Declarative Syntax Definition for Modern Language Workbenches

de Souza Amorim, Eduardo

DOI
10.4233/uuid:43d7992a-7077-47ba-b38f-113f5011d07f

Publication date
2019

Document Version
Final published version

Citation (APA)
de Souza Amorim, E. (2019). *Declarative Syntax Definition for Modern Language Workbenches*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:43d7992a-7077-47ba-b38f-113f5011d07f

Declarative Syntax Definition for Modern Language Workbenches

DISSERTATION

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus Prof.dr.ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates,
to be defended publicly on
Wednesday 19 June 2019 at 12:30 o'clock
by

Luis Eduardo DE SOUZA AMORIM

MSc Computer Science, Universidade Federal de Viçosa, Brazil
born in Unaí, Brazil

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof.dr. E. Visser	Delft University of Technology, promotor
Prof.dr. S. T. Erdweg	Johannes Gutenberg University Mainz, promotor

Independent members:

Dr. E. van Wyk	University of Minnesota, United States
Prof.dr. R. Lämmel	University of Koblenz-Landau, Germany
Prof. E. Scott	Royal Holloway, University of London, United Kingdom
Prof.dr. M.G.J. van den Brand	Eindhoven University of Technology
Prof.dr. K.G. Langendoen	Delft University of Technology
Prof.dr. A. van Deursen	Delft University of Technology, reserve member

The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Cover by: Lonely tree on a green hill, Peter Heeling (`https://skitterphoto.com/photos/394/lonely-tree-on-a-green-hill`) CC0 1.0 Universal (CC0 1.0) Public Domain Dedication

Printed and bound in The Netherlands by:
ProefschriftMaken || `www.proefschriftmaken.nl`.

ISBN 978-94-6366-171-3

Contents

Samenvatting **v**

Summary **vii**

1 Introduction **1**
1.1 Programming Languages 1
1.2 Language Definition 1
1.3 Language Workbenches 4
1.4 The Thesis in More Detail 6
1.5 Research Overview and Contributions 7
1.6 Origin of Chapters 9

I Declarative Disambiguation **11**

2 Declarative Disambiguation of Expression Grammars **13**
2.1 Introduction 13
2.2 Grammars and Ambiguities 19
2.3 Infix Expression Grammars 30
2.4 Prefix Expression Grammars 44
2.5 Postfix Expressions 53
2.6 Distfix Expressions 60
2.7 Indirect Recursion 66
2.8 Grammar Transformation 74
2.9 Implementation 81
2.10 Evaluation 89
2.11 Related Work 94
2.12 Conclusion 97

3 Deep Priority Conflicts in the Wild: A Pilot Study **99**
3.1 Introduction 99
3.2 A Primer on Declarative Disambiguation 101
3.3 Reasoning about Deep Priority Conflicts 105
3.4 Evaluation 109
3.5 Threats to Validity 117
3.6 Related Work 118
3.7 Conclusion and Future Work 120

4 Towards Zero-Overhead Disambiguation of Deep Priority Conflicts 121
4.1 Introduction . . . 121
4.2 Disambiguating Priority Conflicts . . . 123
4.3 Data-dependent Contextual Grammars . . . 129
4.4 Evaluation . . . 136
4.5 Related Work . . . 139
4.6 Conclusions . . . 142

II Declarative Syntax Definition 145

5 Declarative Specification of Layout-Sensitive Languages 147
5.1 Introduction . . . 147
5.2 Background . . . 149
5.3 Layout Declarations . . . 151
5.4 Layout-Sensitive Parsing . . . 155
5.5 Layout-Sensitive Pretty-Printing . . . 159
5.6 Evaluation . . . 165
5.7 Related Work . . . 168
5.8 Future Work . . . 169
5.9 Conclusion . . . 170

6 Principled Syntactic Code Completion 171
6.1 Introduction . . . 171
6.2 State of the Art of Syntactic Completion . . . 174
6.3 Completion by Rewriting Placeholders . . . 176
6.4 Code Expansion by Placeholder Inference . . . 182
6.5 Code Completion for Incorrect Programs . . . 188
6.6 Evaluation . . . 192
6.7 Related Work . . . 193
6.8 Future Work . . . 195
6.9 Conclusion . . . 195

7 Conclusion 197
7.1 The Thesis Revisited . . . 197
7.2 Suggestions for Future Work . . . 199

Bibliography 201

Curriculum Vitae 219

List of Publications 221

Samenvatting

Programmeertalen zijn een belangrijk onderdeel van de informatica, waarmee programmeurs het gedrag van computersystemen kunnen beheersen, bepalen en veranderen. Maar het ontwerpen, implementeren en onderhouden van programmeertalen vereist ook een behoorlijke inspanning. Gelukkig zijn er declaratieve aanpakken voor het definiëren van programmeertalen, die de ontwikkeling en implementatie ervan vergemakkelijken.

Om een programmeertaal te definiëren is de eerste stap meestal het specificeren van de *syntax* (of notatie). Syntaxdefinitieformalismen zijn gebaseerd op *grammatica's*: sets van regels die bepalen welke woorden tot de taal behoren, en hoe deze woorden moeten worden gestructureerd om geldige programma's te construeren. Grammatica's zijn *multifunctioneel*, omdat ze een begrijpbare documentatiebron zijn en tegelijkertijd kunnen worden gebruikt om taalimplementaties van af te leiden. *Taalontwikkelomgevingen* (language workbenches) helpen de taalontwikkelaars om (prototypen van) programeertalen te ontwikkelen door syntactische functionaliteiten af te leiden uit een syntaxdefinitieformalisme.

Het gebruik van declaratieve syntaxdefinities in een taalontwikkelomgeving levert nog steeds veel uitdagingen op. Om een volledig declaratieve syntaxdefinitie mogelijk te maken moeten parsers en andere gereedschappen ondersteuning bieden voor grammatica's in hun natuurlijke vorm. Oftewel, ze moeten dubbelzinnige grammatica's kunnen accepteren. Parsers die *dubbelzinnige grammatica's* aanvaarden hebben vaak geen duidelijke semantiek voor het ondubbelzinnig maken, en dit reduceert de parseerprestaties en beperkt de talen die succesvol kunnen worden geïmplementeerd. Aanvullend op het parseren moeten editorfunctionaliteiten zoals pretty-printing en codevoltooiing (*code completion*) vaak met de hand worden geïmplementeerd, waardoor de kosten van het onderhouden en evolueren van de taal oplopen.

Ons doel is om declaratieve syntaxdefinities te gebruiken om doelmatig de syntax van programmeertalen te definiëren en efficiënte gereedschappen te genereren. Om de bovenstaande problemen aan te pakken introduceren we een nieuwe semantiek voor het ondubbelzinnig maken van contextvrije grammatica's, en dan specifiek de subgroep van grammatica's voor het definiëren van expressies. We onderzoeken hoe vaak deze dubbelzinnigheden zich voordoen in programma's in de praktijk, en tonen de noodzaak aan voor het efficiënt ondubbelzinnig kunnen maken. Ten slotte implementeren we deze semantiek, waarbij we een parser genereren die vrijwel zonder overhead met dubbelzinnige grammatica's overweg kan.

Daarnaast ontwikkelen we een techniek om automatisch parsers en pretty-printers voor *opmaakgevoelige* talen af te leiden uit de syntaxdefinitie. De mogelijkheid om opmaakgevoelige talen declaratief te specificeren laat ons belangrijke vraagstukken aanpakken, waaronder bruikbaarheid, prestaties, en

gereedschapsondersteuning, die de opname van dit soort talen in gereedschappen zoals taalontwikkelomgevingen verhinderen.

Ten slotte stellen we een principiële aanpak voor waarmee syntactische codevoltooiing uit de syntaxdefinitie kan worden afgeleid. De huidige implementaties van voltooiingsfunctionaliteiten zijn vaak ad hoc, ondeugdelijk, en incompleet. Door een principiële aanpak te gebruiken kunnen we redeneren over de deugdelijkheid en compleetheid van codevoltooiing, waardoor een pad wordt onthuld naar rijkere editorfunctionaliteiten in taalimplementaties.

