

# Multi-Purpose Syntax Definition with **SDF3**

**Eelco Visser**



**SEFM | 'CWI, Amsterdam' | September 16, 2020**

**UvA/CWI**  
**1995**



**TU Delft**  
**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  $\alpha \rightarrow \mathcal{A}$ . 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  $w = \alpha_0 \rightarrow \alpha_1 \rightarrow \dots \rightarrow \alpha_n = S$  and each step has the form  $\alpha_i \beta_i \gamma_i \rightarrow \alpha_i \mathcal{B}_i \gamma_i$  where  $\beta_i \rightarrow \mathcal{B}_i$  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

## Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza<sup>1</sup> and Eelco Visser<sup>2</sup><sup>1</sup> Australian National University, Australia<sup>2</sup> 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.

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<sup>+</sup>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  $\alpha \rightarrow \mathcal{A}$ . 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  $w = \alpha_0 \rightarrow \alpha_1 \rightarrow \dots \rightarrow \alpha_n = S$  and each step has the form  $\alpha_i \beta_i \gamma_i \rightarrow \alpha_i \mathcal{B}_i \gamma_i$  where  $\beta_i \rightarrow \mathcal{B}_i$  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<sup>+</sup>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

Building Program Optimizers with Rewriting Strategies\*

Eelco Visser<sup>1</sup>, Zine-e  
Pacific

<sup>1</sup> Dept. of Comp. Science and Engineering, Oregon  
<sup>2</sup> Dept. of Computer Science, Portland  
visser@acm.org,

**Abstract**

We describe a language for defining term rewriting  
gies, and its application to the production of prog  
-izers. Valid transformations on program terms  
scribed by a set of rewrite rules; rewriting strate  
sl to describe when and how the various rules sh  
plied in order to obtain the desired optimization  
parating rules from strategies in this fashion make  
to reason about the behavior of the optimizer as  
pared to traditional monolithic optimizer impl  
ns. We illustrate the expressiveness of our lang  
ng it to describe a simple optimizer for an ML-li  
diate representation.  
The basic strategy language uses operators suc  
ntial composition, choice, and recursion to bui  
mers from a set of labeled unconditional rewr  
also define an extended language in which t  
ditions and contextual rules that arise in real

Program Transformation with Stratego/XT

Rules, Strategies, Tools, and Systems in Stratego/XT 0.9

Eelco Visser

Institute of Information and Comp  
P.O. Box 80089 3508 TB,  
visser@  
http://www.strat

**Abstract.** Stratego/XT is a framework systems aiming to support a wide range work consists of the transformation lang transformation tools. Stratego is based control of programmable rewriting stra for the infrastructure of transformation printing. The framework addresses the e from the specification of transformations systems. This chapter gives an overview composition of transformation systems the abstraction levels of rules, strategies.

1 Introduction

Program transformation, the automatic ma the context of compilation for the implem ers [28]. While compilers are rather special systems are becoming widespread. In the the generation of programs from specificat neering process. In refactoring [21], transf in order to improve its design. Other appli migration and reverse engineering. The co increase programmer productivity by autom

With the advent of XML, transformation of programming language processing, mak any scenario where structured data play a rol are applicable in document processing. In tu (ASP) for the generation of web-pages in of program generators such as Jostraca [3 concrete syntax of the object language are i

Stratego/XT is a framework for the dev to support a wide range of program trans transformation language Stratego and the X ego is based on the paradigm of rewriting ing strategies. The XT tools provide facilit

C. Lengauer et al. (Eds.): Domain-Specific Program G  
© Springer-Verlag Berlin Heidelberg 2004

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

WebDSL: A Case Study in Domain-Specific Language Engineering

**Abstract.** The the productivity erplate code. It requires a smoo requires techn methodology fo design patterns to tackle comm

**Abstract.** Spoofax is a language workbench for efficient, agile de opment of textual domain-specific languages with state the-art IDE support. Spoofax integrates language proces techniques for parser generation, meta-programming, IDE development into a single environment. It uses con declarative specifications for languages and IDE service this paper we describe the architecture of Spoofax and troduce idioms for high-level specifications of language mantics using rewrite rules, showing how analyses ca reused for transformations, code generation, and editor vices such as error marking, reference resolving, and cor

PIE: A Domain-Specific Language for Interactive Software Development Pipelines

Gabriël Konat<sup>a</sup>, Michael J. Steindorfer<sup>a</sup>, Sebastian Erdweg<sup>a</sup>, and Eelco Visser<sup>a</sup>

<sup>a</sup> Delft University of Technology, The Netherlands

**Abstract**  
**Context.** Software development pipelines are used for automating essential parts of soft processes, such as build automation and continuous integration testing. In particular, int which process events in a live environment such as an IDE, require *timely* results for low- and *persistence* to retain low-latency feedback between restarts.  
**Inquiry.** Developing an incrementalized and persistent version of a pipeline is one way to latency, but requires implementation of dependency tracking, cache invalidation, and other error-prone techniques. Therefore, interactivity complicates pipeline development if time tence become responsibilities of the pipeline programmer, rather than being supported b system. Systems for programming incremental and persistent pipelines exist, but do not fo velopment, requiring a high degree of boilerplate, increasing development and maintaina  
**Approach.** We develop Pipelines for Interactive Environments (PIE), a Domain-Specific Lan and runtime for developing interactive software development pipelines, where ease of de cus. The PIE DSL is a statically typed and lexically scoped language. PIE programs are com implementing the API, which the PIE runtime executes in an incremental and persistent w  
**Knowledge.** PIE provides a straightforward programming model that enables direct and c of pipelines without boilerplate, reducing the development and maintenance effort of pip pipeline programs can be embedded into interactive environments such as code editors an timely feedback at a low cost.  
**Grounding.** Compared to the state of the art, PIE reduces the code required to express an in by a factor of 6 in a case study on syntax-aware editors. Furthermore, we evaluate PIE in ty complex interactive software development scenarios, demonstrating that PIE can handle co pipelines in a straightforward and concise way.  
**Importance.** Interactive pipelines are complicated software artifacts that power many in such as continuous feedback cycles in IDEs and code editors, and live language develop workbenches. New pipelines, and evolution of existing pipelines, is frequently necessary. Th for easily developing and maintaining interactive pipelines, such as PIE, is important.

ACM CCS 2012

▪ Software and its engineering → Domain specific languages; Development frameworks a environments; Source code generation; Runtime environments;

**Keywords** domain-specific language, pipeline, interactive software development, increme

The Art, Science, and Engineering of Programming

Submitted December 1, 2017

Published March 29, 2018

Intrinsically-Typed Definitional Interpreters for Imperative Languages

CASPER BACH POULSEN, Delft University of Technology, The Netherlands  
ARJEN ROUVOET, Delft University of Technology, The Netherlands  
ANDREW TOLMACH, Portland State University, USA  
ROBERT K. DEERERS, Delft University of Technology, The Netherlands

**Abstract.** semantics of an object language in terms of the (well-known) semantics anding and validation of the semantics through execution. Combining rate type system requires a separate type safety proof. An alternative nguages, is to use a dependently-typed language to encode the object in of the abstract syntax. Using such intrinsically-typed abstract syntax type checker to verify automatically that the interpreter satisfies type s, and in particular to languages

A Theory of Name Resolution

Pierre Neron<sup>1</sup>, Andrew Tolmach<sup>2</sup>, Eelco Visser<sup>1</sup>, and Guido Wachsmuth<sup>1</sup>

<sup>1</sup> Delft University of Technology, The Netherlands,  
{p.j.m.neron,e.visser,g.wachsmuth}@tudelft.nl  
<sup>2</sup> Portland State University, Portland, OR, USA  
tolmach@pdx.edu

**Abstract.** We describe a language-independent theory for name binding and resolution, suitable for programming languages with complex scop ing rules including both lexical scoping and modules. We formulate name plution as a two-stage problem. First a language-independent scope ph is constructed using language-specific rules from an abstract syn tree. Then references in the scope graph are resolved to correspond declarations using a language-independent resolution process. We reduce a resolution calculus as a concise, declarative, and language-dependent specification of name resolution. We develop a resolution withm that is sound and complete with respect to the calculus. Based

A Constraint Language for Static Semantic Analysis Based on Scope Graphs

Hendrik van Antwerpen  
TU Delft, The Netherlands  
h.vanantwerpen@tudelft.nl

Pierre Néron  
TU Delft, The Netherlands  
p.j.m.neron@tudelft.nl

Andrew Tolmach  
Portland State University, USA  
tolmach@pdx.edu

Eelco Visser  
TU Delft, The Netherlands  
visser@acm.org

Guido Wachsmuth  
TU Delft, The Netherlands  
guwac@acm.org

**Abstract**

In previous work, we introduced *scope graphs* as a formalism for describing program binding structure and performing name resolution in an AST-independent way. In this paper, we show how to use scope graphs to build static semantic analyzers. We use *constraints* extracted from the AST to specify facts about binding, typing, and initialization. We treat name and type resolution as separate building blocks, but our approach can handle language constructs—such as record field access—for which binding and typing are mutually dependent. We also refine and extend our previous scope graph theory to address practical concerns including ambiguity checking and support for a wider range of scope relationships. We describe the details of constraint generation for a model language that illustrates many of the interesting static analysis issues associated with modules and records.

**Categories and Subject Descriptors** D.3.1 [Programming Languages]: Formal Definitions and Theory; D.3.2 [Programming Languages]: Language classifications; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.3.4 [Programming Languages]: Processors; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages; D.2.6 [Software Engineering]: Programming Environments

**Keywords** Language Specification; Name Binding; Types; Domain Specific Languages; Meta-Theory

1. Introduction

Language workbenches [6] are tools that support the implementation of full-fledged programming environments for (domain-specific) programming languages. Ongoing research investigates how to reduce implementation effort by factoring out language-independent implementation concerns and providing high-level

meta-languages for the specification of syntactic and semantic aspects of a language [18]. Such meta-languages should (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) be declarative, but be realizable by practical algorithms and tools; (iv) be factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs as well as to correct ones.

In recent work we showed how name resolution for lexically-scoped languages can be formalized in a way that meets these criteria [14]. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its basic building blocks are *scopes*, which correspond to sets of program points that behave uniformly with respect to resolution. A scope contains identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by edges representing lexical nesting or import of named collections of declarations such as modules or records. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a largely language-independent way. A *resolution calculus* gives a formal definition of what it means for a reference to resolve to a declaration. Resolutions are described as paths in the scope graph obeying certain (language-specific) criteria; a given reference may resolve to one or many declarations (or to none). A derived *resolution algorithm* computes the set of declarations to which each reference resolves, and is sound and complete with respect to the calculus.

In this paper, we refine and extend the scope graph framework of [14] to a full framework for static semantic analysis. In essence, this involves uniting a type checker with our existing name resolution machinery. Ideally, we would like to keep these two aspects separated as much as possible for maximum modularity. And indeed, for many language constructs, a simple two-stage approach—name resolution using the scope graph followed by a separate type checker—works well. But for all recursive expressions and

SDF3

Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza<sup>1</sup> and Eelco Visser<sup>2</sup>

<sup>1</sup> Australian National University, Australia  
<sup>2</sup> 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.

Introduction

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

In this paper, we give an overview of the syntax definition formalism SDF3 d its tool ecosystem that supports the multi-purpose interpretation of syntax initions. The paper does not present any new technical contributions, but is the first paper to give a (high-level) overview of all aspects of SDF3 and ves as a guide to the literature. SDF3 is the third generation in the SDF aily of syntax definition formalisms, which were developed in the context of e ASF+SDF [5], Stratego/XT [10], and Spoofax [38] language workbenches. The first SDF [23] supported modular composition of syntax definition, a dit correspondence between concrete and abstract syntax, and parsing with the l class of context-free grammars enabled by the Generalized-LR (GLR) pars algorithm [56,44]. Its programming environment, as part of the ASF+SDF taEnvironment [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

SDF2

SDF3

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<sup>+</sup>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  $\alpha \rightarrow A$ . 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  $w = \alpha_0 \rightarrow \alpha_1 \rightarrow \dots \rightarrow \alpha_n = S$  and each step has the form  $\alpha_i \beta_i \gamma_i \rightarrow \alpha_i B_i \gamma_i$  where  $\beta_i \rightarrow B_i$  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<sup>+</sup>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



Multi-Purpose Syntax Definition with SDF3

Luís Eduardo Amorim de Souza<sup>1</sup> and Eelco Visser<sup>2</sup>

<sup>1</sup> Australian National University, Australia

<sup>2</sup> 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

## 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

## 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]

## 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

## Spoofox 2

## 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

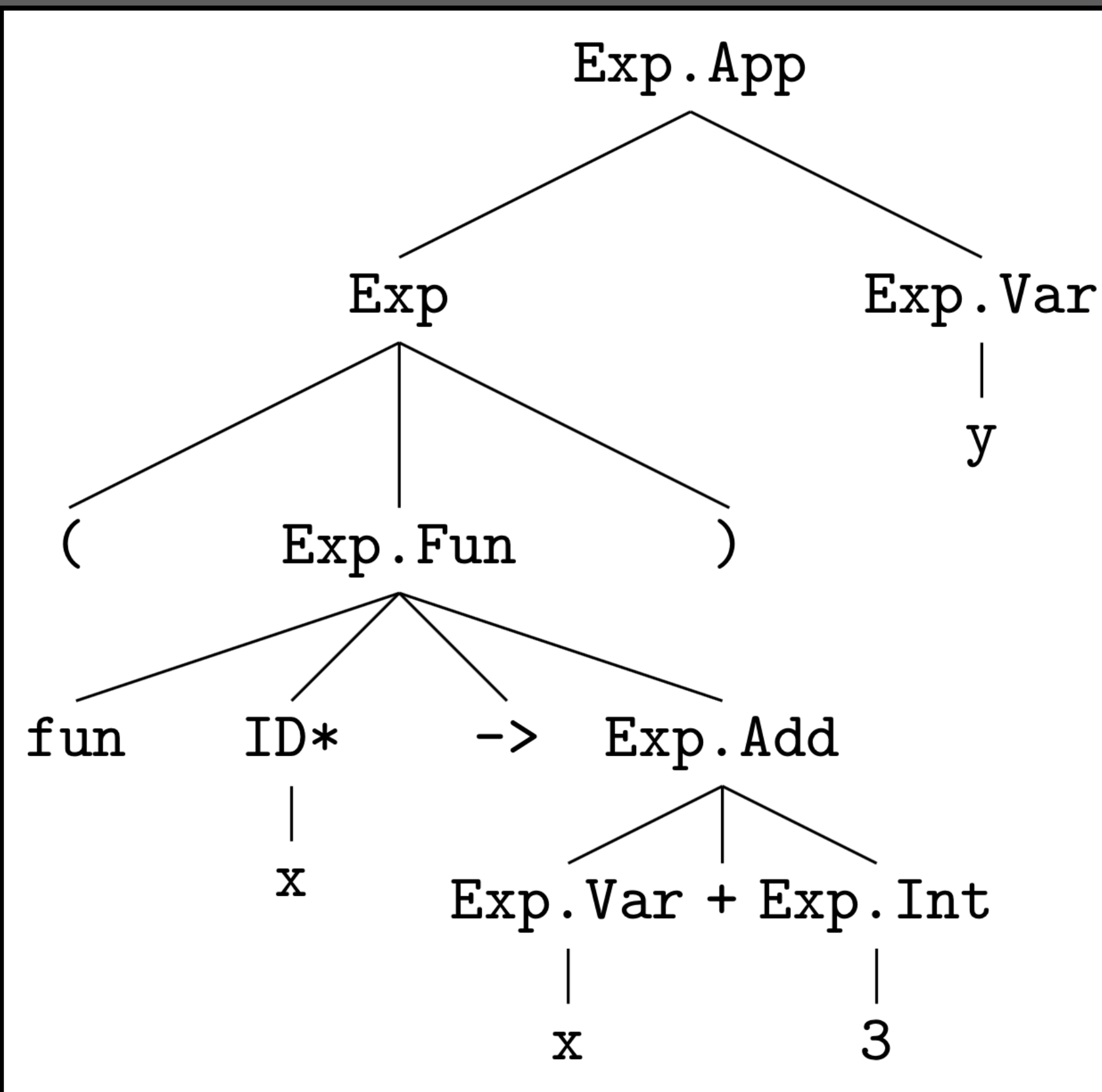
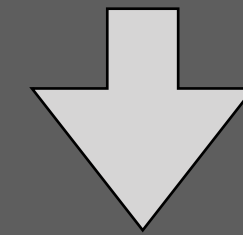
Exp.Var = ID

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 $\Rightarrow$ Abstract Syntax Tree

## context-free syntax

Exp = "(" Exp ")" {**bracket**}

Exp.Int = INT

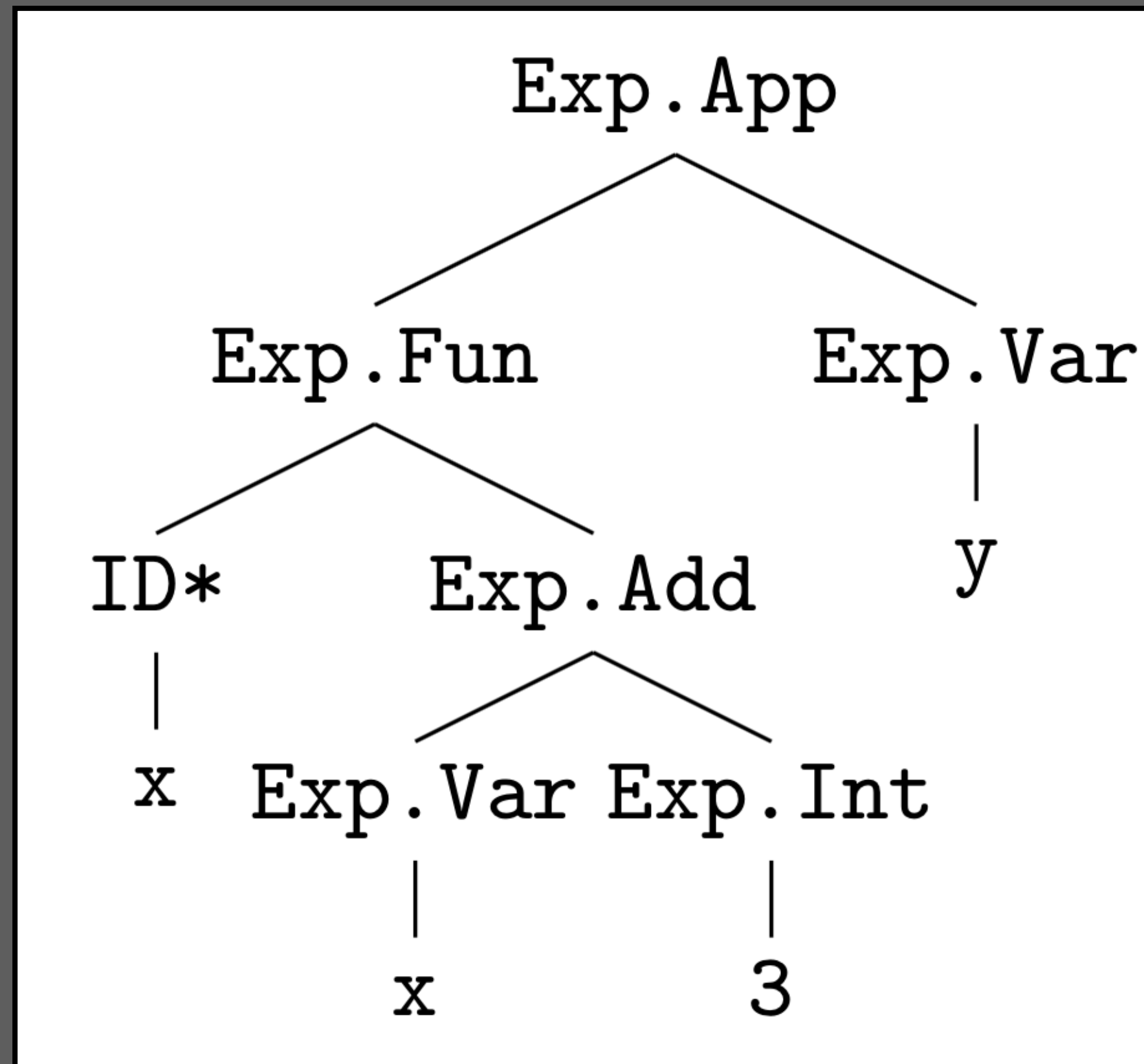
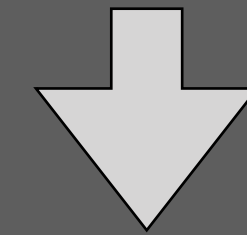
Exp.Var = ID

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

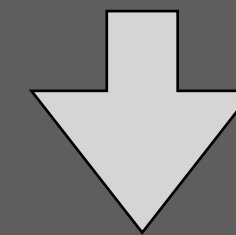
Exp.Var = ID

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 $\simeq$ Algebraic Signature

## context-free syntax

Exp = "(" Exp ")" {**bracket**}

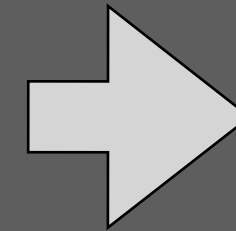
Exp.Int = INT

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

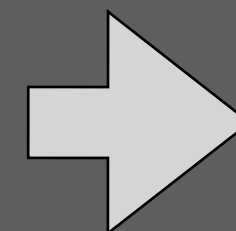
Var : ID → Exp

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
```

Syntax = Structure

Language Designers  
focus on  
Structure of Programs

# Formatting Templates

# parse = (implode ; format)<sup>-1</sup>

## 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>  
>

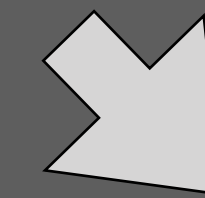
Bnd.Bnd = <<ID> = <Exp>>

**let**

  inc = **fun** x → x + 1

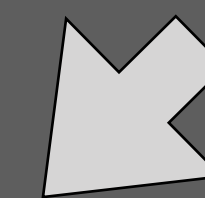
**in**

  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

# Parsing + Formatting Declaratively

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

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

# Syntactic Completion = Rewriting Incomplete Programs

```
let
  inc = fun x → x + 1
in
  inc $Exp
```

- + Var
- + Fun
- + Let
- + Add
- + IfT
- + IfF

```
let
in $Exp
```

Explicit incompleteness: extend language with placeholders

Completion: rewrite placeholders

Templates: Formatting proposals

