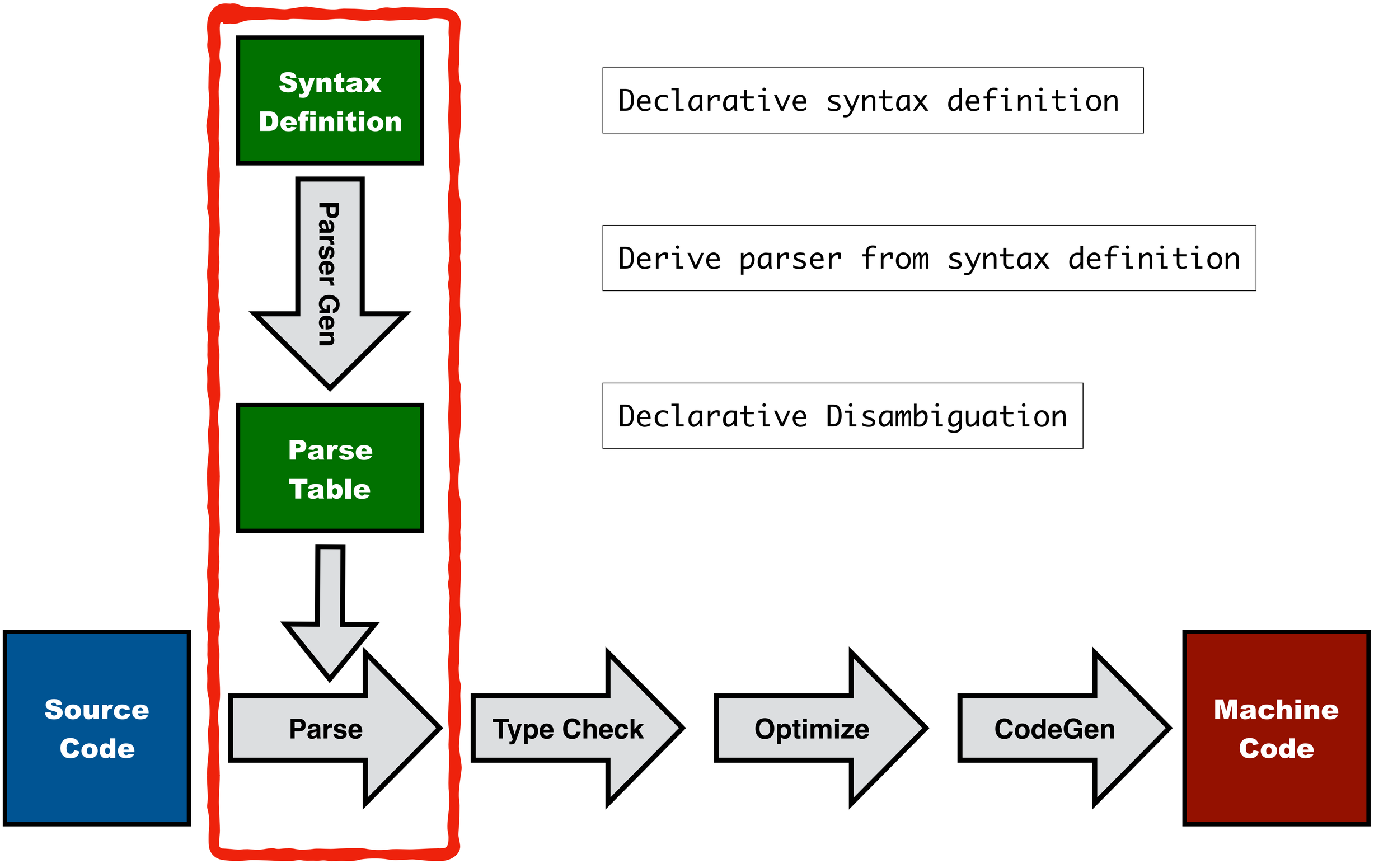
Declare Your Language

Chapter 15:
Declarative Disambiguation

Luís Eduardo de Souza Amorim

IN4303 Compiler Construction
TU Delft
September 2017

TU Delft


Syntax Definition
Parser Gen
Parse Table
Source Code
Parse
Type Check
Optimize
CodeGen
Machine Code
Declarative syntax definition
Derive parser from syntax definition
Declarative Disambiguation

This Lecture

~~**Lexical syntax**~~

- ~~defining the syntax of tokens / terminals including layout~~
- ~~making lexical syntax explicit~~

~~**Formatting specification**~~

- ~~how to map (abstract syntax) trees to text~~

~~**Syntactic completion**~~

- ~~proposing valid syntactic completions in an editor~~

~~**Parsing**~~

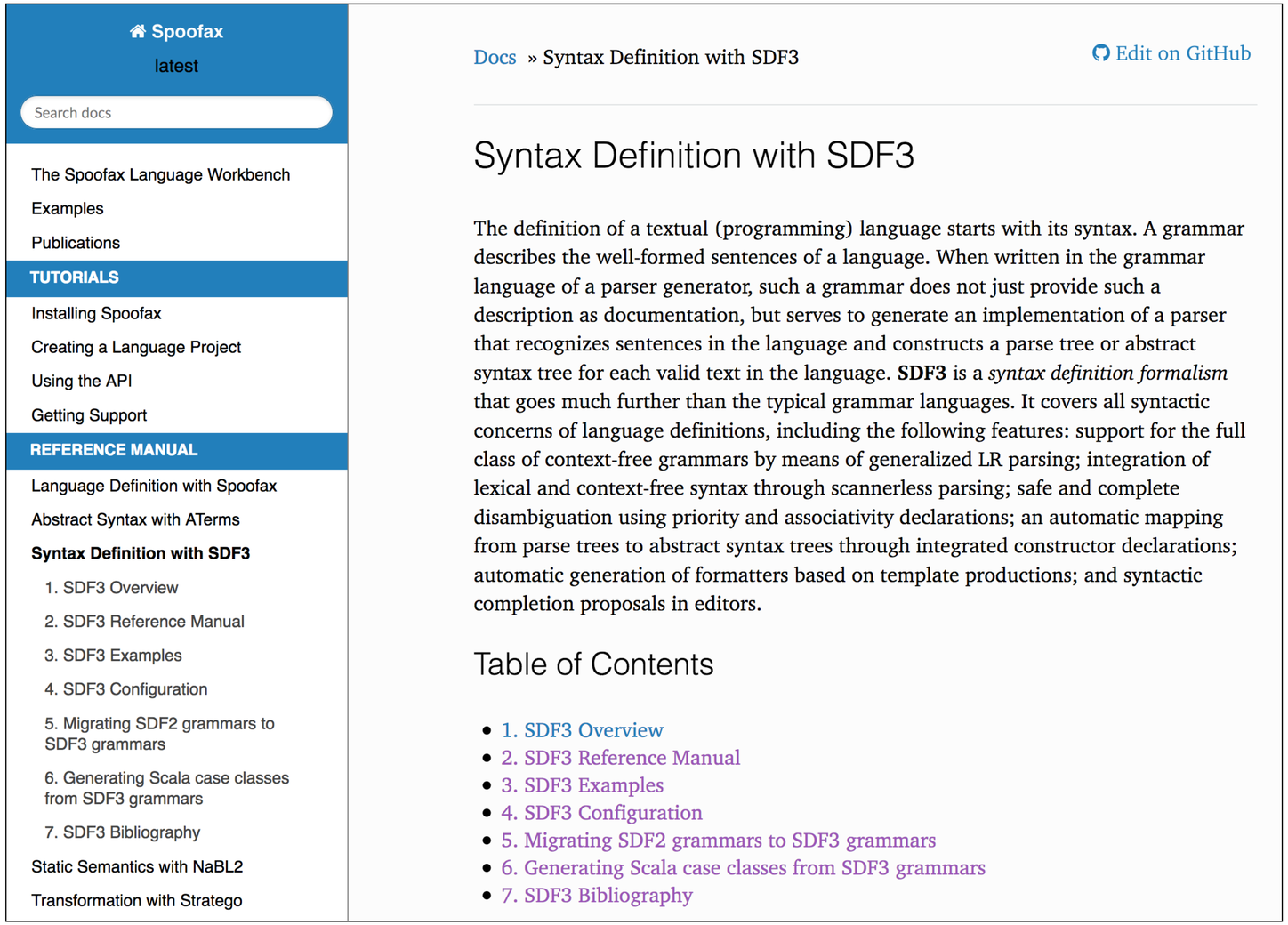
- ~~interpreting a syntax definition to map text to trees~~

Declarative Disambiguation

- solving conflicts when parsing expression languages

Reading Material

The SDF3 syntax definition formalism is documented at the metaborg.org website.



http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/index.html

SDF2 Disambiguation

ASMICS Workshop on Parsing Theory, 1994

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.9812

Using Filters for the Disambiguation of Context-free Grammars*

Paul Klint[§¶]
paulk@cwi.nl

Eelco Visser[§]
visser@fwi.uva.nl

Abstract

An ambiguous context-free grammar defines a language in which some sentences have multiple interpretations. For conciseness, ambiguous context-free grammars are frequently used to define even completely unambiguous languages and numerous disambiguation methods exist for specifying which interpretation is the intended one for each sentence. The existing methods can be divided in 'parser specific' methods that describe how some parsing technique deals with ambiguous sentences and 'logical' methods that describe the intended interpretation without reference to a specific parsing technique.

We propose a framework of *filters* to describe and compare a wide range of disambiguation problems in a parser-independent way. A filter is a function that selects from a set of parse trees (the canonical representation of the interpretations of a sentence) the intended trees. The framework enables us to define several general properties of disambiguation methods.

The expressive power of filters is illustrated by several case studies. Finally, a start is made with the study of efficient implementation techniques for filters by exploiting the commutativity of parsing steps and filter steps for certain classes of filters.

Key words & phrases: context-free grammars, generalized parsing, disambiguation, filters

1 Introduction

In the last two decades we have seen the successful development of theory and implementation techniques for efficient, deterministic, parsing of languages defined by context-free grammars. As a consequence, the LL(k) and LR(k) grammar classes and associated parsing algorithms are now dominating the field.

Using parsing techniques based on these subclasses of the context-free grammars has, however, several draw backs. First of all, syntax definitions may need to be brought into an acceptable, but often unnatural, form that obeys the restrictions imposed by the grammar class being used. More importantly, subclasses of the context-free grammars are not closed under composition, e.g., composing two LR(1) grammars does not necessarily yield an LR(1) grammar. Only the class of context-free grammars itself can support the composition of grammars which is essential for the support and development of modular grammar definitions.

The use of natural, modular, grammars is becoming feasible due to the recent advances in parsing technology for arbitrary context-free grammars. Unfortunately, when leaving the established field of deterministic parsing one encounters a next obstacle: the language defined by a grammar may become *ambiguous* and mechanisms are needed to disambiguate

*Partial support received from NWO project 612-317-420: Incremental parser generation and context-dependent disambiguation, a multi-disciplinary perspective.

*This is Technical Report P9426 (December 23, 1994) from[§] (ftp://ftp.fwi.uva.nl/pub/programming-research/reports/1994/P9426.ps.Z). This paper also appeared in the *Proceedings of the ASMICS Workshop on Parsing Theory*, Milano, 13 & 14 October 1994.

[§]Programming Research Group, University of Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands, http://www.fwi.uva.nl/fwi/research/vg3/

[¶]CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, http://www.cwi.nl/[~gipe/]

SDF3 Disambiguation

Technical Report TUD-SERG-2017-014.
Delft University of Technology.

Declarative Disambiguation of Deep Priority Conflicts

LUÍS EDUARDO DE SOUZA AMORIM, Delft University of Technology
TIMOTHÉE HAUDEBORG, ENS Rennes
EELCO VISSER, Delft University of Technology

Declarative disambiguation using precedence and associativity declarations supports concise and extensible definition of the syntax of programming languages. However, for non-trivial cases such as low precedence prefix and postfix operators, the semantics of such declarations is not well-defined. In this paper, we define a new safe and complete semantics for priority and associativity declarations. We extend a safe semantics for one-level tree patterns with a formal definition of *deep priority conflicts* that are not covered by fixed-depth patterns. We show how this semantics can be used to resolve ambiguities such as dangling else and longest-match. Furthermore, we extend the approach to productions with indirect recursion. We have implemented the semantics in a parser generator for SDF3 and evaluated the approach by applying it to the grammars of seven languages.

ACM Reference format:
Luís Eduardo de Souza Amorim, Timothée Haudeborg, and Eelco Visser. 2016. Declarative Disambiguation of Deep Priority Conflicts. 1, 1, Article 1 (January 2016), 43 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Context-free grammars provide a concise, high-level, and well-understood formalism to document the syntax of programming languages. Grammars play a dual role in the description of programming languages. On the one hand, grammars are used to describe the *structure* of programs, i.e. their well-formed trees. For this purpose, the (abstract) syntax definitions in reference manuals and academic papers are often *ambiguous*, since that allows concise descriptions and a direct correspondence between abstract syntax trees and grammar rules. On the other hand, grammars are also used to describe the mapping from sentences to trees. For this purpose, a grammar should unambiguously identify the structure of a program text.

Many common ambiguities arise from operator precedence and associativity of expressions in programming languages. Reference manuals typically address such ambiguities by separately declaring the precedence and associativity of operators, or by encoding the precedence of the operators in the grammar itself. The grammar used throughout the Java SE 7 Specification (Gosling et al. 2013) follows the first approach. However, a different grammar is used as the basis for the reference implementation of the parser for Java SE 7. The Java SE 8 specification (Gosling et al. 2014), on the other hand, uses the second approach. Even though only one grammar is presented in the Java 8 specification, the grammar is less concise since productions are duplicated to solve ambiguities that arise from combining operators or statements.[1]

[1]For example, to solve dangling-else ambiguities, the Java grammar creates a new non-terminal StatementNoShortIf, duplicating the productions for statements excluding IfThenStatement, as shown in https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.9. In the case of operators, the grammar contains different non-terminals that encode the precedence levels of each operator.

2016. XXXX-XXXX/2016/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Deep Priority Conflicts in the Wild

https://doi.org/10.1145/3136014.3136020

Deep Priority Conflicts in the Wild: A Pilot Study

Luís Eduardo de Souza Amorim
Delft University of Technology
The Netherlands
l.e.desouzaamorim-1@tudelft.nl

Michael J. Steindorfer
Delft University of Technology
The Netherlands
m.j.steindorfer@tudelft.nl

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Abstract

Context-free grammars are suitable for formalizing the syntax of programming languages concisely and declaratively. Thus, such grammars are often found in reference manuals of programming languages, and used in language workbenches for language prototyping. However, the natural and concise way of writing a context-free grammar is often ambiguous.