Summary

Programming languages are one of the key components of computer science, allowing programmers to control, define, and change the behaviour of computer systems. However, programming languages require considerable effort to design, implement, and maintain. Fortunately, declarative approaches can be used to define programming languages facilitating their development and implementation.

Commonly, the first step to define a programming language consists of specifying its *syntax*. Syntax definition formalisms are based on *grammars*, defining rules that specify the words that belong to a language and how these words must be structured to construct valid programs. Grammars are *multipurpose*, i.e., they provide an understandable source of documentation, and can also be used to derive language implementations. *Language workbenches* assist language engineers to develop and prototype programming languages by deriving syntactic services from a syntax definition formalism.

Many challenging problems still exist when using declarative syntax definitions in a language workbench. To enable truly declarative syntax definitions, parsers and other tools must support grammars in their natural form, i.e., they must be able to handle *ambiguous grammars*. Parsers that support ambiguous grammars lack a clear semantics for disambiguation, restricting their parsing performance and the languages they can successfully implement. Complementary to parsing, editor services such as pretty-printing and code completion, often need to be implemented by hand, increasing the cost of maintaining and evolving a language.

Our goal is to use declarative syntax definitions to effectively define the syntax of programming languages and generate efficient tools. To address the above problems, we propose a new semantics for disambiguating context-free grammars, particularly the subset of grammars that define expressions. We study how often these ambiguities occur in real programs, showing the need for efficient disambiguation. Finally, we implement this semantics, generating a parser that performs disambiguation with near-zero performance overhead.

Moreover, we develop a technique to automatically derive parsers and pretty-printers for *layout-sensitive languages* from the syntax definition. By enabling the declarative specification of layout-sensitive languages, we tackle important issues, including usability, performance, and tool support, which prevent the adoption of these languages in tools such as language workbenches.

Finally, we propose a principled approach to derive syntactic code completion from the syntax definition. The current implementation of completion services is often ad-hoc, unsound, and incomplete. By using a principled approach, we are able to reason about soundness and completeness of code completion, opening up a path to richer editing services in language implementations.

Introduction

1

This dissertation shows that declarative syntax definitions can be used to effectively define the syntax of programming languages and generate efficient tools.

1.1 PROGRAMMING LANGUAGES

Programming languages are one of the key components of computer science. They allow one to control, define and change the behaviour of computer systems by introducing abstractions over hardware-specific tasks such as accessing memory, executing commands, and controlling hardware devices. At their core, different programming languages have the same goal—allow programmers to write computer instructions—however, each programming language may have specific characteristics according to its *application domain*.

Some programming languages such as Pascal [139], C [70], Scala [92], Haskell [86], and Java [55] are considered *general-purpose programming languages*, since they are designed for writing software for many application domains. Other languages, on the other hand, so-called *domain-specific languages*, introduce abstractions for tasks in a specific domain. For instance, the numerical analysis language MATLAB [88], the query language SQL [87], the modelling language UML [102], and the mark-up language YAML [20] are considered domain-specific languages.

Programming languages, both general-purpose and domain-specific, still require considerable effort in their design, implementation, maintenance, and evolution. Declarative approaches can be used to define programming languages to facilitate their design and implementation.

1.2 LANGUAGE DEFINITION

A language definition often consists of the specification of the language's *syntax* and *semantics*. In programming languages, the description of a language's syntax corresponds to a set of rules that govern how the sentences (programs) of a language can be correctly constructed. Meanwhile, the specification of the language's semantics describes the meaning of syntactic constructs, restricting its set of valid programs to programs that are semantically meaningful.

Most programming languages have their syntax defined by a *grammar*. More specifically, a *context-free grammar* is a syntax definition formalism introduced by Noam Chomsky that enables formal specifications of the syntax of programming languages [37]. Grammars consist of rules, called *productions*, which define the structure of the sentences in a programming language. *Recognizers*, and *parsers* are tools that use grammars to verify whether a sentence belongs

to a language, or to construct an abstract representation in the form of a tree, to represent this structure. Grammars are *declarative* in their essence, since they describe *what* such language tools should do, not necessarily *how* to do it [69].

1.2.1 *Context-free Syntax*

Traditionally, languages are implemented using handwritten parsers, which are optimized for the language at hand. However, this approach results in high maintainability and extensibility costs. Furthermore, handwritten parsers often result in a mismatch between the language specification and language implementation. *Parser generators* address this issue by generating a parser from a language specification such as a context-free grammar, integrating language specification and language implementation. Ideally, parser generators can be reused for multiple grammars, and grammars can be reused for different purposes, other than parser generation.

It is possible to distinguish parser generators by the classes of grammars they support. Traditional parser generators such as YACC [64] and ANTLR [95, 96] only support certain grammar subclasses such as LR, LL, or LALR. Even though restricting grammars to a certain class may provide benefits such as better performance, this approach comes at a price, as grammars become *parser specifications* rather than *language specifications*, thus losing their declarative aspect. Furthermore, parsers based on LL(k) grammars do not support left recursion, whereas parsers based on LR(k) grammars have to deal with shift/reduce conflicts in parse tables. In both cases, these parsers are restricted by the number of symbols that need to be considered to take a certain action, the lookahead k.

Another major drawback of restricting grammars to a certain subclass relates to *language composition*. When composing grammars from a certain subclass, it is not guaranteed that the resulting grammar will belong to the same subclass [61]. This inhibits the modularity of syntax definitions and limits the composition of grammars.

An alternative to maintaining the declarative nature of syntax definition formalisms is to use *generalized parsers*. Generalized parsers such as generalized LL (GLL) [109], or generalized LR (GLR) [125] support the full class of context-free grammars, thus allowing modular language specifications and language composition. Furthermore, generalized parsers handle unlimited lookahead by allowing multiple interpretations of the input to be parsed in parallel. This feature also allows generalized parsers to parse *ambiguous grammars*. Generalized parsers such as GLL may have cubic worst-case complexity for ambiguous grammars, but they often have a linear behaviour for most grammars of programming languages [125, 109].

1.2.2 *Lexical Syntax*

Another way to classify parser generators is with respect to how they deal with *lexical syntax*, that is, how parser generators verify that words in a language

have been correctly constructed. Lexical syntax is often specified using *character-level grammars*, which are based on a less expressive subset of context-free grammars called *regular grammars*. Regular grammars use regular expressions to determine how to combine characters to form the words in a language.

Because character-level grammars often require arbitrary length lookahead to parse, conventional parsers based on LL or LR grammars handle grammars that incorporate productions to specify lexical syntax [28]. Thus, these parsers separate the processing of context-free syntax and lexical syntax by using a scanner, or lexer, to tokenize the input, such that the parser can operate on these tokens.

The separation of lexical and context-free syntax has many implications. First, grammars need to be split between lexical and context-free syntax, with different expressive powers, which are often defined using two different syntax definition formalisms. Moreover, because lexical syntax is restricted to regular grammars, constructs such as nested comments cannot be expressed. In fact, this problem occurs whenever lexical constructs need context information to properly tokenize the input. This problem can be tackled by introducing a lexical state, creating a mechanism to allow lexer and parser to interact, but this increases the complexity of the implementation, and is also detrimental to the declarative nature of syntax definitions.

To avoid the need for a separate scanner, *scannerless parsers* process programs at the level of characters. They use a single syntax definition for both lexical and context-free syntax and can only be implemented using generalized parsing algorithms [69]. For instance, the scannerless GLR parsing algorithm (SGLR) [130] uses the syntax definition formalism SDF [58, 131], which enables a language specification that combines lexical and context-free syntax.

1.2.3 *Disambiguation*

Grammars in their natural and declarative form are often ambiguous [10], which means that parser generators need to be equipped with mechanisms to efficiently handle the various types of ambiguities that may occur in syntax definitions. There are three ways that ambiguities can be handled by parser generators: treating an ambiguity as conflict, ignoring ambiguities, or embracing ambiguities [69].