```
let
  inc = fun x → x + 1
in
  inc let y = $Exp
```

```
in $Exp
```

- + Var
- + Sub
- + Fun
- + Let
- + App
- + IfT

```
$Exp $Exp
```

Soundness:  
Only syntactically correct proposals

Completeness:  
Reach all programs

**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

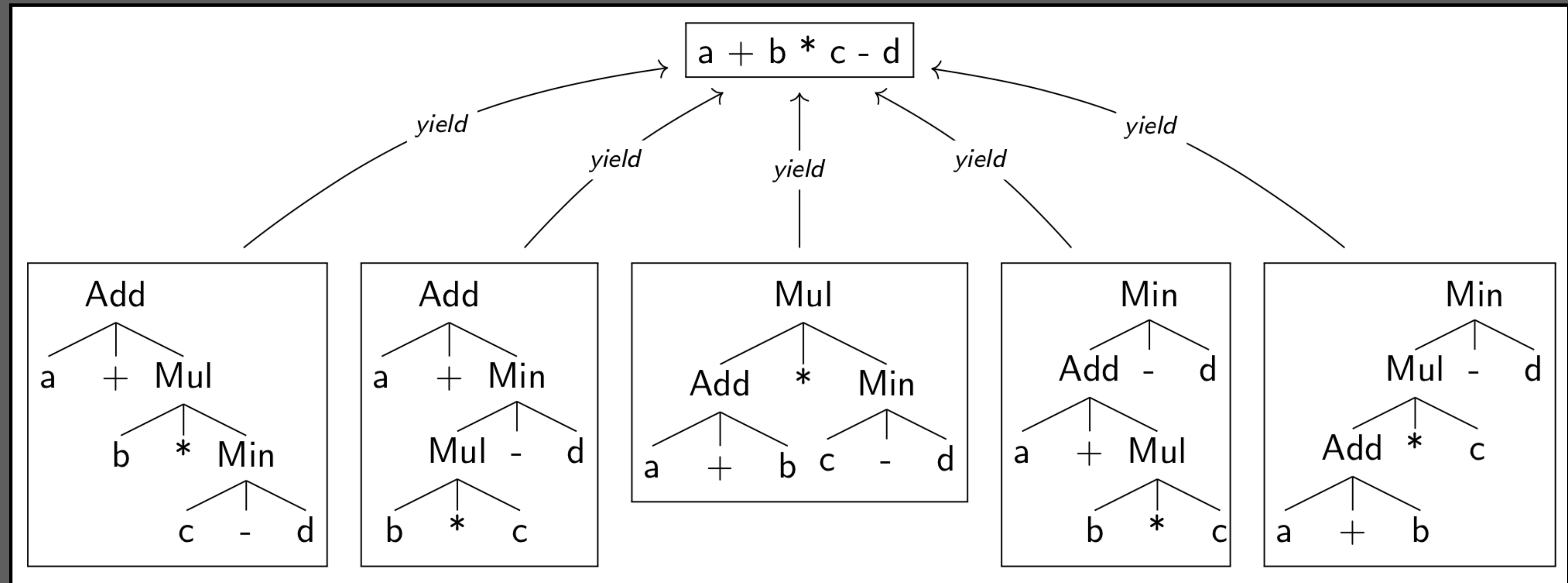
Exp.Int = INT

Exp.Var = ID

Exp.Min =  $\langle\langle\text{Exp}\rangle - \langle\text{Exp}\rangle\rangle$

Exp.Add =  $\langle\langle\text{Exp}\rangle + \langle\text{Exp}\rangle\rangle$

Exp.Mul =  $\langle\langle\text{Exp}\rangle * \langle\text{Exp}\rangle\rangle$



# Disambiguation with Associativity and Priority Rules

## context-free syntax

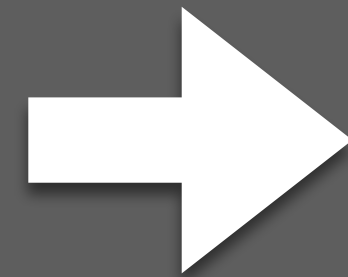
Exp.Int = INT

Exp.Var = ID

Exp.Min = <<Exp> - <Exp>>

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

Exp.Mul = <<Exp> \* <Exp>>



## context-free syntax

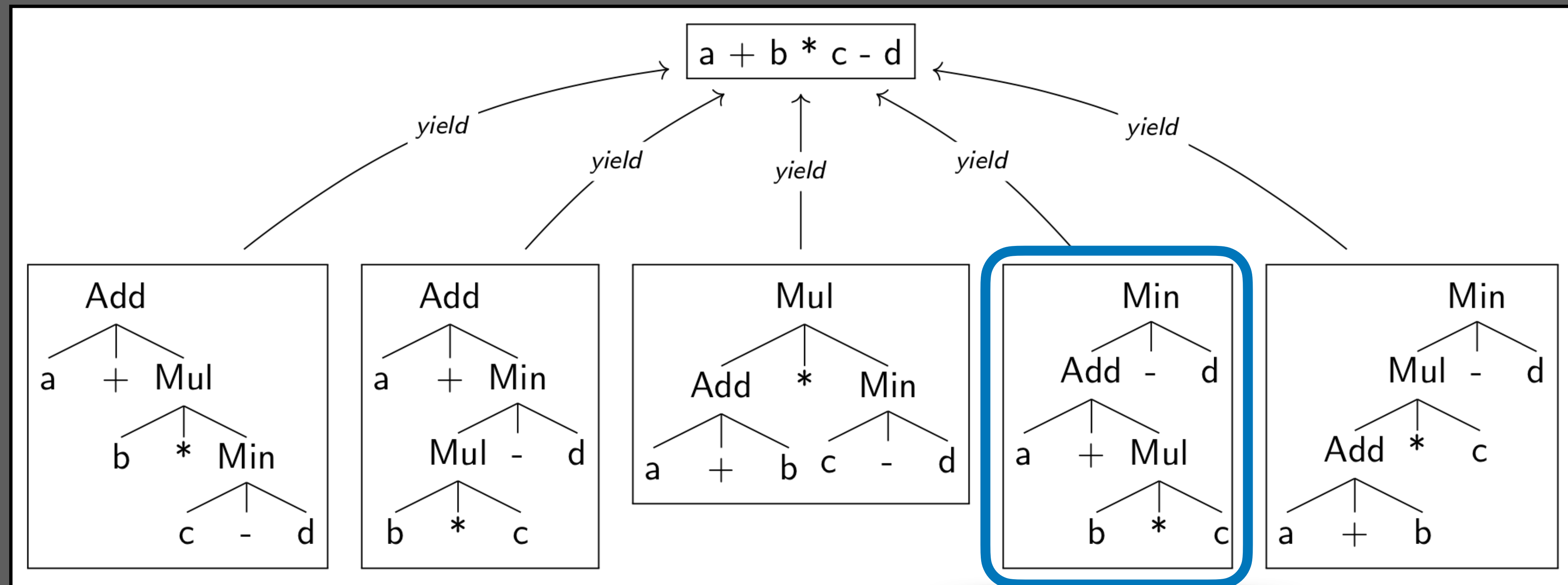
Exp.Min = <<Exp> - <Exp>> {left}

Exp.Add = <<Exp> + <Exp>> {left}

Exp.Mul = <<Exp> \* <Exp>> {left}

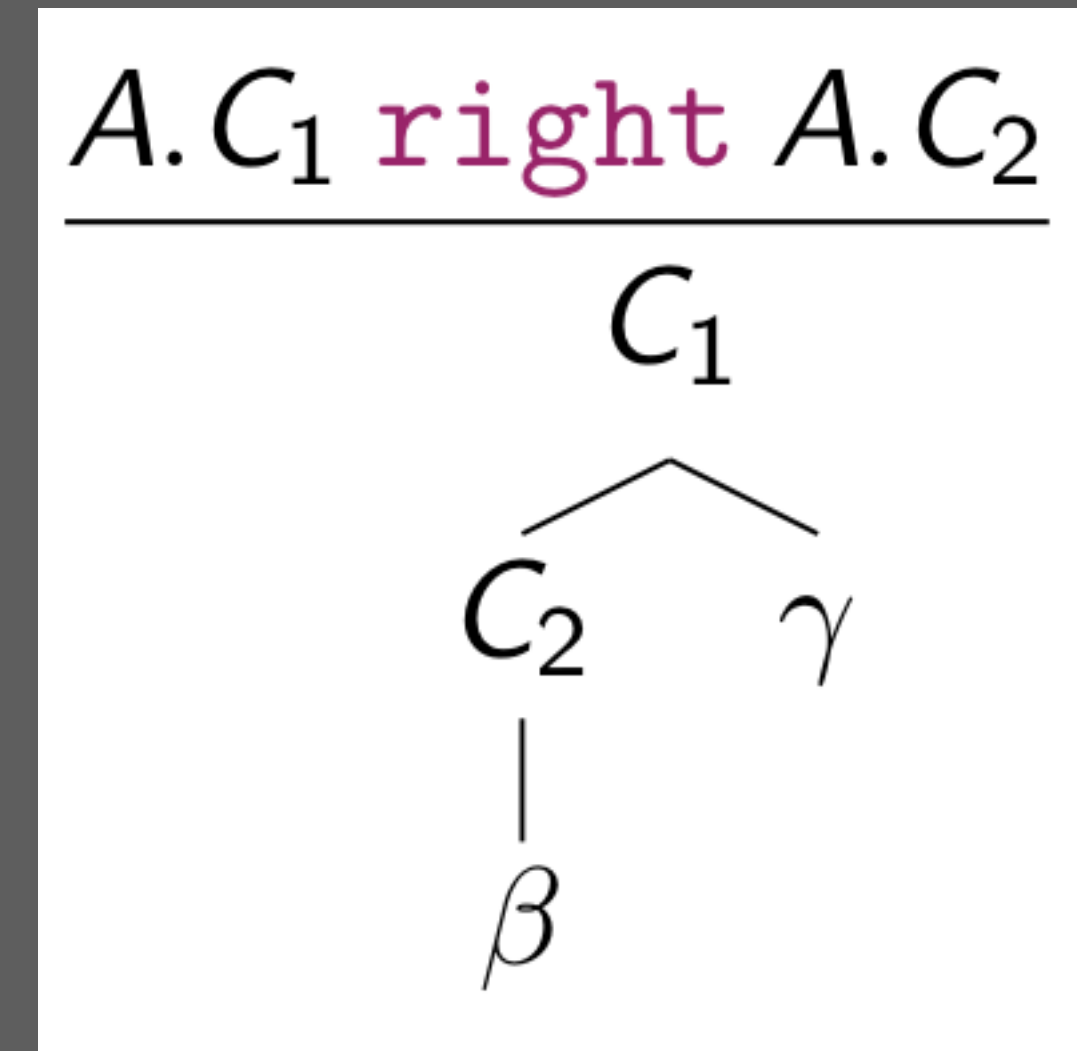
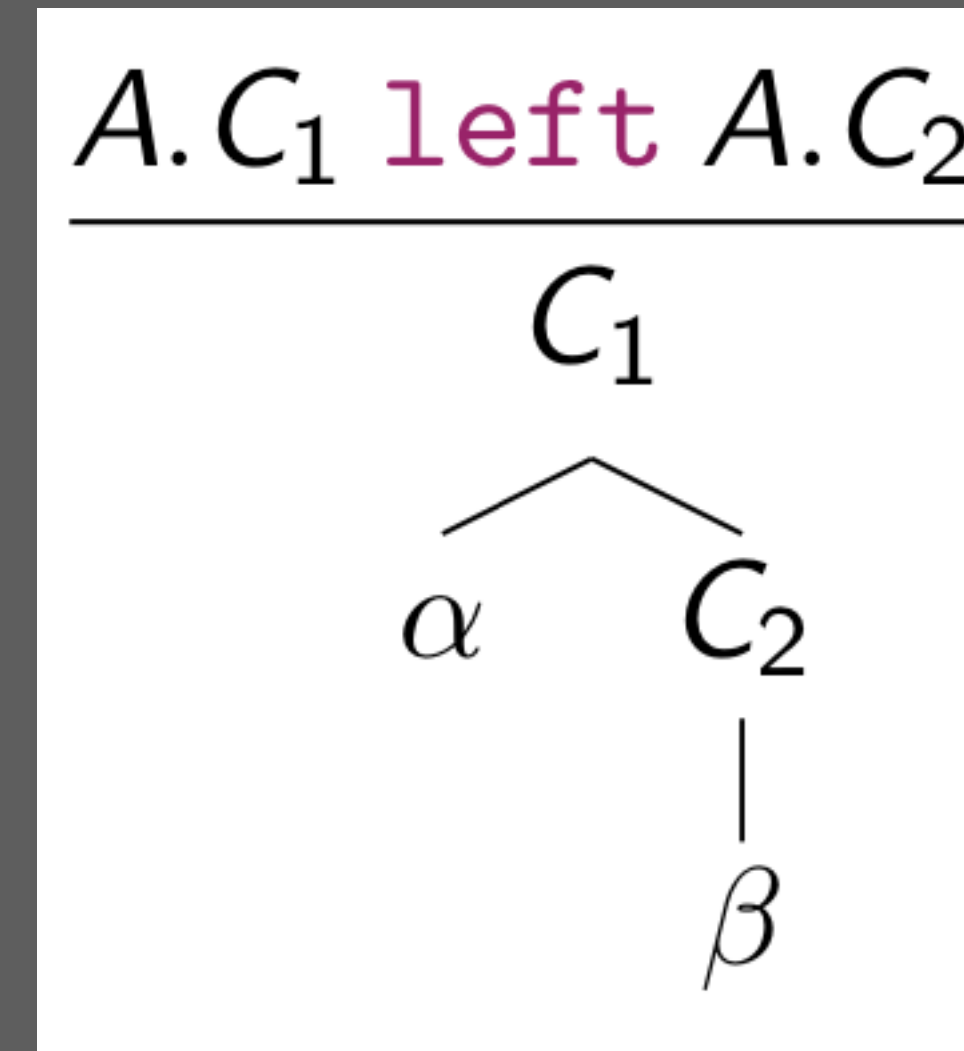
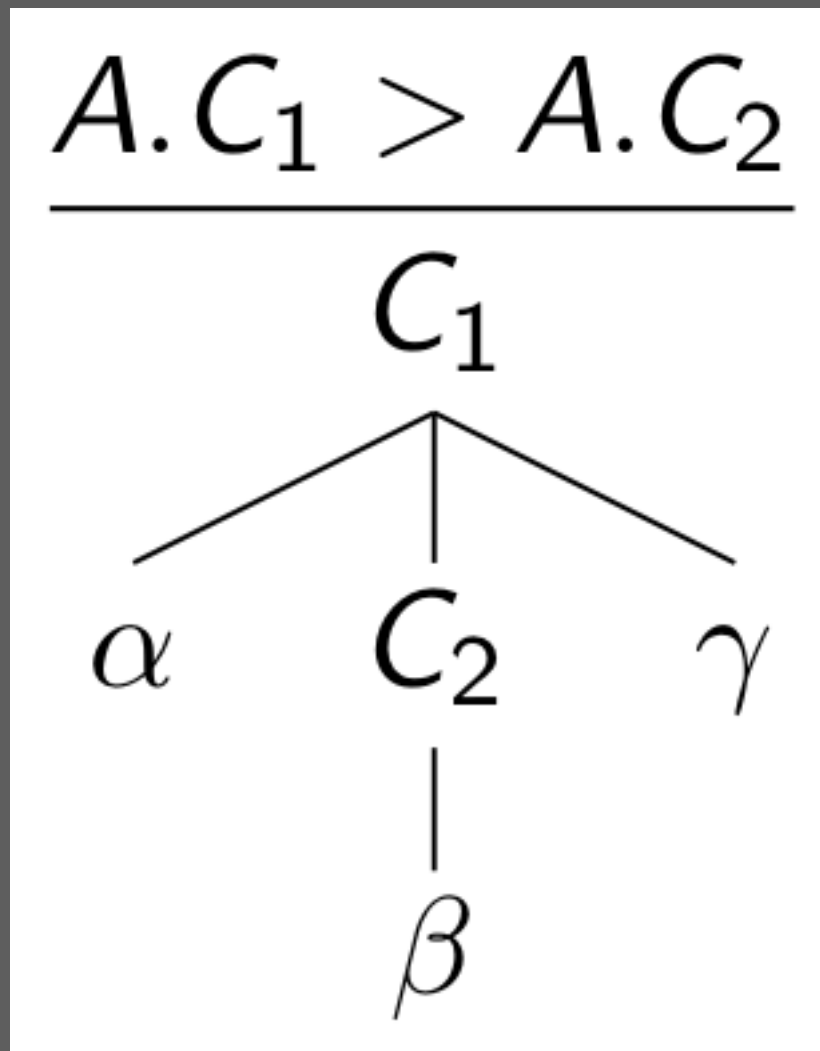
## context-free priorities

Exp.Mul > {left: Exp.Min Exp.Add}



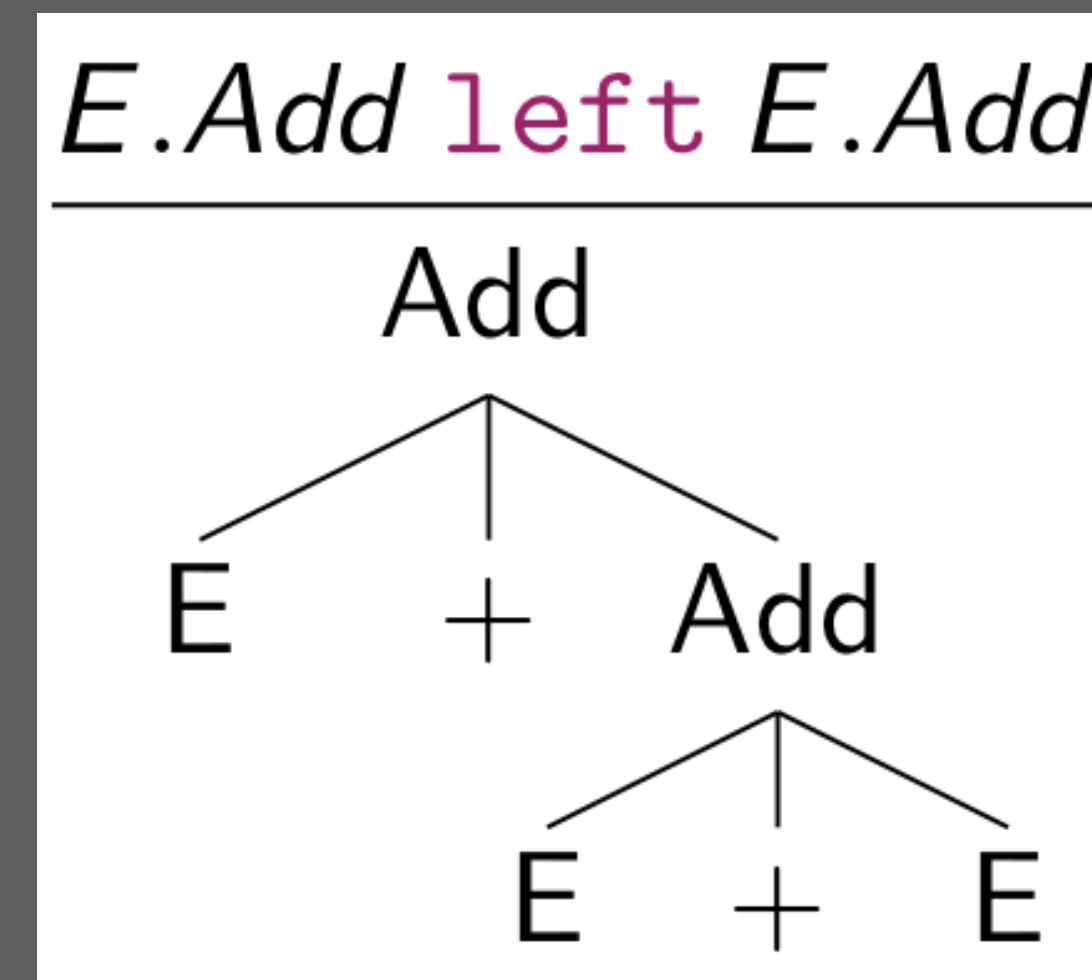
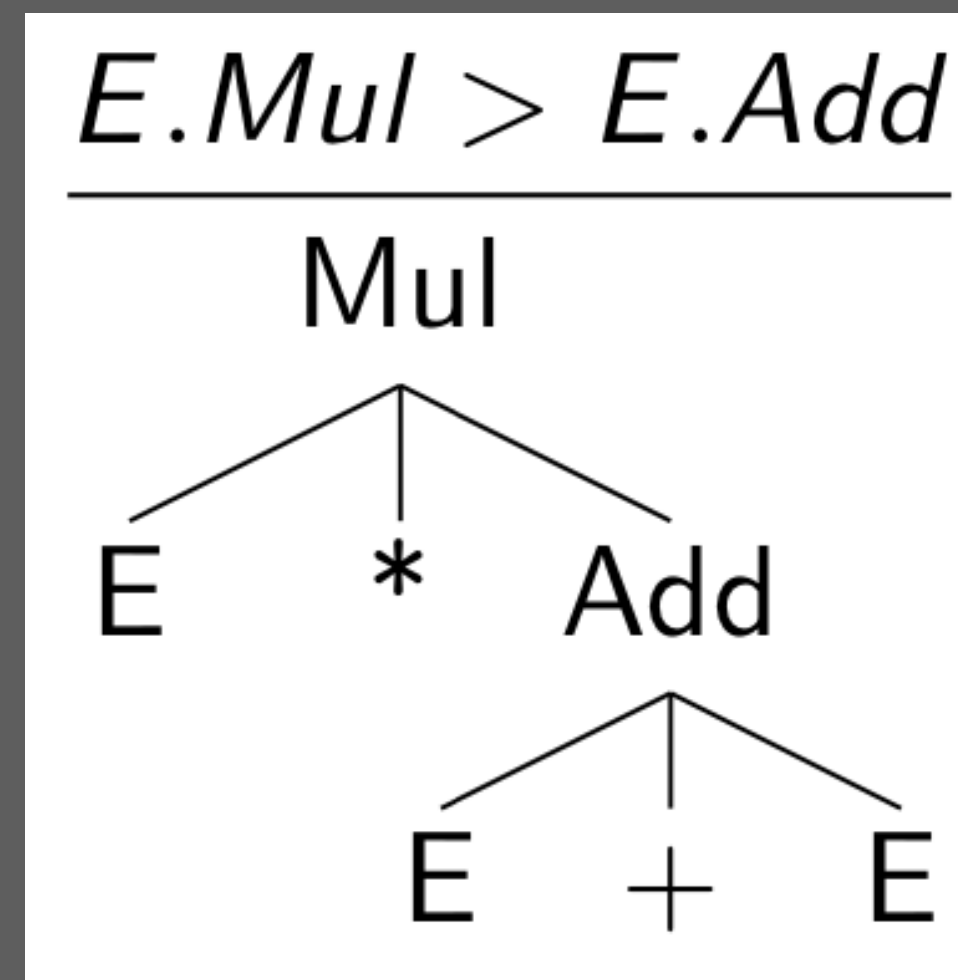
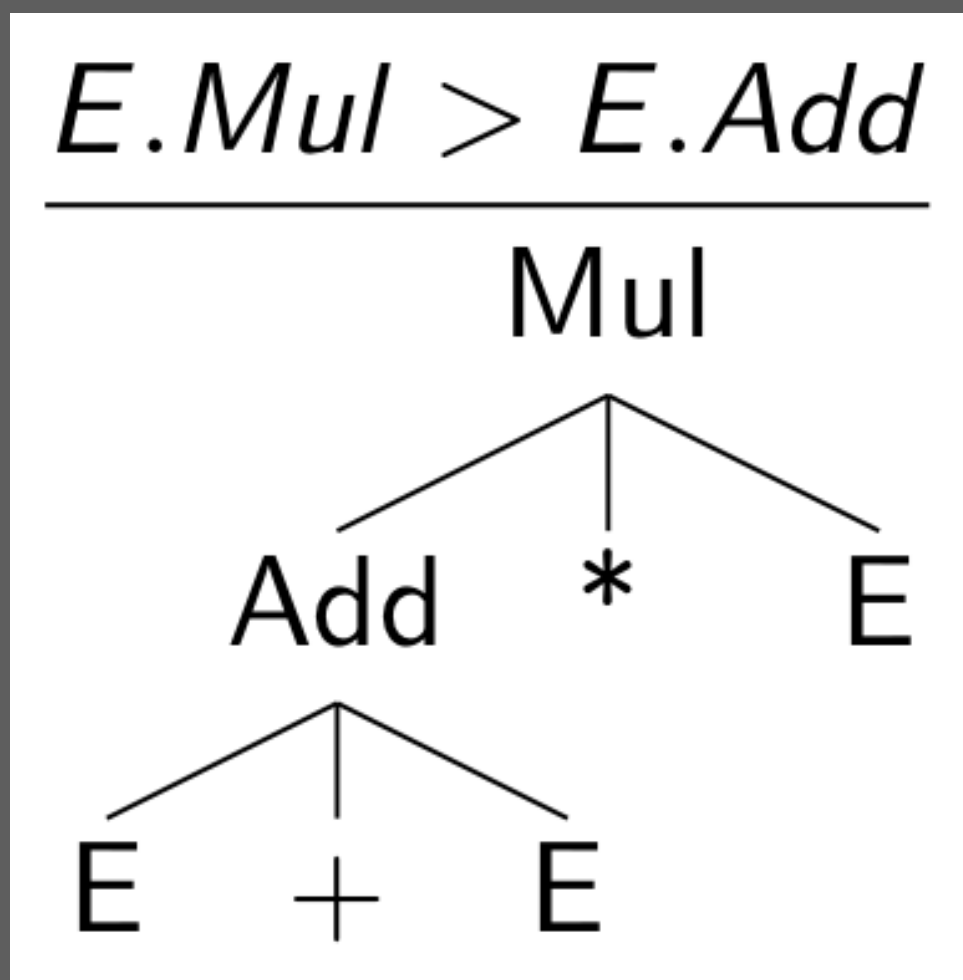
# Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules



Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances



# Disambiguation by Subtree Exclusion

## context-free syntax

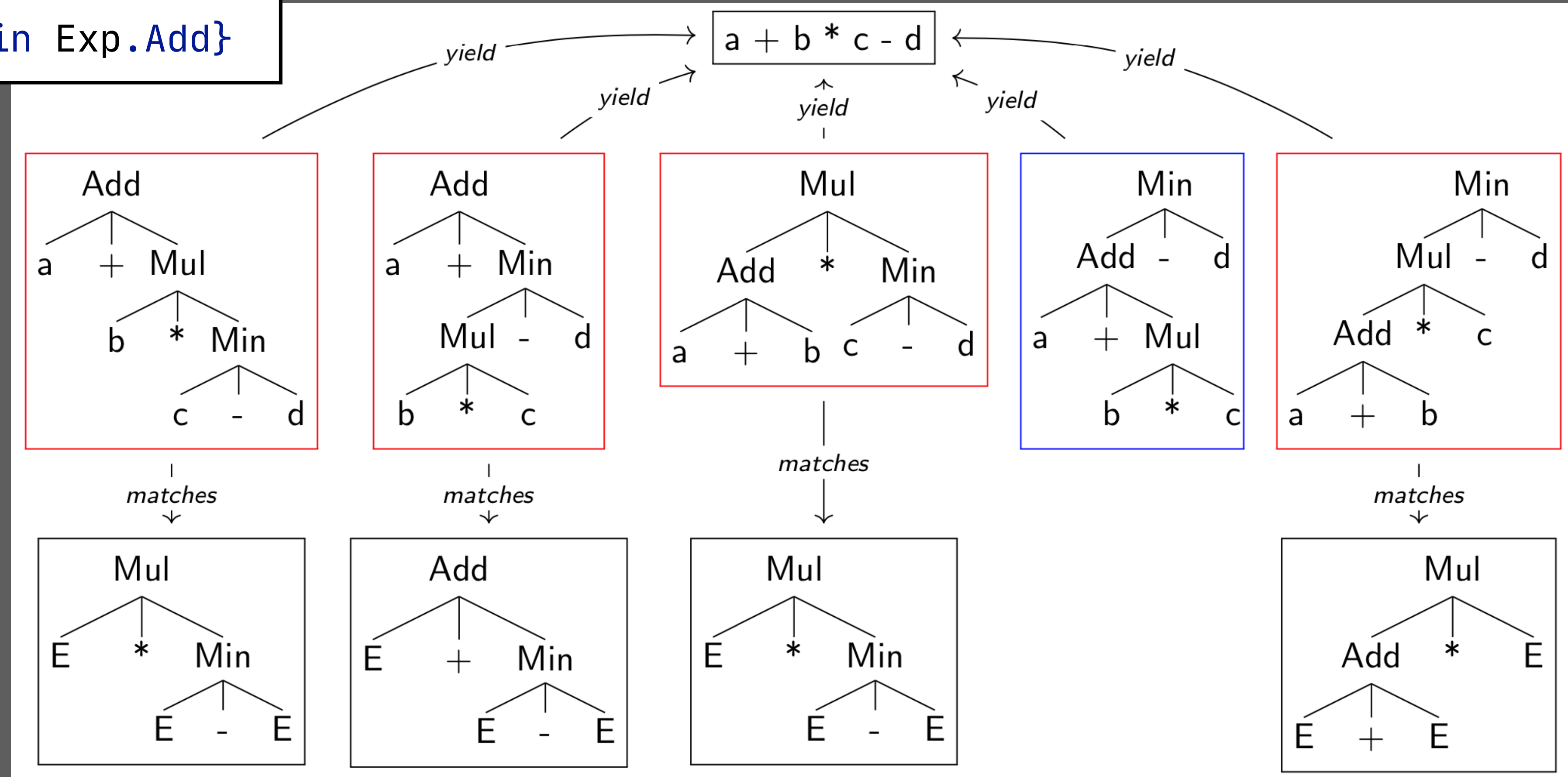