Safe and complete declarative disambiguation of operator precedence and associativity conflicts guarantees that all ambiguities arising from combining the operators of the language are resolved. Ambiguities can occur due to *shallow conflicts*, which can be captured by one-level tree patterns, and *deep conflicts*, which require more elaborate techniques. Approaches to solve deep priority conflicts include grammar transformations, which may result in large unambiguous grammars, or may require adapted parser technologies to include data-dependency tracking at parse time.

In this paper we study deep priority conflicts *"in the wild"*. We investigate the efficiency of grammar transformations to solve deep priority conflicts by using a lazy parse table generation technique. On top of lazily-generated parse tables, we define metrics, aiming to answer how often deep priority conflicts occur in real-world programs and to what extent programmers explicitly disambiguate programs themselves. By applying our metrics to a small corpus of popular open-source repositories we found that in OCaml, up to 17% of the source files contain deep priority conflicts.

CCS Concepts • **Software and its engineering → Syntax**; *Parsers*;

Keywords Disambiguation, operator precedence, declarative syntax definition, grammars, empirical study.

ACM Reference Format:
Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. 2017. Deep Priority Conflicts in the Wild: A Pilot Study. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136020

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136020

1 Introduction

In software engineering, the Don't Repeat Yourself (DRY) principle means that *"every piece of knowledge must have a single, unambiguous, authoritative representation within a system"* [11]. While in theory context-free grammars come close to fulfilling this principle for declaratively formalizing the syntax of a programming language, they still fail to deliver it in practice [13].

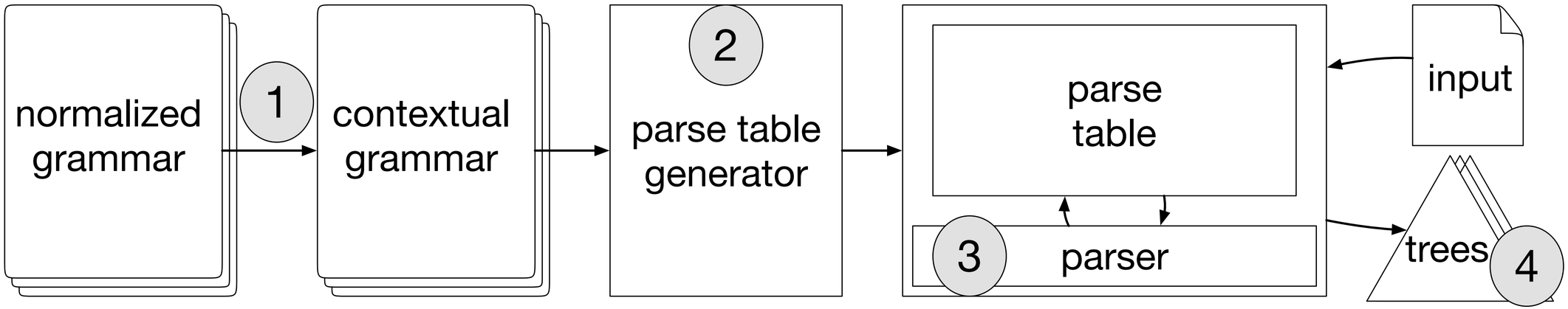
Natural and concise ways of writing a context-free grammar are often ambiguous and lead to Write Everything Twice (WET) solutions, i.e., the direct opposite of DRY. For example, the reference manual of the Java SE 7 edition [6] contains a natural and concise context-free reference grammar that describes the language, but a different grammar is used as the basis for the reference implementation. The refined Java SE 8 specification [7] contains a single unambiguous grammar, at the price of losing conciseness and readability.

A long-standing research topic in the parsing community is how to declaratively disambiguate concise expression grammars of programming languages. To address this issue, formalisms such as YACC [12] or SDF2 [22] extend context-free grammars with precedence and associativity declarations. In YACC, precedence is defined by a global ranking on the tokens of operators, and interpreted as choosing an alternative that solves a conflict in a parse table (i.e., a conflict should be resolved in favor of a specific action given a certain lookahead token). SDF2, on the other hand, constructs a partial order among productions using priority relations, deriving filters that reject conflicting patterns from the resulting tree. Because it supports the full class of context-free grammars and character-level grammars to enable modular syntax definitions and language composition, the YACC solution cannot be applied, which poses additional challenges when developing a solution to disambiguate SDF2 grammars.

Two desired properties for declarative disambiguation of precedence and associativity conflicts using SDF2 priorities are *safety* and *completeness*. To strive towards safety and

When to disambiguate?

Disambiguation Times using SDF3 and SGLR



Disambiguating Priority Conflicts

```
Exp.Mul = Exp "*" Exp
Exp.Add = Exp "+" Exp
Exp.Num = NUM
Exp     = "(" Exp ")"
```

NUM + NUM + NUM

```
amb(
  Add(Num(), Add(Num(), Num()))
, Add(Add(Num(), Num()), Num())
)
```

```
Exp.Mul = Exp "*" Exp
Exp.Add = Exp "+" Exp
Exp.Num = NUM
Exp     = "(" Exp ")"
```

NUM + NUM + NUM

```
amb(
  Add(Num(), Add(Num(), Num()))
, Add(Add(Num(), Num()), Num())
)
```

```
Exp.Add = Exp "+" • Exp
Exp.Mul = • Exp "*" Exp
Exp.Add = • Exp "+" Exp
Exp.Num = • NUM
Exp     = • "(" Exp ")"
```

Exp

```
Exp.Add = Exp "+" Exp •
Exp.Mul = Exp • "*" Exp
Exp.Add = Exp • "+" Exp
```

shift/reduce conflict

NUM + NUM + NUM

```
amb(
  Add(Num(), Add(Num(), Num()))
, Add(Add(Num(), Num()), Num())
)
```

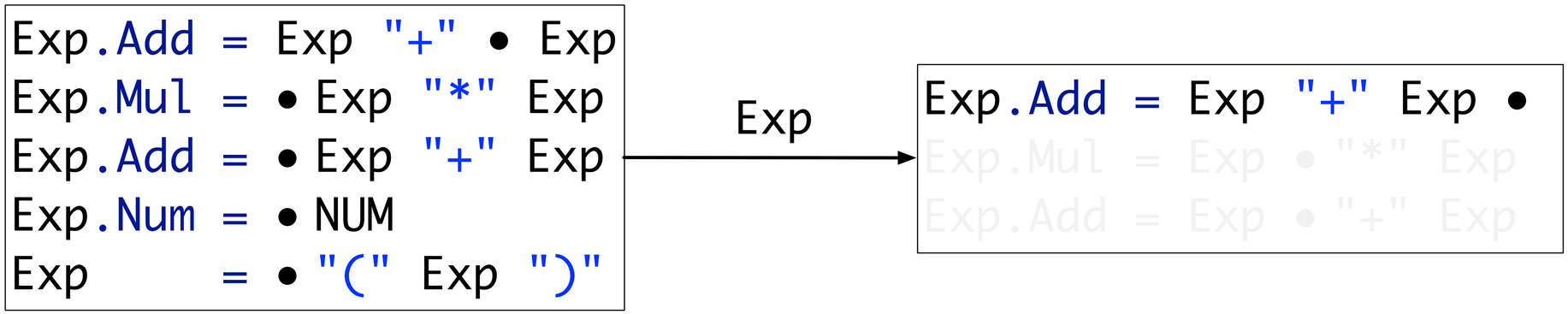
```
Exp.Add = Exp "+" • Exp
Exp.Mul = • Exp "*" Exp
Exp.Add = • Exp "+" Exp
Exp.Num = • NUM
Exp     = • "(" Exp ")"
```

Exp

```
Exp.Add = Exp "+" Exp •
Exp.Mul = Exp • "*" Exp
Exp.Add = Exp • "+" Exp
```

NUM + NUM + NUM

```
amb(
  Add(Num(), Add(Num(), Num()))
, Add(Add(Num(), Num()), Num())
)
```



```
Exp.Add    = Exp "+" Term
Exp        = Term
Term.Mul   = Term "*" Factor
Term       = Factor
Factor.Num = NUM
Factor     = "(" Exp ")"
```

NUM + NUM + NUM

```
Add(Add(Num(), Num()), Num())
```

```
Exp.Add    = Exp "+" Term
Exp        = Term
Term.Mul   = Term "*" Factor
Term       = Factor
Factor.Num = NUM
Factor     = "(" Exp ")"
```

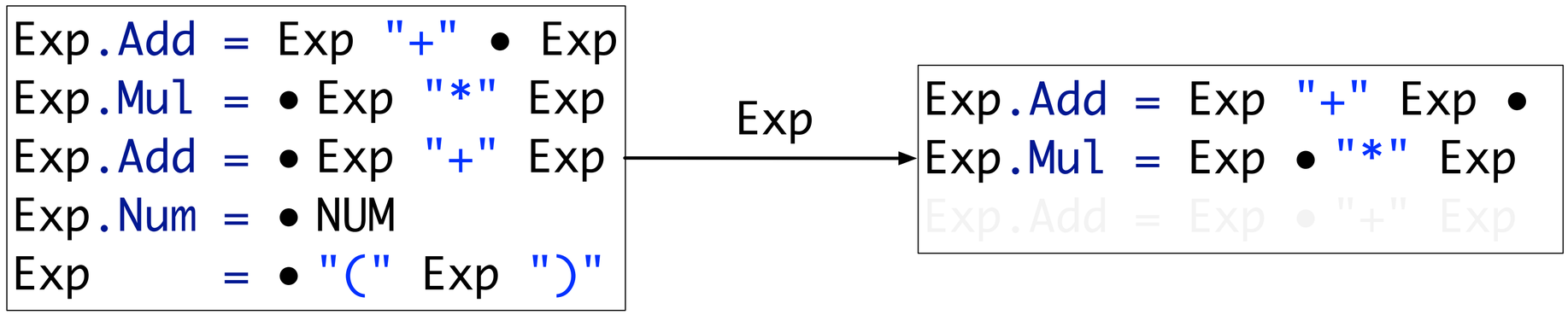
```
Exp.Add     = Exp "+" • Term
Term.Mul    = • Term "*" Factor
Term        = • Factor
Factor.Num  = • NUM
Factor      = • "(" Exp ")"
```

Term

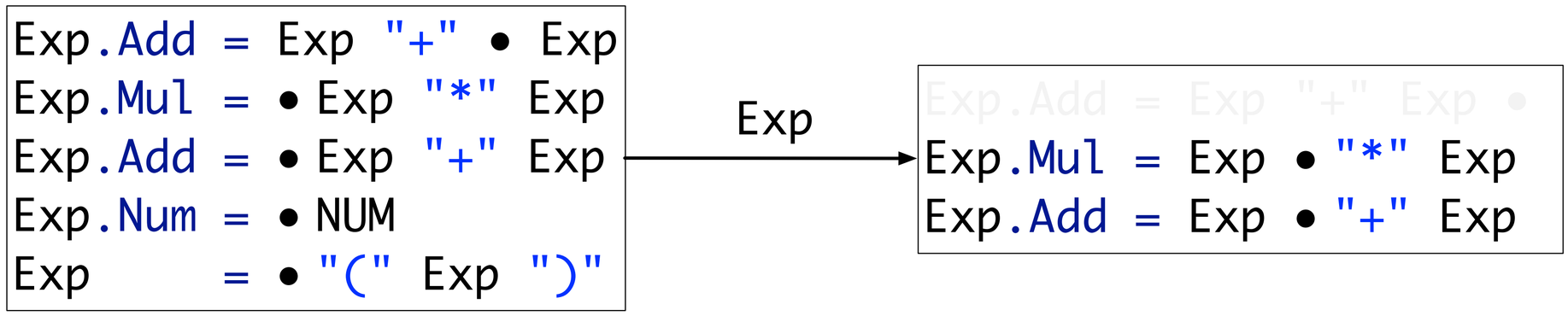
```
Exp.Add  = Exp "+" Term •
Term.Mul =  Term • "*" Factor
```

Disambiguation in YACC

```
%left  '+'
%left  '*'

%%

expr    :       expr  '+'  expr
        |       expr  '*'  expr
        |       '(' expr ')'
        |       INT
        ;
```




```
%left  '+' => when '+' is the lookahead token, and
there is a shift/reduce conflict, reduce.
```




```
%right  '+' => when '+' is the lookahead token,
and there is a shift/reduce conflict, shift.
```

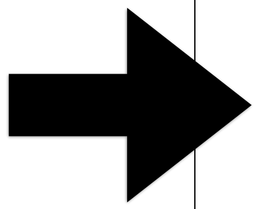
```
Exp.Add = Exp Layout? "+" Layout? • Exp
Exp.Mul = • Exp Layout? "*" Layout? Exp
Exp.Add = • Exp Layout? "+" Layout? Exp
Exp.Num = • NUM
Exp     = • "(" Layout? Exp Layout? ")"
```

Exp

```
Exp.Add = Exp Layout? "+" Layout? Exp •
Exp.Mul = Exp • Layout? "*" Layout? Exp
Exp.Add = Exp • Layout? "+" Layout? Exp
Layout? = ...
```

With scannerless parsers, the lookahead token represent next character.
Because layout is explicit, the next token/character might be "hidden" by Layout.