Because parsers based on LL and LR grammars are deterministic, they cannot handle ambiguous grammars. To determine whether a grammar is ambiguous or not is an undecidable problem [53]. Thus, when conflicts (points where it is not possible to determine which action the parser should take) do *not* occur when using LL and LR parsers, the grammar is deterministic. However, when a conflict does occur, it is not a guarantee that an ambiguity has been detected. Nevertheless, conflicts provide a good indication about whether the grammar is ambiguous or not.

Solving conflicts can be quite challenging. First, non-local conflicts can be introduced by making minor modifications to the grammar. Thus, evolving a syntax definition can become a time-consuming and tedious task. Furthermore,

conflict resolution is often done by modifying the parser definition to the point where the original structure is lost, which makes the definition hard to maintain. Finally, modifying the grammar to perform disambiguation often produces a larger grammar that is neither as declarative nor as concise as the original grammar.

Packrat parsers [49] are recursive-descent parsers that use memoization to efficiently perform backtracking when parsing programs. Packrat parsers are based on parsing expression grammars (PEGs) [50], which is a class of grammars that are not a strict subset of context-free grammars and cannot be used to express all grammars in this class. PEGs use a strict order of productions, that is, the first alternative that matches the input is always performed, preventing any non-determinism from occuring. Thus, ambiguities are ignored by PEGs, and cannot be defined.

The main problem when using PEGs is that for productions that contain multiple alternatives and have joint patterns to recognize expressions, it is not always clear to determine which language a definition describes because of the ordering of the alternatives. This can lead to subtle errors, where the intention of the language designer is not the one represented in the syntax definition. Because detection of disjointness is also undecidable [50], such problems are difficult to capture.

Generalized parsers embrace ambiguities, being able to handle ambiguous grammars, and producing a *parse forest* containing all possible interpretations of the input. These parsers are often equipped with disambiguation mechanisms to produce a single parse tree as result.

A syntax definition used by a generalized parser is usually extended with *disambiguation rules*. For them to be effective, disambiguation rules are also designed to be declarative and to naturally adhere to context-free grammars. Disambiguation can then occur as a grammar transformation, when generating a parse table, during parsing, or after parsing by selecting a tree that meets certain criteria. For instance, the syntax definition formalism SDF [58] has been equipped with many declarative disambiguation rules to effectively handle different forms of ambiguities that occur in syntax definitions, both in lexical and context-free syntax. Even parser generators based on LR parsers such as YACC [64] use declarative rules to deal with common ambiguities.

1.3 LANGUAGE WORKBENCHES

Despite being at the core of language implementations, parsers are *only one* of the many tools to consider when designing a language. Tools such as *language workbenches* can be used to assist language engineers to develop and prototype domain-specific and general-purpose programming languages. The term language workbench was introduced by Martin Fowler [51] to characterize tools that support the *efficient* definition, implementation, maintenance, evolution and composition of languages going beyond parsing and code generation. They allow a better editing experience to language users, providing various editing services that are present in integrated development environments (IDEs).

Examples of early language workbenches include SEM [122], Metaview [114], MetaPlex [35], and MetaEdit [113]. Language workbenches are still being developed to provide better experiences to language engineers, and to support more language features. Some examples of modern language workbenches are JastAdd [44], Rascal [72, 73], Spoofax [68], and Xtext [47]. These are examples of so-called *textual workbenches*, since they use parsers to validate the syntax of programs as the user types in a text editor. *Projectional workbenches*, on the other hand, allow users to manipulate abstract syntax trees directly, which are then shown back to the programmer in a particular format. Intentional Programming [112], and the Meta Programming System (MPS) [135] are examples of projectional language workbenches. Modern language workbenches are enjoying significant growth in number and diversity, driven by both industry and academia [46].

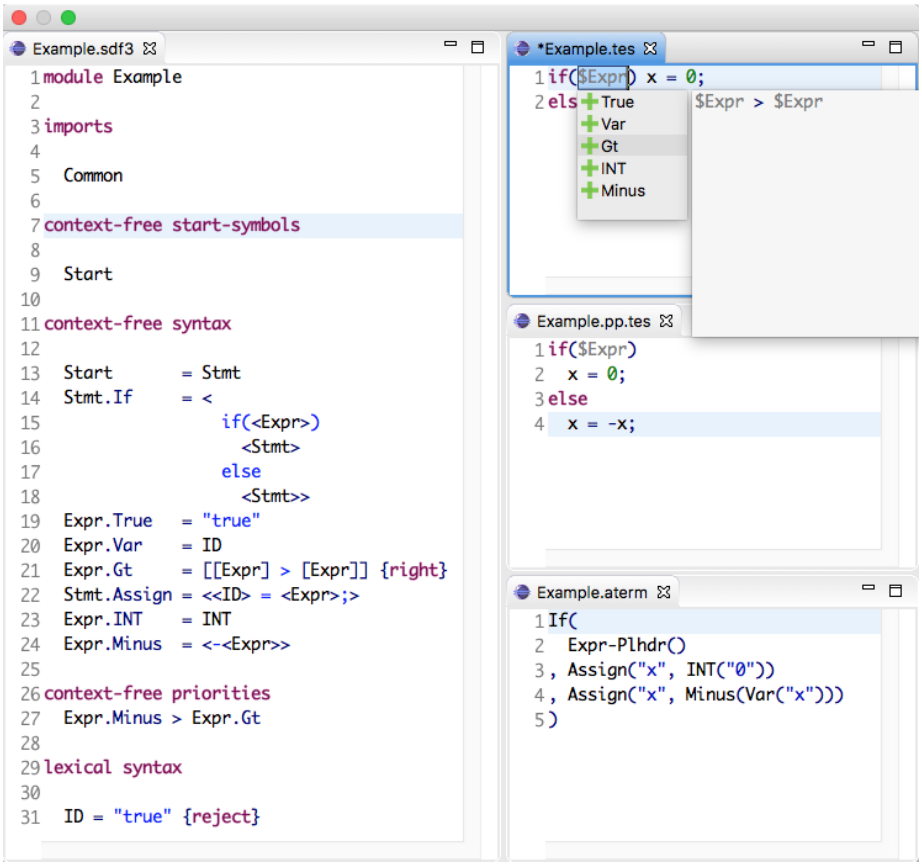


Figure 1.1 A screenshot of the Spoofax Language Workbench.

Editor support is a key component of language workbenches. For instance, when considering a textual language workbench, besides a plain text editor,

a language workbench should provide a selection of syntactic and semantic features such as syntax highlighting, parse error recovery, code folding, pretty-printing, refactoring tools, reference resolution, and code completion.

Figure 1.1 provides an overview of the Spoofax Language Workbench [68], which allows the development of textual languages with full IDE support. From different language specifications for the language's syntax and semantics, Spoofax generates an IDE plugin that can be loaded into the Eclipse or IntelliJ IDEs to support language users. Spoofax provides support for language prototyping, since language developers can load the plugins while the language is being developed, enabling them to simultaneously edit the language specification and programs of the language being developed.

1.4 THE THESIS IN MORE DETAIL

The goal of this dissertation is to present general solutions that support agile development and prototyping of both general and domain-specific programming languages. Despite the evolution of language workbenches and syntax definition formalisms, many challenging problems remain when using declarative language specifications in a language workbench.

Syntax definition formalisms are *multipurpose*. They can be used as a source of documentation, providing a way to easily understand and describe a language's syntax. At the same time, syntax definition formalisms can be used to derive language implementations by generating tools and services in a language workbench. A *declarative* syntax definition formalism requires that the language specification abstracts over the details of the language implementation.