Exp.Min =  $\langle\langle\text{Exp}\rangle - \langle\text{Exp}\rangle\rangle$  {left}

Exp.Add =  $\langle\langle\text{Exp}\rangle + \langle\text{Exp}\rangle\rangle$  {left}

Exp.Mul =  $\langle\langle\text{Exp}\rangle * \langle\text{Exp}\rangle\rangle$  {left}

## context-free priorities

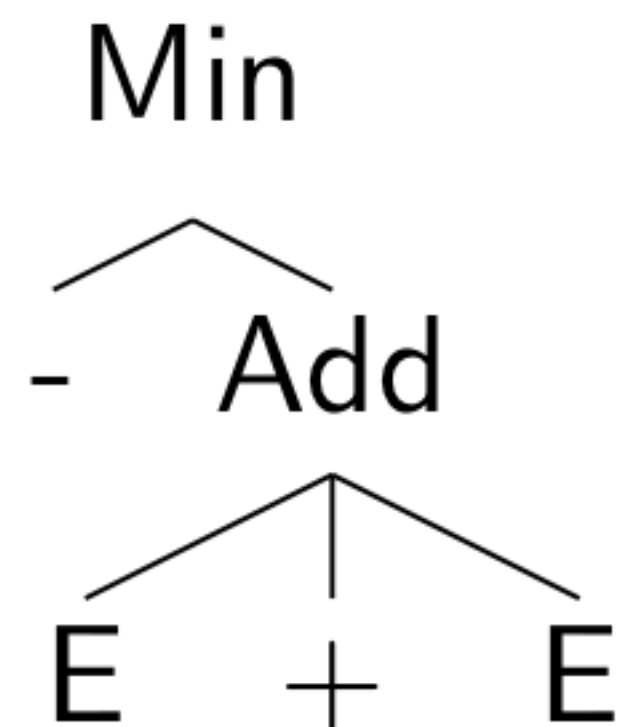
Exp.Mul > {left: Exp.Min Exp.Add}



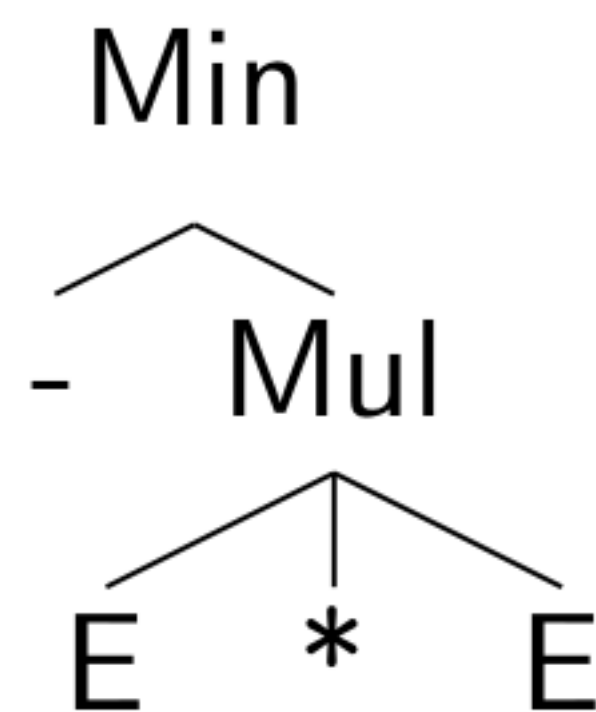
# Safe for High Priority Prefix Operators

Conflict  
Patterns

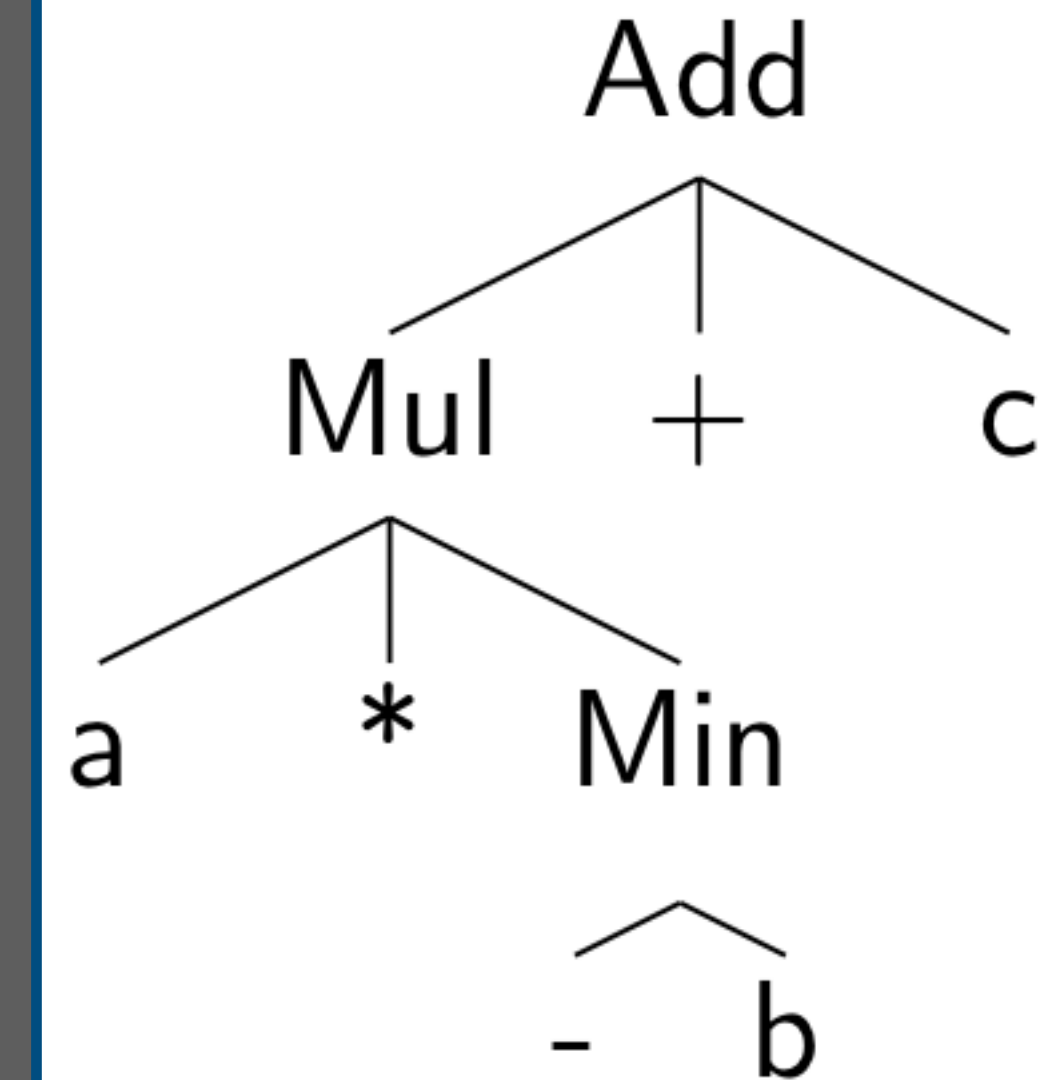
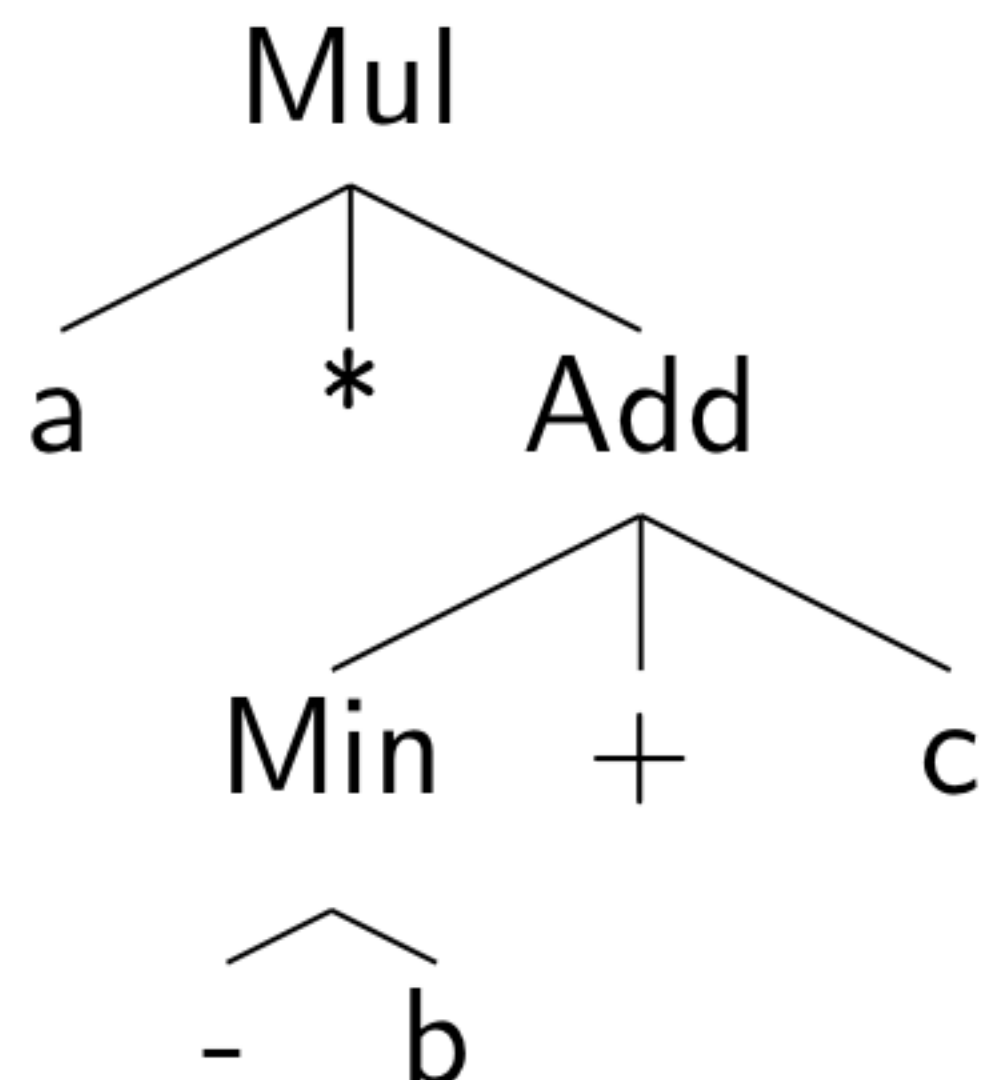
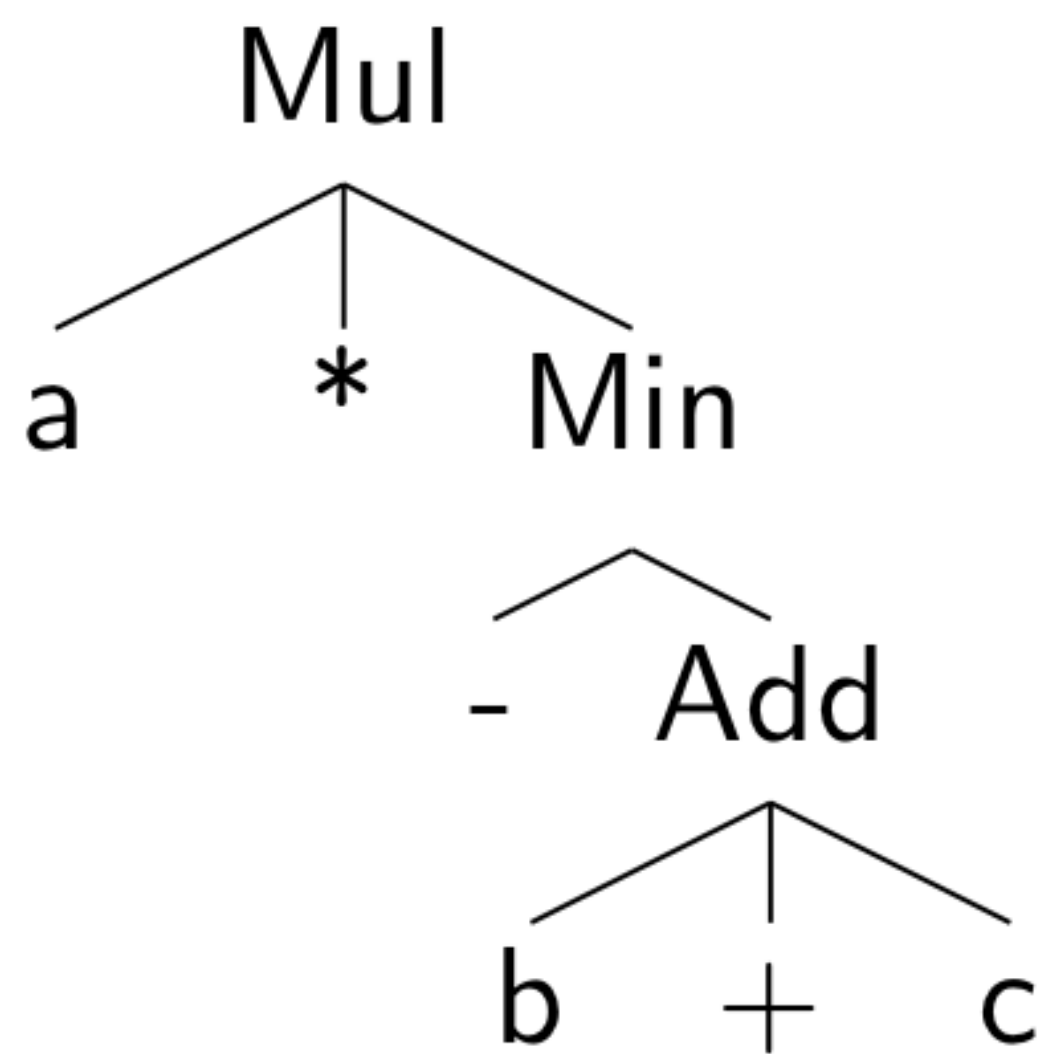
$$\frac{E.Min > E.Add}{}$$



$$\frac{E.Min > E.Mul}{}$$

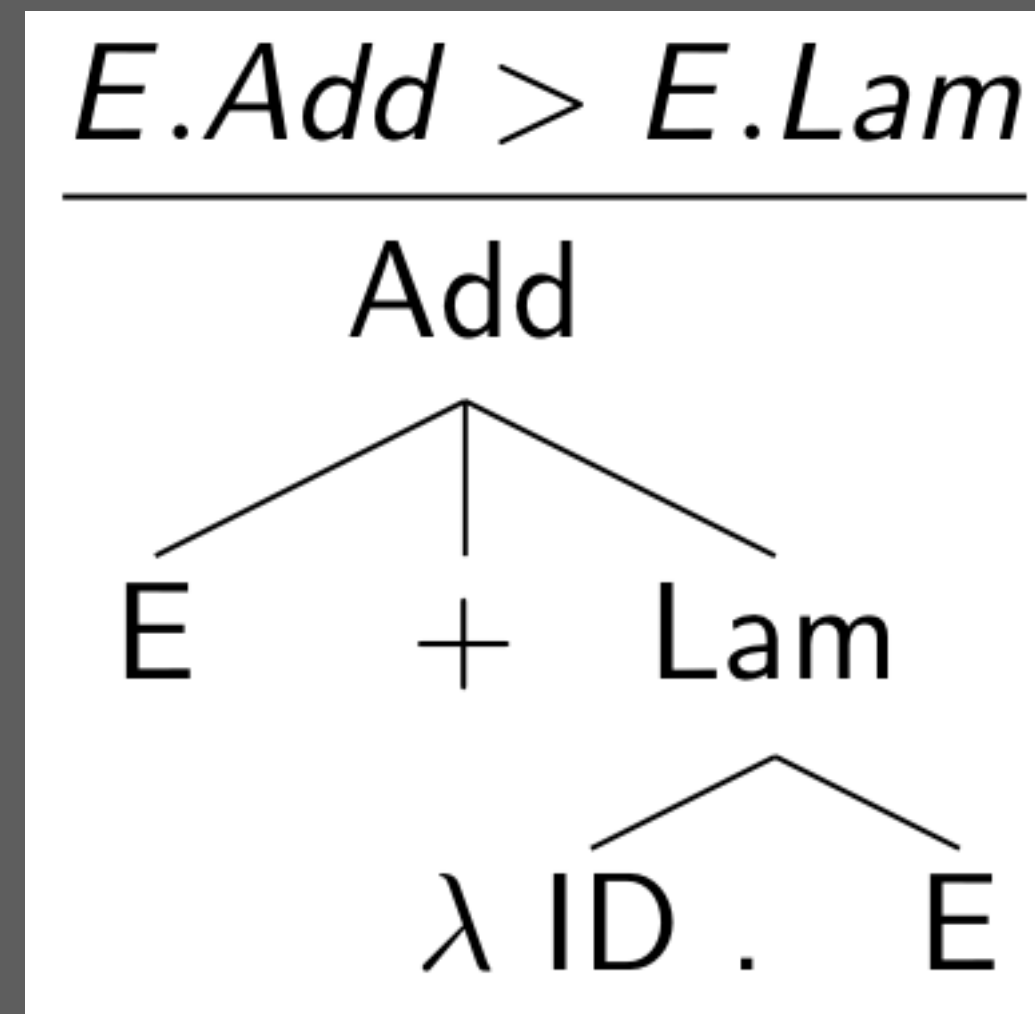
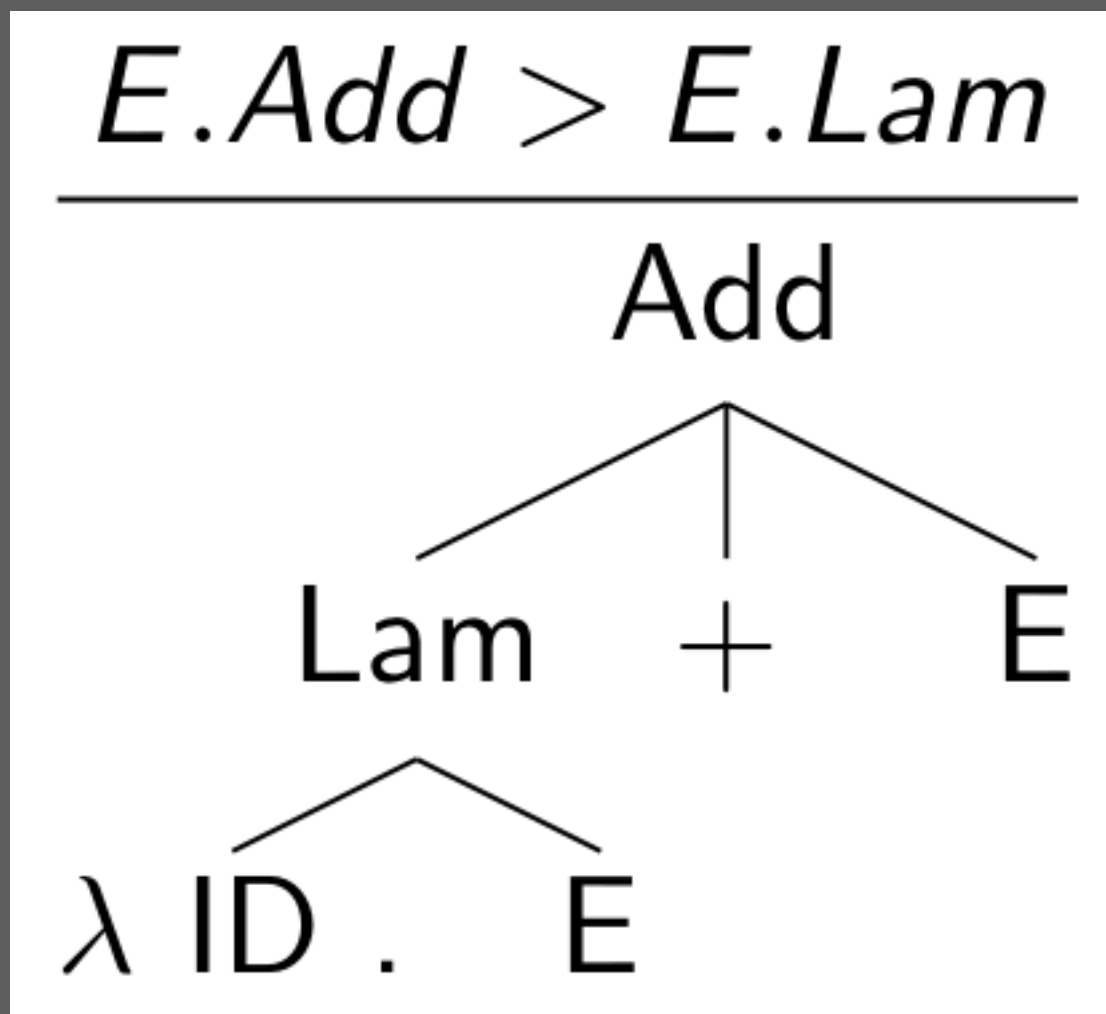


Trees



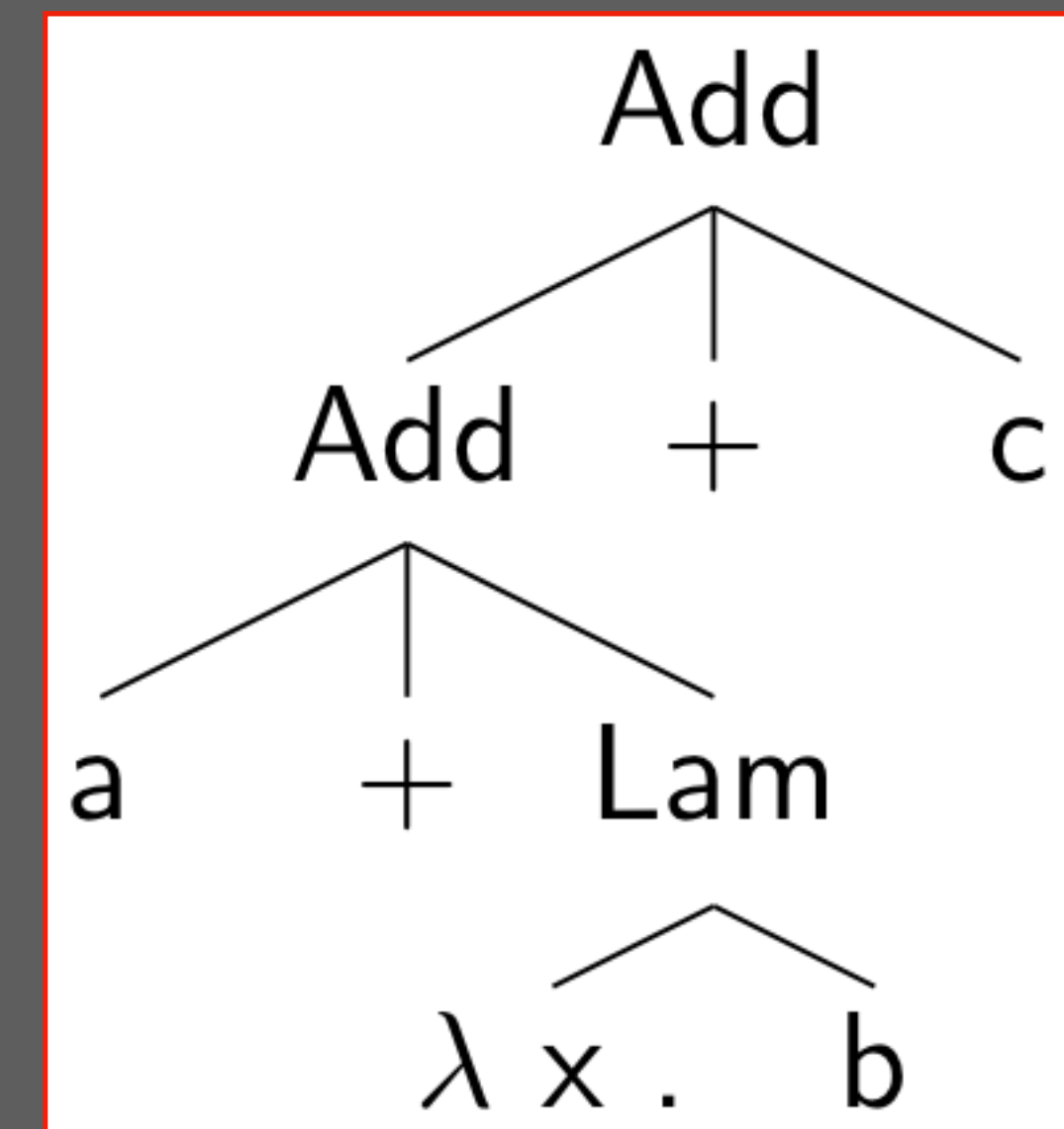
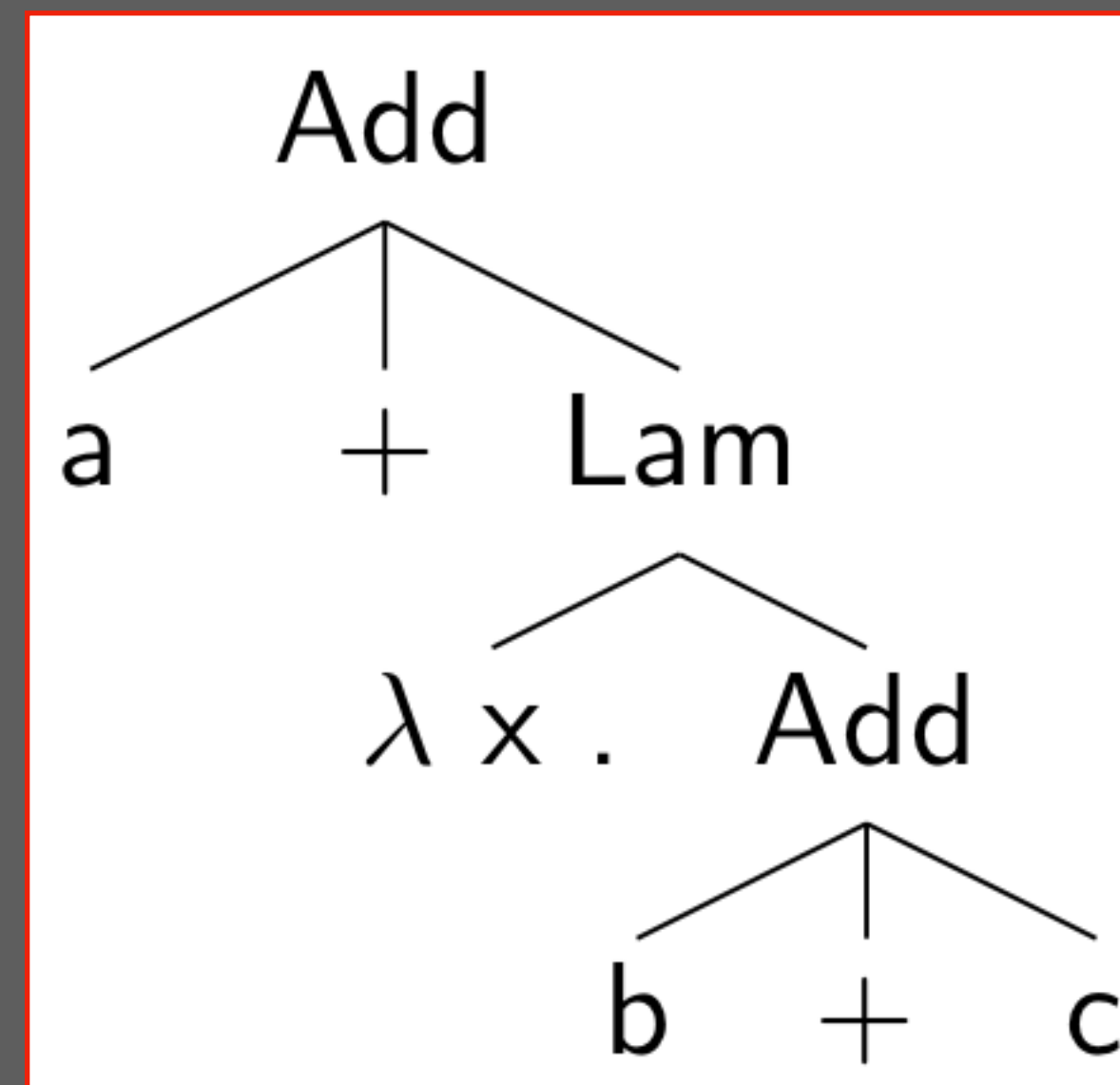
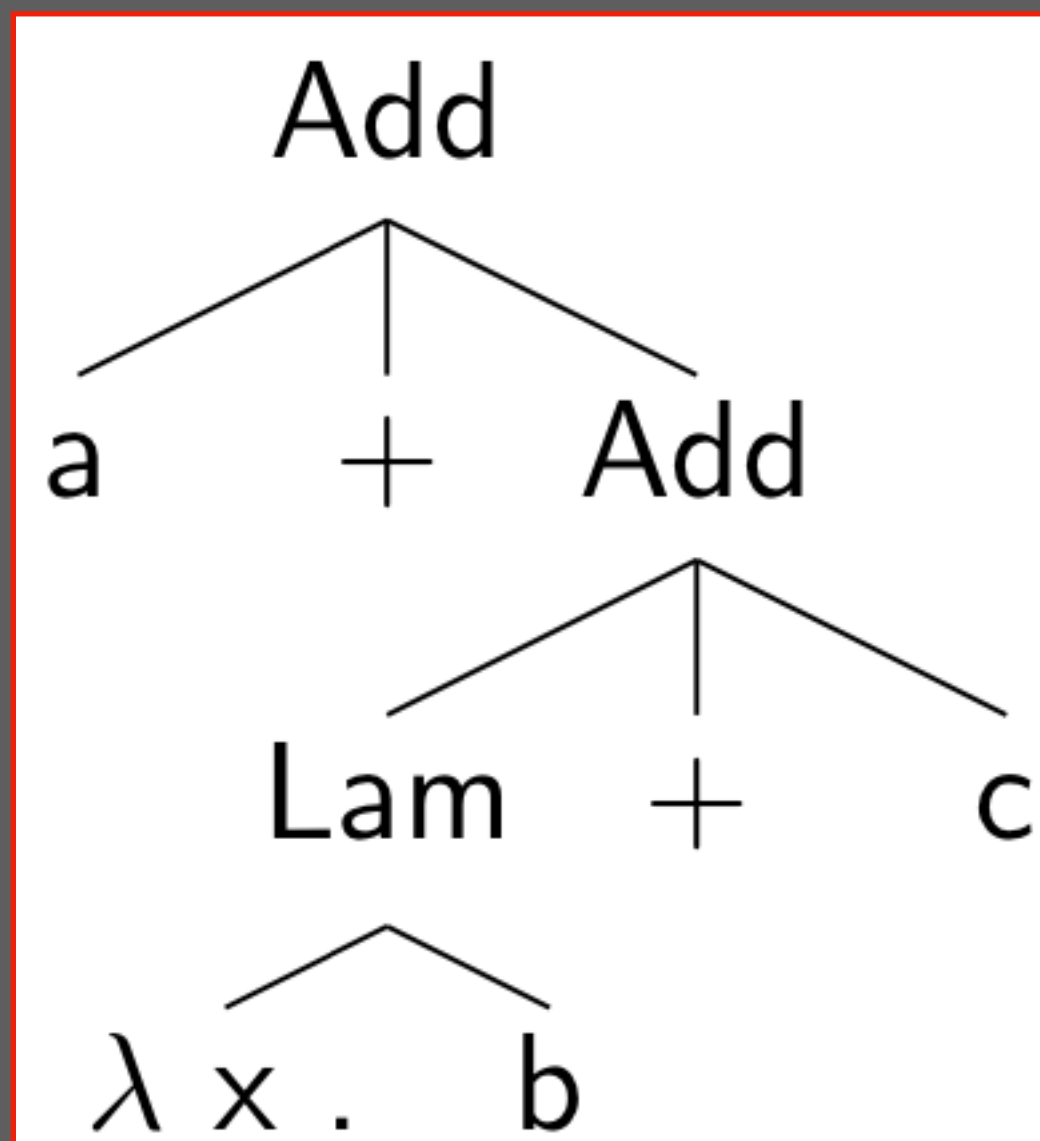
# Unsafe for Low Priority Prefix Operators [SDF2]

Conflict  
Patterns



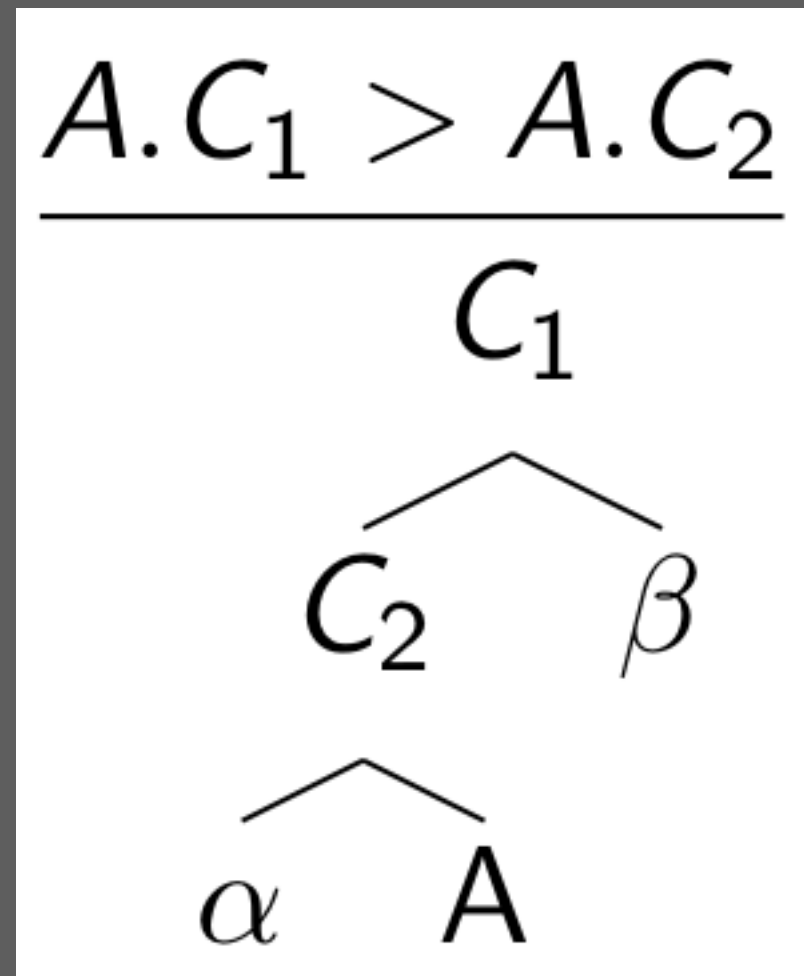
Afroozeh, van den Brand,  
Johnstone, Scott, Vinju: Safe  
specification of operator  
precedence rules. SLE 2013

Trees

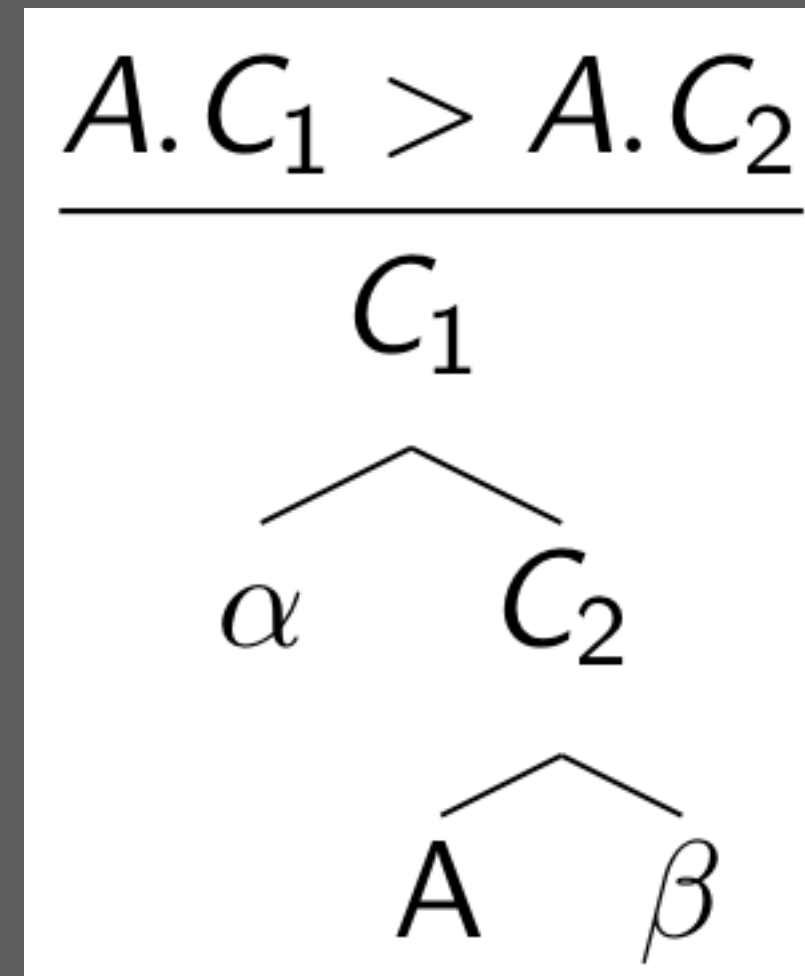


# Safe Subtree Exclusion Rules [SDF3 (2019)]

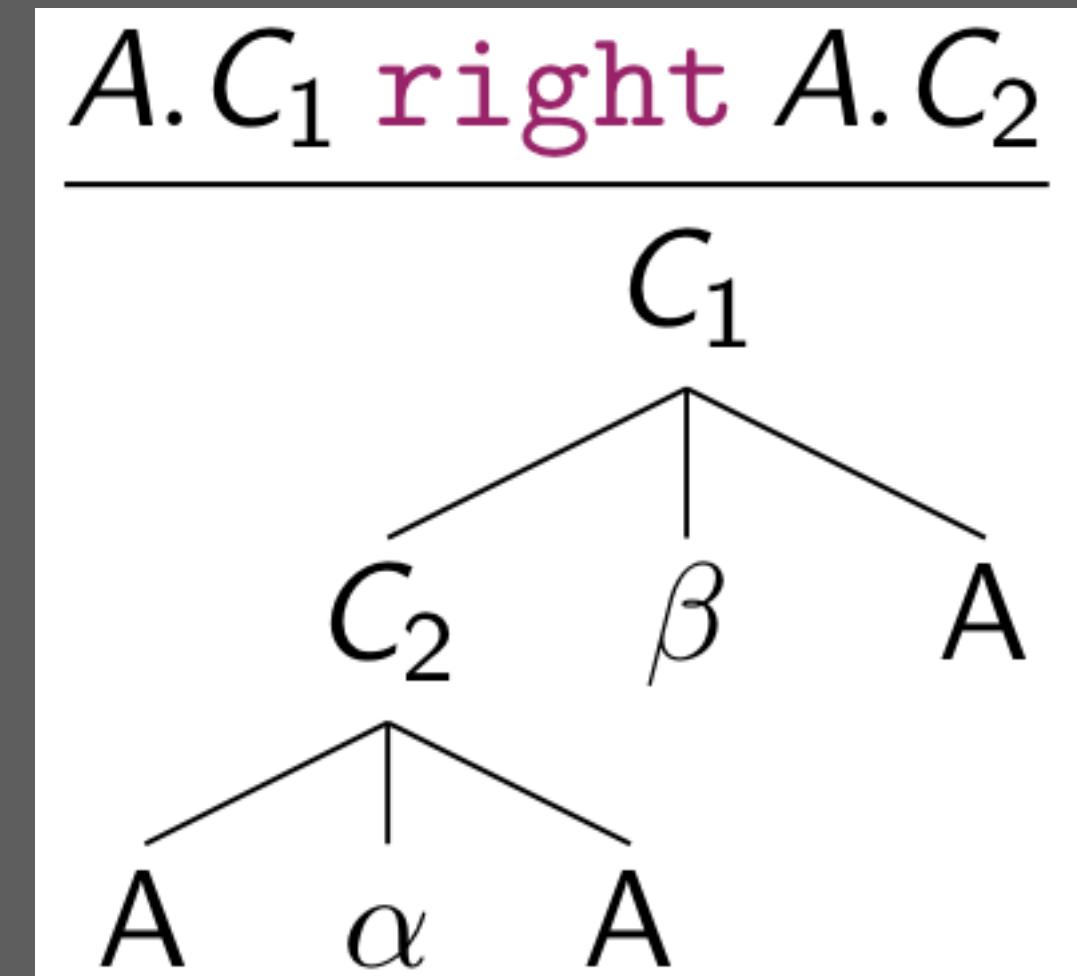
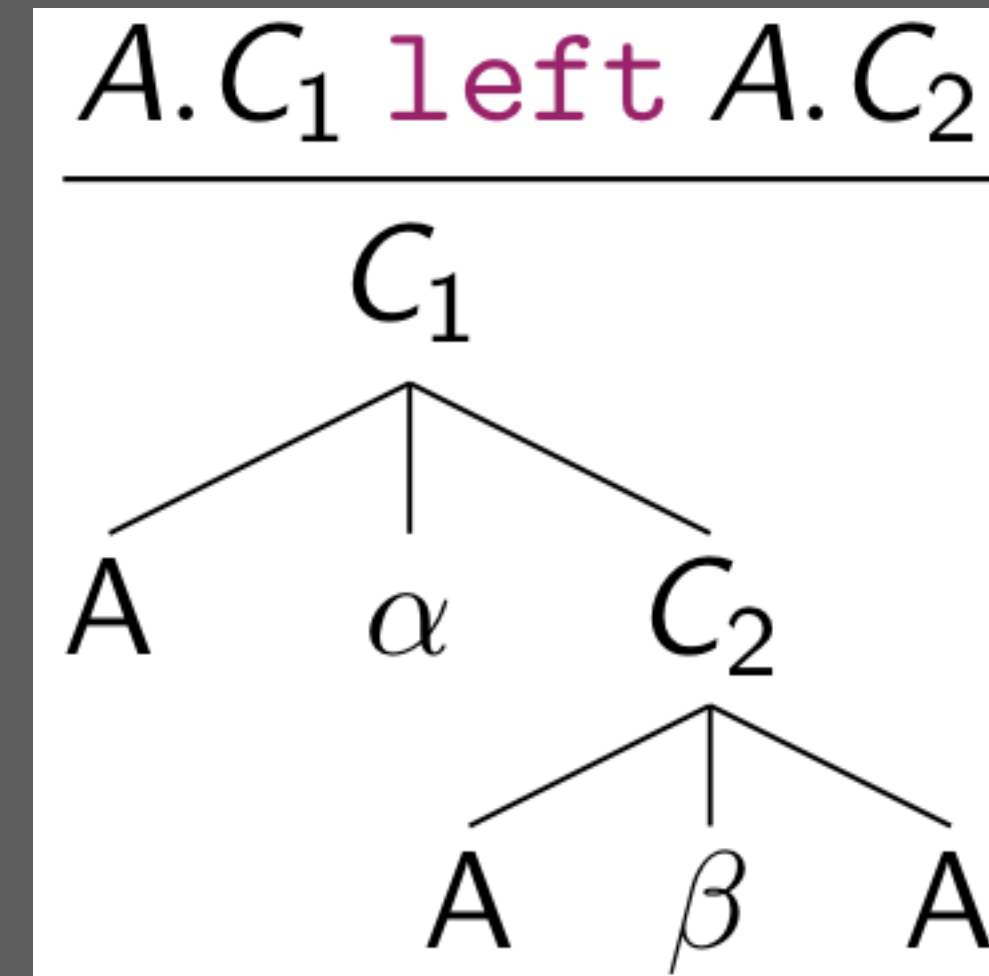
## Rules



Right Recursive in  
Left Recursive Position

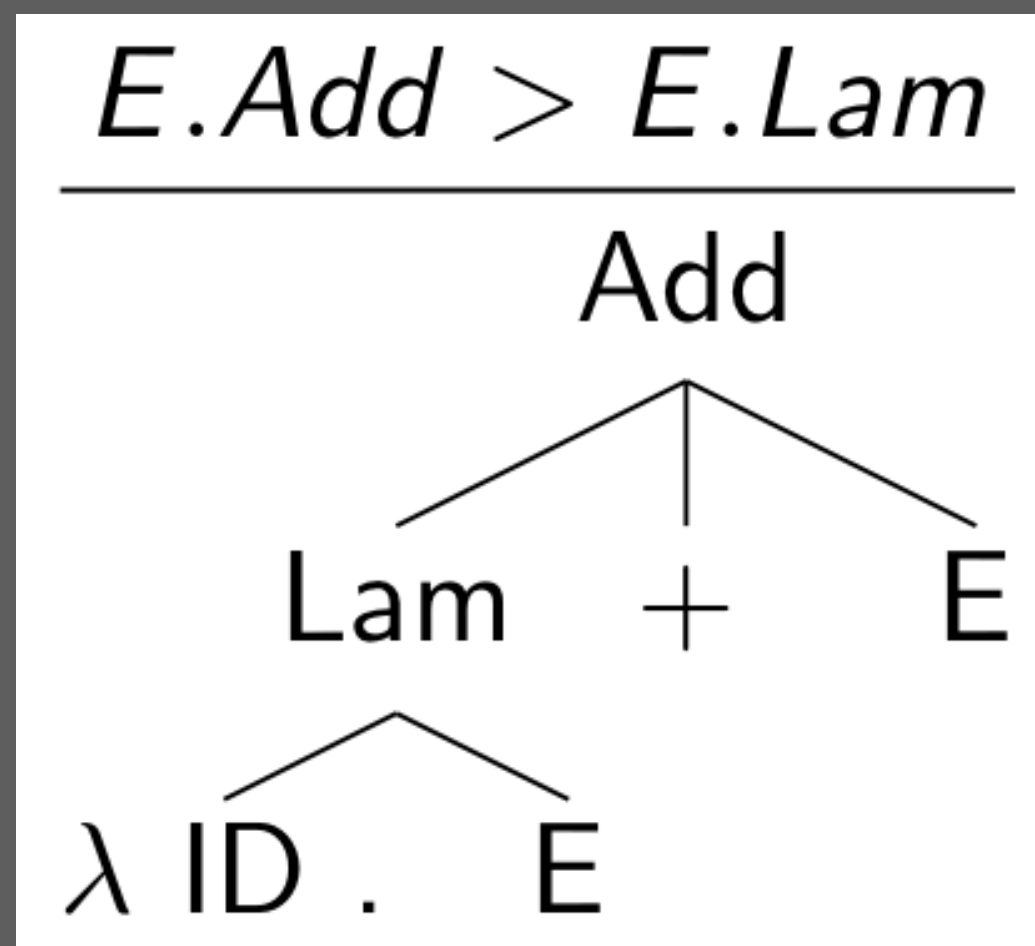


Left Recursive in  
Right Recursive Position

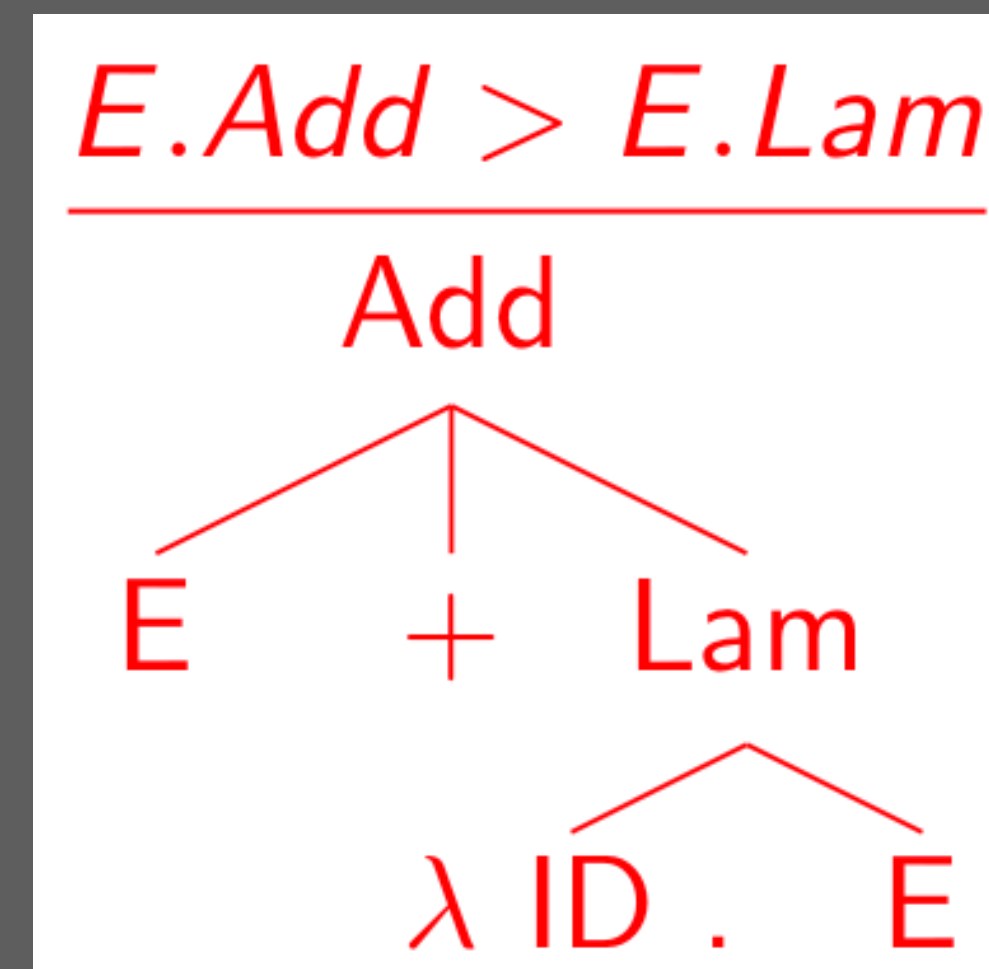


Associativity

## Conflict Patterns



conflict pattern:  
\ right recursive