Disambiguation in SDF3

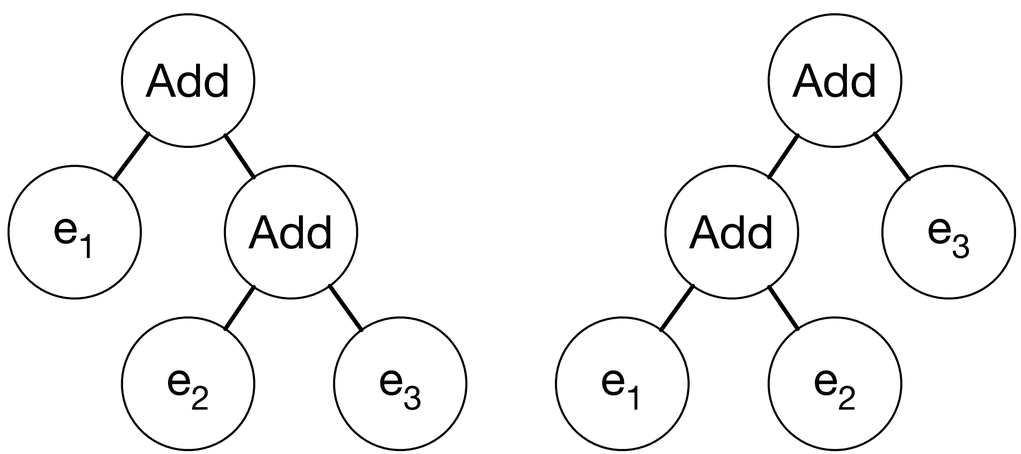


```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

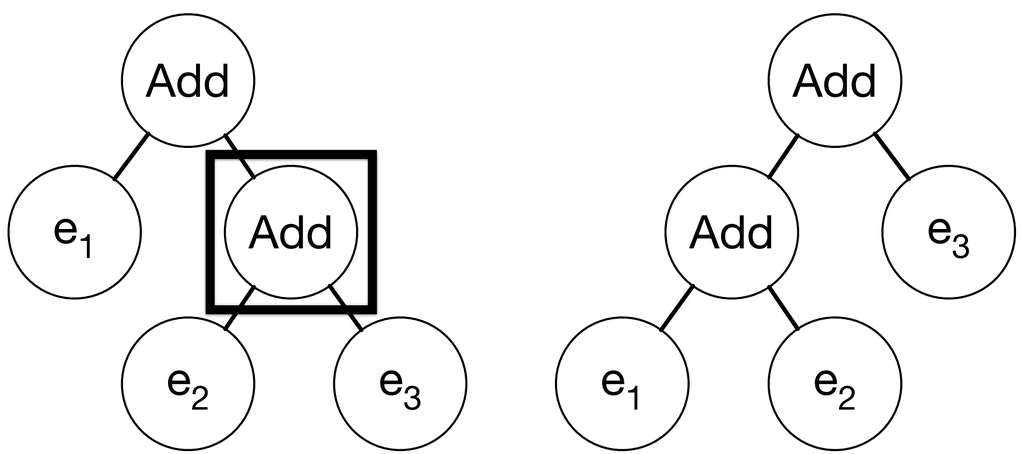
$e_1 + e_2 + e_3$



Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

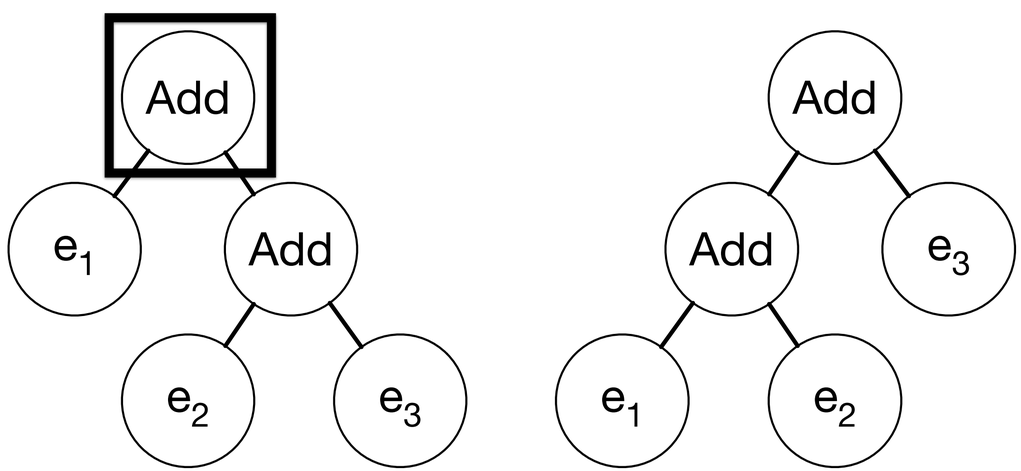
$e_1 + e_2 + e_3$



Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

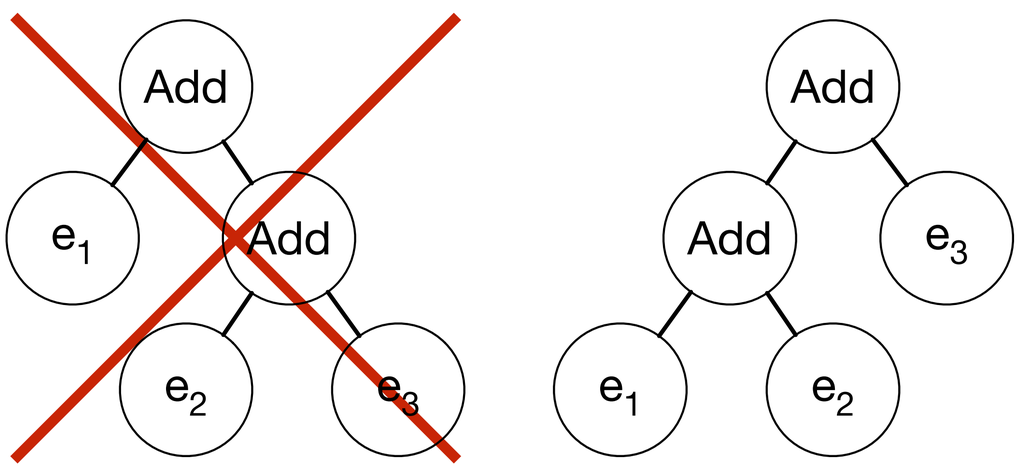
$e_1 + e_2 + e_3$



Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

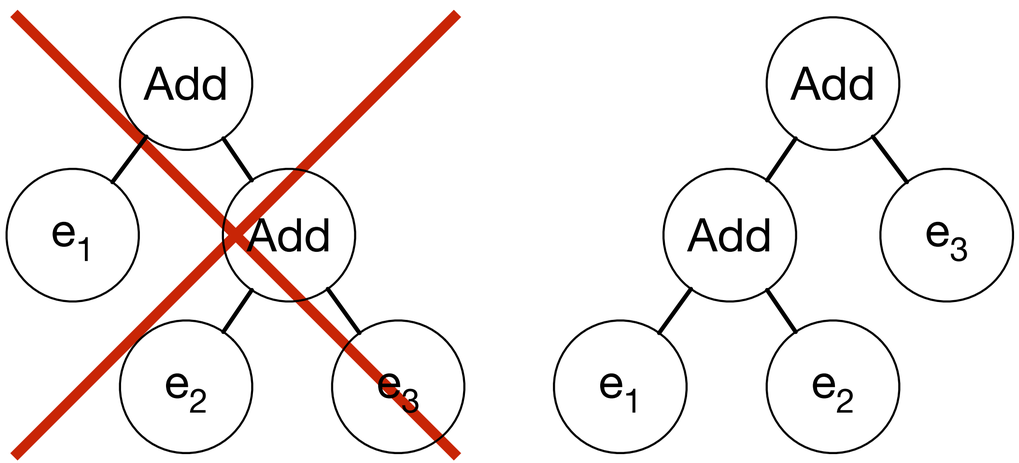
$e_1 + e_2 + e_3$



Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

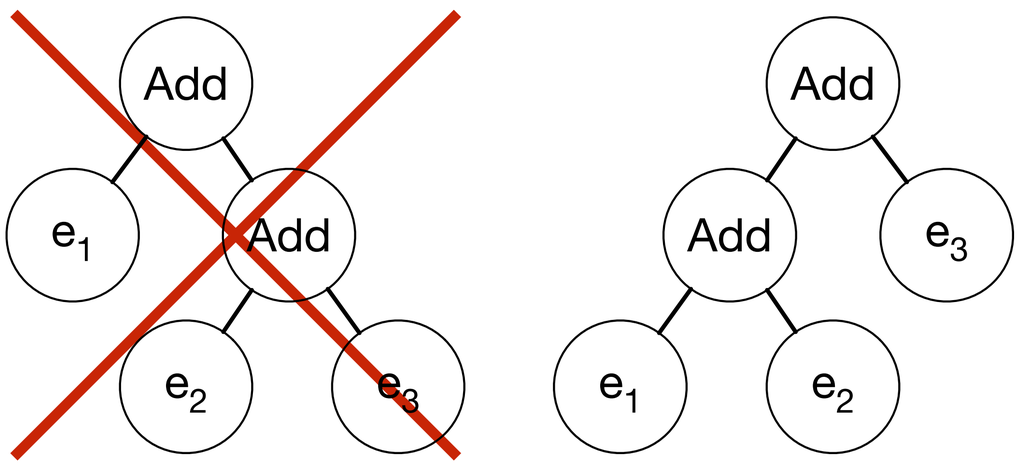
$e_1 + e_2 + e_3$



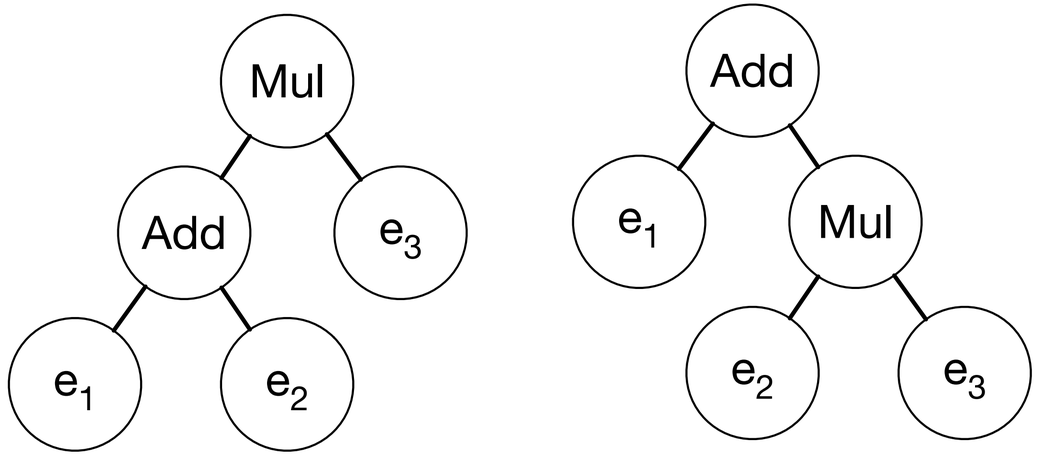
Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

$e_1 + e_2 + e_3$



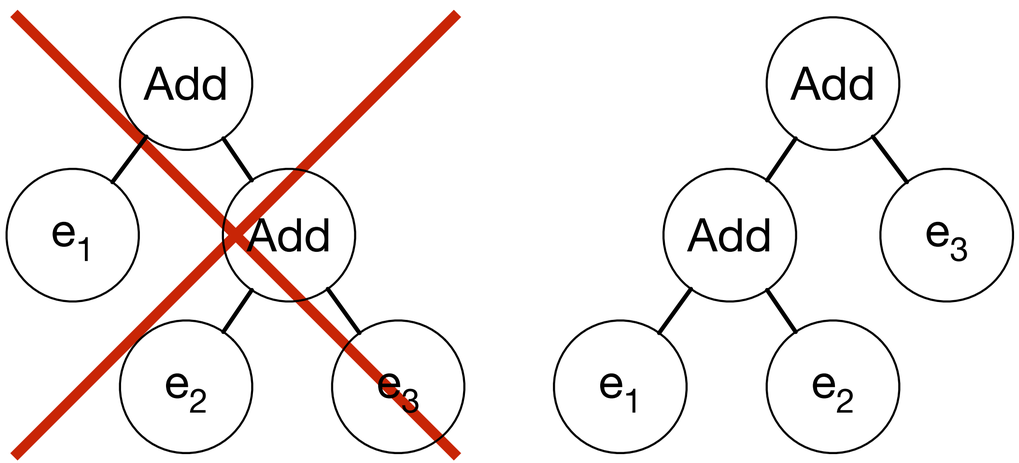
$e_1 + e_2 * e_3$



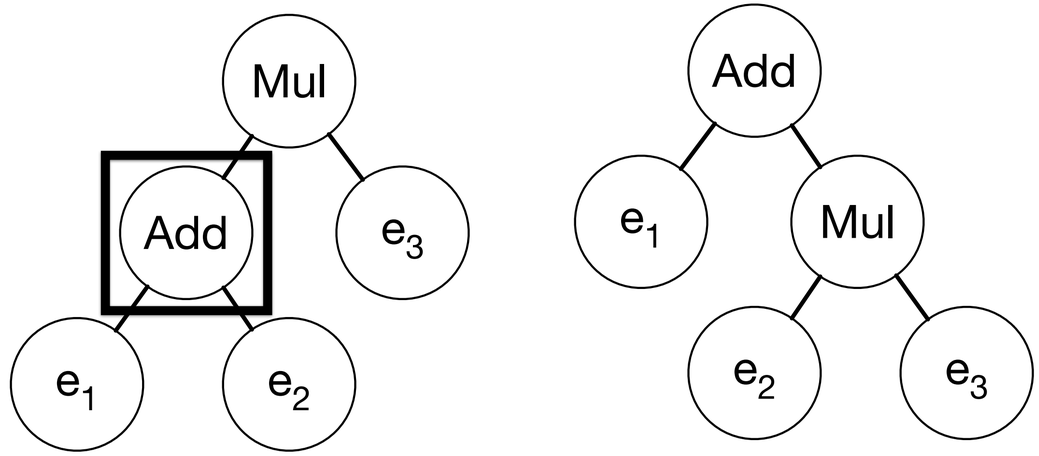
Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

$e_1 + e_2 + e_3$



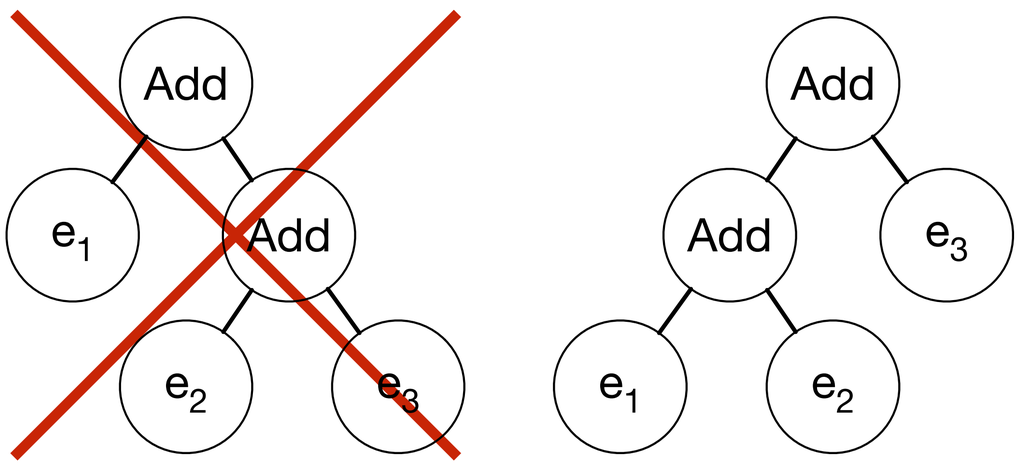
$e_1 + e_2 * e_3$



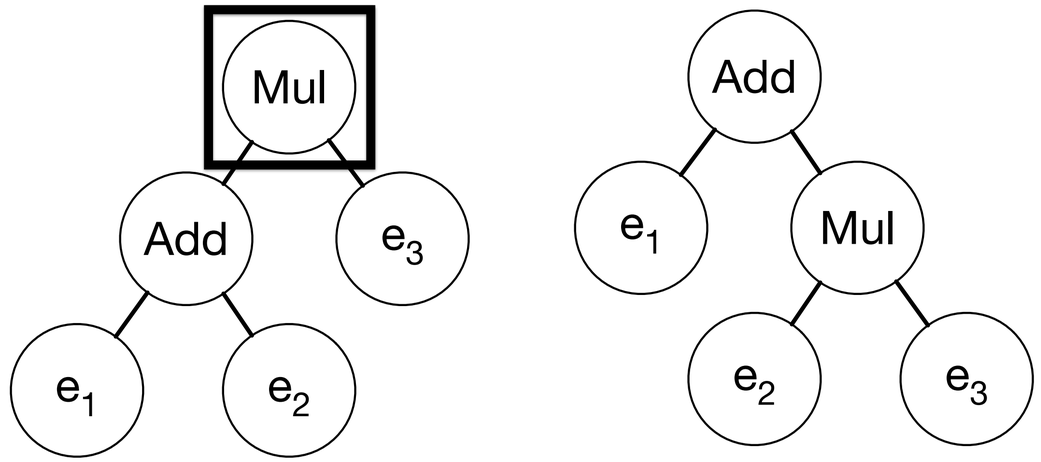
Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

$e_1 + e_2 + e_3$



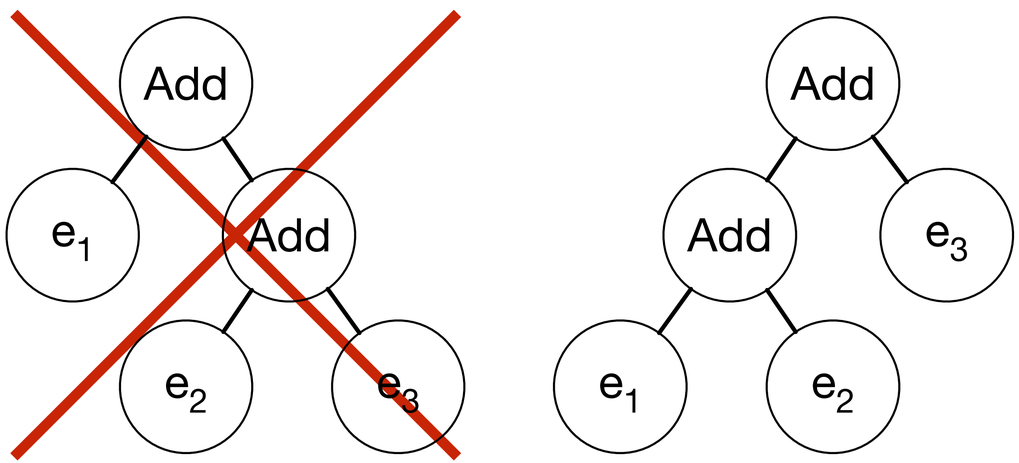
$e_1 + e_2 * e_3$



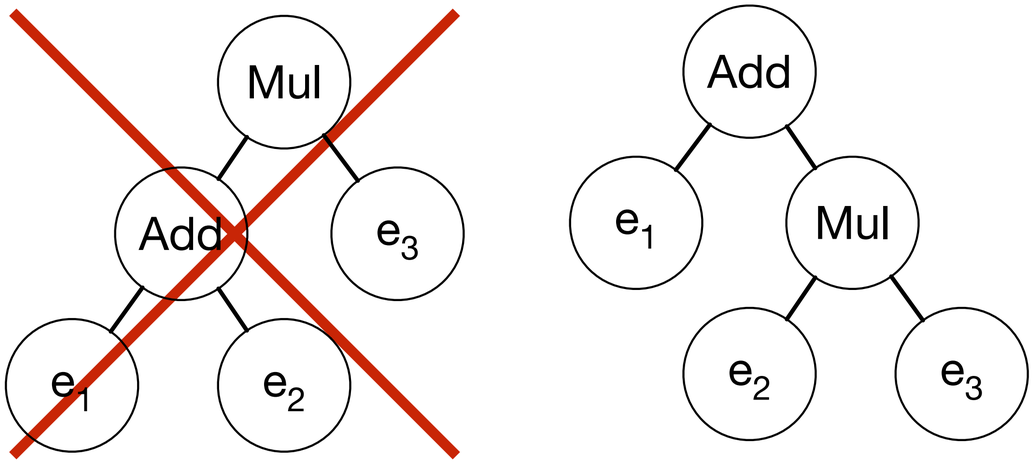
Disambiguation in SDF3