To enable truly declarative syntax definitions, parsers and other tools must support grammars in their natural and declarative form, i.e., they must be able to handle *ambiguous grammars*. Most textual language workbenches rely on parser generators based on a specific class of context-free grammars. Syntax definition formalisms based on generalized parsers lack a clear semantics for disambiguation, which restrict the languages these formalisms can successfully specify, and may also affect the performance of the generated parser.

Complementary to parsing, another important element in a language workbench consists of *pretty-printing*. A pretty-printer transforms abstract syntax trees back into text, and is used by many other tools in a language workbench such as refactoring tools, code completion, and code generation. Pretty-printers are often written by hand, or written using a different specification from the syntax definition. Both approaches are detrimental to evolving and maintaining a language, since changes to the syntax definition need to be reflected in the pretty-printer specification.

Code completion is one of the most used editor services in an IDE. However, both IDEs and language workbenches use ad-hoc approaches to code completion that are *unsound* and/or *incomplete*. *Syntactic code completion* is based on a language's syntax, and enables language discoverability, helping programmers to code faster by avoiding syntax errors. A *principled* approach

to code completion is necessary to produce a *sound and complete* completion service.

Thus, the lack of a clear semantics for disambiguation, the lack of an efficient implementation of this semantics, and the need to manually implement tools such as parsers, pretty-printers, and code completion services prevent effectively using syntax definition formalisms to define languages in language workbenches.

Our vision is that syntax definition formalisms can be used in language workbenches to declaratively specify different programming languages, and at the same time, implement efficient tools such as parsers, pretty-printers, and syntactic code completion.

1.5 RESEARCH OVERVIEW AND CONTRIBUTIONS

In this section, we give an overview of the research described in this dissertation, listing our core contributions. Throughout this dissertation, we present the design and implementation of the syntax definition formalism *SDF3*, used in the Spoofax Language Workbench [68]. SDF3 is based on context-free grammars, extended with constructs that enable declarative specifications of parsers and pretty-printers. SDF3 has evolved from SDF [58] to serve the needs of modern language workbenches and at the same time improve various issues of its predecessor, SDF2 [131].

1.5.1 *Declarative Disambiguation*

In the first part of this dissertation we focus on disambiguation of context-free grammars, in particular, addressing ambiguities that occur in *expression grammars*.

Safe and Complete Semantics for Disambiguating Expression Grammars As mentioned previously, the concise, declarative, natural specifications of context-free grammars are often ambiguous. One common source of ambiguities in grammars of programming languages is related to *operator priority and associativity* in expression grammars. Recent work [5] has shown that the disambiguation techniques of SDF2 for handling operator precedence ambiguities are *unsafe*, *incomplete* and *limited*. It is unsafe because it rejects unambiguous sentences, it is incomplete because it cannot handle all combinations of operators, and it is limited because it only considers conflicts that occur one level deep in a resulting tree. In order to address these issues, we propose a new semantics for SDF3 declarative disambiguation in *Chapter 2*. Our semantics provide *safe* disambiguation by only rejecting trees that are part of an ambiguity. Furthermore, we make our semantics *complete* by addressing a specific type of conflicts in expression grammars: *deep priority conflicts*. In conclusion, *the semantics described in this chapter enables declarative syntax definitions to effectively define the syntax of programming languages.*

Empirical Study on Deep Priority Conflicts Despite our efforts to provide safe and complete disambiguation of expression grammars, there was no data about how often deep priority conflicts occur in practice, or the efficiency of the techniques to disambiguate deep conflicts. Thus, in *Chapter 3*, we present a study that investigates deep priority conflicts in two different programming languages: Java, and OCaml. As a statement-based language, Java provides a good example of a language that only contains a few deep conflicts in its specification. In contrast, OCaml, as an expression-based language represents the other side of the spectrum, a language with many deep priority conflicts. Our study investigated disambiguation of deep priority conflicts using a grammar transformation technique, considering programs from the top 10 trending repositories on GitHub for each language. In conclusion, *our study indicated that deep priority conflicts* do occur *in real programs, demonstrating the need for efficient disambiguation.*

Efficient Disambiguation of Deep Priority Conflicts Grammar transformation techniques to address disambiguation may have a large impact on the performance of the parser. In *Chapter 4* we address efficient disambiguation of deep priority conflicts by using *data-dependent grammars*. We implement the semantics for disambiguation described in Chapter 2, disambiguating deep priority conflicts at parse time. The technique consists of creating a data-dependent grammar that encodes deep priority conflicts. A data-dependent parser constructs parse trees propagating information about how the trees are constructed. This information is then used to forbid trees that violate a particular disambiguation rule. Since disambiguation occurs at parse time, it is not necessary to transform the grammar creating additional productions that affect the performance of the parser. In conclusion, *our data-dependent approach for disambiguation generates efficient parsers from declarative syntax definitions.*

1.5.2 *Declarative Syntax Definition*

In the second part of this dissertation we investigate the specification of layout-sensitive languages, and syntactic code completion.

Layout Declarations Layout-sensitive languages characterizes languages that must obey particular *indentation rules*, i.e., languages in which the indentation of a program influences how the program should be parsed. While layout-sensitive languages, such as Haskell and Python, have been widely adopted, tools such as parsers and pretty-printers still need to be handwritten for these languages. In *Chapter 5*, we detail an approach to declaratively specify *layout-sensitive languages*. We introduce *layout declarations* as a mechanism to declaratively specify indentation rules in the syntax definition, evaluating our approach using the Haskell programming language. In conclusion, *layout declarations enable deriving efficient layout-sensitive parsers and pretty-printers.*

Principled Syntactic Code Completion Syntactic code completion enable users to change the source code by inserting code snippets corresponding to fragments of the language's syntax. In a language workbench, a syntactic completion

service assists both novice and experience language users, enabling language discoverability, and avoiding syntax errors. Existing code completion systems are ad-hoc and neither sound nor complete, meaning that they produce wrong results, and do not support the entire language's syntax. We discuss *syntactic code completion* in *Chapter 6*. We propose a principled approach for syntactic code completion based on placeholders, which can be derived from a syntax definition. We adopt a divide and conquer approach that enables us to address code completion for (i) incomplete programs, i.e., programs with placeholders, (ii) complete programs without placeholders, and (iii) programs with syntax errors. In conclusion, *our approach assists generating a completion service that is both sound and complete.*

1.5.3 *Methodology*

In this dissertation we are interested in answering what is a better way to do/create/modify/evolve syntactic tools when implementing programming languages. Answering this question requires developing methods to automatically derive efficient tools from a syntax definition. We follow the iterative process described in the memorandum on design-oriented information systems research, published by Österle et al. [94]. This process consists of four phases: analysis, design, evaluation, and diffusion.

In the analysis phase we identify and study problems when using syntax definition formalisms to define programming languages in tools such as language workbenches. In the design phase, we design new approaches that solve the problems we identified, implementing new solutions, as shown in each of the chapters of this dissertation. We also perform a thorough comparison with related work contrasting the main differences between our approach and other solutions in the literature.

In the evaluation phase, we evaluate each of our solutions by applying it to real world programs and programming languages. We also develop formal proofs to ensure that particular characteristics hold when using the solution proposed. When considering the performance of a technique, we experimentally evaluate our solution, measuring performance differences, and discussing to what degree we improve performance.

Finally, in the diffusion phase, we publish our findings as research papers to journals and conferences, producing software artifacts that are reviewed and can be used to perform further research. In the next section, we list the research papers that originated from the work in this dissertation.

1.6 ORIGIN OF CHAPTERS

All chapters are directly based on peer-reviewed (or under ongoing review) publications at programming languages and software engineering conferences and workshops. Therefore, all chapters can be read independently of each other. Despite having independent core contributions, there is some redundancy in the chapters when stating the background, related work, motivation and

examples. Nevertheless, this redundancy has not been removed allowing reading each of the chapters independently.

