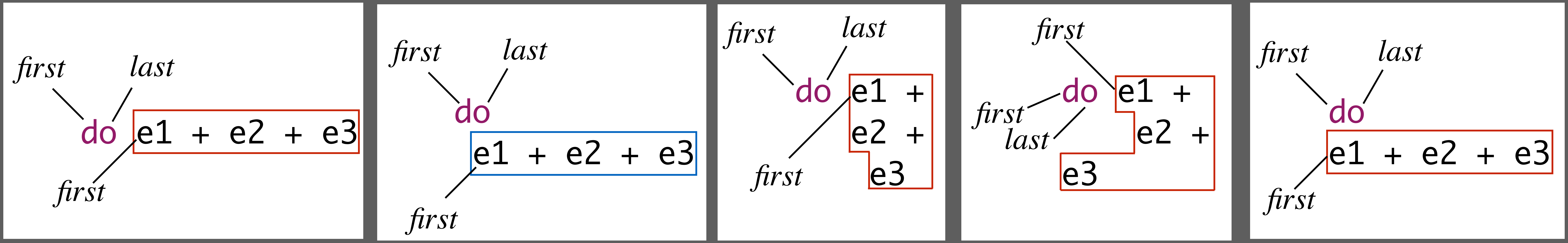


Semantics

$$\frac{y.first.col > x.first.col}{indent\ x\ y}$$

Newline + Indentation

```
context-free syntax
 Exp.Do  = "do" exp:Exp {layout(newline-indent "do" exp)}
  Exp.Add = Exp "+" Exp {left}
  Exp.Id  = ID
```

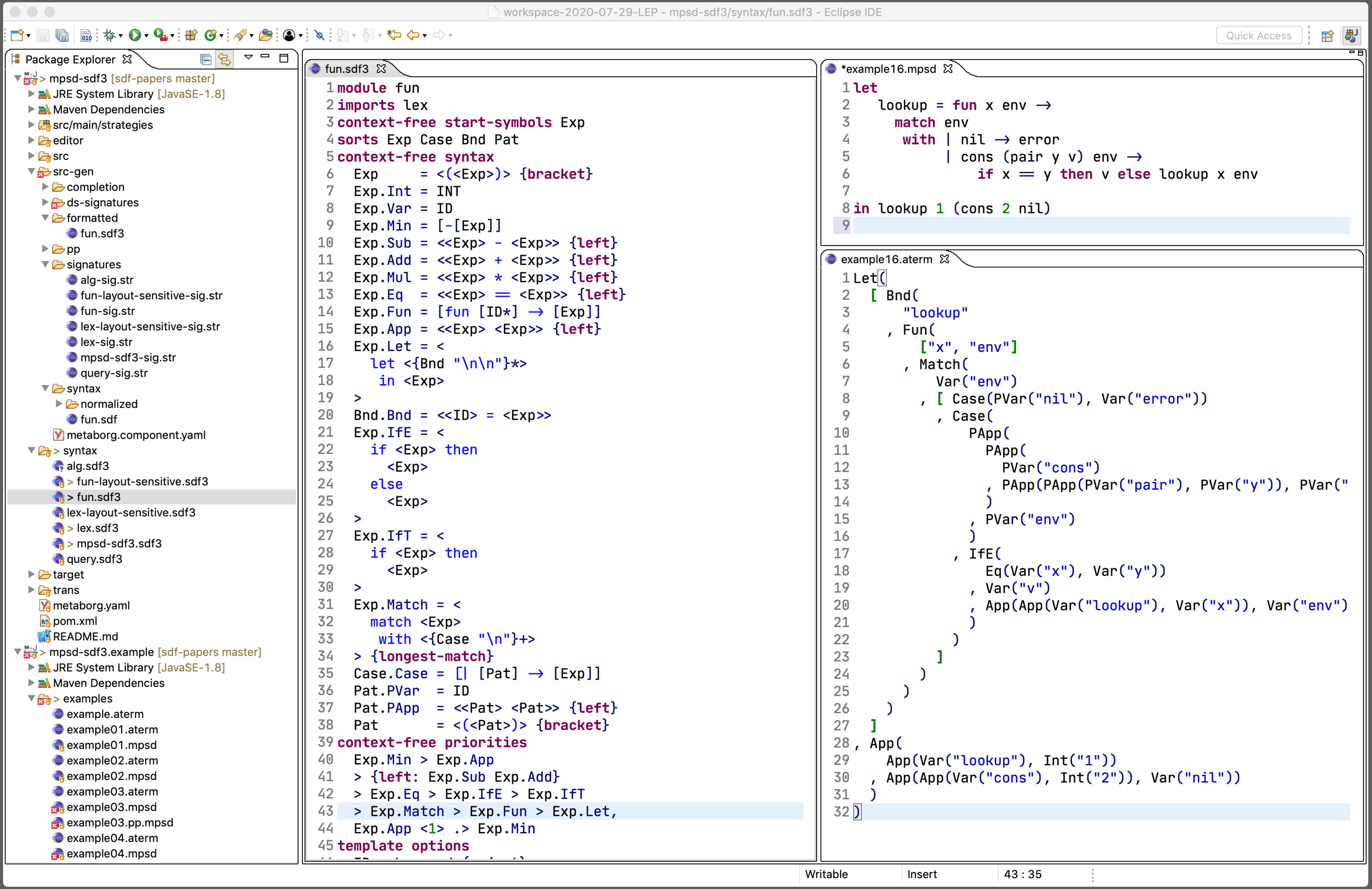


Semantics

$$\frac{\texttt{y.first.col > x.first.col \&\& y.first.line > x.last.line}}{\texttt{newline-indent x y}}$$

How does program layout disambiguate structure?