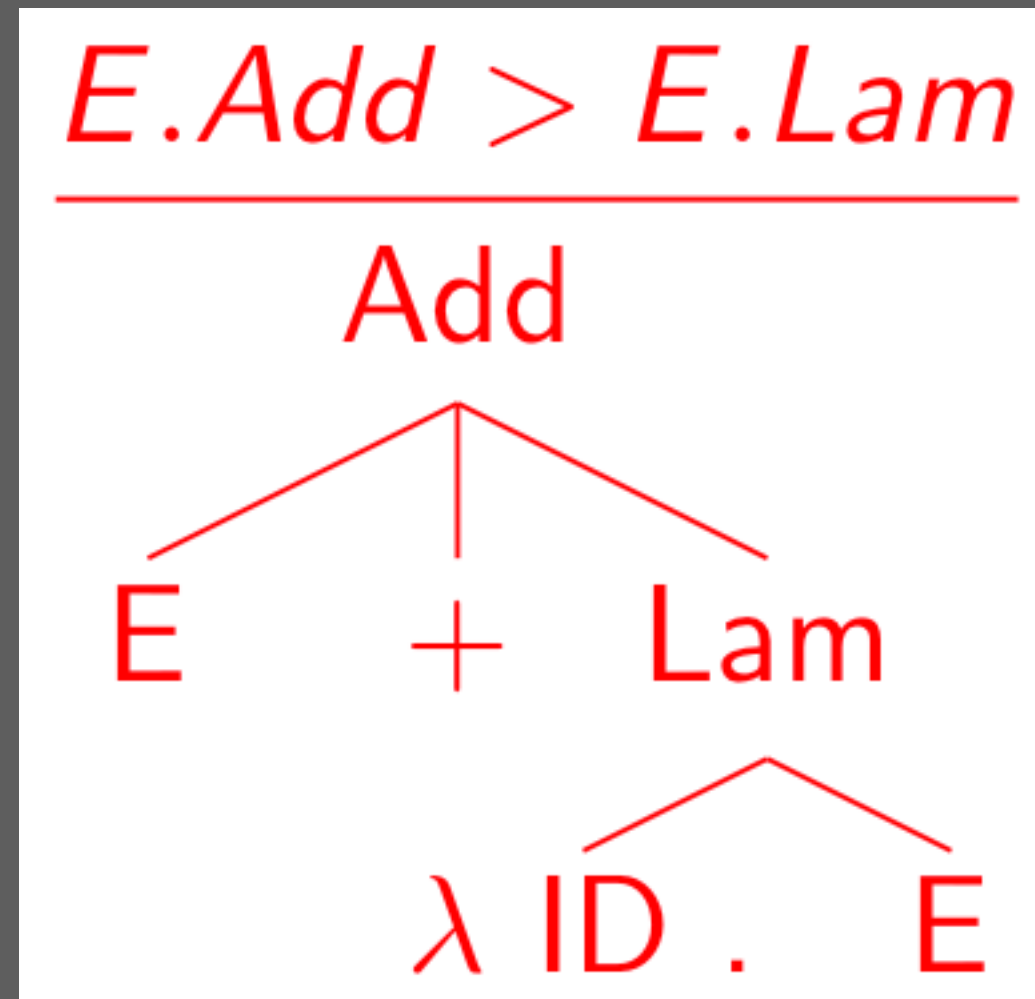
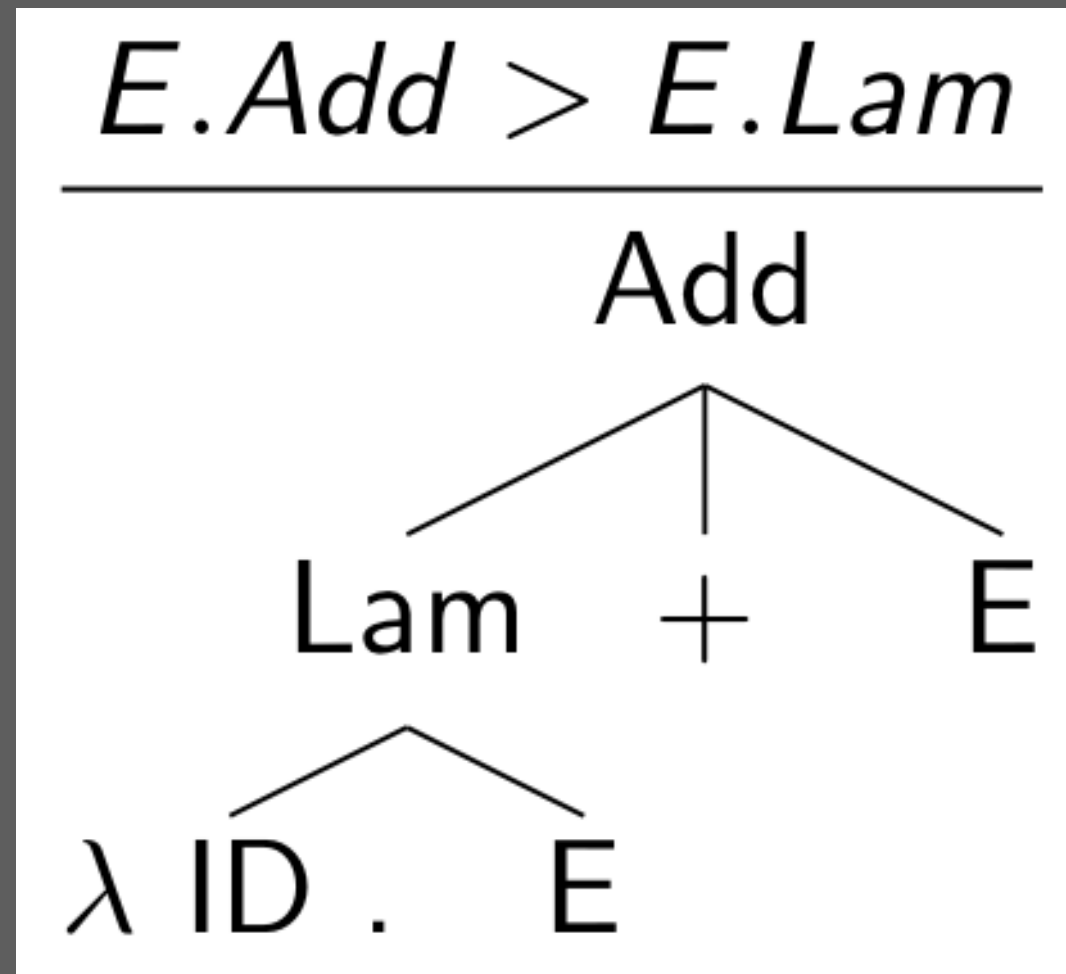


Amorim, Visser: A direct  
semantics of declarative  
disambiguation rules.  
(Under revision)

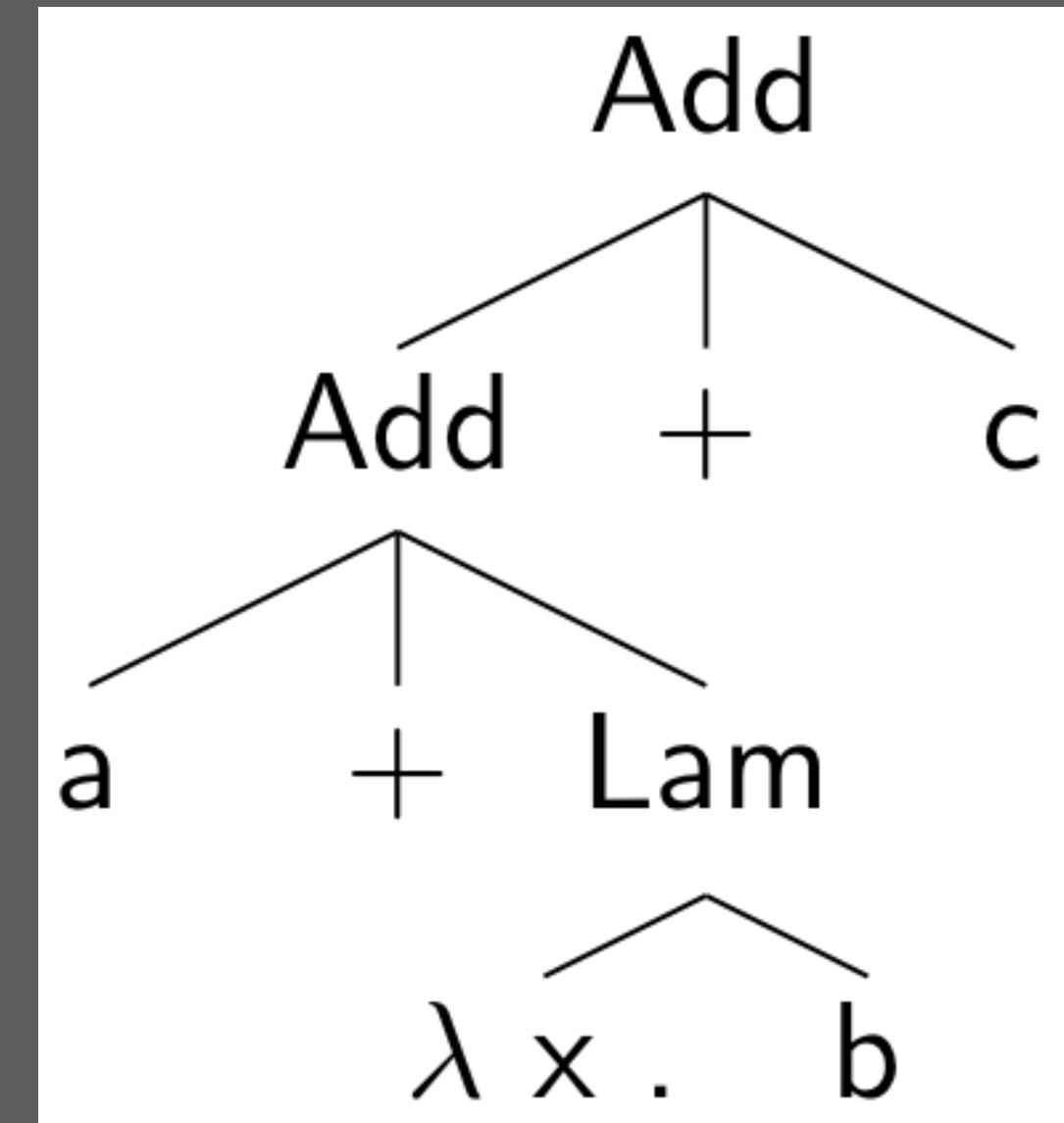
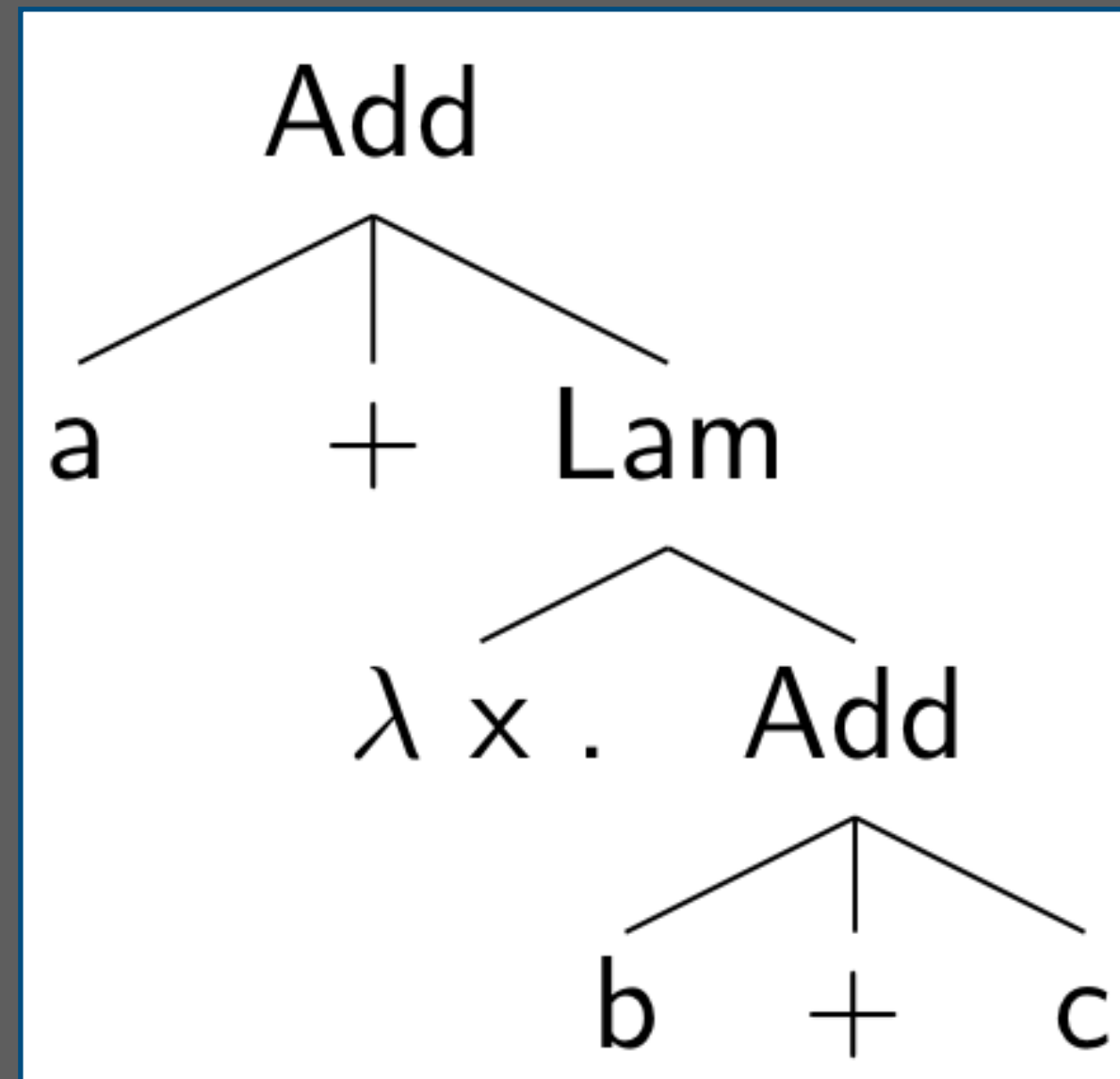
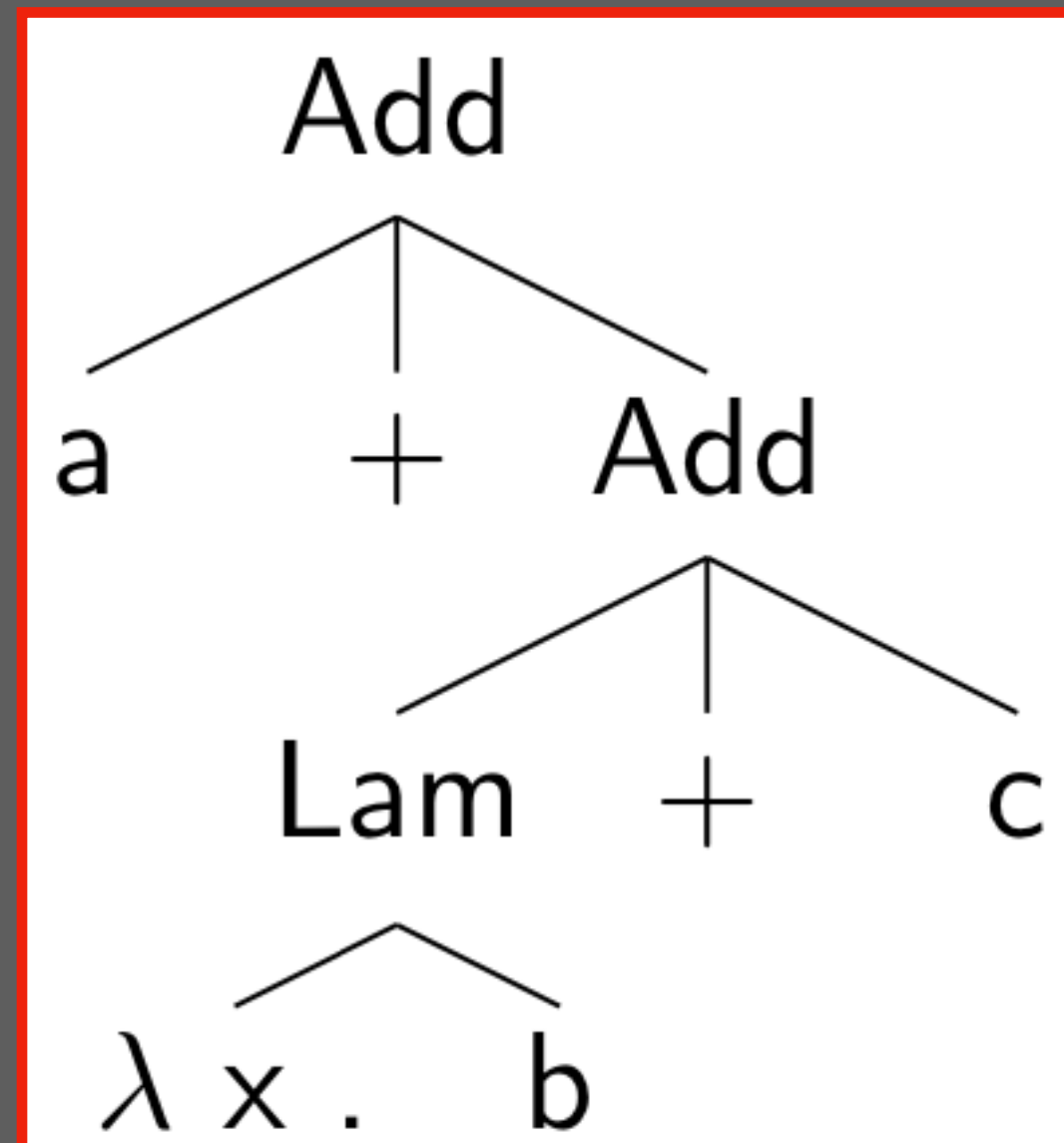
not a conflict pattern:  
\ not left recursive

# Shallow Interpretation: Safe for Low Priority Prefix Operators

Conflict  
Patterns

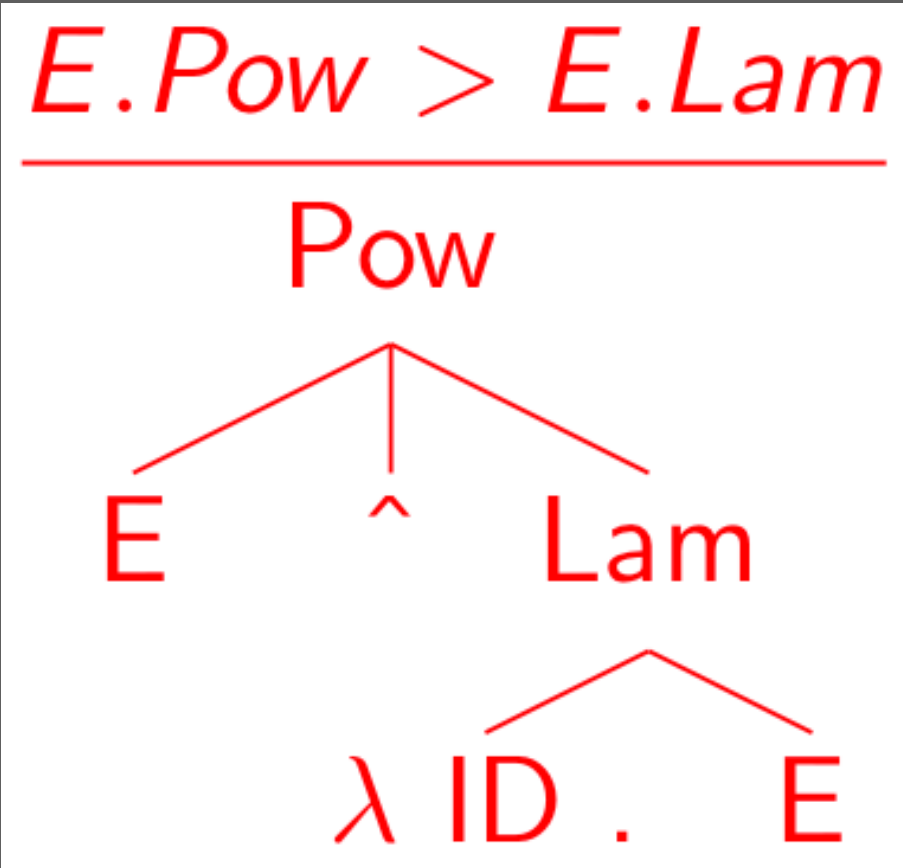
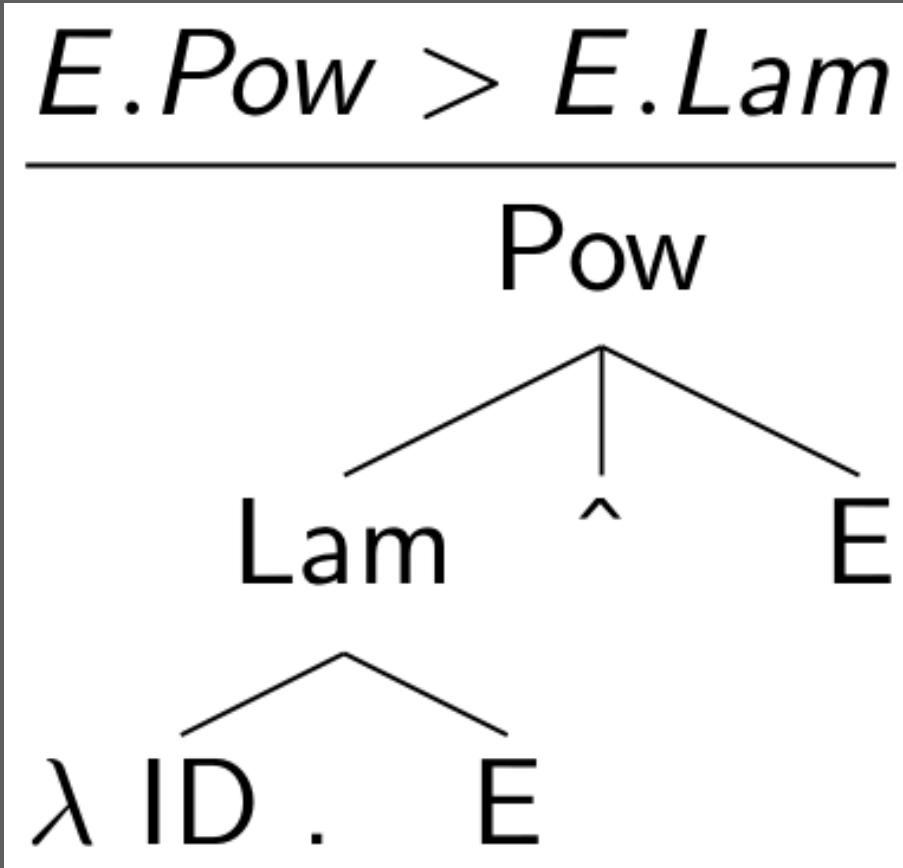
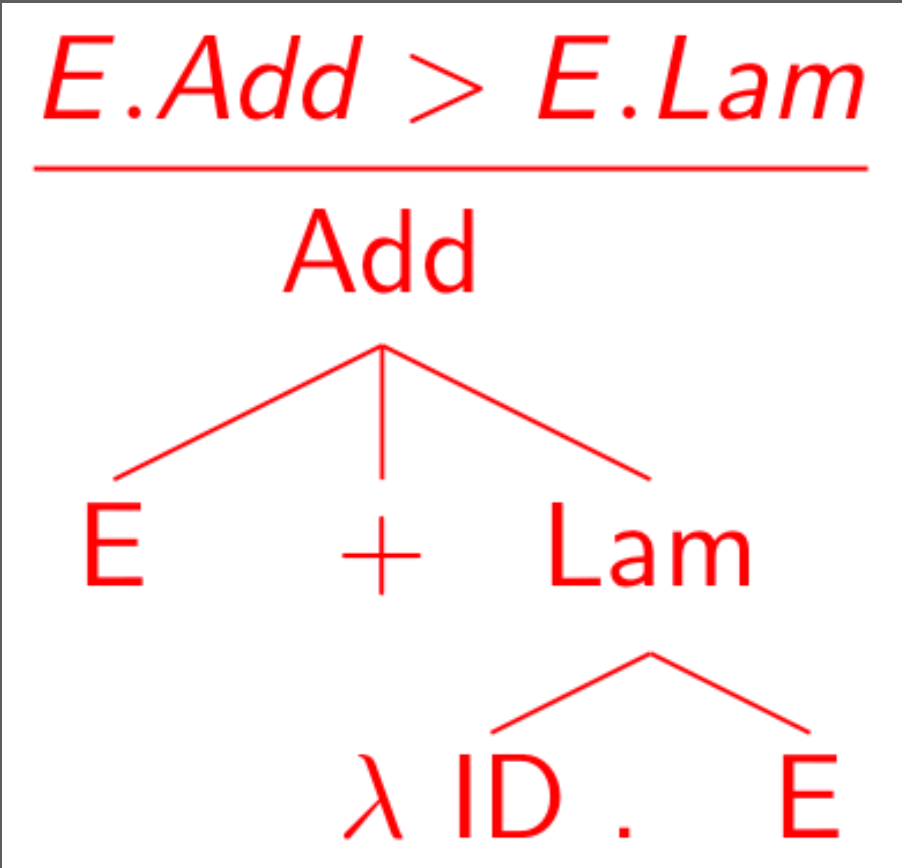
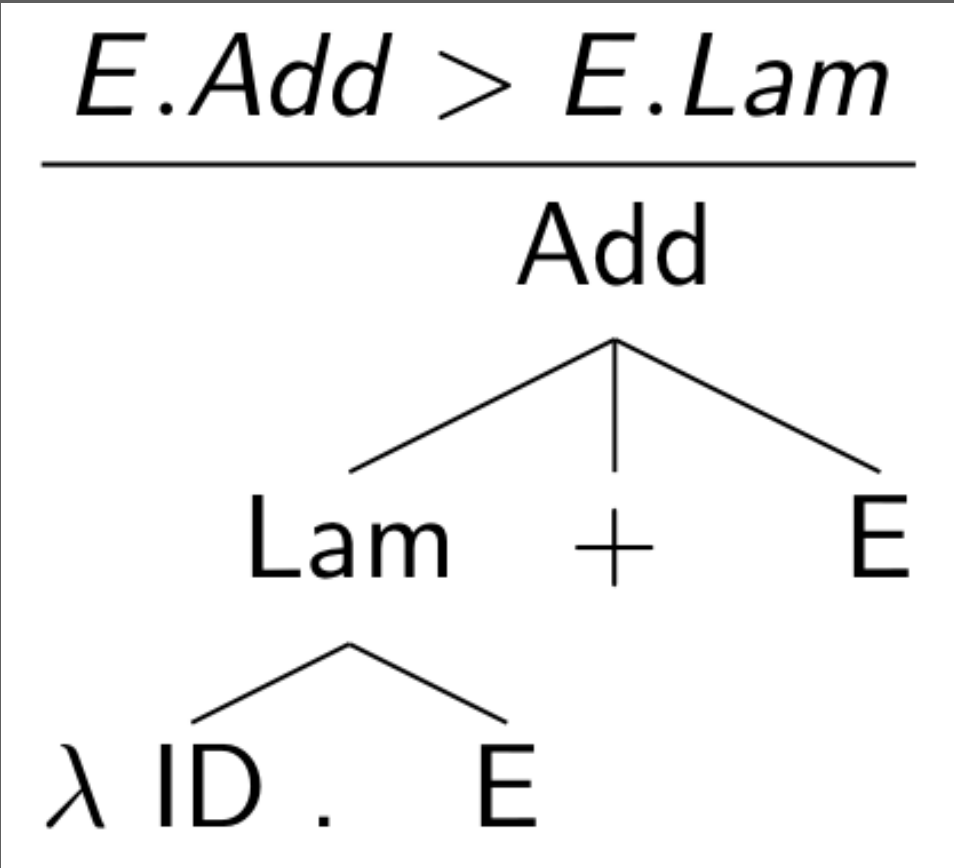


Trees



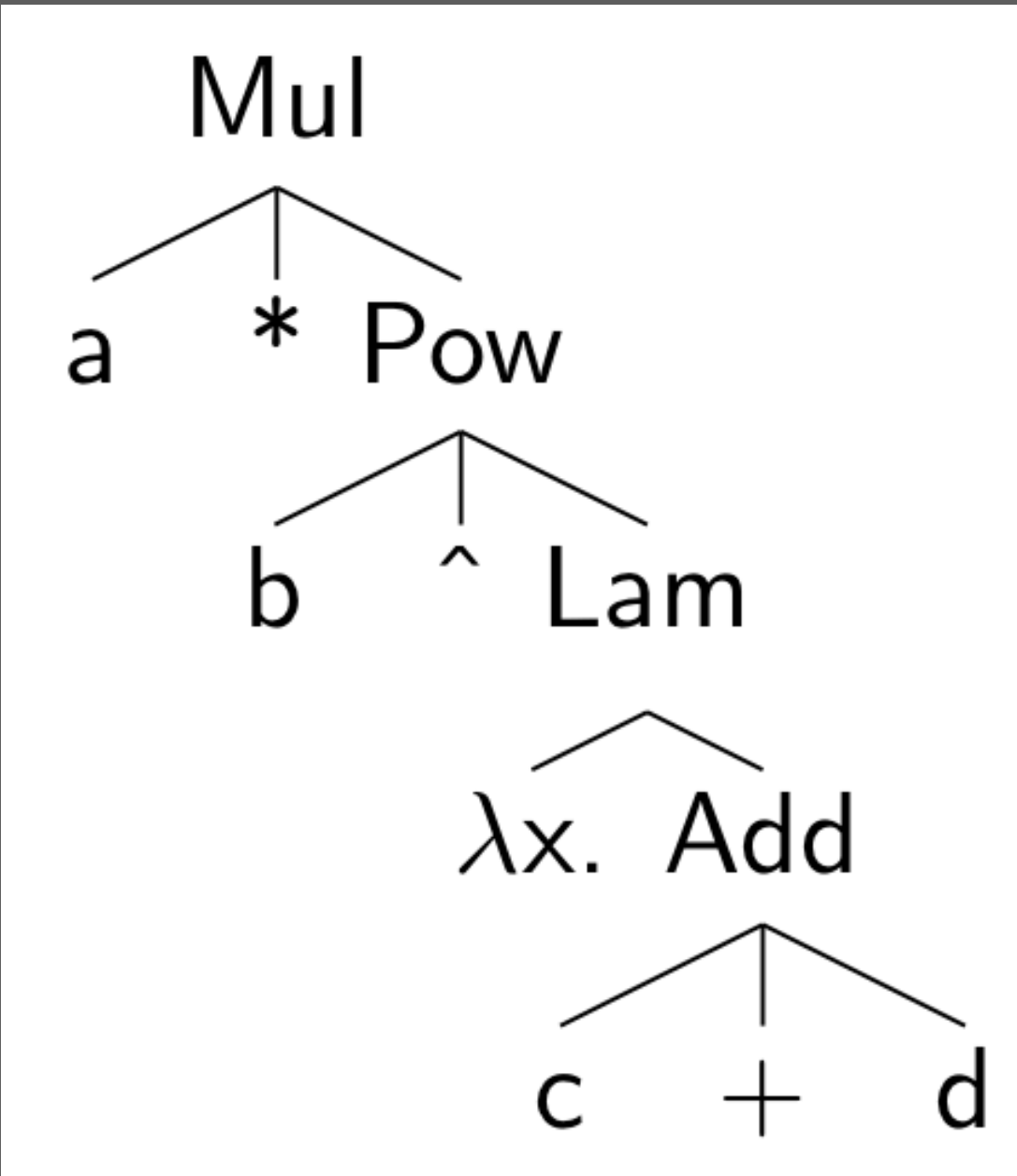
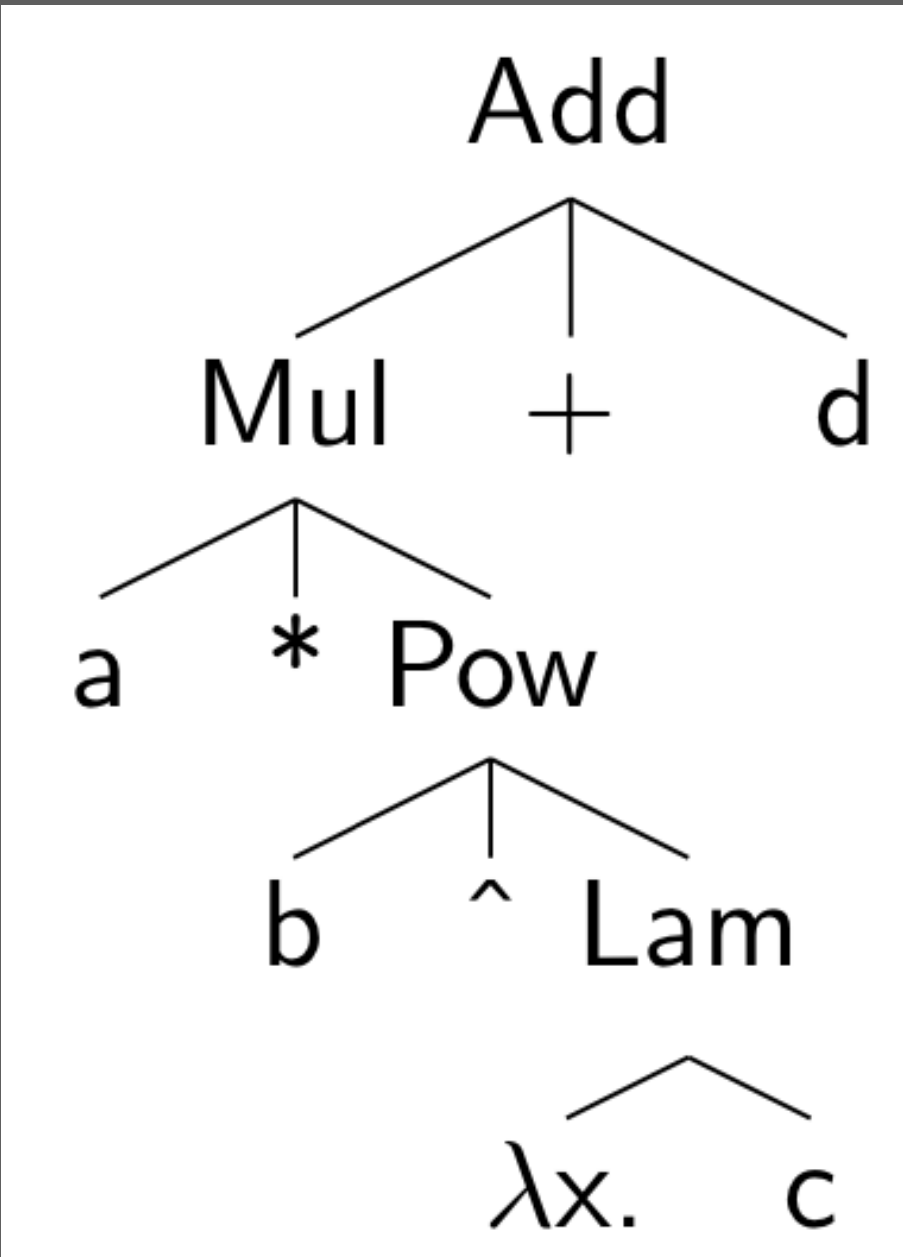
# Shallow Interpretation: Incomplete for Low Priority Prefix Operators

Conflict  
Patterns

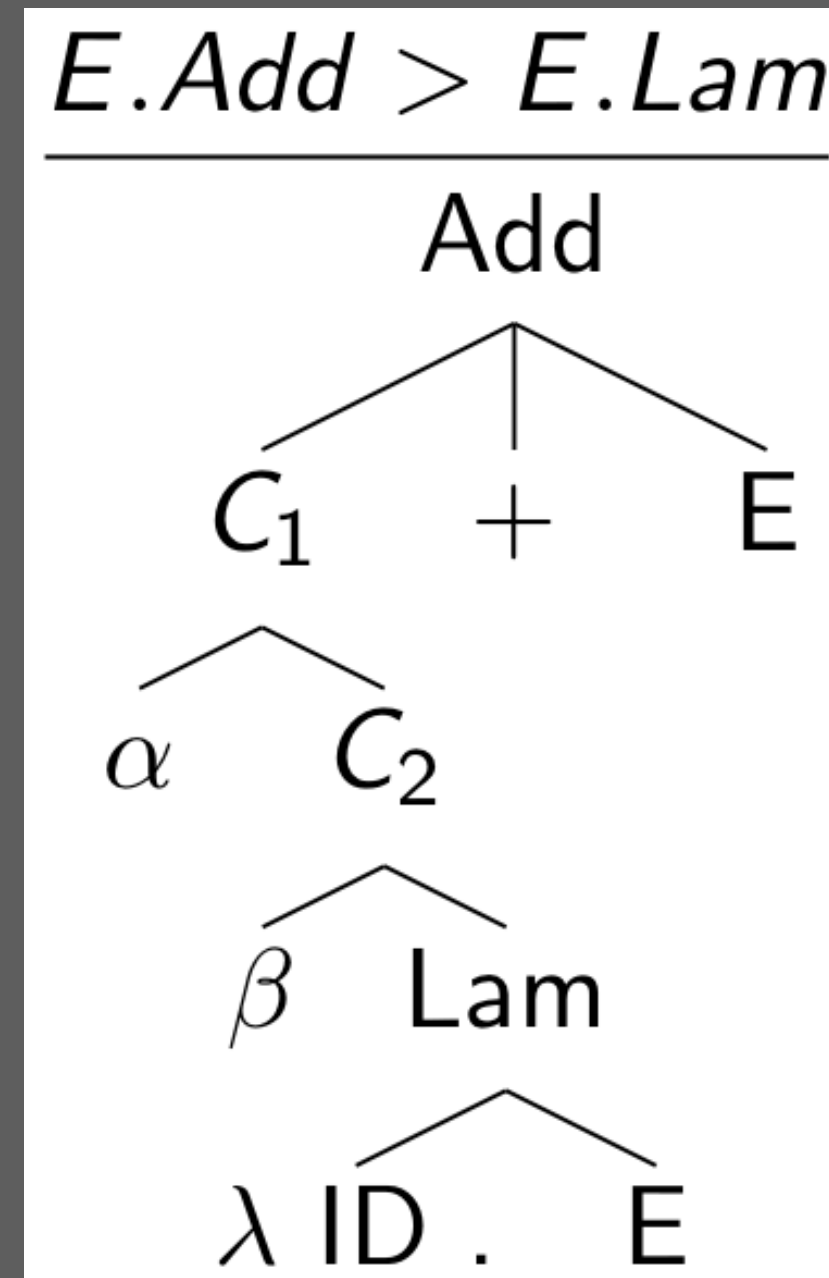
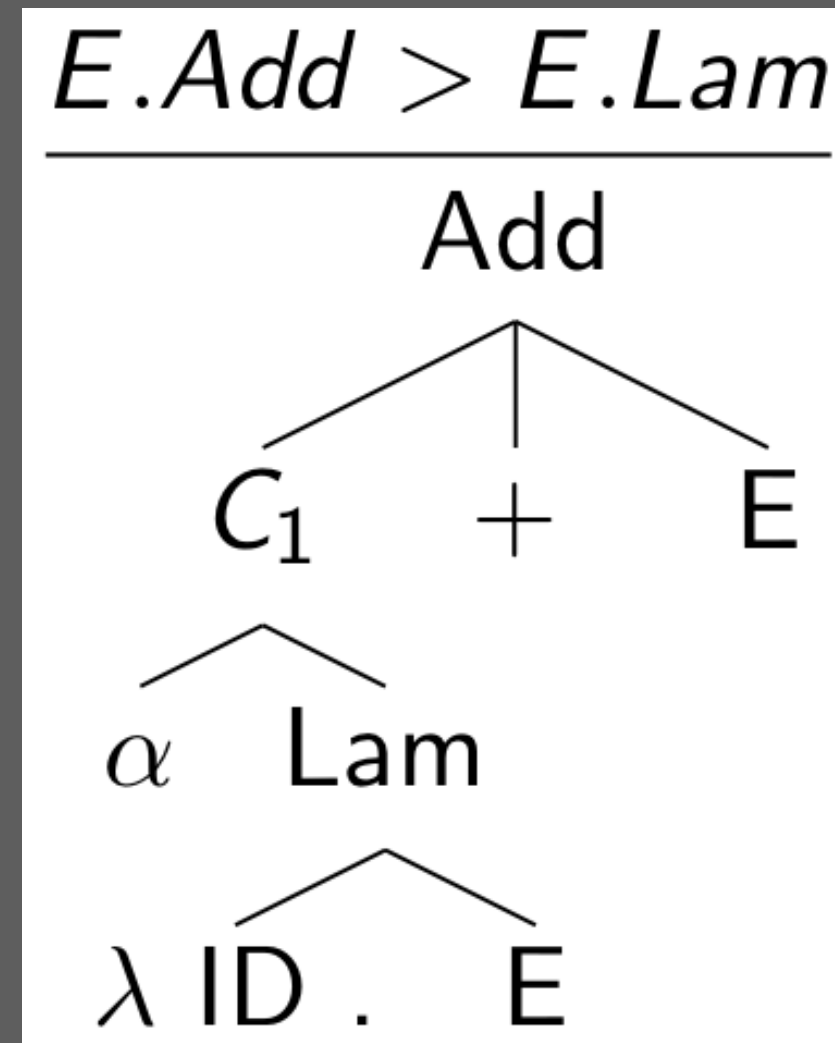
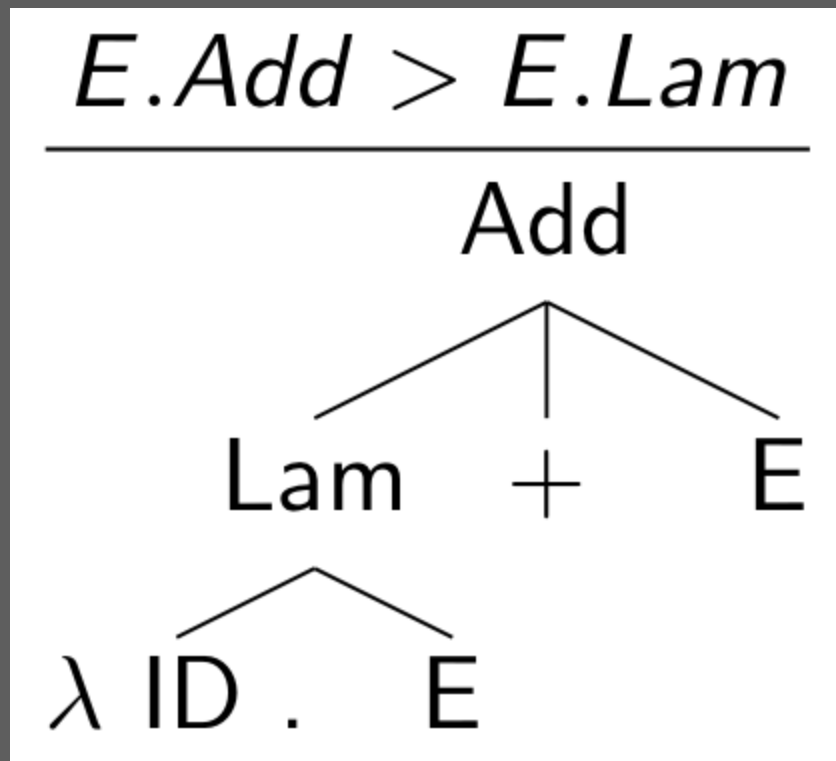


Trees

`a * b ^ λ x. c + d`



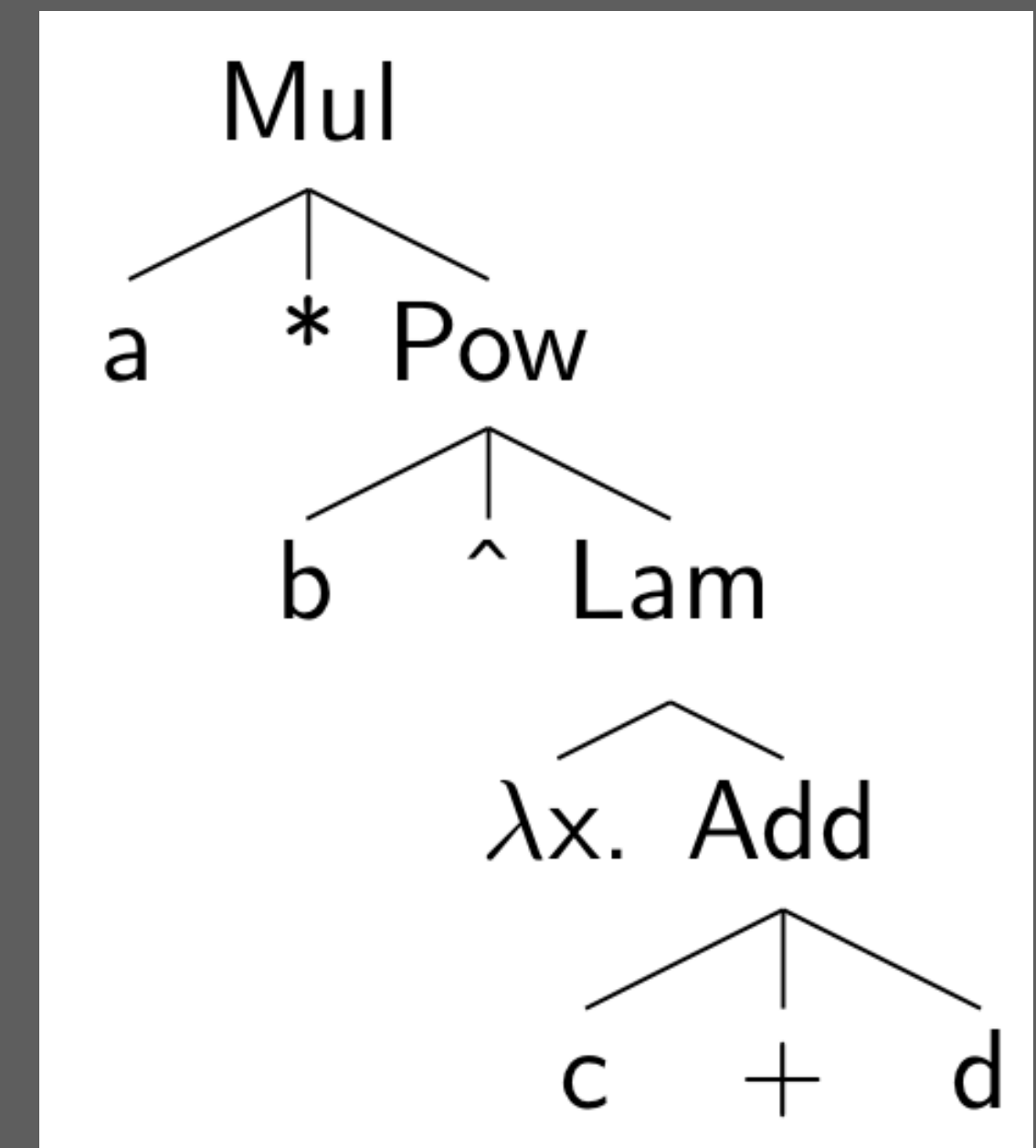
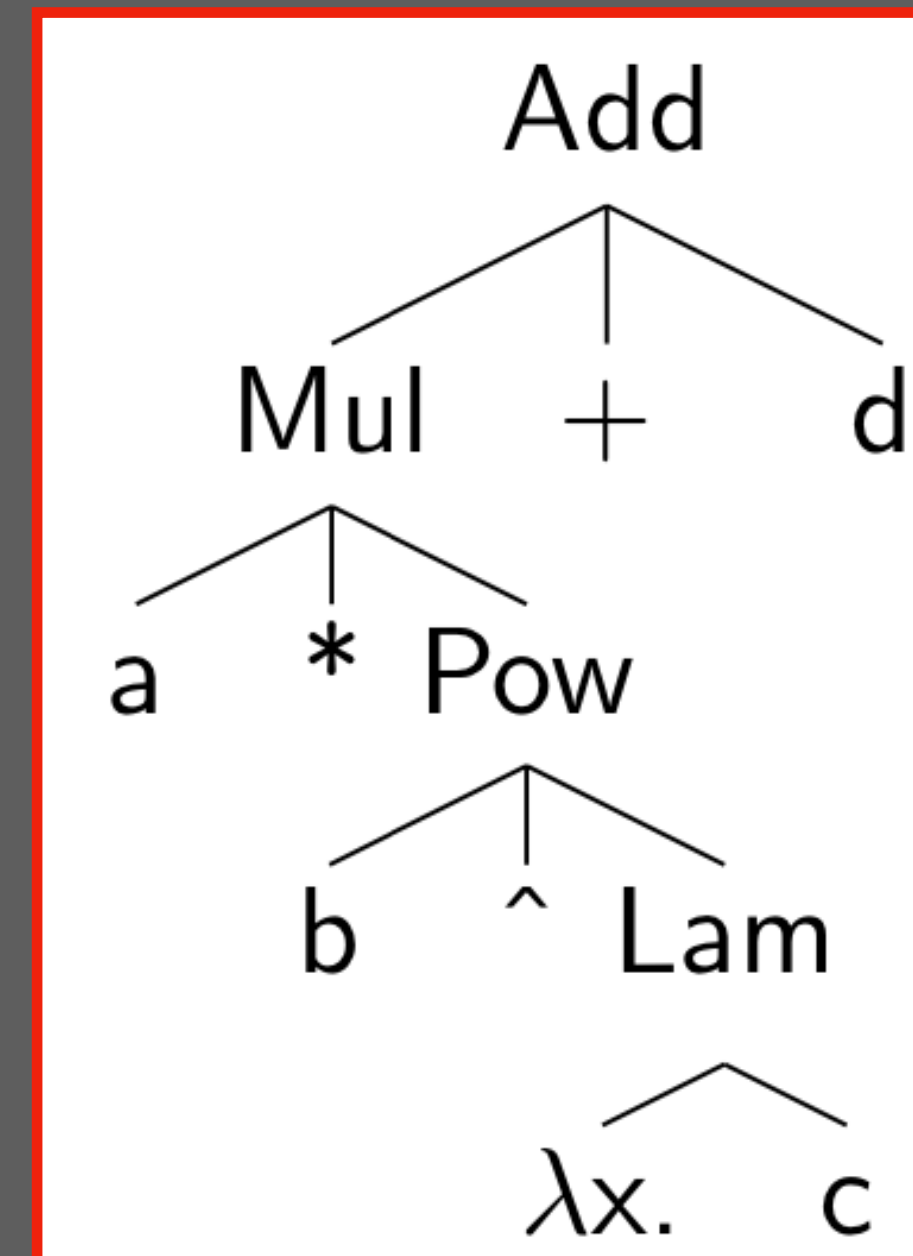
# Deep Priority Conflicts: Match Subpattern in Right-Most Subtree



...

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

Infinite set of conflict patterns



# Safe and Complete Disambiguation Rules

## context-free syntax

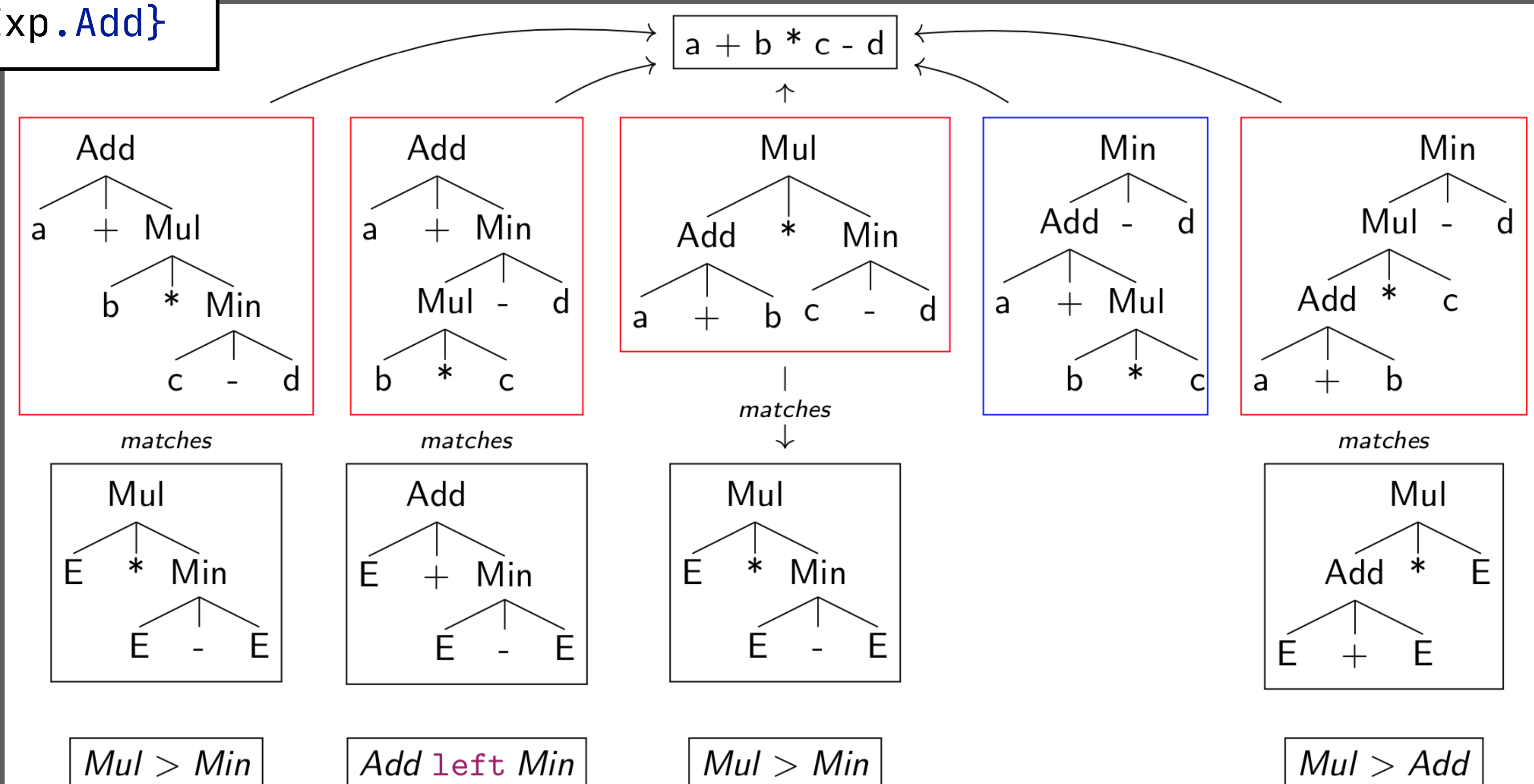
Exp.Min =  $\langle\langle\text{Exp}\rangle - \langle\text{Exp}\rangle\rangle$  {left}

Exp.Add =  $\langle\langle\text{Exp}\rangle + \langle\text{Exp}\rangle\rangle$  {left}

Exp.Mul =  $\langle\langle\text{Exp}\rangle * \langle\text{Exp}\rangle\rangle$  {left}