```
context-free syntax

  Exp.Add = Exp "+" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}

context-free priorities

  Exp.Mul > Exp.Add
```

$e_1 + e_2 + e_3$



$e_1 + e_2 * e_3$



```
Exp.Add = Exp Layout? "+" Layout? • Exp
Exp.Mul = • Exp Layout? "*" Layout? Exp
Exp.Add = • Exp Layout? "+" Layout? Exp
Exp.Num = • NUM
Exp     = • "(" Layout? Exp Layout? ")"
```

```
Exp.Mul
Exp.Num
Exp {bracket}
```

```
Exp.Add = Exp Layout? "+" Layout? Exp •
Exp.Mul = Exp • Layout? "*" Layout? Exp
Layout  = • ...
```

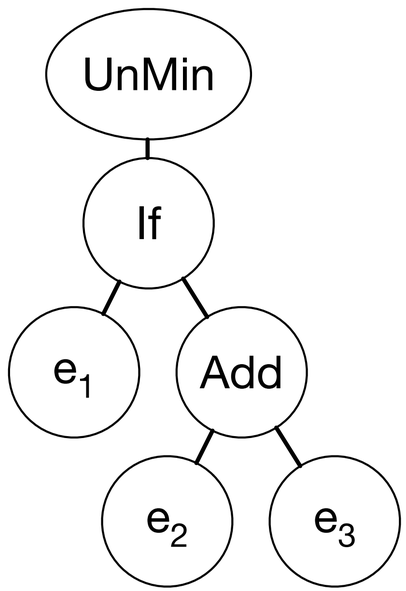
Deep Priority Conflicts

Operator-style

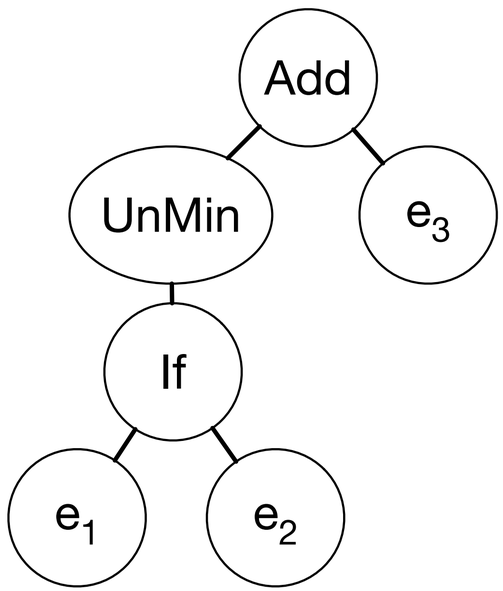
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```

Deep Priority Conflict!

- if e_1 then e_2 + e_2



- if e_1 then (e_2 + e_3)



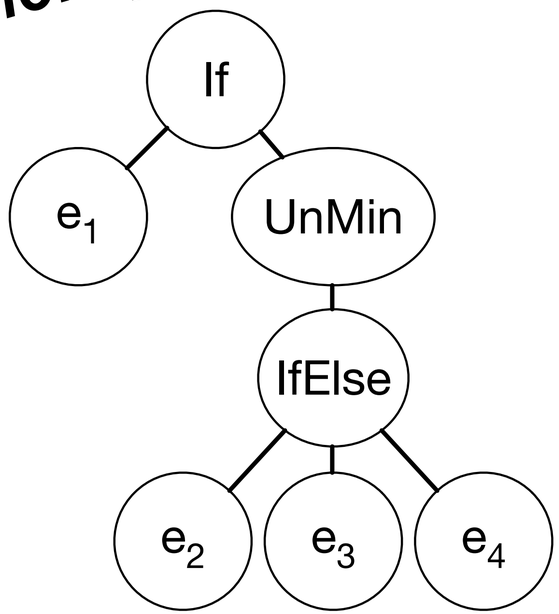
- (if e_1 then e_2) + e_3

Dangling Else

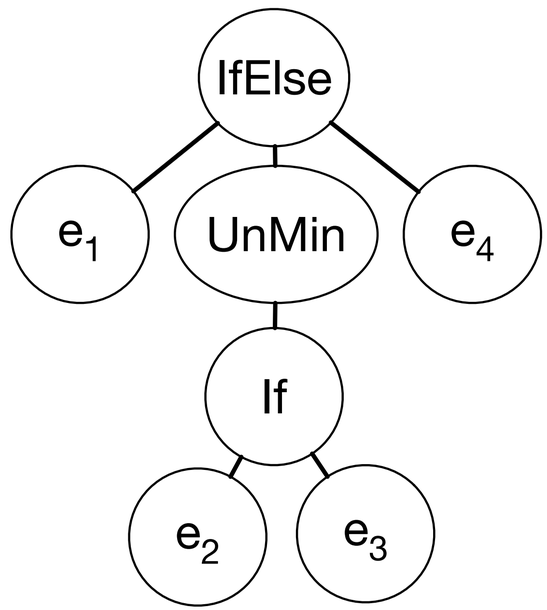
```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Int    = INT
```

`if` e_1 `then` `-` `if` e_3 `then` e_4 `else` e_5

Deep Priority Conflict!



`if` e_1 `then` `-` `(if` e_2 `then` e_3 `else` e_4`)`



`if` e_1 `then` `-` `(if` e_2 `then` e_3`)` `else` e_4

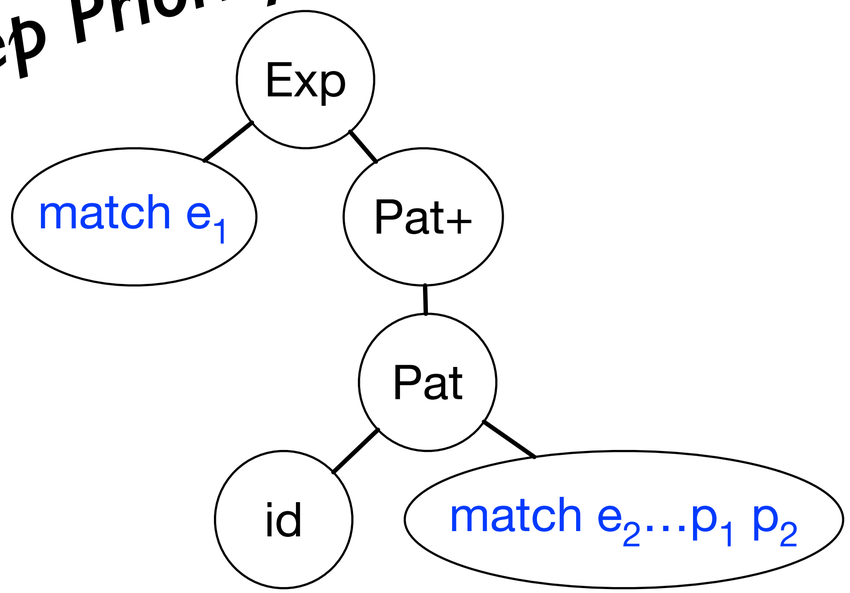
Longest Match

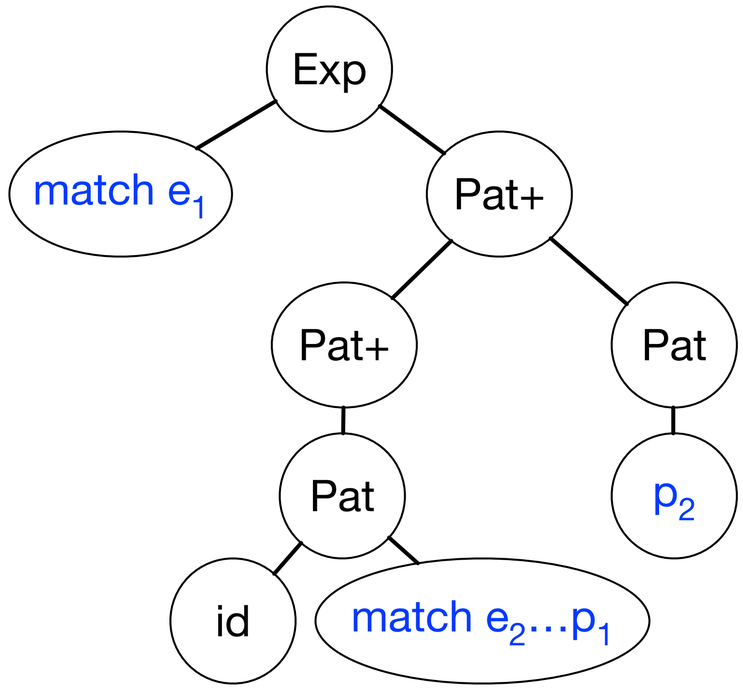
```
context-free syntax

  Exp.Match   = "match" Exp "with" Pat+
  Pat.Pattern = ID "->" Exp
  Exp.UnMin   = "-" Exp
  Exp.Int     = INT
```