Spoofax Language Workbench


workspace-2020-07-29-LEP - mpsd-sdf3/syntax/fun.sdf3 - Eclipse IDE
Quick Access
Package Explorer
> mpsd-sdf3 [sdf-papers master]
JRE System Library [JavaSE-1.8]
Maven Dependencies
src/main/strategies
editor
src
src-gen
completion
ds-signatures
formatted
fun.sdf3
pp
signatures
alg-sig.str
fun-layout-sensitive-sig.str
fun-sig.str
lex-layout-sensitive-sig.str
lex-sig.str
mpsd-sdf3-sig.str
query-sig.str
syntax
normalized
fun.sdf
metaborg.component.yaml
> syntax
alg.sdf3
> fun-layout-sensitive.sdf3
> fun.sdf3
lex-layout-sensitive.sdf3
> lex.sdf3
> mpsd-sdf3.sdf3
query.sdf3
target
trans
metaborg.yaml
pom.xml
README.md
> mpsd-sdf3.example [sdf-papers master]
JRE System Library [JavaSE-1.8]
Maven Dependencies
> examples
example.aterm
example01.aterm
example01.mpsd
example02.aterm
example02.mpsd
example03.aterm
example03.mpsd
example03.pp.mpsd
example04.aterm
example04.mpsd
fun.sdf3
module fun
imports lex
context-free start-symbols Exp
sorts Exp Case Bnd Pat
context-free syntax
Exp = <(<Exp>)> {bracket}
Exp.Int = INT
Exp.Var = ID
Exp.Min = [-[Exp]]
Exp.Sub = <<Exp> - <Exp>> {left}
Exp.Add = <<Exp> + <Exp>> {left}
Exp.Mul = <<Exp> * <Exp>> {left}
Exp.Eq = <<Exp> == <Exp>> {left}
Exp.Fun = [fun [ID*] -> [Exp]]
Exp.App = <<Exp> <Exp>> {left}
Exp.Let = <
let <{Bnd "\n\n"}*>
in <Exp>
>
Bnd.Bnd = <<ID> = <Exp>>
Exp.IfE = <
if <Exp> then
<Exp>
else
<Exp>
>
Exp.IfT = <
if <Exp> then
<Exp>
>
Exp.Match = <
match <Exp>
with <{Case "\n"}+>
> {longest-match}
Case.Case = [| [Pat] -> [Exp]]
Pat.PVar = ID
Pat.PApp = <<Pat> <Pat>> {left}
Pat = <(<Pat>)> {bracket}
context-free priorities
Exp.Min > Exp.App
> {left: Exp.Sub Exp.Add}
> Exp.Eq > Exp.IfE > Exp.IfT
> Exp.Match > Exp.Fun > Exp.Let,
Exp.App <1> .> Exp.Min
template options
*example16.mpsd
let
lookup = fun x env ->
match env
with | nil -> error
| cons (pair y v) env ->
if x == y then v else lookup x env
in lookup 1 (cons 2 nil)
example16.aterm
Let(
[Bnd(
"lookup"
, Fun(
["x", "env"]
, Match(
Var("env")
, [Case(PVar("nil"), Var("error"))
, Case(
PApp(
PApp(
PVar("cons")
, PApp(PApp(PVar("pair"), PVar("y")), PVar("
)
, PVar("env")
)
, IfE(
Eq(Var("x"), Var("y"))
, Var("v")
, App(App(Var("lookup"), Var("x")), Var("env")
)
)
]
)
)
)
]
, App(
App(Var("lookup"), Int("1"))
, App(App(Var("cons"), Int("2")), Var("nil"))
)
)
Writable
Insert
43 : 35

Conclusion

Multi-Purpose Syntax Definition with SDF3

High-Level Declarative Domain-Specific Language

- Context-free grammars extended with
- Constructors
- Template productions
- Disambiguation rules
- Layout constraints
- All syntactic aspects of language in one specification

Multi-Purpose Interpretation

- Parsing, Recovery, Syntax Highlighting, Formatting, Completion, Fuzzing, Testing, Parenthesis Insertion, Signature Generation, …
- Possible because high-level and declarative

A work in progress

Generalization: Multi-Purpose Language Definition

High-Level Declarative Domain-Specific Language

- Declarative semantics
- Abstracts from implementation details
- All aspects of language in one specification

Multi-Purpose Interpretation

- Many tools from one specification
- Execution, Generation, Fuzzing, Analysis, Completion, Reverse Engineering, …

Other Spoofax Meta-Languages

Statix

- static semantics (w/ scope graphs)

Dynamix

- dynamic semantics

FlowSpec

- data-flow analysis

Stratego

- transformation strategies

Other Domains

WebDSL

- web programming

IceDust

- declarative data modeling
- derivation of incremental computation

CSX

- configuration space exploration

A Vision for Formal Methods

Domain-Specific Language

- encodes rules of the domain
- *declarative semantics: formally specified, easy to understand*
- users focus on domain programs

Multiple Interpretations

- operational semantics: sound wrt declarative semantics
- *intrinsically verified (sound by construction)*
- *operational semantics $\Rightarrow$ implementation*

Language Designer's Workbench

- helps you put this all together with meta-DSLs

Sooner than another 25 years … ?