## context-free priorities

Exp.Mul > {left: Exp.Min Exp.Add}



# Unsafe: Too Many Disambiguation Rules

## context-free syntax

Exp.Min =  $\langle\langle\text{Exp}\rangle - \langle\text{Exp}\rangle\rangle$  {left}

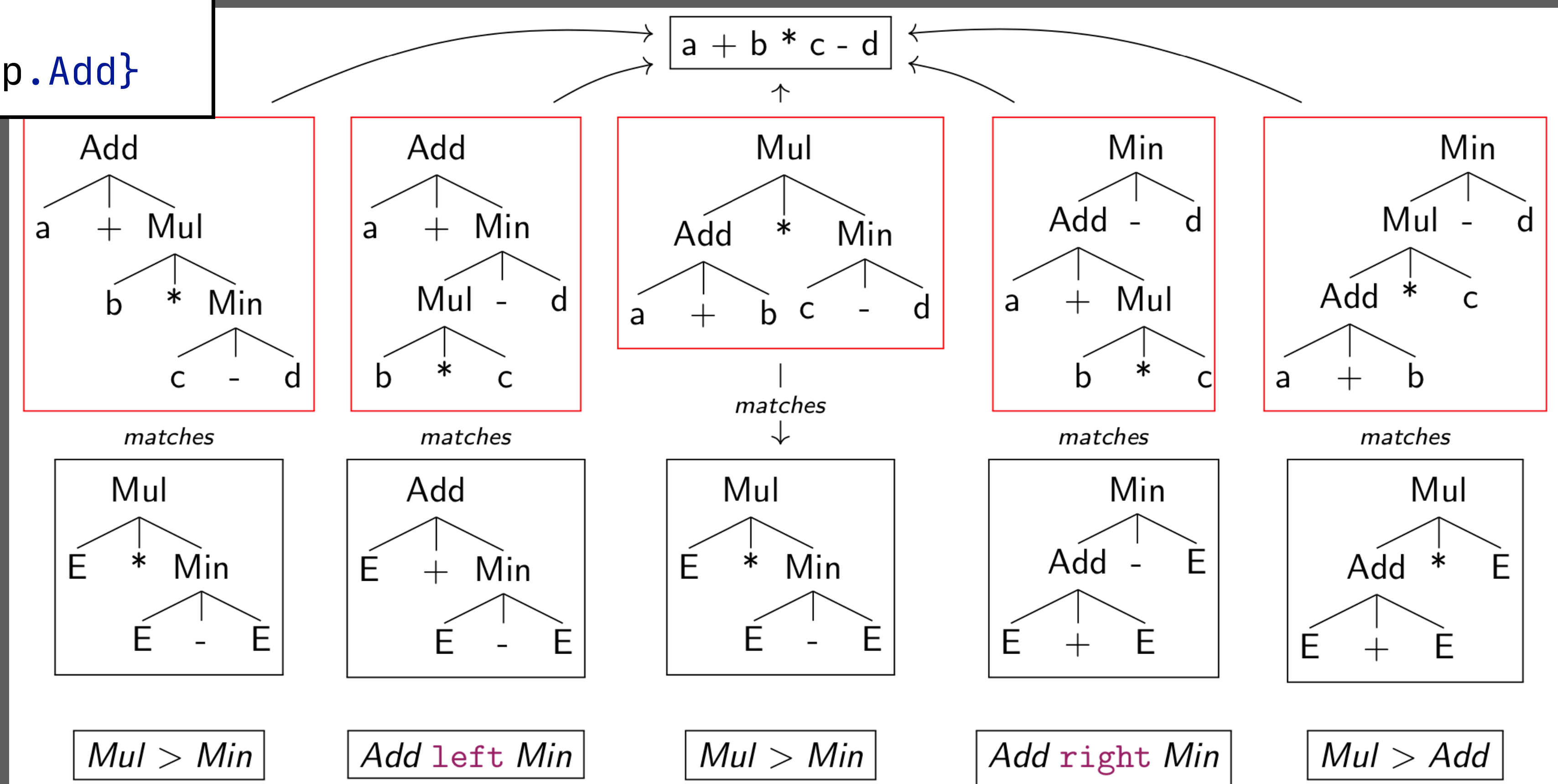
Exp.Add =  $\langle\langle\text{Exp}\rangle + \langle\text{Exp}\rangle\rangle$  {left}

Exp.Mul =  $\langle\langle\text{Exp}\rangle * \langle\text{Exp}\rangle\rangle$  {left}

## context-free priorities

Exp.Mul

> {left, right: Exp.Min Exp.Add}



# Incomplete: Too Few Disambiguation Rules

## context-free syntax

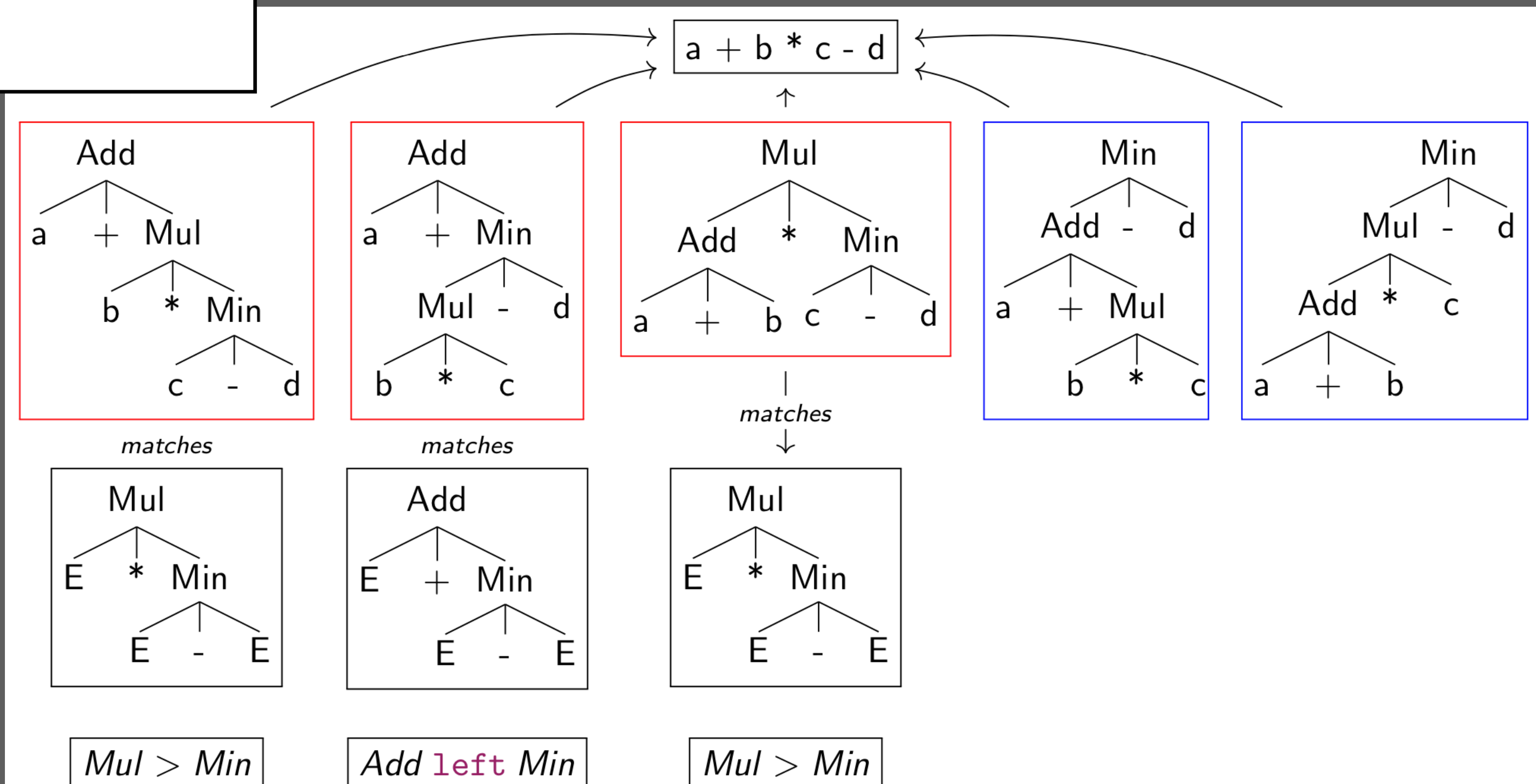
Exp.Min =  $\langle\langle\text{Exp}\rangle - \langle\text{Exp}\rangle\rangle$  {left}

Exp.Add =  $\langle\langle\text{Exp}\rangle + \langle\text{Exp}\rangle\rangle$  {left}

Exp.Mul =  $\langle\langle\text{Exp}\rangle * \langle\text{Exp}\rangle\rangle$  {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<sup>-1</sup> (Insert Necessary Parentheses)

## context-free syntax

Exp = <(<Exp>)> {**bracket**}

Exp.Int = INT

Exp.Var = ID

Exp.Add = <<Exp> + <Exp>> {**left**}

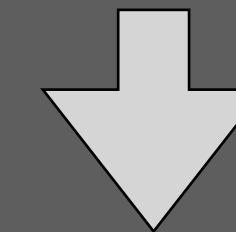
Exp.Let = <let <Bnd\*> in <Exp>>

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

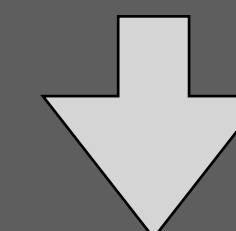
## context-free priorities

Exp.Add > Exp.Let

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



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

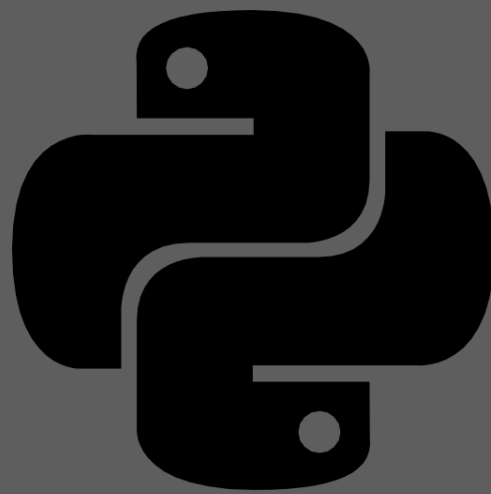


a + (**let**  
 x = b  
**in**  
 c) + d

# 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

x = do 9 + 4  