```
Pat+ = Pat+ Pat
Pat+ = Pat
```

match e_1 with id -> match e_2 with p_1 p_2

Deep Priority Conflict!





match e_1 with id -> (match e_2 with p_1 p_2)

match e_1 with id -> (match e_2 with p_1) p_2

Contextual Grammars

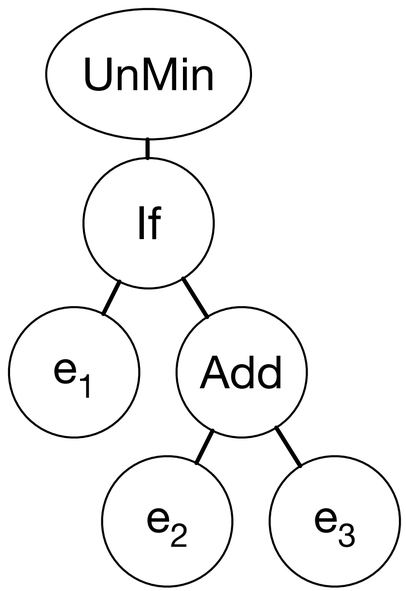
Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```

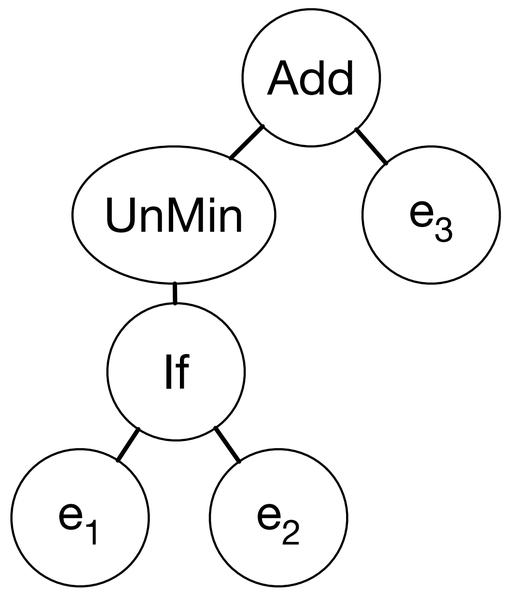
Priority Conflict!

- if e_1 then e_2 + e_2

Deep Priority Conflict!



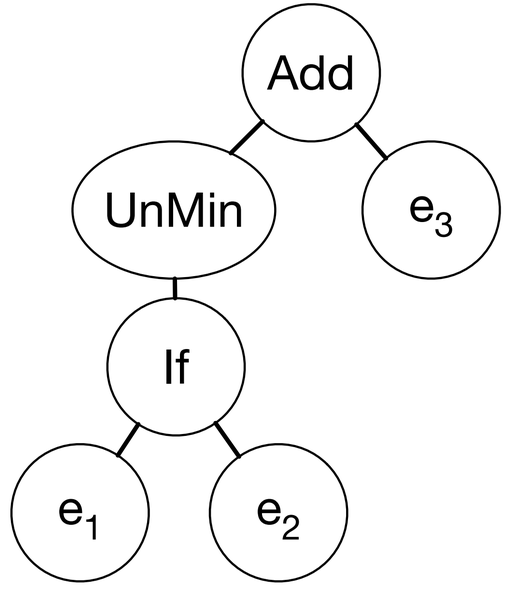
- if e_1 then (e_2 + e_2)



- (if e_1 then e_2) + e_3

Operator-style

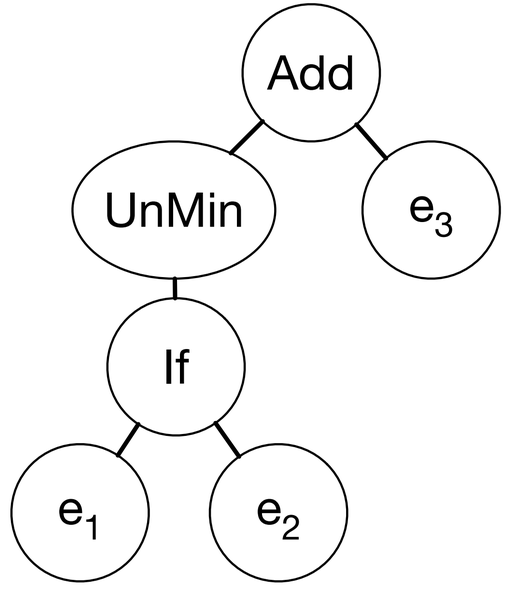
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



- (if e_1 then e_2) + e_3

Operator-style

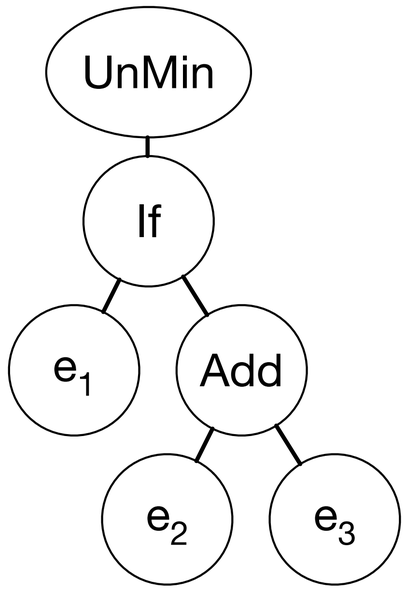
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

  Exp{Exp.If}.UnMin = "-" Exp{Exp.If}
  Exp{Exp.If}.Add   = Exp{Exp.If} "+" Exp{Exp.If} {left}
  Exp{Exp.If}.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



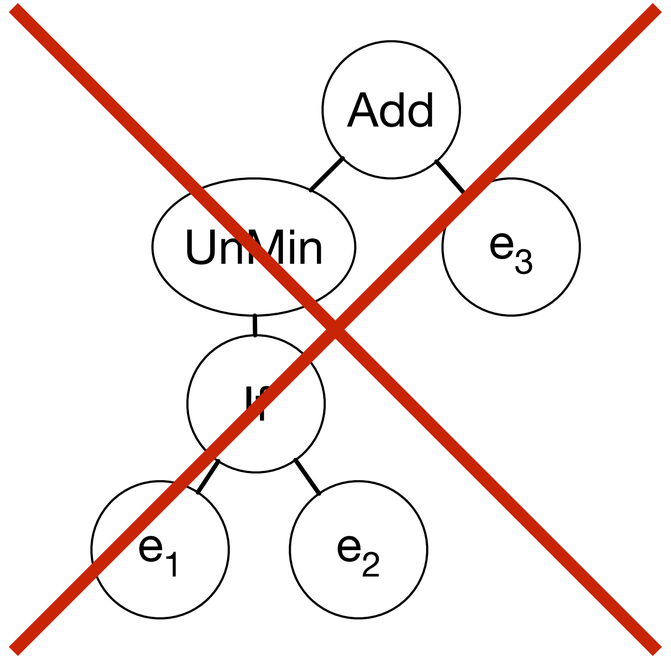
- (if e_1 then e_2) + e_3

Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

  Exp{Exp.If}.UnMin = "-" Exp{Exp.If}
  Exp{Exp.If}.Add   = Exp{Exp.If} "+" Exp{Exp.If} {left}
  Exp{Exp.If}.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



- `if e1 then (e2 + e3)`



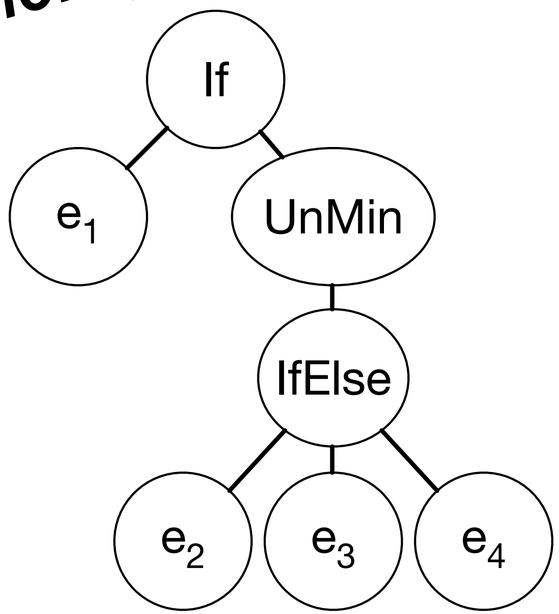
- `(if e1 then e2) + e3`

Dangling Else

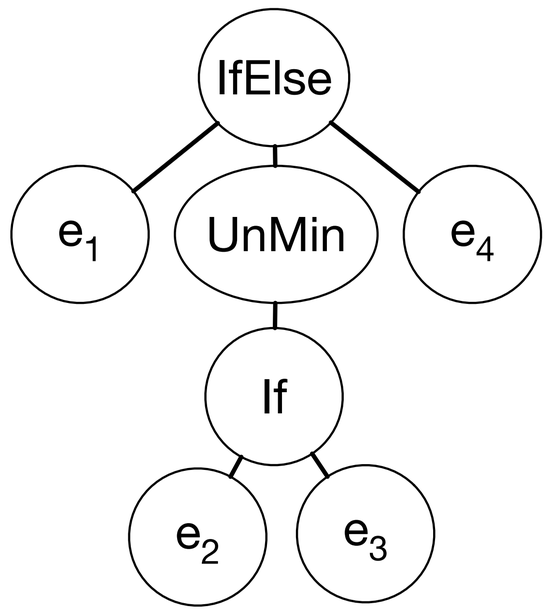
```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Int    = INT
```

`if` e_1 `then` `-` `if` e_3 `then` e_4 `else` e_5

Deep Priority Conflict!



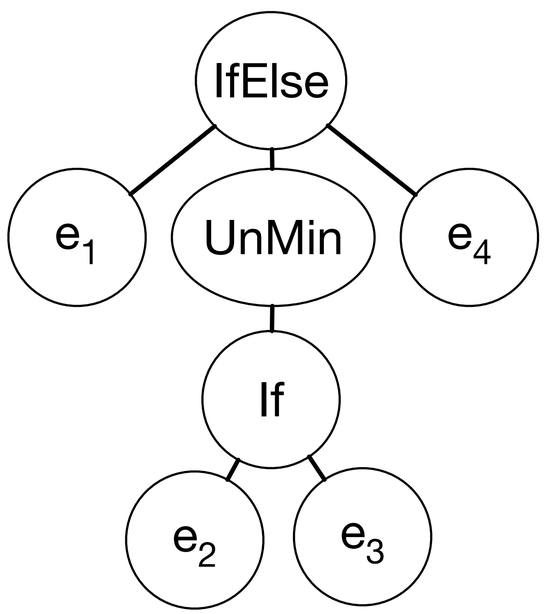
`if` e_1 `then` `-` `(if` e_2 `then` e_3 `else` e_4`)`



`if` e_1 `then` `-` `(if` e_2 `then` e_3`)` `else` e_4

Dangling Else

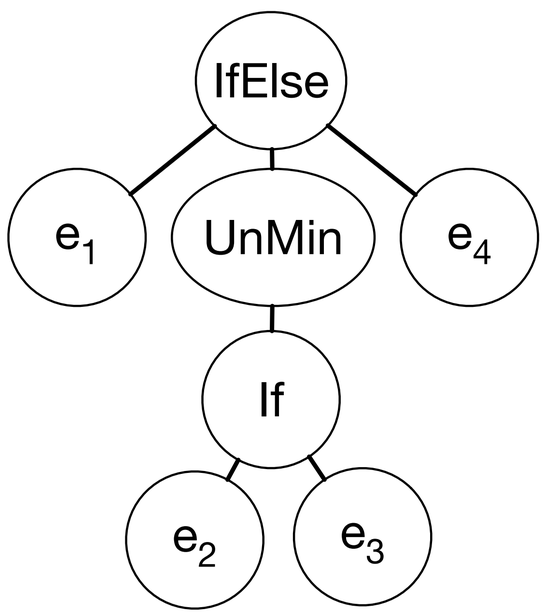
```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Int    = INT

context-free priorities

  Exp.IfElse > Exp.If
```



`if` e_1 `then -` `(``if` $e_2$ `then` $e_3$`)` `else` e_4

Dangling Else

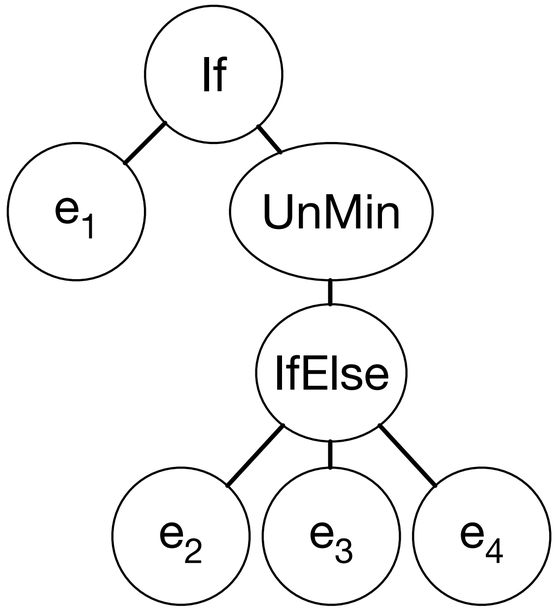
```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Int    = INT

  Exp{Exp.If}.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp{Exp.If}
  Exp{Exp.If}.UnMin  = "-" Exp{Exp.If}
  Exp{Exp.If}.Int    = INT

context-free priorities

  Exp.IfElse > Exp.If
```



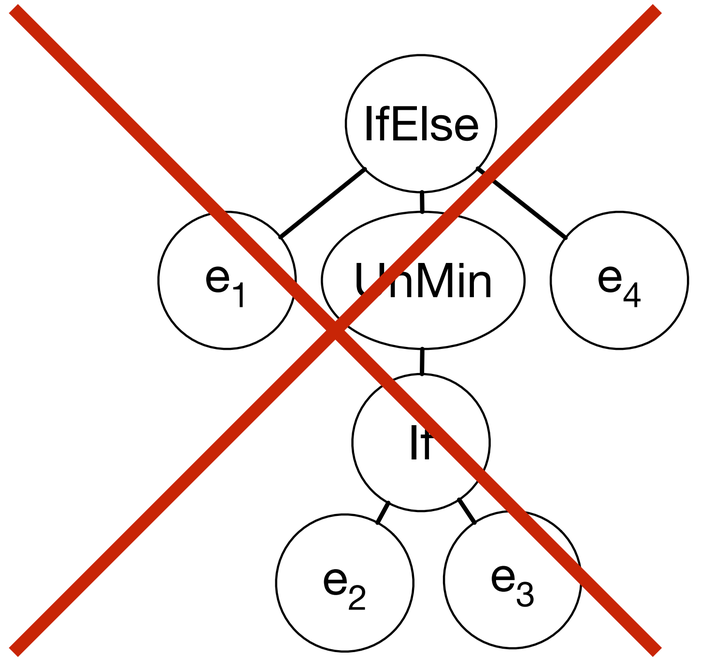

```
if e1 then - (if e2 then e3) else e4
```

Dangling Else

```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Int    = INT

  Exp{Exp.If}.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp{Exp.If}
  Exp{Exp.If}.UnMin  = "-" Exp{Exp.If}
  Exp{Exp.If}.Int    = INT

context-free priorities

  Exp.IfElse > Exp.If
```



`if e1 then - (if e2 then e3 else e4)`



`if e1 then - (if e2 then e3) else e4`

$G_{\{OS,\ DE\}}$