- Chapter 2 is currently under submission at the *ACM Transactions of Programming Languages and Systems (TOPLAS)* journal as the paper *A Direct Semantics for Declarative Disambiguation of Expression Grammars* [116].
- Chapter 3 has been published as the paper *Deep Priority Conflicts in the Wild: A Pilot Study* [118] at the *International Conference on Software Language Engineering (SLE) 2017*.
- Chapter 4 has been published at the journal *The Art, Science, and Engineering of Programming (<Programming>)* and presented at the *<Programming> 2018* conference as the paper *Towards Zero-Overhead Disambiguation of Deep Priority Conflicts* [119].
- Chapter 5 has been published at *SLE 2018* as the paper *Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-Printing Layout-Sensitive Languages* [117].[1]
- Chapter 6 has been published as the paper *Principled Syntactic Code Completion Using Placeholders* [115], presented at *SLE 2016*.

[1] This paper was awarded the *distinguished paper*, for most notable paper, as determined by the PC chairs based on the recommendations of the programme committee.

Part I

Declarative Disambiguation

Declarative Disambiguation of Expression Grammars

2

ABSTRACT

Context-free grammars in reference manuals and academic papers are often ambiguous, yet, they provide concise descriptions and a direct correspondence between abstract syntax trees and grammar rules. A major part of such ambiguities arises from the subset of a grammar that specifies *expressions*. Disambiguation of expressions in context-free grammars by means of priority and associativity declarations enables a direct correspondence between grammar and abstract syntax trees, more concise grammars, and a better expression of intent than encoding associativity and priority in the grammar.

There is no standardized, declarative semantics of disambiguation with associativity and priority declarations. Indirect approaches to the semantics use a translation into another formalism (such as parse tables or tree automata), inhibiting generalization and/or understanding. A direct approach is to define the semantics of priority and associativity declarations by means of subtree exclusion patterns, which are independent of a particular parsing algorithm or grammar transformation. However, existing definitions of direct approaches are not safe, and/or do not cover all cases of ambiguous expressions.

In this chapter, we provide a *direct* semantics of disambiguation by means of associativity and priority declarations that is *safe* and *complete*, and not limited to simple expression grammars. We define a semantics for *safe* disambiguation with priority and associativity in terms of shallow subtree exclusion patterns, filtering trees only when the input is ambiguous. We extend the semantics with a formal definition of *deep priority conflicts*, covering additional ambiguities in more complex expression grammars that cannot be solved by fixed-depth patterns, such as ambiguities due to low precedence prefix or postfix operators, dangling prefix, dangling suffix, longest match, and indirect recursion. We have implemented the semantics in a parser generator for SDF3, evaluating the approach by applying it to the grammars of five languages. Finally, we demonstrate that our approach is practical by measuring the performance of a parser that implements our disambiguation techniques, applying it to a corpus of real-world Java and OCaml programs.

2.1 INTRODUCTION

Context-free grammars provide a concise, high-level, and well-understood formalism for the specification and documentation of the syntax of programming languages. Grammars play a dual role in such descriptions. On the one hand,

a grammar describes the *structure* of programs in a language, i.e. the set of *trees* that represent its well-formed programs. On the other hand, grammars also specify *parsers*, i.e. the mapping from sentences to trees. These roles pose conflicting requirements on grammars. For the purpose of describing structure, the (abstract) syntax definitions in reference manuals and academic papers are often *ambiguous*, providing concise descriptions and a direct correspondence between abstract syntax trees and grammar rules. For the purpose of semantic evaluation, a grammar should *unambiguously* identify the structure of a program text.

```
context-free syntax
  Exp.Var       = ID
  Exp.Int       = INT
  Exp           = "(" Exp ")" {bracket}
  Exp.Add       = Exp "+" Exp {left}
  Exp.Sub       = Exp "-" Exp {left}
  Exp.Mul       = Exp "*" Exp {left}
  Exp.Minus     = "-" Exp
  Exp.Lambda    = "\\" ID "." Exp
  Exp.Inc       = Exp "++"
  Exp.If        = "if" Exp "then" Exp
  Exp.IfElse    = "if" Exp "then" Exp "else" Exp
  Exp.Subscript = Exp "[" Exp "]"
  Exp.While     = "while" Exp "do" Exp "done"
  Exp.App       = Exp Exp {left}
  Exp.Function  = "function" PMatch+ {longest-match}
  PMatch.Clause = ID "->" Exp
context-free priorities
  {Exp.Subscript Exp.Inc} > Exp.App > Exp.Minus >
  Exp.Mul > {left: Exp.Add Exp.Sub} > Exp.IfElse >
  {Exp.If Exp.Lambda Exp.Function}
```

Figure 2.1 Expression grammar with declarative disambiguation rules.

Associativity and Priority In mathematics, instead of explicitly indicating the structure of expressions using parentheses, parsing the structure is determined based on conventions. Anyone with high school mathematics has learned to read an expression such as $a * b + c/d$ as $(a * b) + (c/d)$ and not as $a * ((b + c)/d)$ since multiplication and division have higher priority[1] than addition. Similarly, $a + b + c$ is the same as $(a + b) + c$ and $a + (b + c)$ since addition is associative. However, $a - b - c$ should be read as $(a - b) - c$ and not as $a - (b - c)$, since subtraction is left associative. The expression sub-languages of programming languages have evolved from these notational conventions. For example, the name of the FORTRAN programming language is derived from *formula translation* and its compiler applied these conventions to determine

[1] While *precedence* is often used in the context of expression disambiguation, we use its synonym *priority*, which has been used in the SDF family of grammar formalisms for that concept [58].

the order of computation of the sub-expressions [16]. Thus, associativity and priority are well known conventions, familiar to programmers before they write their first program.

Conceptually, then, we learn the structure of expressions based on an ambiguous grammar. Operators such as addition and multiplication combine (the values of) two expressions into a new expression. The productions for addition and multiplication in the *context-free syntax* section of the syntax definition in Figure 2.1 reflect this structure. Only when *combining* multiple operators in an expression do we (need to) apply *disambiguation* rules to determine their structure (or order of evaluation). Programming languages have extended expression notation beyond the simple infix, prefix, and postfix operators of mathematical notation. Figure 2.1 presents a syntax definition with an eclectic collection of examples of such constructs.

Encoding Associativity and Priority. There are essentially two approaches to encode associativity and priority in the grammar of a programming language.

First, associativity and priority can be encoded in the grammar by using a different non-terminal for each priority level, and making productions left or right recursive in order to encode associativity, resulting in an unambiguous grammar. This encoding obscures the *conceptual* structure of expressions and the notion of associativity and priority. Moreover, it does not scale well to more complex expression forms such as dangling else and low priority prefix operators (which are prevalent in functional languages), requiring duplication of productions [2].

The second approach to address expression disambiguation is to define an ambiguous expression grammar in combination with separate *associativity and priority rules* [42]. This route is often taken by reference manuals of programming languages; a table enumerates the operators at each priority level and for each level indicates associativity. Grammar formalisms such as YACC [64] and SDF [58, 128, 28] formally integrate priority and associativity declarations (which we will also refer to as disambiguation rules, for brevity). Figure 2.1 illustrates the use of associativity and priority declarations in SDF3. This approach preserves the conceptual structure of expressions in the grammar and applies the familiar notion of associativity and priority.

Both approaches are used in practice. The grammar in the Java SE 7 Specification [54] uses an ambiguous grammar with a separate (informal) specification of associativity and priority. However, an explicitly disambiguated grammar is used as the basis for the reference implementation of the parser for Java SE 7. In contrast, the Java SE 8 specification [55] explicitly encodes associativity and priority in the grammar. As a result, the grammar is less concise since productions are duplicated to solve ambiguities that arise from combining operators or statements.[2]

[2]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 separate non-terminals to encode the priority levels of operators.