      \* 3

main = do putStrLn \$  
          show (x \* 2)  
                  *first*      *right*  
                  *left*      *last*

# Alignment with Layout Constraints

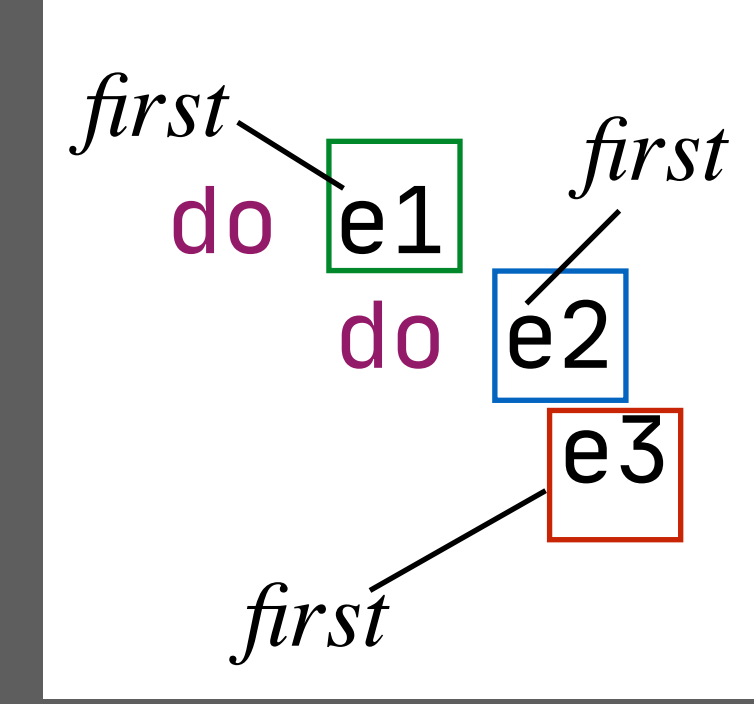
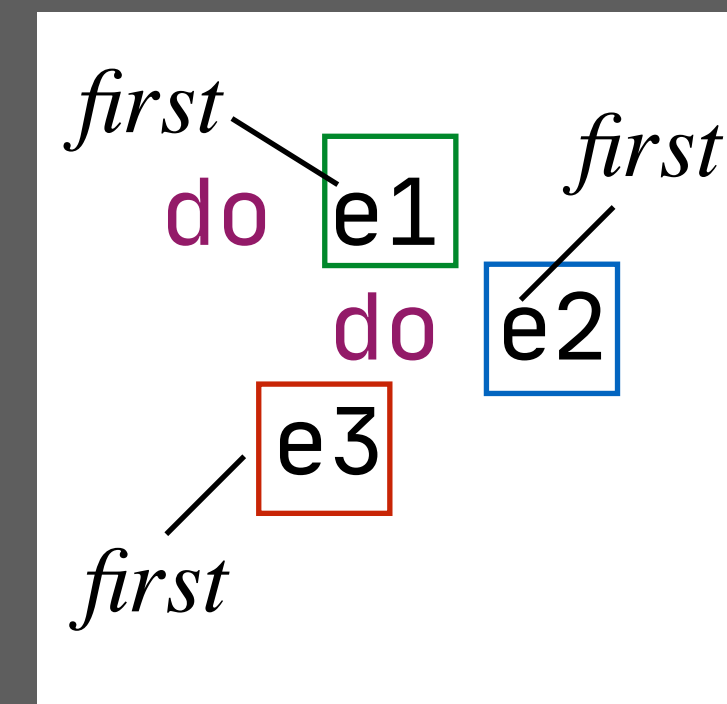
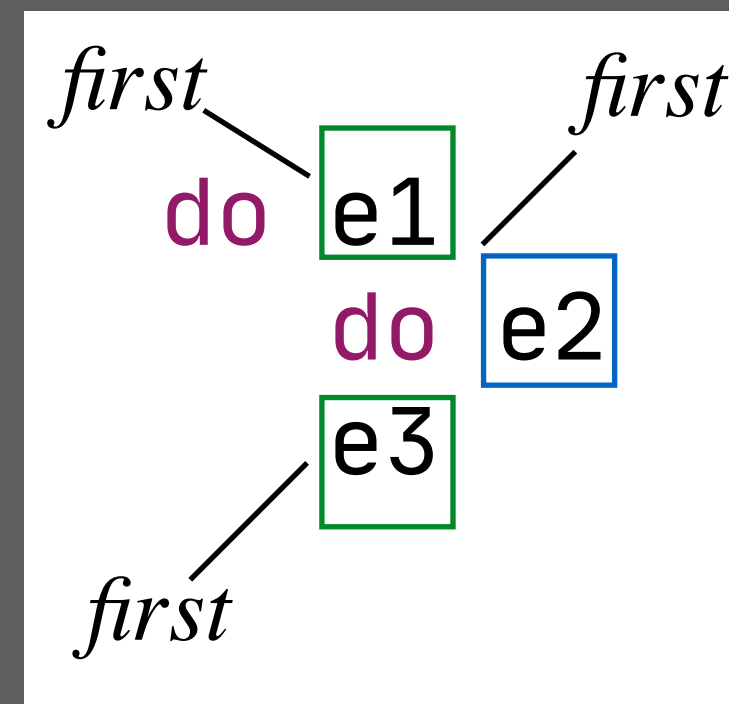
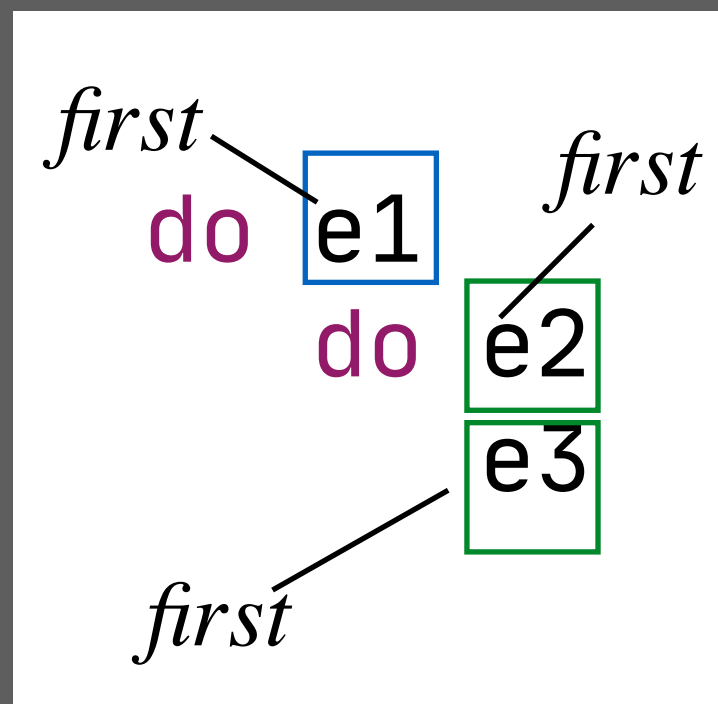
context-free syntax

Exp.Do = "do" ExpList

ExpList.Cns = Exp

ExpList.Lst = ExpList Exp {layout(1.first.col = 2.first.col)}

Exp.Id = ID



# Alignment Declaration

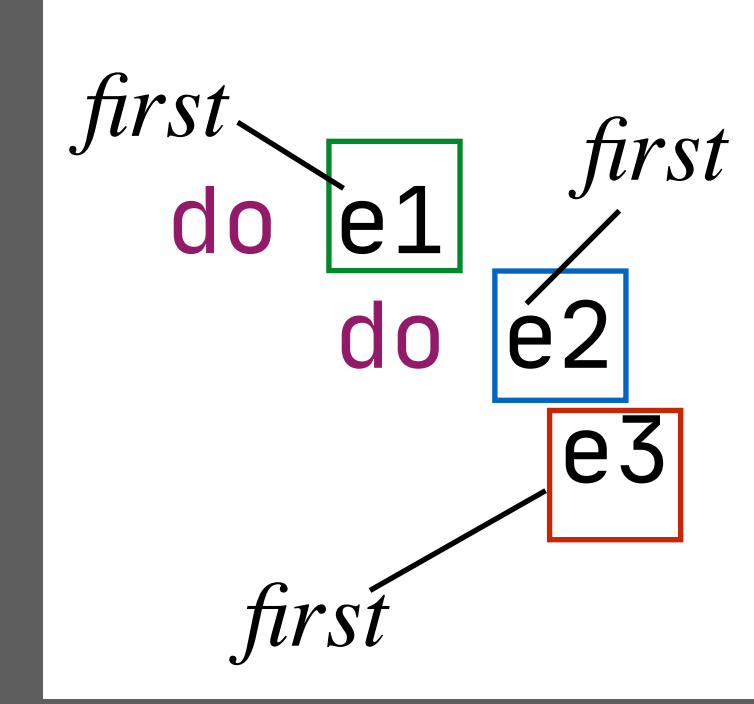
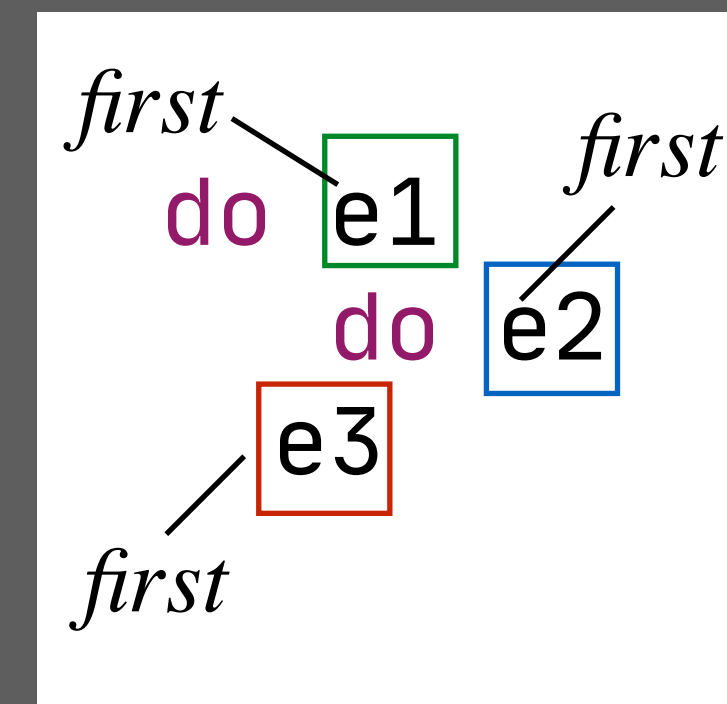
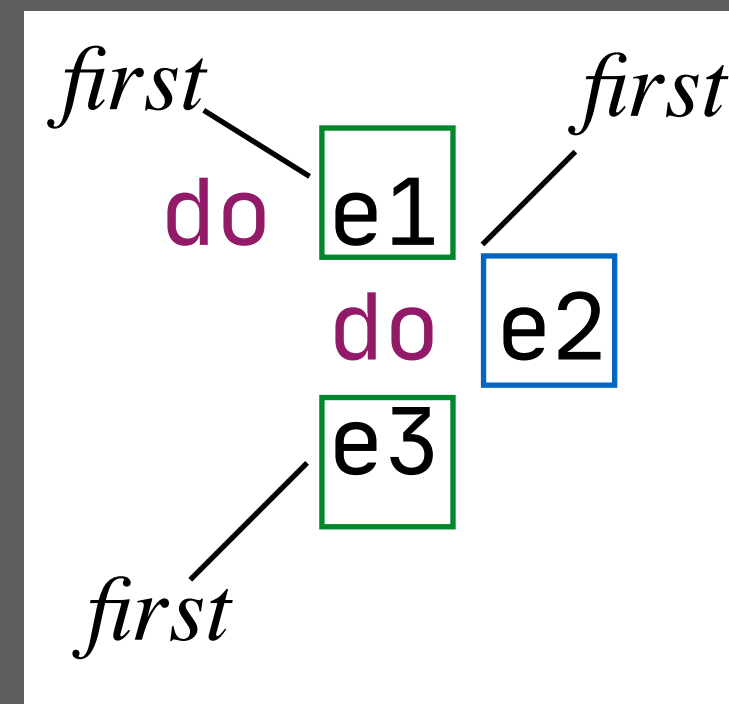
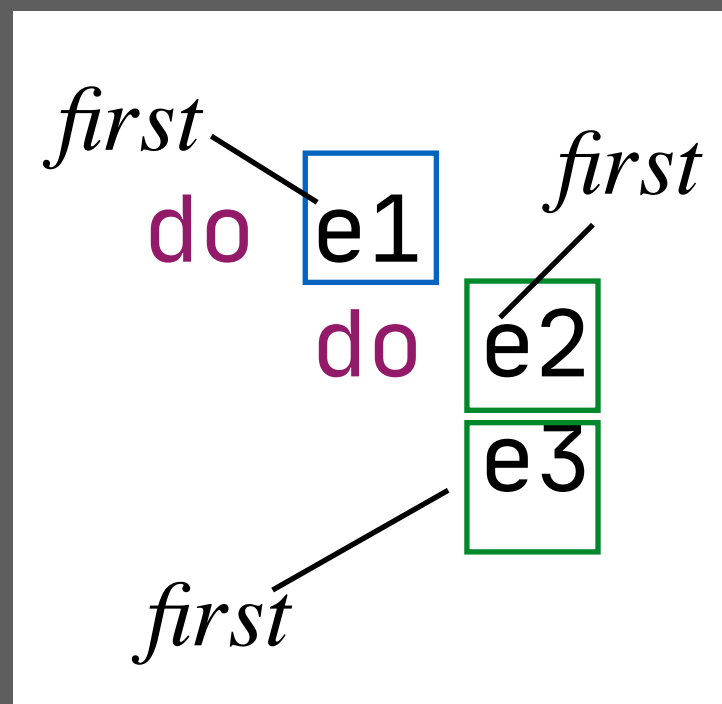
context-free syntax

Exp.Do = "do" ExpList

ExpList.Cns = Exp

ExpList.Lst = exps:ExpList exp:Exp {layout(align exps exp)}

Exp.Id = ID



Semantics

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

# List Alignment Declaration

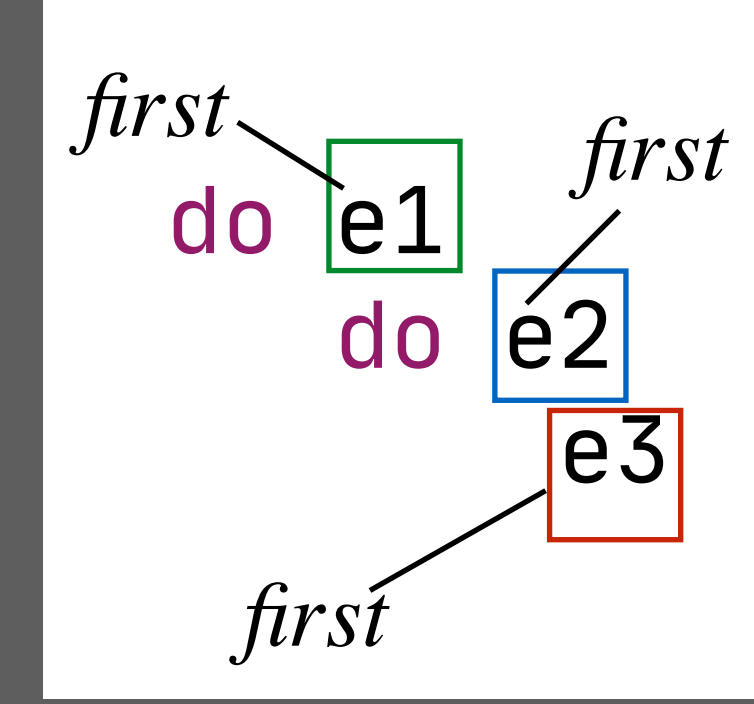
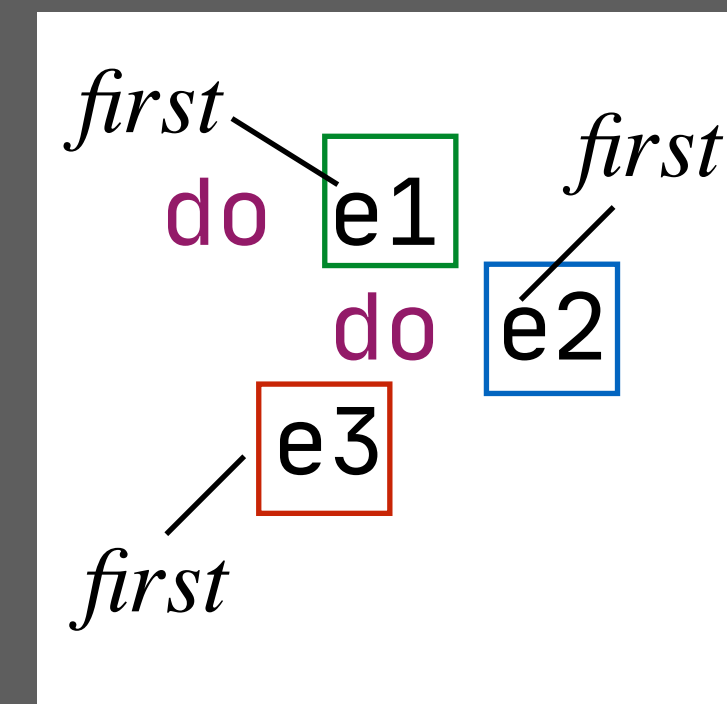
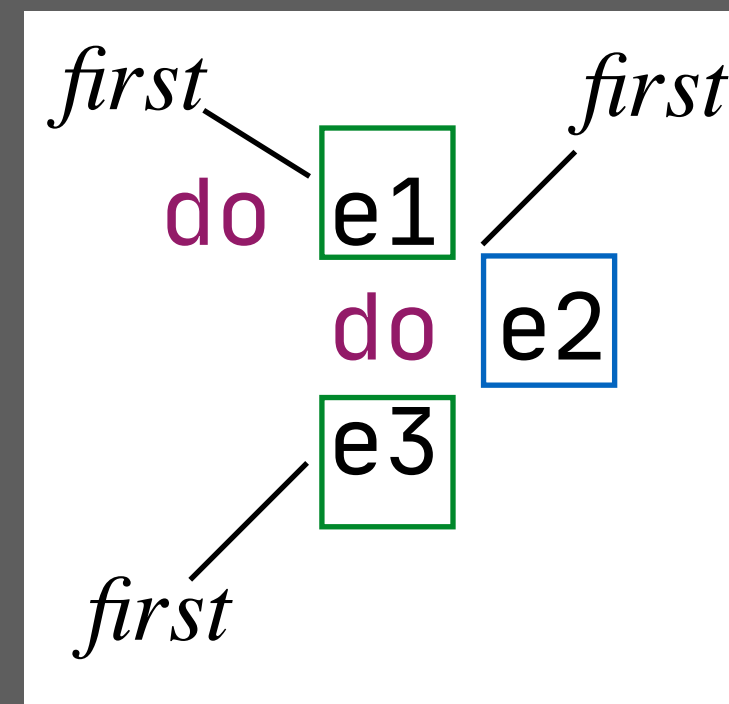
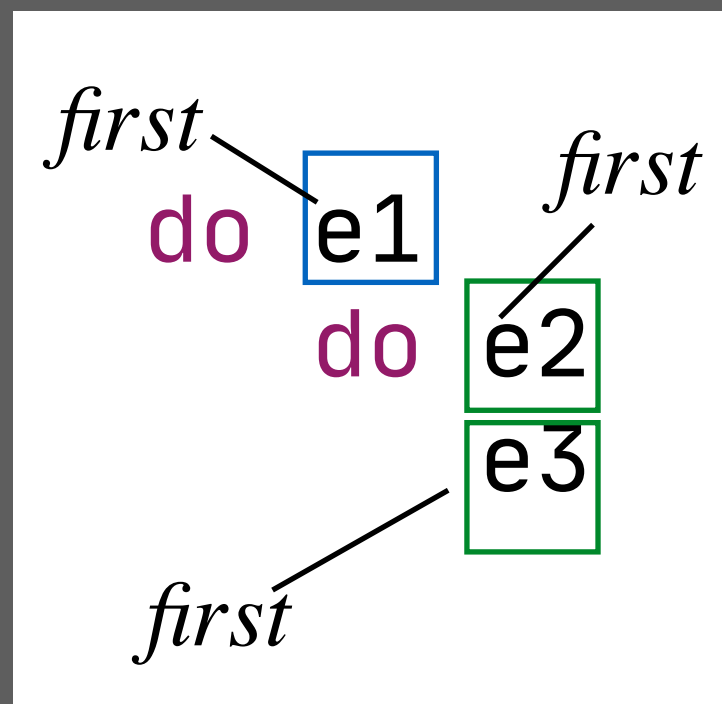
context-free syntax

Exp.Do = "do" exps:Exp+ {layout(align-list exps)}

Exp.Id = ID

Exp+ = Exp+ Exp // normalized

Exp+ = Exp // productions

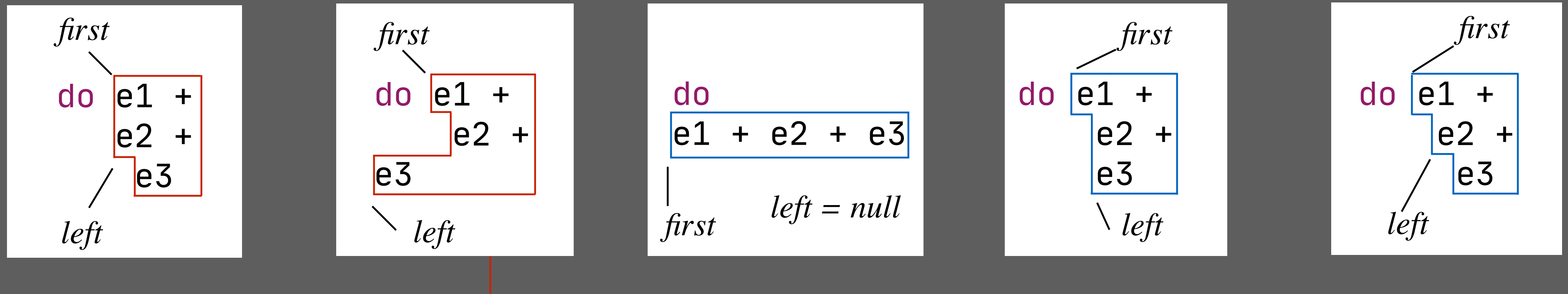


Semantics

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

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.

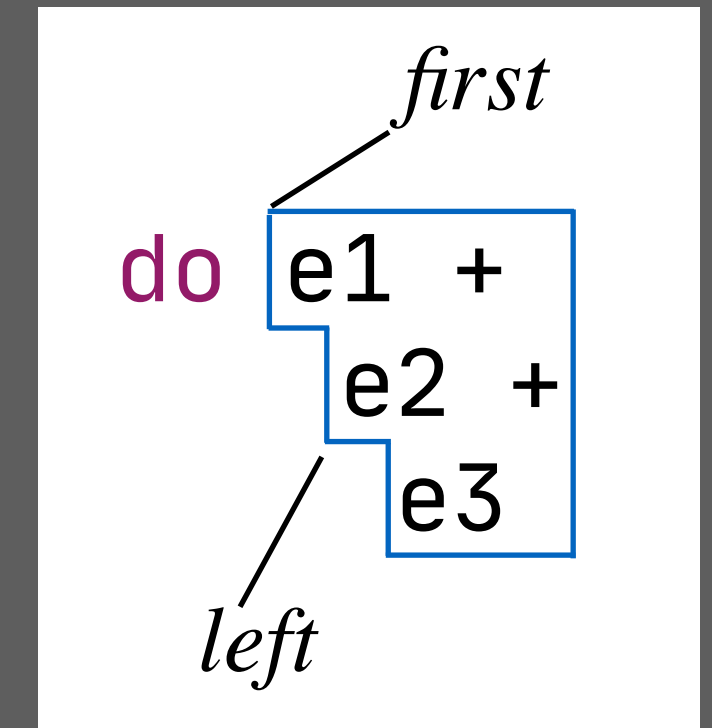
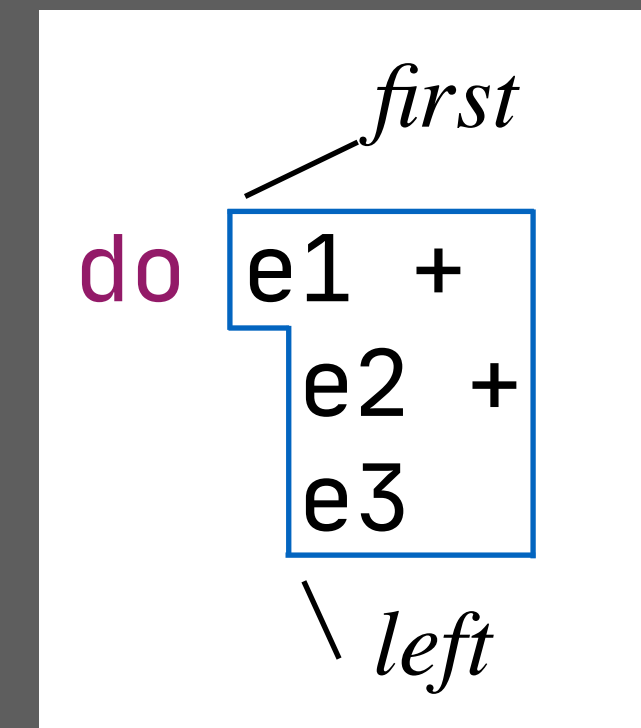
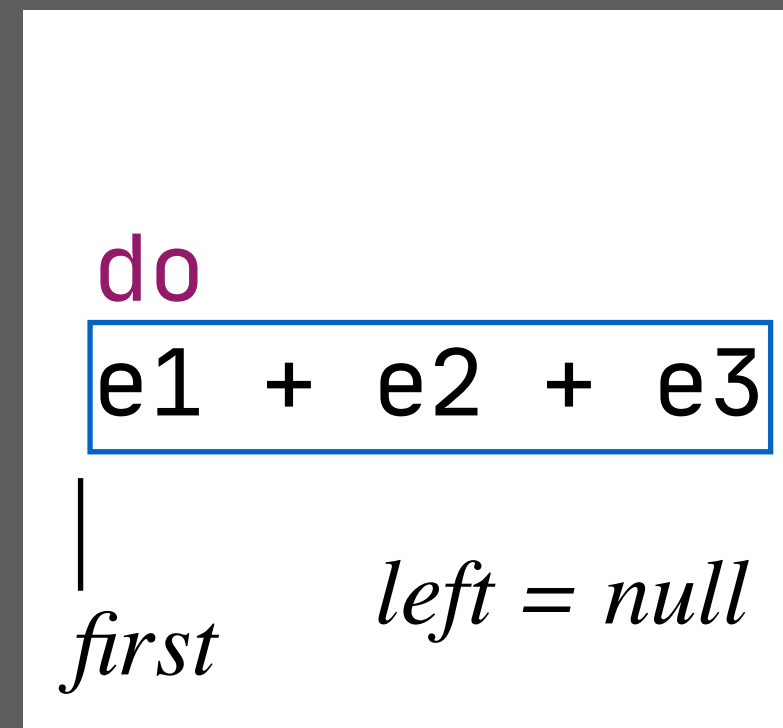
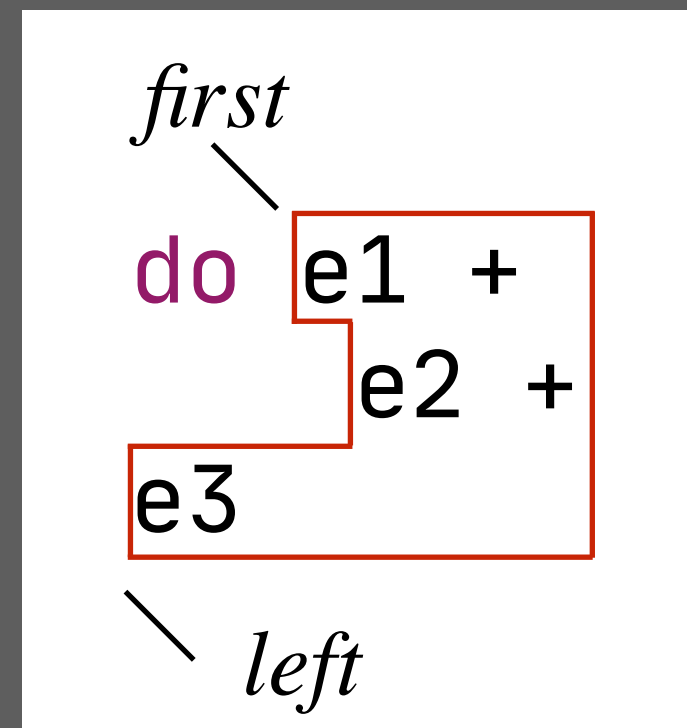
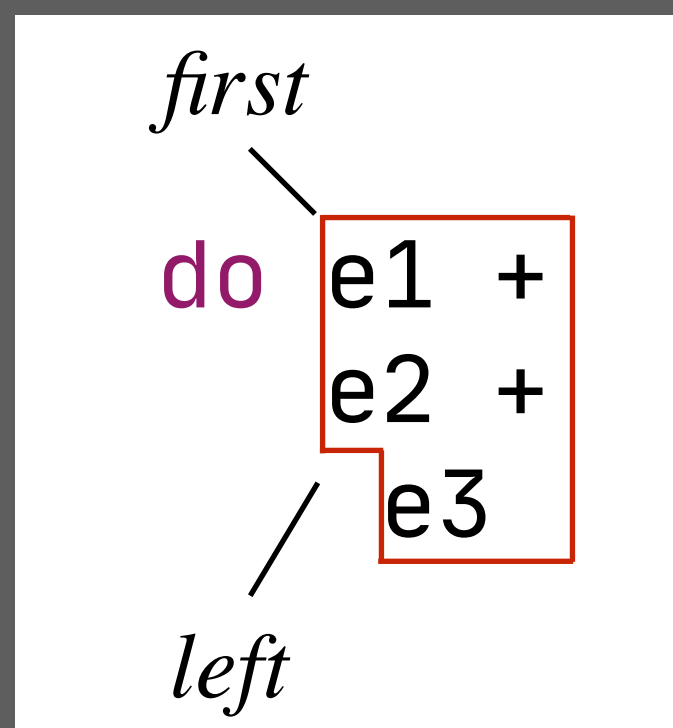