```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp {left}
  Exp.Int    = INT

  Exp{Exp.If}.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp{Exp.If}
  Exp{Exp.If}.UnMin  = "-" Exp{Exp.If}
  Exp{Exp.If}.Int    = INT
  Exp{Exp.If}.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp{Exp.If} {left}

  Exp{Exp.If, Exp.IfElse}.UnMin  = "-" Exp{Exp.If, Exp.IfElse}
  Exp{Exp.If, Exp.IfElse}.Int    = INT
  Exp{Exp.If, Exp.IfElse}.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp{Exp.If, Exp.IfElse} {left}


context-free priorities

  Exp.UnMin > Exp.Add > Exp.IfElse > Exp.If
```

Deep Priority Conflicts in the Wild

TU Delft

Experimental Setup

Java 8

OCaml 4.04

10 top trending projects

3296 OCaml files
9935 Java files

https://github.com/MetaBorgCube/deep-conflicts-in-the-wild

Research Questions

1. *To what extent do deep priority conflicts occur in real-world programs?*

2. *How do deep priority conflicts impact the efficiency of declarative disambiguation techniques that rely on grammar transformations?*

3. *To what extent do programmers use brackets for disambiguating priority conflicts explicitly?*

$G_{\{OS,\ DE\}}$

```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp {left}
  Exp.Int    = INT

  Exp{Exp.If}.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp{Exp.If}
  Exp{Exp.If}.UnMin  = "-" Exp{Exp.If}
  Exp{Exp.If}.Int    = INT
  Exp{Exp.If}.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp{Exp.If} {left}

  Exp{Exp.If, Exp.IfElse}.UnMin  = "-" Exp{Exp.If, Exp.IfElse}
  Exp{Exp.If, Exp.IfElse}.Int    = INT
  Exp{Exp.If, Exp.IfElse}.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp{Exp.If, Exp.IfElse} {left}