Semantics of Associativity and Priority. While the semantics of pure context-free grammars is well-understood, there is no standardized semantics of context-free grammars extended with associativity and priority declarations. Defining such a semantics requires answers to the following questions:

- What is the meaning of a set of disambiguation rules for a grammar? That is, what are the parse trees associated with sentences in the language of the disambiguated grammar? To what extent is that meaning independent of particular implementation strategies?
- Is a set of disambiguation rules safe? That is, do the disambiguation rules preserve the language of the grammar they disambiguate? Is it necessary for disambiguation rules to be safe, or can rules exclude sentences?
- Is a set of disambiguation rules complete? That is, do the rules identify at most one parse tree for each sentence in the language? It is not obvious that this question is decidable, since, in general, it is undecidable to determine whether a context-free grammar is ambiguous or not [36, 48, 34].
- What is the coverage of disambiguation rules? That is, what classes of ambiguity do the rules solve?
- What is an effective implementation strategy for disambiguation rules?
- What is the notational overhead of disambiguation rules? That is, is the use of rules more effective than an encoding in the grammar?

We distinguish *indirect* and *direct* approaches to characterize the semantics of disambiguation rules.

Indirect Semantics of Associativity and Priority Indirect approaches use a translation into another formalism. For instance, in YACC, operator priority is defined by declaring a linear priority ordering on operators and declaring the associativity of individual operators. The semantics of these declarations is defined indirectly through their interpretation in solving shift-reduce conflicts in parse tables. "If there is a shift/reduce conflict, and both the grammar rule and the input character have priority and associativity associated with them, then the conflict is resolved in favour of the action (shift or reduce) associated with the higher priority. [64]" Thus, understanding the semantics of priority declarations in a YACC grammar requires understanding LR table generation, the nature of shift/reduce conflicts, and the mapping of disambiguation declarations to conflict resolution. YACC's approach is also limited to the LR class of context-free grammars. For example, it does not generalize to *character-level* grammars, in which individual characters are the tokens; in this case, decisions cannot be taken using the next token(s).

Adams and Might [4] propose *tree automata* [39] to describe disambiguation policies. A tree automaton describes the trees that should or should not be produced by a parser. Intersecting the context-free grammar with the tree automaton produces a transformed context-free grammar that does not produce

tree structures forbidden by the automaton. The tree automata approach is very expressive and can be used to carve out a rich set of subtree patterns from grammars. Adams and Might [4] give examples of the use of tree automata to disambiguate associativity and priority, but they do not provide a general semantics for such rules in terms of tree automata. From those examples, it is clear that tree automata are not a useful formalism to directly define disambiguation rules for individual grammars since they require a regular tree pattern for each pair of productions. It may be possible to use tree automata as a target for compiling disambiguation rules, but determining the correctness of such a translation requires a semantics for the latter in the first place. Furthermore, disambiguation by tree automata is, intentionally, not guaranteed to be safe or complete. Finally, tree automata are tightly coupled to the implementation of disambiguation by means of grammar transformation. As we will show in Section 2.10, disambiguation by grammar transformation can lead to very large grammars and correspondingly large parse tables.

Direct Semantics of Associativity and Priority By contrast, a direct approach to disambiguation is independent of a particular parsing algorithm or grammar transformation. For instance, the SDF2 [131, 128] syntax definition formalism defines the semantics of disambiguation declarations *directly* through interpretation as *disambiguation filters* [74] that select among the possible parse trees for a sentence. Specifically, SDF2 uses *parse tree patterns* [123, 130, 28] to exclude trees of a certain shape as disambiguation filter. For example, the prototypical ambiguity between addition and multiplication is resolved by excluding trees that have the shape `[[`t_1 `+` t_2`]` `*` t_3`]` resulting in trees with shape `[`t_1 `+ [`t_2 `*` t_3`]]` to be selected as unambiguous result of parsing t_1 `+` t_2 `*` t_3. These tree patterns directly follow from disambiguation declarations and provide a natural explanation of operator priority. Another advantage of using a direct semantics is that it does not preclude specific implementation techniques (as in the case of YACC), enabling, for example, generalization to character-level grammars.

Afroozeh et al. [5] show that the SDF2 semantics for operator priority and associativity is *unsafe*, *incomplete*, and *limited*. The semantics is *unsafe* because it rejects trees that are unambiguous and belong to the language defined by the grammar. It is *incomplete* because it is not powerful enough to disambiguate some combinations of operators. It is *limited* because it only considers conflicts that occur one level deep in a resulting tree. To address these issues, Afroozeh et al. [5] define a new interpretation of SDF2 priorities that is safe, expressed by means of a grammar transformation. However, their solution unnecessarily modifies the grammar in the presence of shallow conflicts, and is still indirect as they rely on other methods than priority and associativity declarations to solve ambiguities such as dangling prefix, dangling suffix, and longest match.

Contributions In this chapter, we present the first *direct* semantics of disambiguation by means of associativity and priority declarations that is *safe* and *complete* for *expression grammars*. The chapter makes the following contributions:

- We formalize the semantics of associativity and priority rules for classes of *expression grammars*, subsets of the class of context-free grammars that correspond to the structure of the expression sub-languages of programming languages. While the ambiguity of arbitrary context-free grammars is undecidable, when considering embedded expression grammars, we can guarantee safe and complete disambiguation. We study classes of expression grammars of increasing complexity: *infix* expression grammars, *prefix* and infix expression grammars, *basic* (infix, prefix, postfix) expression grammars , *distfix* expression grammars, *indirectly recursive* distfix expression grammars and indirectly recursive distfix expression grammars *with hidden recursion*.

- We define an *extraction of expression grammars* from larger context-free grammars, such that the techniques from this chapter can be applied to the embedded expression grammars in larger programming language grammars.

- We identify the concept of *harmless overlap* between grammar productions, in order to ensure that expression grammars do not have inherent ambiguities that are not solvable by associativity and priority declarations, while not completely ruling out overlap between productions.

- We define the semantics for associativity and priority rules by means of *subtree exclusion patterns*. We adapt the existing rules for subtree exclusion to be *safe* and extend them to a large class of expression grammars with harmless overlap, relying on *deep priority conflict patterns*.

- We prove that the semantics is *safe*, i.e. preserves the language of the underlying expression grammar, and *complete*, i.e., it solves all ambiguities in the underlying expression grammar. Methodologically, we prove safety and completeness using parse trees (and in particular using the notion of trees under subtree exclusion) instead of derivations, which simplifies reasoning. A prerequisite for completeness is that the set of disambiguation rules is *total*. We define totality for all expression grammar classes and implement a check for totality in order to detect missing disambiguation rules in a grammar.

- We define a canonical implementation of the semantics by means of a grammar transformation to *contextual grammars* that preserves the tree structure of the original ambiguous grammar. We also show that our canonical implementation correctly implement the semantics defined in the chapter, and that it terminates when transforming the original grammar.

- We discuss two additional implementations of the semantics in a parser generator for SDF3. The first one combines grammar transformation with disambiguation during parse table generation, and the second one postpones disambiguation of deep priority conflicts to parse time,

using a data-dependent parsing algorithm. Both implementations focus on deep priority conflicts. Shallow conflicts are solved during parse table generation in the style of SDF2 [126, 130] but adapted to the safe semantics.

- We have evaluated the approach by applying it to the grammars of five languages. We have also evaluated the performance of the parser that implements disambiguation by grammar transformation against disambiguation by data-dependency, applying both variations of the parser to a corpus containing 9935 Java and 3296 OCaml real-world source files.

Outline We proceed as follows. In Section 2.2 we introduce the basic notions for our discussion of ambiguities in expression grammars, formally defining concepts such as context-free grammars, parsing, parse trees, and ambiguities as we use them in the rest of the chapter. In Section 2.3 we discuss disambiguation of *infix expression grammars*, presenting declarative disambiguation rules and existing techniques for implementing and interpreting these definitions, and discussing a semantics for declarative disambiguation of infix grammars based on subtree exclusion that is safe and complete. In Section 2.4 we discuss *prefix expression grammars*, introducing a semantics to disambiguate grammars that also include prefix operators, and introducing *deep priority conflicts*. In Section 2.5, we consider expression grammars with prefix, postfix, and infix operators, called *basic expression grammars*. In Section 2.6, we extend these to *distfix expressions* and in Section 2.7, to distfix expression grammars with *indirect recursion*.