# Offside with Layout Constraints

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.

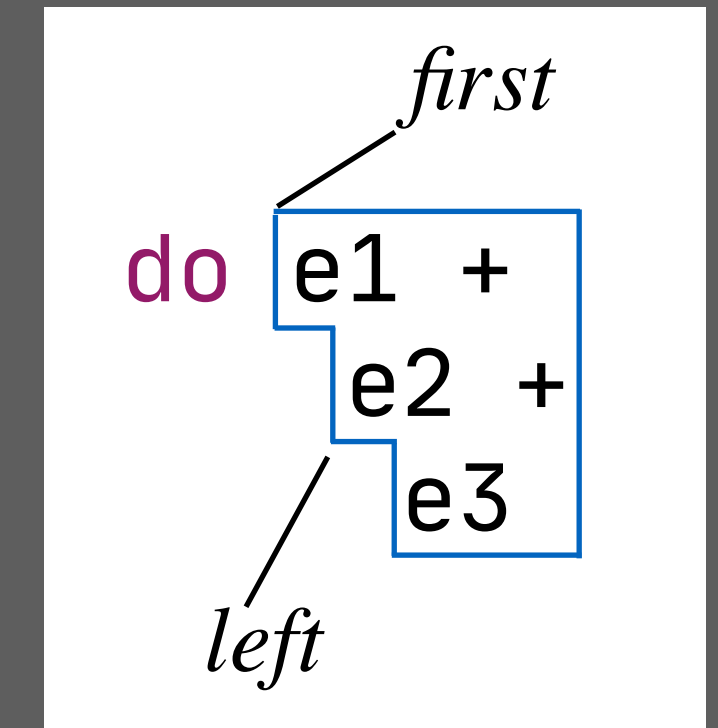
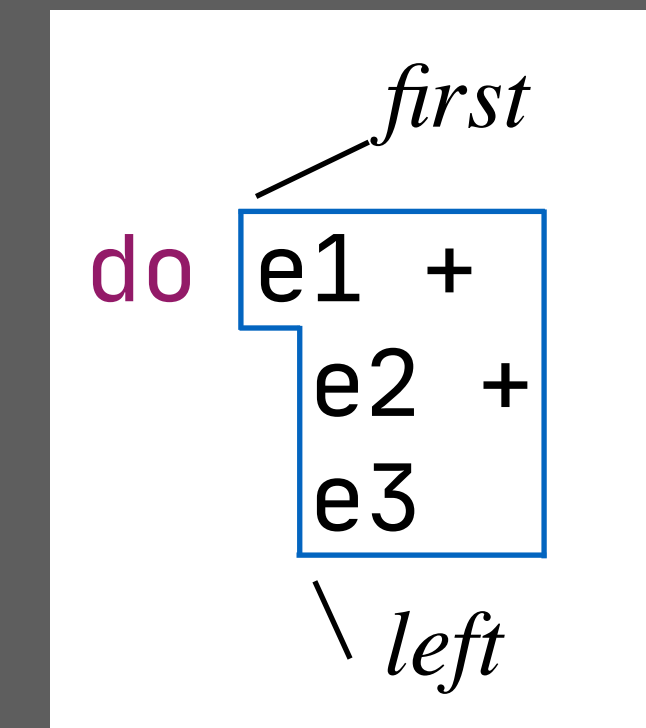
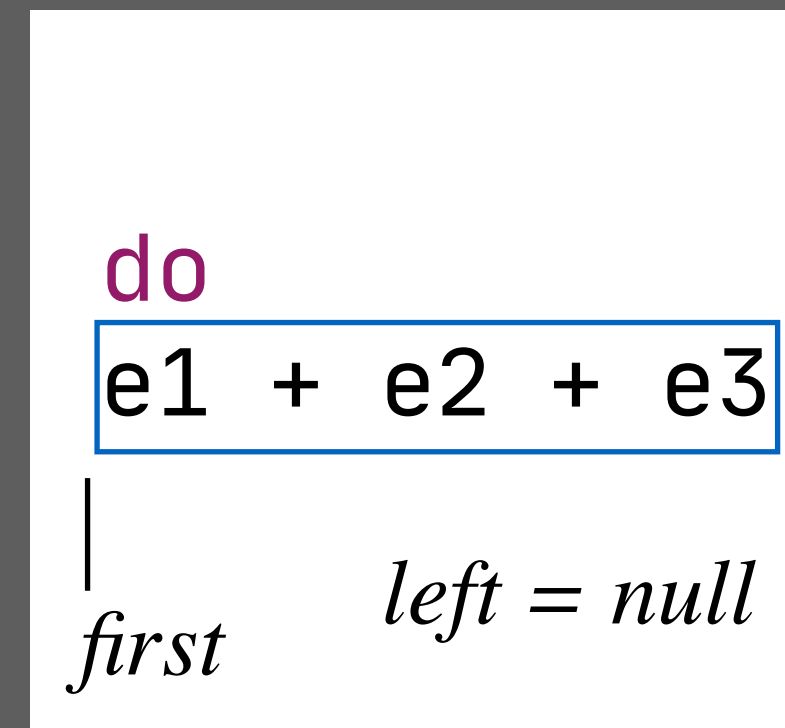
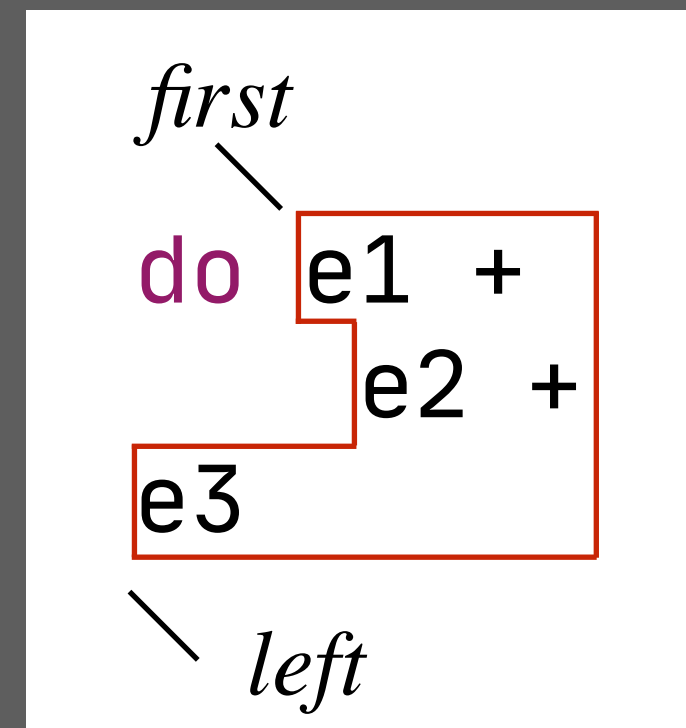
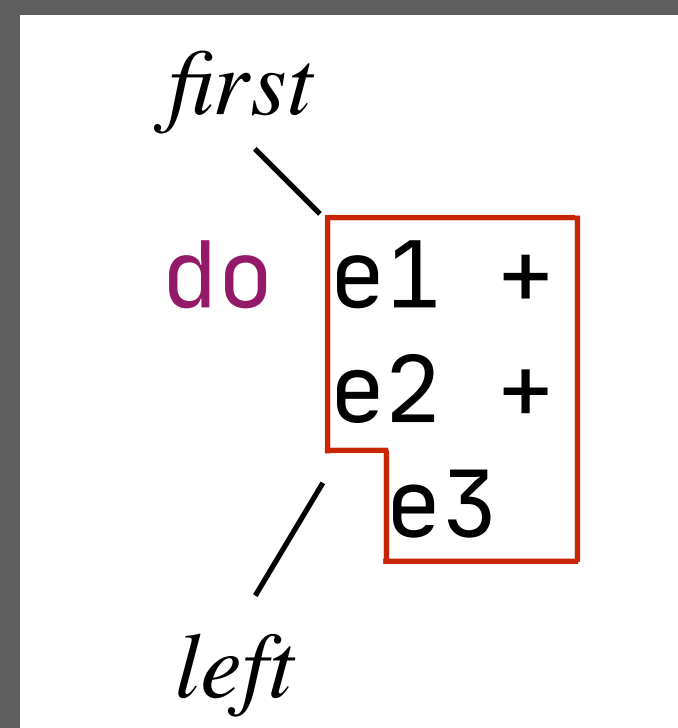
# Offside

## context-free syntax

Exp.Do = "do" exp:Exp {layout(offside exp)}

Exp.Add = Exp "+" Exp {left}

Exp.Id = ID



## Semantics

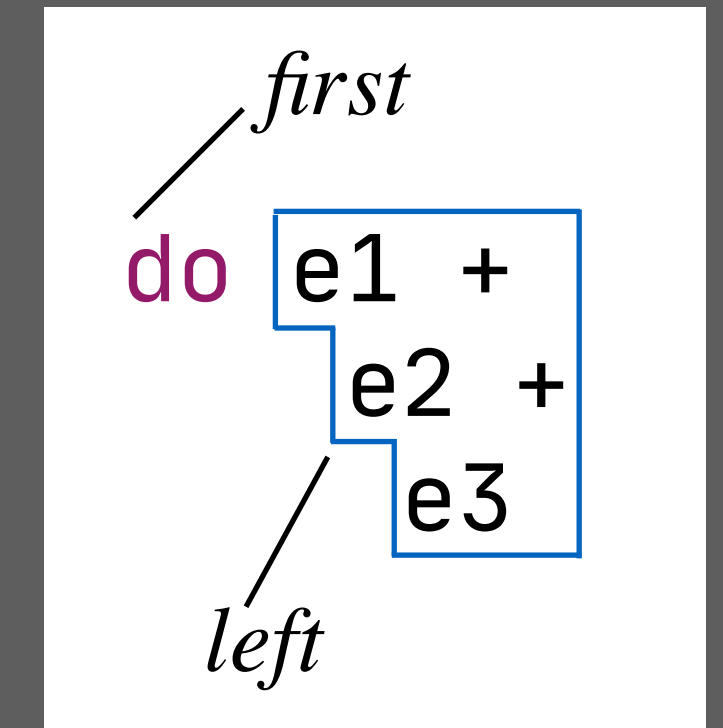
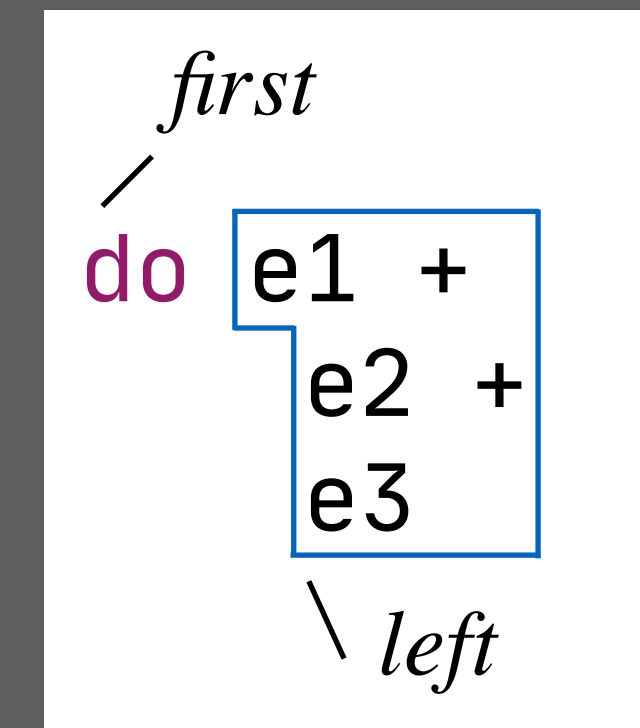
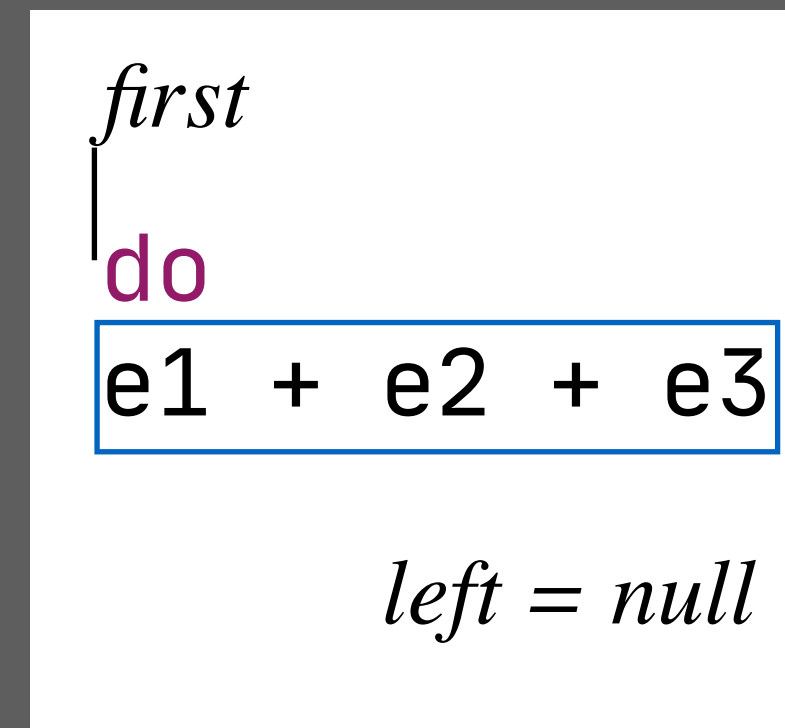
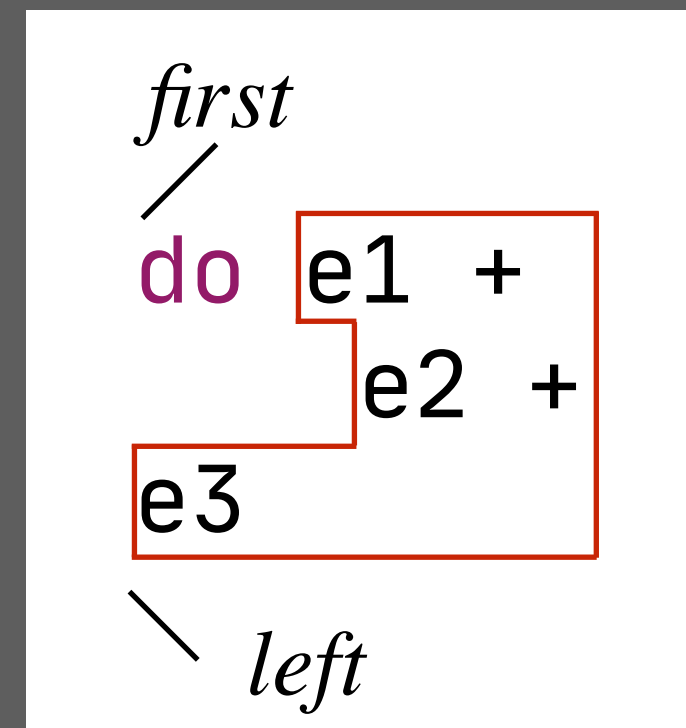
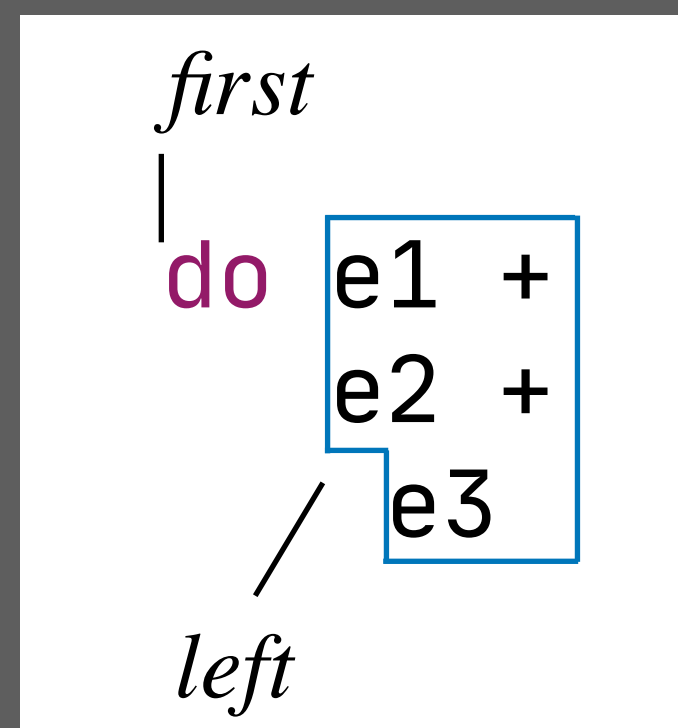
$x.\text{left.col} > x.\text{first.col}$

offside x

# Relative Offside

## context-free syntax

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



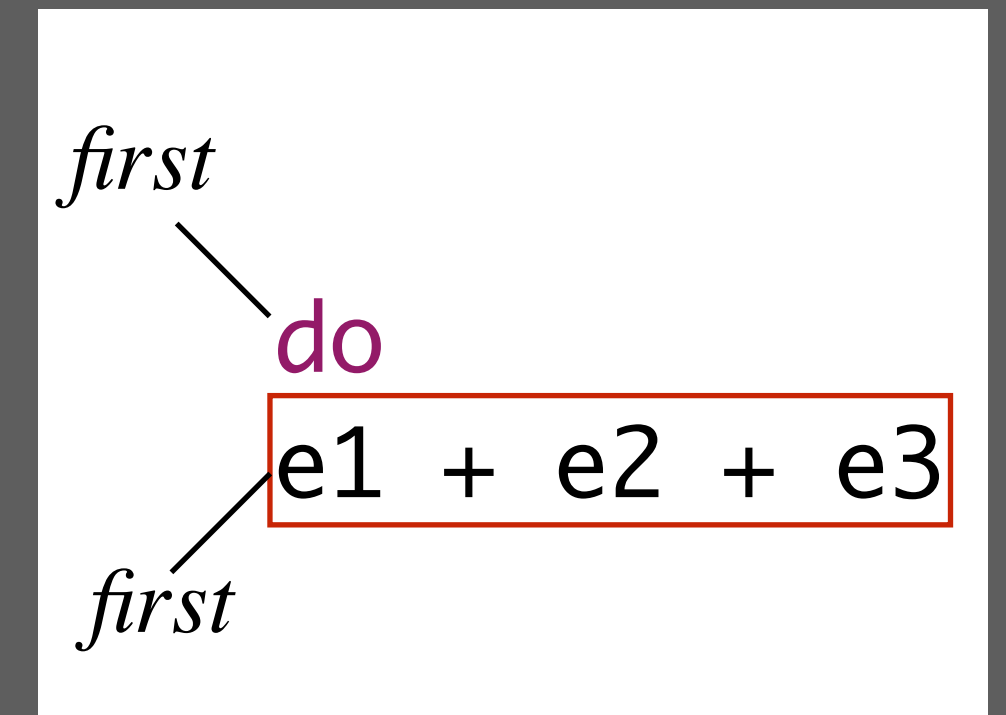
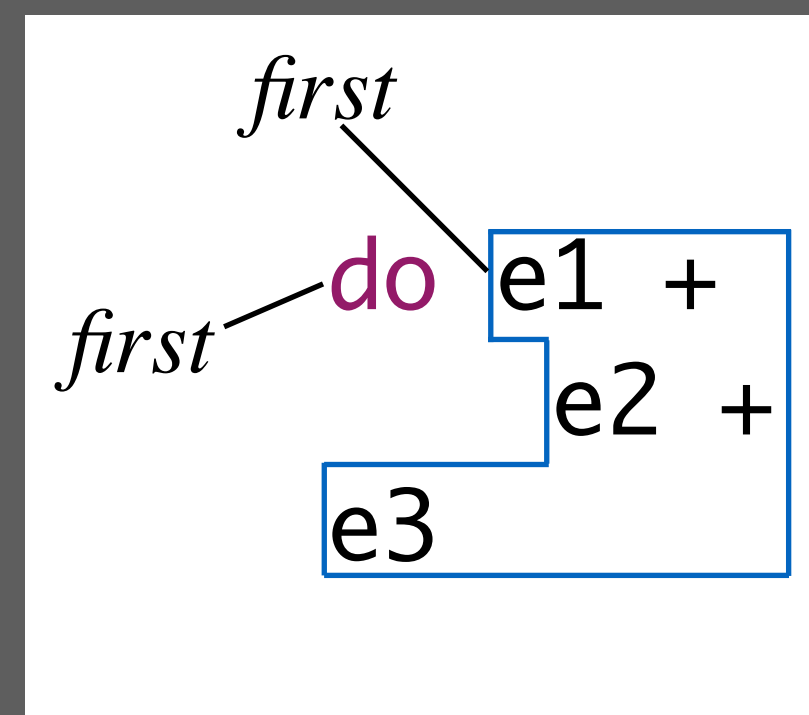
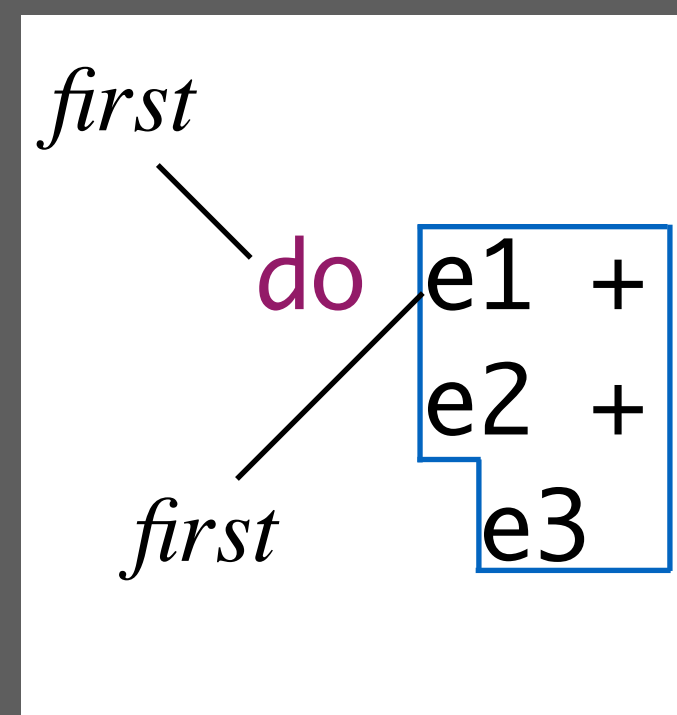
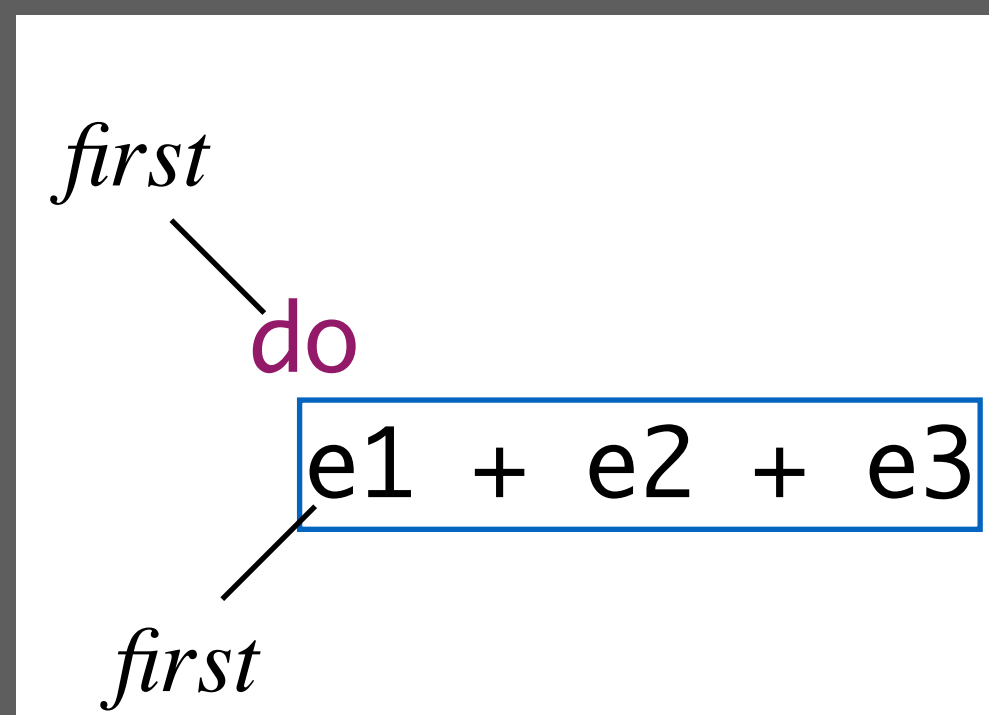
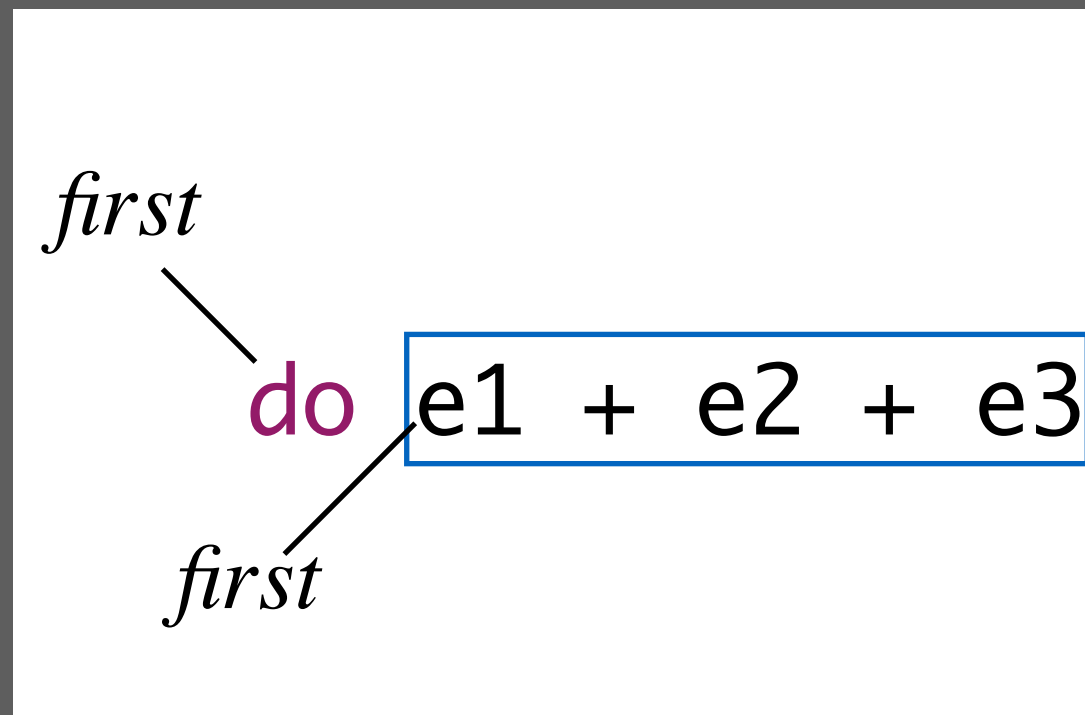
Semantics

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

# Indentation

## context-free syntax

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



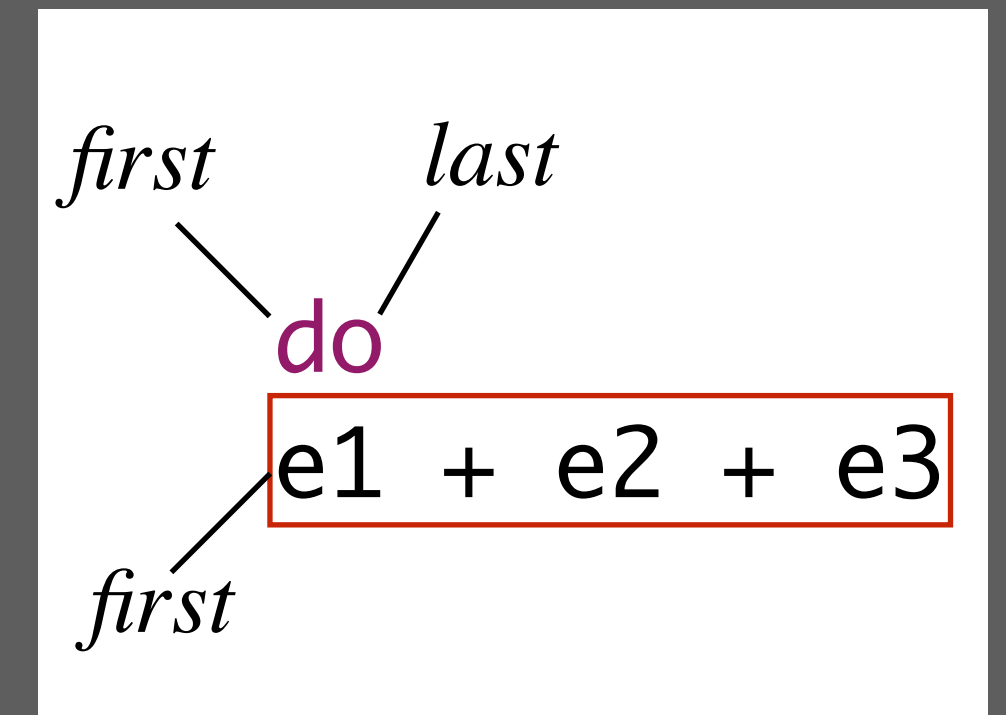
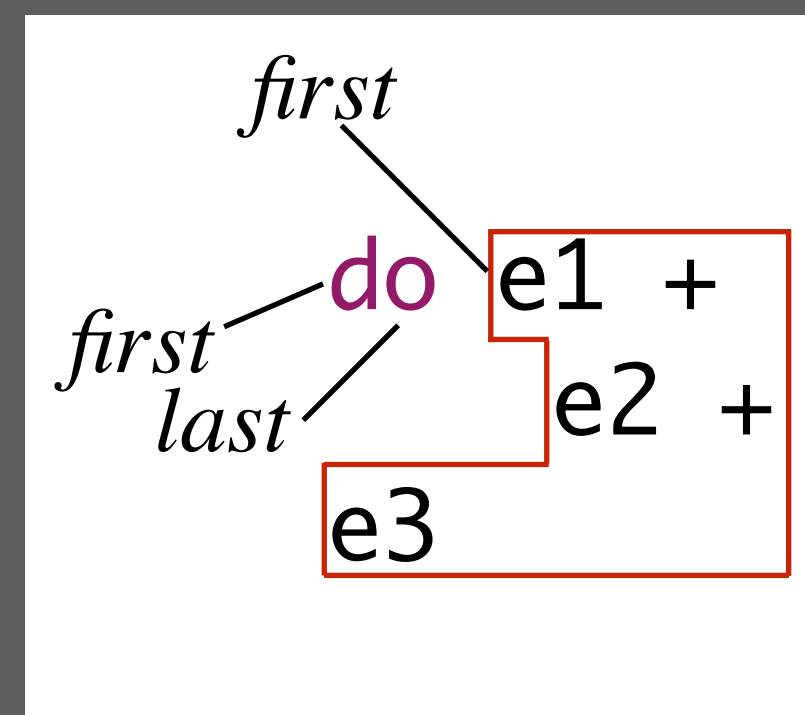
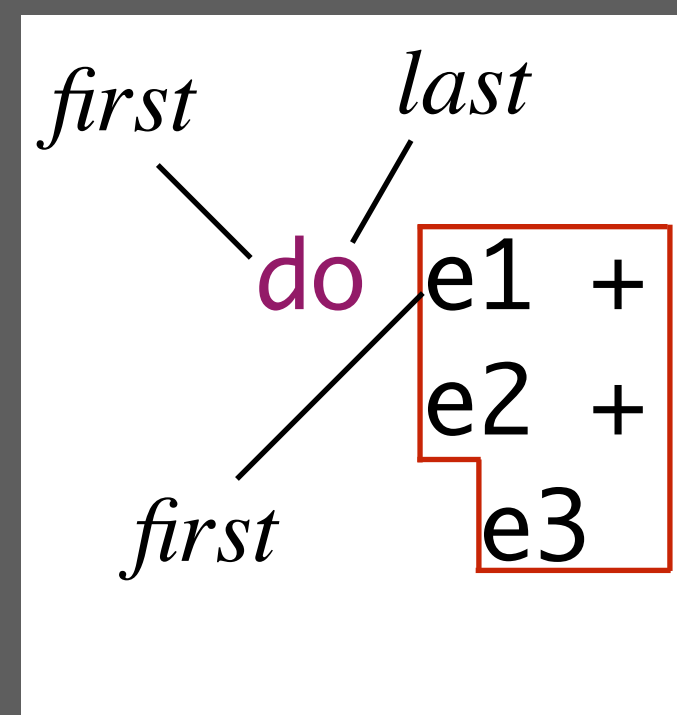
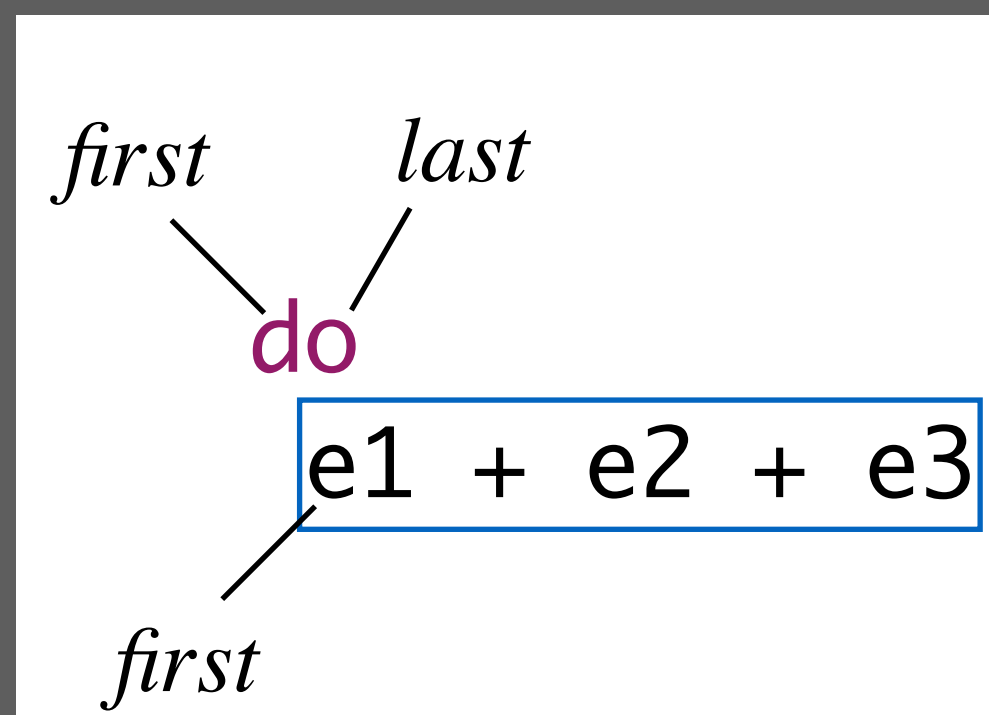
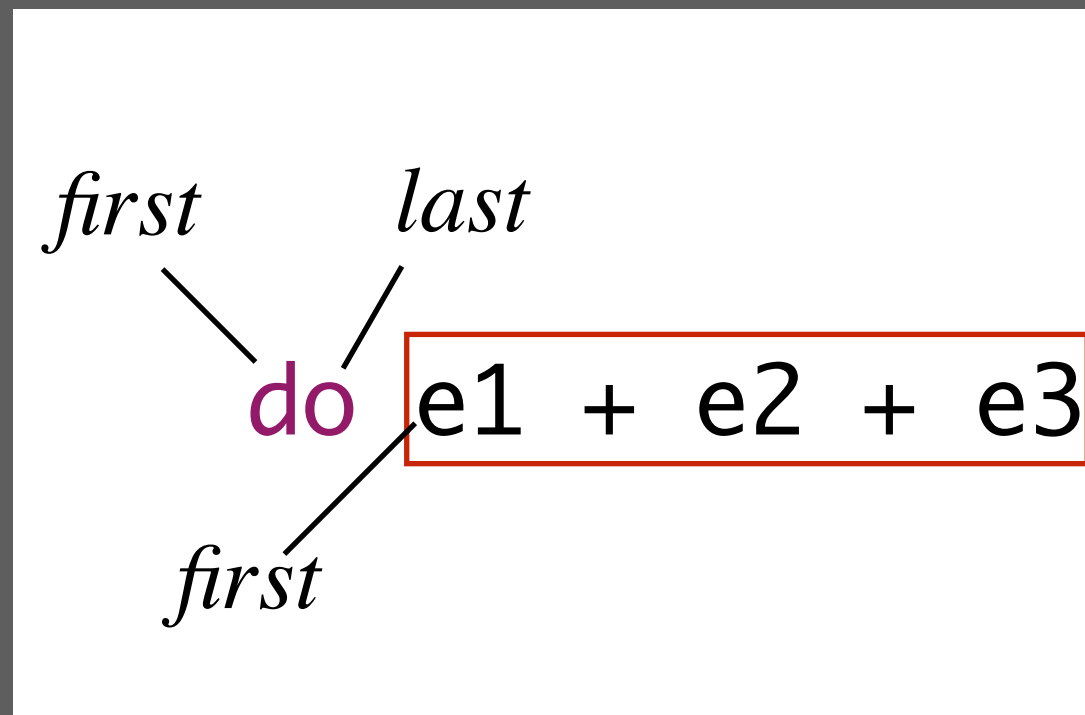
Semantics

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

# Newline + Indentation

## context-free syntax

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



## Semantics