context-free priorities

  Exp.UnMin > Exp.Add > Exp.IfElse > Exp.If
```

Hypotheses

H1 More ambiguities in OCaml programs because of its expression-oriented grammar.

H2 The majority of conflicts in OCaml are longest match because many expressions can contain pattern matches.

H3 Deep conflicts are sparse in Java.

H4 Overall, deep conflicts are sparse and do not occur frequently across programs of both languages.

Project	Affected Files
Matisse	0/41
RxJava	0/1469
aurora-imui	0/55
gitpitch	0/45
kotlin	0/3854
leetcode	0/94
litho	0/510
lottie-android	0/109
spring-boot	2/3444
vlayout	0/46
All	**2/9667**

Project	Affected Files	Deep Priority Conflicts			
		Total Number	Operator Style	Dangling Else	Longest Match
FStar	6/160	6	33%	0%	67%
bincat	5/26	26	58%	0%	42%
bucklescript	85/885	305	50%	1%	49%
coq	158/417	441	35%	1%	64%
flow	52/305	117	37%	0%	63%
infer	33/234	52	23%	0%	77%
ocaml	112/909	275	28%	1%	71%
reason	4/36	14	7%	0%	93%
spec	4/40	5	100%	0%	0%
tezos	71/149	416	79%	0%	21%
All	**530/3161**	**1657**	**48%**	**1%**	**51%**

Project	Affected Files
Matisse	0/41
RxJava	0/1469
aurora-imui	0/55
gitpitch	0/45
kotlin	0/3854
leetcode	0/94
litho	0/510
lottie-android	0/109
spring-boot	2/3444
vlayout	0/46
All	**2/9667**

Project	Affected Files	Deep Priority Conflicts			
		Total Number	Operator Style	Dangling Else	Longest Match
FStar	6/160	6	33%	0%	67%
bincat	5/26	26	58%	0%	42%
bucklescript	85/885	305	50%	1%	49%
coq	158/417	441	35%	1%	64%
flow	52/305	117	37%	0%	63%
infer	33/234	52	23%	0%	77%
ocaml	112/909	275	28%	1%	71%
reason	4/36	14	7%	0%	93%
spec	4/40	5	100%	0%	0%
tezos	71/149	416	79%	0%	21%
All	**530/3161**	**1657**	**48%**	**1%**	**51%**

H1 - OCaml has more conflicts

Project	Affected Files
Matisse	0/41
RxJava	0/1469
aurora-imui	0/55
gitpitch	0/45
kotlin	0/3854
leetcode	0/94
litho	0/510
lottie-android	0/109
spring-boot	2/3444
vlayout	0/46
All	**2/9667**

Project	Affected Files	Deep Priority Conflicts			
		Total Number	Operator Style	Dangling Else	Longest Match
FStar	6/160	6	33%	0%	67%
bincat	5/26	26	58%	0%	42%
bucklescript	85/885	305	50%	1%	49%
coq	158/417	441	35%	1%	64%
flow	52/305	117	37%	0%	63%
infer	33/234	52	23%	0%	77%
ocaml	112/909	275	28%	1%	71%
reason	4/36	14	7%	0%	93%
spec	4/40	5	100%	0%	0%
tezos	71/149	416	79%	0%	21%
All	**530/3161**	**1657**	**48%**	**1%**	**51%**

H2 - More Longest Match Ambiguities

Project	Affected Files
Matisse	0/41
RxJava	0/1469
aurora-imui	0/55
gitpitch	0/45
kotlin	0/3854
leetcode	0/94
litho	0/510
lottie-android	0/109
spring-boot	2/3444
vlayout	0/46
All	**2/9667**

Project	Affected Files	Deep Priority Conflicts			
		Total Number	Operator Style	Dangling Else	Longest Match
FStar	6/160	6	33%	0%	67%
bincat	5/26	26	58%	0%	42%
bucklescript	85/885	305	50%	1%	49%
coq	158/417	441	35%	1%	64%
flow	52/305	117	37%	0%	63%
infer	33/234	52	23%	0%	77%
ocaml	112/909	275	28%	1%	71%
reason	4/36	14	7%	0%	93%
spec	4/40	5	100%	0%	0%
tezos	71/149	416	79%	0%	21%
All	**530/3161**	**1657**	**48%**	**1%**	**51%**

H3 - Deep Conflicts are sparse in Java

Project	Affected Files
Matisse	0/41
RxJava	0/1469
aurora-imui	0/55
gitpitch	0/45
kotlin	0/3854
leetcode	0/94
litho	0/510
lottie-android	0/109
spring-boot	2/3444
vlayout	0/46
All	**2/9667**

Project	Affected Files	Deep Priority Conflicts: Total Number	Operator Style	Dangling Else	Longest Match
FStar	6/160	6	33%	0%	67%
bincat	5/26	26	58%	0%	42%
bucklescript	85/885	305	50%	1%	49%
coq	158/417	441	35%	1%	64%
flow	52/305	117	37%	0%	63%
infer	33/234	52	23%	0%	77%
ocaml	112/909	275	28%	1%	71%
reason	4/36	14	7%	0%	93%
spec	4/40	5	100%	0%	0%
tezos	71/149	416	79%	0%	21%
All	**530/3161**	**1657**	**48%**	**1%**	**51%**

H4 - Deep Conflicts occur relatively often in OCaml

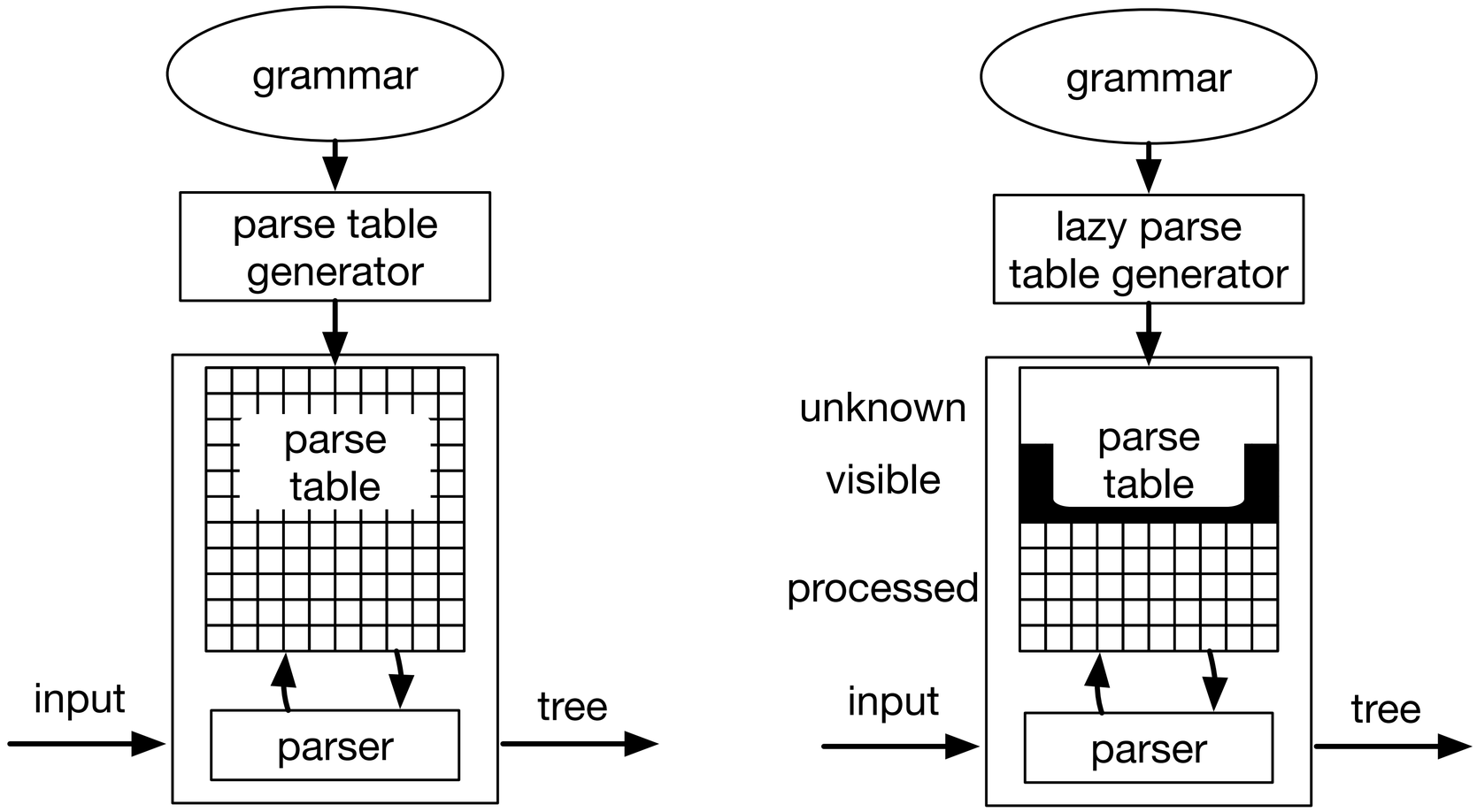
Research Questions

1. *To what extent do deep priority conflicts occur in real-world programs?*

2. *How do deep priority conflicts impact the efficiency of declarative disambiguation techniques that rely on grammar transformations?*

3. *To what extent do programmers use brackets for disambiguating priority conflicts explicitly?*

Lazy Parse Table Generation



Jan Heering, Paul Klint, and Jan Rekers. 1989. Incremental Generation of Parsers. In PLDI. 179–191.

Hypotheses

H5 For both languages, only (a minor) part of the grammar productions and parse table states are exercised, even after parsing all programs in the corpuses.

H6 Due to its expression-oriented syntax, OCaml programs contain considerably more brackets than Java programs.

H7 The majority of brackets in Java and OCaml are necessary to disambiguate shallow conflicts.

Grammar and Parse Table Coverage

	Grammar		Parse Table		
				Lazy Expansion	
	# Prod.	Used	# States	Proc.	Visible
Java	3420	59%	20200	36%	46%
OCaml	1916	55%	4674	49%	57%

Grammar and Parse Table Coverage

	Grammar			Parse Table	
				Lazy Expansion	
	# Prod.	Used	# States	Proc.	Visible
Java	3420	59%	20200	36%	46%
OCaml	1916	55%	4674	49%	57%

H5 - Only part of the parse states/productions are used

Research Questions

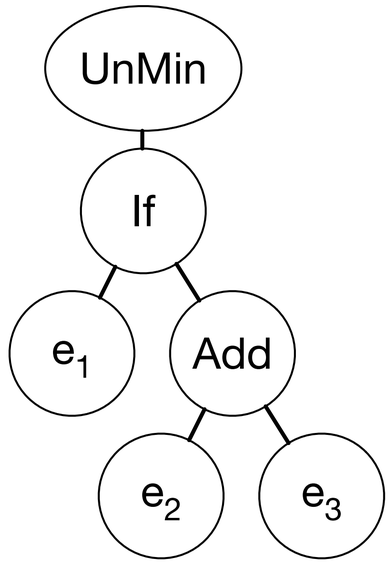
1. *To what extent do deep priority conflicts occur in real-world programs?*

2. *How do deep priority conflicts impact the efficiency of declarative disambiguation techniques that rely on grammar transformations?*

3. *To what extent do programmers use brackets for disambiguating priority conflicts explicitly?*

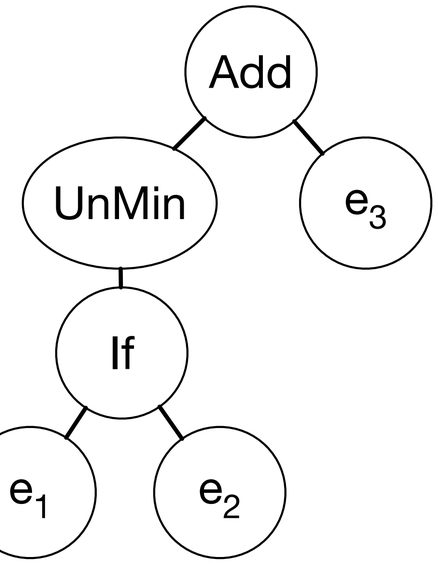
Counting Brackets

- `if` e_1 `then` e_2 `+` e_2



- `if` e_1 `then (`e_2 `+` e_3`)`

Redundant bracket



- `(if` e_1 `then` e_2`)` `+` e_3

Bracket that disambiguates a deep conflict

Hypotheses

H5 For both languages, only (a minor) part of the grammar productions and parse table states are exercised, even after parsing all programs in the corpuses.

H6 Due to its expression-oriented syntax, OCaml programs contain considerably more brackets than Java programs.

H7 The majority of brackets in Java and OCaml are necessary to disambiguate shallow conflicts.

Project	Disamb. with brackets		
	Redundant Brackets	Shallow Conflicts	Deep Conflicts
Matisse	2	33	0
RxJava	110	398	0
aurora-imui	20	57	0
gitpitch	62	1	0
kotlin	4286	4892	0
leetcode	37	30	0
litho	151	297	0
lottie-android	19	134	0
spring-boot	507	630	0
vlayout	89	285	0
All	**5283**	**6757**	**0**

Project	Disamb. with brackets		
	Redundant Brackets	Shallow Conflicts	Deep Conflicts
Matisse	21917	12487	607
RxJava	602	2735	28
aurora-imui	14589	29238	924
gitpitch	23106	56083	1039
kotlin	3307	13374	278
leetcode	4083	10720	376
litho	9502	35010	737
lottie-android	413	1194	25
spring-boot	194	1293	15
vlayout	454	6969	130
All	**78167**	**169103**	**4159**

Project	Disamb. with brackets		
	Redundant Brackets	Shallow Conflicts	Deep Conflicts
Matisse	2	33	0
RxJava	110	398	0
aurora-imui	20	57	0
gitpitch	62	1	0
kotlin	4286	4892	0
leetcode	37	30	0
litho	151	297	0
lottie-android	19	134	0
spring-boot	507	630	0
vlayout	89	285	0
All	**5283**	**6757**	**0**

Project	Disamb. with brackets		
	Redundant Brackets	Shallow Conflicts	Deep Conflicts
Matisse	21917	12487	607
RxJava	602	2735	28
aurora-imui	14589	29238	924
gitpitch	23106	56083	1039
kotlin	3307	13374	278
leetcode	4083	10720	376
litho	9502	35010	737
lottie-android	413	1194	25
spring-boot	194	1293	15
vlayout	454	6969	130
All	**78167**	**169103**	**4159**

H6 - More brackets in OCaml

Project	Disamb. with brackets		
	Redundant Brackets	Shallow Conflicts	Deep Conflicts
Matisse	2	33	0
RxJava	110	398	0
aurora-imui	20	57	0
gitpitch	62	1	0
kotlin	4286	4892	0
leetcode	37	30	0
litho	151	297	0
lottie-android	19	134	0
spring-boot	507	630	0
vlayout	89	285	0
All	**5283**	**6757**	**0**

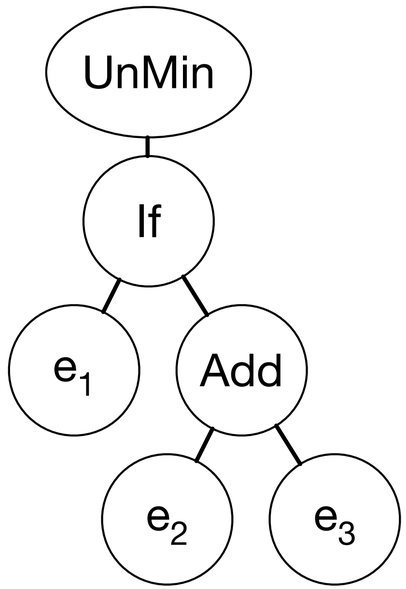
Project	Disamb. with brackets		
	Redundant Brackets	Shallow Conflicts	Deep Conflicts
Matisse	21917	12487	607
RxJava	602	2735	28
aurora-imui	14589	29238	924
gitpitch	23106	56083	1039
kotlin	3307	13374	278
leetcode	4083	10720	376
litho	9502	35010	737
lottie-android	413	1194	25
spring-boot	194	1293	15
vlayout	454	6969	130
All	**78167**	**169103**	**4159**

H7 - More brackets to disambiguate shallow conflicts

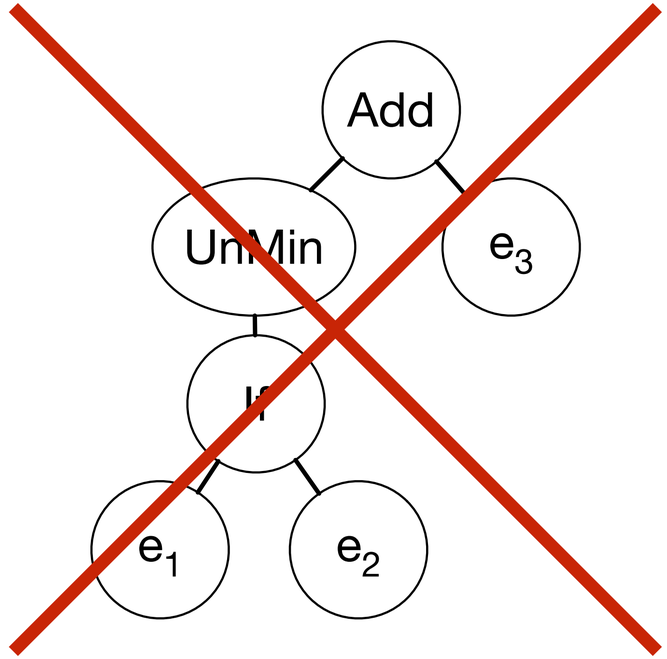
Data-dependent Contextual Grammars

Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

  Exp{Exp.If}.UnMin = "-" Exp{Exp.If}
  Exp{Exp.If}.Add   = Exp{Exp.If} "+" Exp{Exp.If} {left}
  Exp{Exp.If}.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



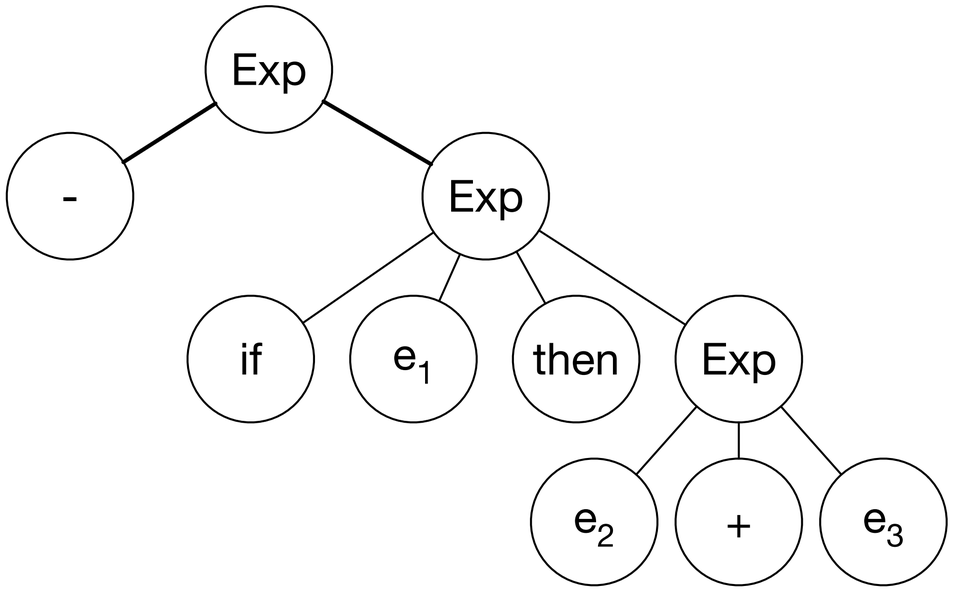
- `if` e_1 `then` (`(`e_2 `+` e_3`)`



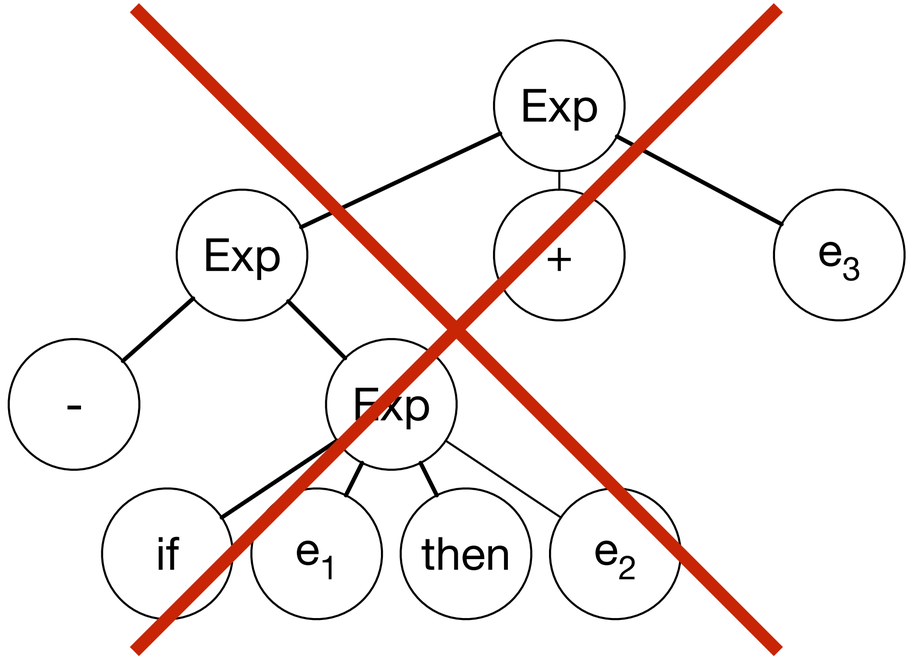
- `(``if` $e_1$ `then` $e_2$`)` `+` e_3

Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

  Exp{Exp.If}.UnMin = "-" Exp{Exp.If}
  Exp{Exp.If}.Add   = Exp{Exp.If} "+" Exp{Exp.If} {left}
  Exp{Exp.If}.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



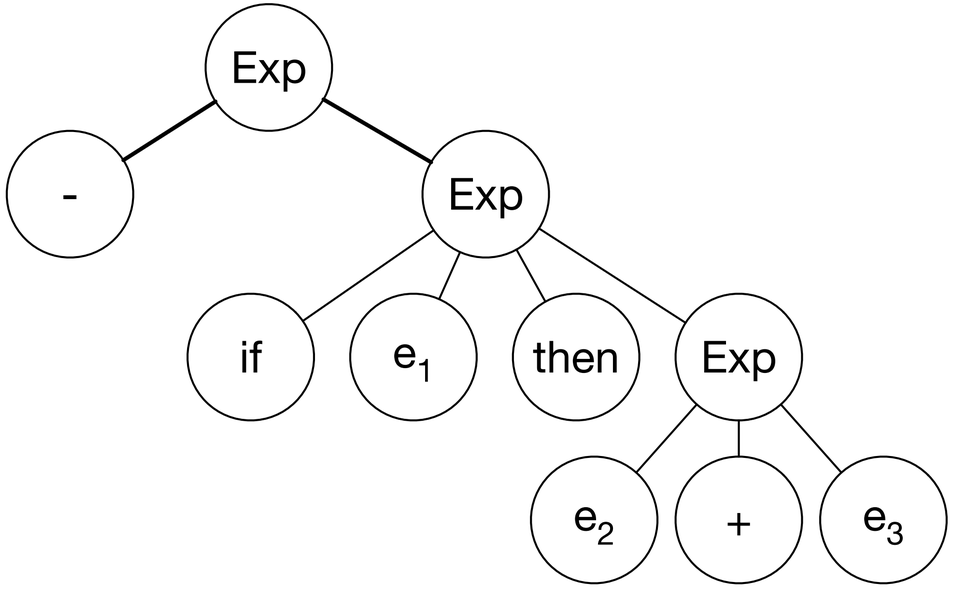