In Section 2.8 we present a canonical implementation of the semantics presented in the previous chapters using a grammar transformation. In Section 2.9 we discuss how to detect expression grammars, recalling disambiguation of shallow conflicts as implemented by SDF2, and a more efficient implementation of our semantics using data-dependent parsing. Finally, in Section 2.10 we evaluate our approach, discussing related work in Section 2.11, and concluding in Section 2.12.

2.2 GRAMMARS AND AMBIGUITIES

In this section we define (the notation for) context-free grammars, well-formed terms, parsing, abstract syntax trees, tree patterns, derivation steps, and matching, as we will use them in this chapter. We also discuss the general notions of ambiguities, ambiguous grammars, *disambiguation filters*, and their properties.

2.2.1 *Grammars and Parsing*

Context-free grammars are usually written in some concrete grammar formalism such as BNF [17]. In this chapter we use the SDF3 syntax definition

```
context-free syntax
  Exp.Add       = Exp "+" Exp
  Exp.Sub       = Exp "-" Exp
  Exp.Mul       = Exp "*" Exp
  Exp.Var       = ID
```

Figure 2.2 An example of an SDF3 grammar for arithmetic expressions.

formalism as grammar notation[3]. SDF3 extends BNF with explicit abstract syntax tree constructors for productions, which also provides a convenient way to identify grammar productions in disambiguation directives. Figure 2.2 shows an SDF3 grammar for a simple expression language. Formally, the context-free grammar fragment of SDF3 is defined as follows:

Definition 2.2.1 (Context-free Grammars). *A context-free grammar is a tuple* (Σ, N, P)*, with* Σ *a set of terminal symbols,* N *a set of non-terminals (sorts) and* P *a set of productions of the form* $A.C = \alpha$*, where we use the following notation:* $P(G)$ *to represent the productions of a grammar* G*,* V *to represent* $\Sigma \cup N$*;* X, Y, Z *to represent symbols in* V*;* A*,* B*, and* S *to represent elements of* N*;* a*,* b *to represent elements of* Σ*;* C *to represent constructor names that are used when constructing abstract syntax tree nodes and together with a non-terminal uniquely identify a production;* u, v, w, x, y, z *to represent elements of* Σ^**; and* $\alpha, \beta, \gamma, \phi$ *to represent elements of* V^**, also known as sentential forms.* □

Definition 2.2.2 (Well-formed Trees). *Given a grammar* G*, we inductively define the family of sets of well-formed trees (or terms)* $T^X(G)$ *indexed by* $X \in V$ *as the smallest set such that:*

$$\frac{a \in \Sigma}{a \in T^a(G)} \tag{2.1}$$

$$\frac{A.C = X_1 ... X_n \in P(G) \quad t_i \in T^{X_i}(G) \quad 1 \leq i \leq n}{[A.C = t_1 ... t_n] \in T^A(G)} \tag{2.2}$$

The family of well-formed trees is then defined as $(T^X(G))_{X \in V}$*, or simply* $T(G)$*. The yield of a tree is the concatenation of its leafs, which are terminals. The language defined by a grammar is the family* $L(G) = \{L^X(G) \mid yield(T^X(G)), X \in V\}$ *of sets of strings that are the yields of trees over the grammar.* □

Definition 2.2.3 (Parsing). *Given a grammar* G*, a parser* $\Pi(G)$ *is a mapping from strings to parse trees such that:*

$$\Pi(G)(w) = \{t \in T^X(G) \mid yield(t) = w, X \in V\} \tag{2.3}$$

A parser Π *is non-ambiguous if* $|\Pi(w)| \leq 1$ *for all strings* w*.* □

[3]SDF3 is part of the SDF family of syntax definition formalisms [58], it is an evolution of SDF2 [127, 131], introducing new features such as template productions [133], and addressing problems of its predecessor such as declarative disambiguation.

For example, using the productions in the grammar of Figure 2.2, parsing an addition of two variables produces the tree `[Exp.Add = [Exp.Var = ID] + [Exp.Var = ID]]`.[4]

Definition 2.2.4 (Derivation Steps)**.** *Given a context-free grammar G, the one-step derivation relation $\Rightarrow_G$ is the binary relation of sentential forms $V \times V$ such that:*

$$\frac{\alpha = \lambda A \rho \quad \beta = \lambda \gamma \rho \quad A.C = \gamma \in P(G)}{\alpha \Rightarrow_G \beta} \tag{2.4}$$

The reflexive and transitive closure of $\Rightarrow_G$ is denoted by $\overset{*}{\Rightarrow}_G$. The symbol being expanded in a derivation step is indicated by $\_$. We might also write $\overset{p}{\Rightarrow}_G$ to indicate the production using when expanding a symbol. A leftmost derivation $\underset{lm}{\overset{*}{\Rightarrow}}_G$ is a sequence of derivation steps which always expands the non-terminal at the leftmost position. □

Lemma 2.2.5. *A parse tree directly corresponds to a derivation, modulo the order in which productions are applied.*

Proof. By induction on the number of steps in a derivation, as shown by Aho, Sethi, and Ullman [11]. □

2.2.2 *From Parse Trees to Abstract Syntax Trees*

```
signature
  constructors
    Add : Exp * Exp -> Exp
    Sub : Exp * Exp -> Exp
    Mul : Exp * Exp -> Exp
    Var : ID -> Exp
```

Figure 2.3 Signature

A parser maps a string to a set of parse trees according to a grammar. Parse trees often contain syntactic information that is not necessary in the following phases of a compiler. *Abstract syntax trees (ASTs)* provide a more concise representation of the strucure of a program, abstracting over the syntactic details (literals, layout) of parse trees. Rather than providing *semantic actions* that execute arbitrary code to construct abstract syntax trees, SDF3 includes *constructors* in productions in order to define a standardized mapping from parse trees to abstract syntax trees. For example, the parse tree

```
[Exp.Add = [Exp.Mul = [Exp.Var = a] * [Exp.Var = b]] +
  [Exp.Var = c]]
```

[4] `ID` is a *lexical non-terminal* denoting identifiers and should be defined in the *lexical syntax* section of the grammar. For simplicity, we omit subtrees for lexical non-terminals from our examples or use concrete lexical elements as leafs, e.g., parsing `a + b` produces the tree `[Exp.Add = a + b]` .

for the sentence `a * b + c` according to the grammar of Figure 2.2 is converted to the term[5]:

```
Add(Mul(Var("a"), Var("b")), Var("c"))
```

The structure of abstract syntax trees is captured by an algebraic data type described by a signature. For instance, the signature in Figure 2.3 describes the abstract syntax trees of the grammar of Figure 2.2. For example, the production

```
Exp.Add = Exp "+" Exp
```

is converted to the corresponding signature by taking the non-terminals in the body of the production as arguments of the constructor and leaving out the operator literal. Thus, grammars with constructors define a direct correspondence between abstract syntax trees and parse trees.

It may be desirable to further simplify abstract syntax trees. Such simplifications can be realized by means of a desugaring transformation after parsing. For example, using a rewrite rule in the Stratego rewriting language [29], one would write a desugaring rule

```
desugar : IfThen(e, stmt) -> IfElse(e, stmt, Skip())
```

transforming an if-then AST to an if-then-else AST using the skip statement (for some imaginary language with such constructs).

While we will not further consider abstract syntax in this chapter, the standardized mapping is an important consideration in the design of disambiguation. Ambiguous productions in expression grammars result in a concise abstract syntax, e.g. without constructors corresponding to injection productions of the form `Exp.T2E = Term`. Therefore, a disambiguation mechanism should *preserve* the (abstract syntax) tree structure, which is not the case for mechanisms that *transform* the grammar.