```
y.first.col > x.first.col && y.first.line > x.last.line  
newline-indent x y
```

**How does program layout  
disambiguate structure?**

# Spoofox Language Workbench

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

```

1 module fun
2 imports lex
3 context-free start-symbols Exp
4 sorts Exp Case Bnd Pat
5 context-free syntax
6   Exp      = <(<Exp>)> {bracket}
7   Exp.Int  = INT
8   Exp.Var  = ID
9   Exp.Min  = [-[Exp]]
10  Exp.Sub  = <<Exp> - <Exp>> {left}
11  Exp.Add  = <<Exp> + <Exp>> {left}
12  Exp.Mul  = <<Exp> * <Exp>> {left}
13  Exp.Eq   = <<Exp> = <Exp>> {left}
14  Exp.Fun  = [fun [ID*] → [Exp]]
15  Exp.App  = <<Exp> <Exp>> {left}
16  Exp.Let  = <
17    let <{Bnd "\n\n"}*>
18    in <Exp>
19  >
20  Bnd.Bnd  = <<ID> = <Exp>>
21  Exp.IfE  = <
22    if <Exp> then
23      <Exp>
24    else
25      <Exp>
26  >
27  Exp.IfT  = <
28    if <Exp> then
29      <Exp>
30  >
31  Exp.Match = <
32    match <Exp>
33    with <{Case "\n"}*>
34  > {longest-match}
35  Case.Case = [] [Pat] → [Exp]]
36  Pat.PVar  = ID
37  Pat.PApp  = <<Pat> <Pat>> {left}
38  Pat       = <(<Pat>)> {bracket}
39 context-free priorities
40   Exp.Min > Exp.App
41   > {left: Exp.Sub Exp.Add}
42   > Exp.Eq > Exp.IfE > Exp.IfT
43   > Exp.Match > Exp.Fun > Exp.Let,
44   Exp.App <1> .> Exp.Min
45 template options

```

```

1 let
2   lookup = fun x env →
3     match env
4       with | nil → error
5            | cons (pair y v) env →
6              if x = y then v else lookup x env
7
8 in lookup 1 (cons 2 nil)
9

```

```

1 Let[
2   [ Bnd(
3     "lookup"
4     , Fun(
5       ["x", "env"]
6       , Match(
7         Var("env")
8         , [ Case(PVar("nil"), Var("error"))
9             , Case(
10              PApp(
11                PApp(
12                  PVar("cons")
13                  , PApp(PApp(PVar("pair"), PVar("y")), PVar("
14                    )
15                    , PVar("env")
16                  )
17                , IfE(
18                  Eq(Var("x"), Var("y"))
19                  , Var("v")
20                  , App(App(Var("lookup"), Var("x")), Var("env")
21                )
22              )
23            ]
24          )
25        )
26      )
27    ]
28  , App(
29    App(Var("lookup"), Int("1"))
30    , App(App(Var("cons"), Int("2")), Var("nil"))
31  )
32 ]

```

# 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

## 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 ... ?*