```
- if e1 then (e2 + e2)
```



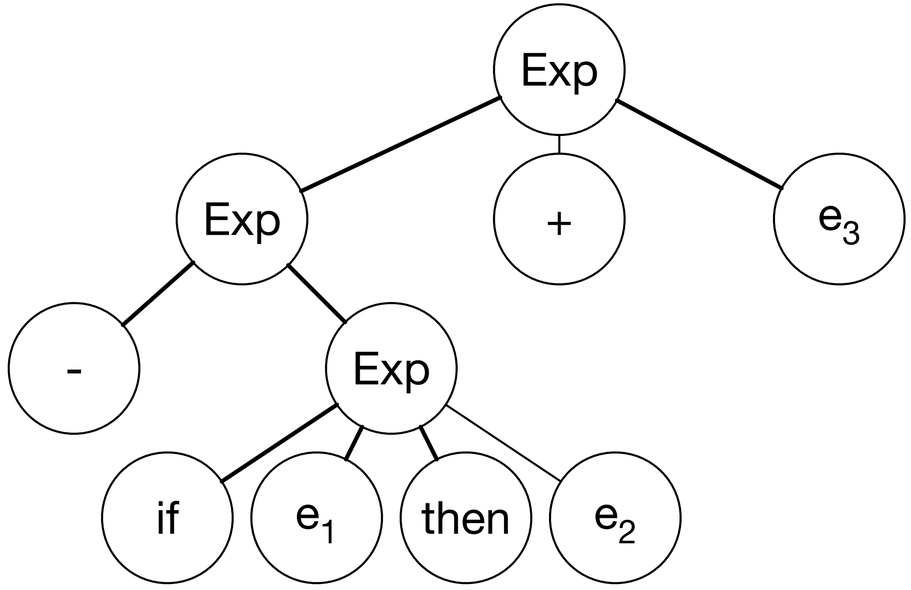

```
- (if e1 then e2) + e2
```

Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



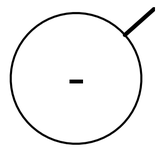
- if e1 then (e2 + e2)



- (if e1 then e2) + e2

Operator-style

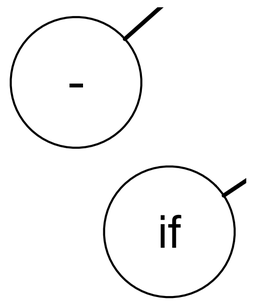
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



- (if e_1 then e_2) + e_2

Operator-style

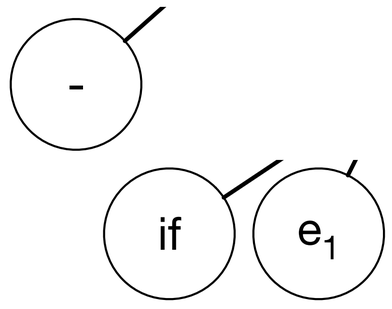
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



- (if e_1 then e_2) + e_2

Operator-style

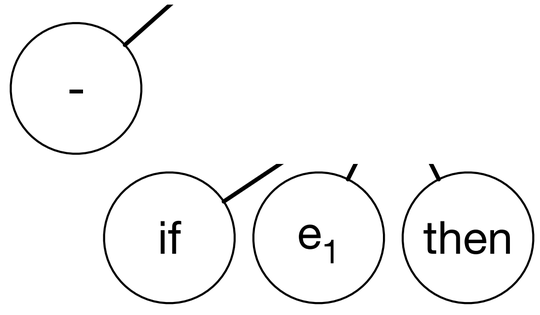
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



- (if e_1 then e_2) + e_2

Operator-style

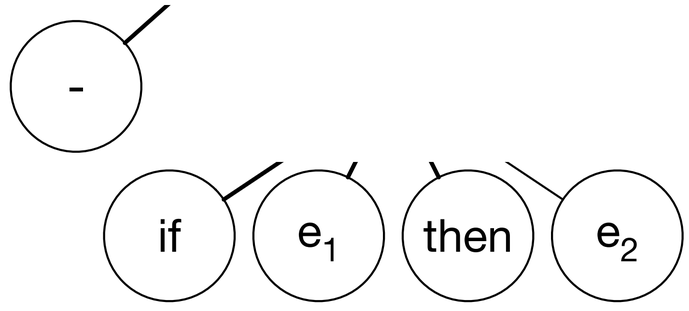
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```




```
- (if e1 then e2) + e2
```

Operator-style

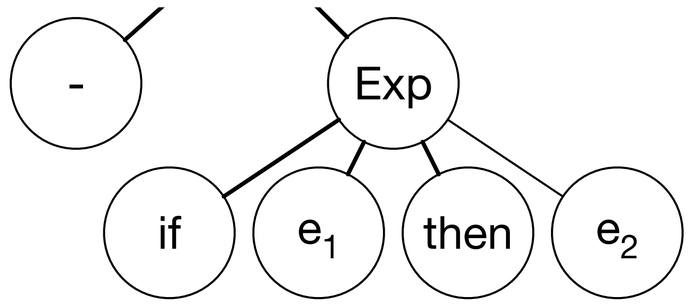
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```




```
- (if e1 then e2) + e2
```

Operator-style

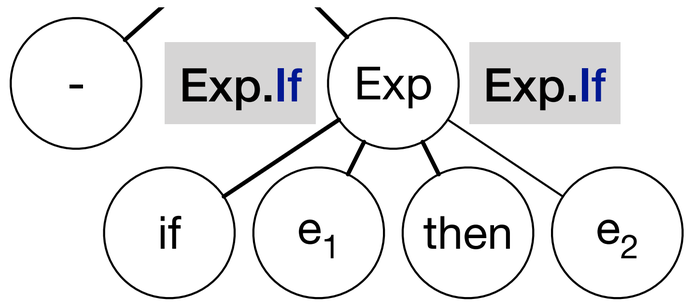
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```




```
- (if e1 then e2) + e2
```

Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```




```
- (if e1 then e2) + e2
```

Operator-style

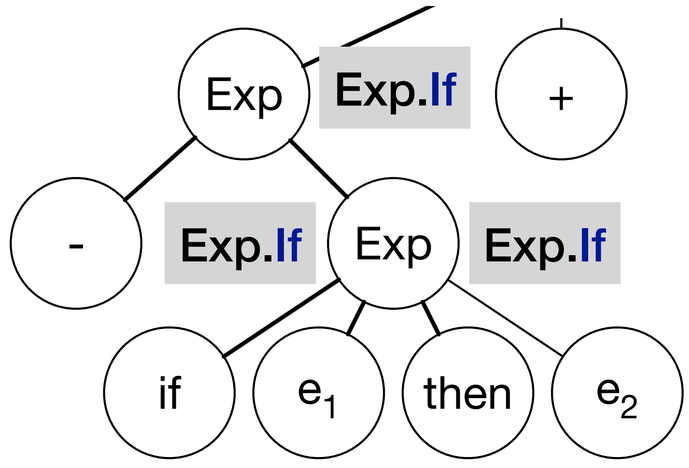
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



```
- (if e1 then e2) + e2
```

Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```




```
- (if e1 then e2) + e2
```

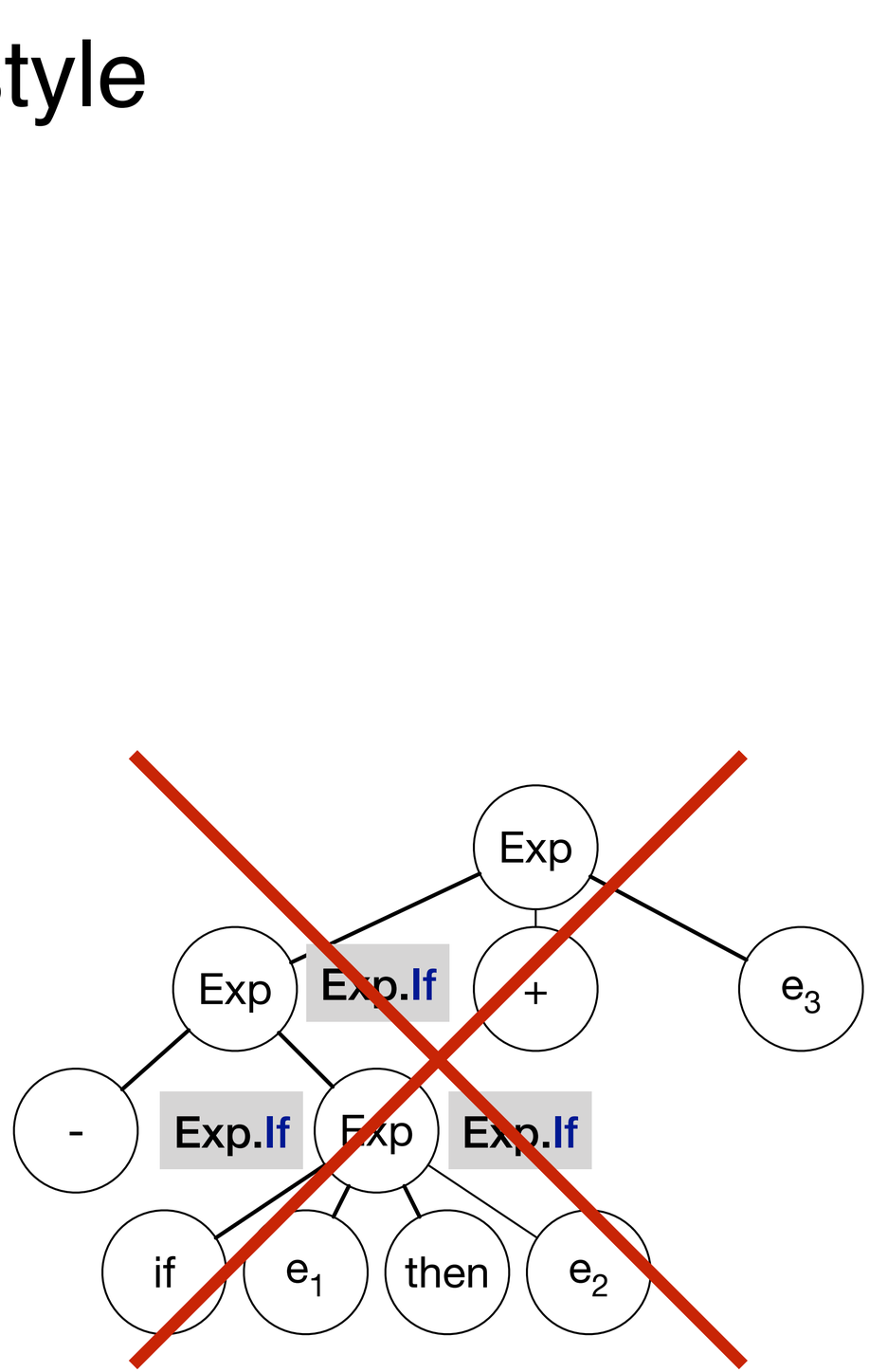
Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```

- (if e_1 then e_2) + e_2

Operator-style

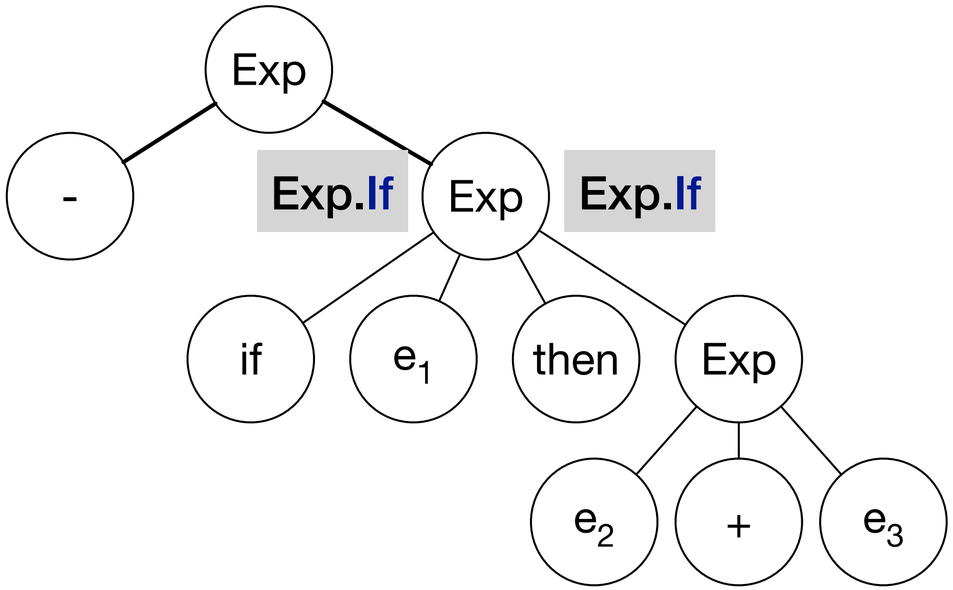
```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```



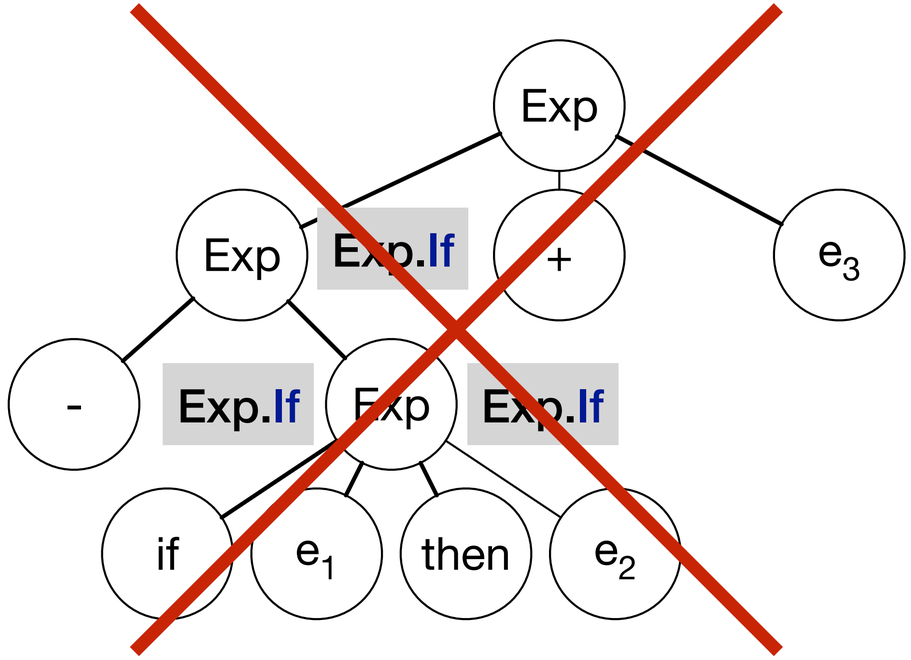
- (if e_1 then e_2) + e_2

Operator-style

```
context-free syntax

  Exp.UnMin = "-" Exp
  Exp.If    = "if" Exp "then" Exp
  Exp.Add   = Exp{Exp.If} "+" Exp {left}
  Exp.Int   = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.If
```




```
- if e1 then (e2 + e2)
```




```
- (if e1 then e2) + e2
```

Contextual Grammars

```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp {left}
  Exp.Int    = INT

  Exp{Exp.If}.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp{Exp.If}
  Exp{Exp.If}.UnMin  = "-" Exp{Exp.If}
  Exp{Exp.If}.Int    = INT
  Exp{Exp.If}.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp{Exp.If} {left}

  Exp{Exp.If, Exp.IfElse}.UnMin  = "-" Exp{Exp.If, Exp.IfElse}
  Exp{Exp.If, Exp.IfElse}.Int    = INT
  Exp{Exp.If, Exp.IfElse}.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp{Exp.If, Exp.IfElse} {left}


context-free priorities

  Exp.UnMin > Exp.Add > Exp.IfElse > Exp.If
```

Data-dependent Contextual Grammars

```
context-free syntax

  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp{Exp.If} "else" Exp
  Exp.UnMin  = "-" Exp
  Exp.Add    = Exp{Exp.If, Exp.IfElse} "+" Exp {left}
  Exp.Int    = INT

context-free priorities

  Exp.UnMin > Exp.Add > Exp.IfElse > Exp.If
```