2.2.3 *Tree Patterns and Matching*

Tree patterns, which extend trees with *non-terminals as leafs* to characterize sets of trees that have a particular structure, are defined as follows:

Definition 2.2.6 (Tree Patterns)**.** *Given a grammar G, we inductively define the family of tree patterns $(TP^X(G))_{X \in V}$ over G as the smallest sets such that:*

$$\frac{X \in V}{X \in TP^X(G)} \tag{2.5}$$

$$\frac{A.C = X_1 ... X_n \in P(G) \quad t_i \in TP^{X_i}(G) \quad 1 \leq i \leq n}{[A.C = t_1 ... t_n] \in TP^A(G)} \tag{2.6}$$

□

A tree pattern p denotes a set of parse trees, namely the set obtained by replacing each non-terminal A in p by the elements of $T^A(G)$. For example, the

[5] To be precise, SDF3 produces terms in the ATerm format [27].

pattern `[Exp.Add = [Exp.Var = ID] + Exp]` represents a *set* of parse trees for addition expressions where the first operand is a variable and the second operand is an arbitrary expression.

A tree pattern corresponds to the derivation tree for a sentential form. For example, consider the derivation $Exp \overset{Add}{\Rightarrow}_G \underline{Exp} + Exp \overset{Add}{\Rightarrow}_G Exp + Exp + Exp$ in which the leftmost expression ($\underline{Exp}$) of each sentential form derives an addition ($Exp + Exp$). This derivation is represented by the tree pattern `[Exp.Add = [Exp.Add = Exp + Exp] + Exp]`. In addition to tree patterns, we also define a mechanism to determine whether a tree *matches* a particular tree pattern.

Definition 2.2.7 (Matching). *Given a grammar G, a tree $t \in T(G)$, and a tree pattern $q \in TP(G)$, t matches q, i.e., $\mathcal{M}(t, q)$, iff:*

$$\frac{a \in \Sigma}{\mathcal{M}(a, a)} \tag{2.7}$$

$$\frac{[A.C = t_1 ... t_n] \in T^A(G)}{\mathcal{M}([A.C = t_1 ... t_n], A)} \tag{2.8}$$

$$\frac{[A.C = t_1 ... t_n] \in T^A(G) \quad [A.C = q_1 ... q_n] \in TP^A(G) \quad \mathcal{M}(t_i, q_i) \quad 1 \leq i \leq n}{\mathcal{M}([A.C = t_1 ... t_n], [A.C = q_1 ... q_n])} \tag{2.9}$$

If Q is a set of patterns then $\mathcal{M}(t, Q)$ if there is some $q \in Q$ such that $\mathcal{M}(t, q)$. □

Using the definition above, the tree

```
[Exp.Add = [Exp.Add = [Exp.Var = ID] + [Exp.Var = ID]] +
  [Exp.Var = ID]]
```

matches the pattern `[Exp.Add = [Exp.Add = Exp + Exp] + Exp]`.

2.2.4 *Ambiguity*

Context-free grammars are suitable to formally specify the syntax of programming languages concisely and declaratively. However, the price to pay for conciseness — for example in grammars found in reference manuals and academic papers — is that they are often ambiguous. While such grammars can be used to describe the *abstract syntax* of a language, they cannot be directly used to generate parsers, since for some sentences, there may be multiple ways to identify their structure according to the grammar. We formally define ambiguous grammars as follows.

Definition 2.2.8 (Ambiguous Grammar). *A grammar G is ambiguous if for some sentence $w \in L(G)$, $|\Pi(G)(w)| > 1$, i.e., parsing a sentence that belongs to the language defined by G produces more than one tree.* □

Lemma 2.2.9. *A grammar G is ambiguous if there exist at least two leftmost derivations that generate a sentence $w \in L(G)$.*

Proof. By Lemma 2.2.5. □

For instance, consider the grammar from Figure 2.2 is ambiguous and the sentence `a + b + c` defining the addition of three variables. Parsing this sentence produces two different trees:

```
[Exp.Add = a + [Exp.Add = b + c]]
[Exp.Add = [Exp.Add = a + b] + c]
```

which correspond to the following leftmost derivations:

(i) $\underline{Exp} \Rightarrow_G \underline{Exp} + Exp \Rightarrow_G a + \underline{Exp} \Rightarrow_G a + Exp + Exp \overset{*}{\underset{lm\ G}{\Rightarrow}} a + b + c$

(ii) $\underline{Exp} \Rightarrow_G \underline{Exp} + Exp \Rightarrow_G Exp + Exp + Exp \overset{*}{\underset{lm\ G}{\Rightarrow}} a + b + c$

Thus, the grammar is ambiguous.

2.2.5 *Explicit Disambiguation*

When grammars are used as a formalism to describe the abstract syntax of well-formed programs, they abstract over disambiguation. However, when grammars are used as a parser specification they should be unambiguous. Thus, for each sentence of the language defined by a context-free grammar, parsing this sentence should produce a single tree. For example, consider again the grammar in Figure 2.2. Even though this grammar is perfectly adequate to identify the abstract syntax of the expressions it defines, it cannot be used to describe the mapping from sentences to trees, as it does not unambiguously identify the structure of an input program. To determine whether a context-free grammar is ambiguous or not is an undecidable problem [36, 48, 34].

One approach to preserve the conciseness of context-free grammars and *avoid* ambiguity of individual programs is to support explicit disambiguation. For instance, in the case of ambiguities due to operator priority and associativity, *parenthesized expressions* allow programmers to specify their preferred precedence in each expression they construct. SDF3 supports defining explicit disambiguation while preserving the abstract syntax of programs by `bracket` productions.

```
context-free syntax
  Exp.Add       = Exp "+" Exp
  Exp.Sub       = Exp "-" Exp
  Exp.Mul       = Exp "*" Exp
  Exp.Var       = ID
  Exp           = "(" Exp ")" {bracket}
```

Figure 2.4 Using brackets to explicitly disambiguate programs.

A production with a `bracket` annotation does not have a constructor and therefore preserves the *abstract syntax* of the program. At the same time, such productions allow programmers to parenthesize expressions, encoding their desired precedence in the expressions they construct. For example, consider the grammar in Figure 2.4 and the ambiguous sentence `a * b + c`. A programmer

may then write `a * (b + c)`, indicating that the addition to the right should have higher priority. Parsing this new sentence produces a single parse tree:

```
[Exp.Mul = a * [Exp = ([Exp.Add = b + c])]]
```

with abstract syntax tree `Mul(a, Add(b, c))`. While this approach can be used to construct alternative *sentences* that are unambiguous, it does not disambiguate the *context-free grammar* itself, since the grammar remains ambiguous for the sentence `a * b + c`.

2.2.6 *Disambiguation Filters*

Ambiguous grammars can be disambiguated by transforming it into an unambiguous grammar (Section 2.3.2) (provided it is not inherently ambiguous). However, the resulting grammars are typically hard to read and maintain and do not preserve the intended abstract syntax tree structure. For this reason we are interested in *disambiguation as a separate concern*. In particular, this chapter is about disambiguation by means of *associativity and priority declarations*, as we will discuss in more detail in Section 2.3.1. To formalize the semantics of such declarations we use the general notion of *disambiguation filters*, which were introduced to reason about a variety of *disambiguation mechanisms* for context-free grammars [74, 123]. We recall the definition by Klint and Visser [74] of a filter that disambiguates a grammar by selecting the preferred trees from a set of trees, and introduce the new notion of well-formed trees under subtree exclusion.

Definition 2.2.10 (Disambiguation Filter)**.** *A filter F for a CFG G is a function $F : \mathcal{P}(T(G)) \to \mathcal{P}(T(G))$ that maps sets of parse trees to sets of parse trees, where $F(\Phi) \subseteq \Phi$ for any $\Phi \subseteq T(G)$. The disambiguation of a CFG G by a filter F is denoted by G/F. The language $L(G/F)$ generated by G/F is defined as follows:*

$$L(G/F) = \{w \in \Sigma^* \mid \exists \Phi \subseteq T(G),\ yield(\Phi) = \{w\},\ F(\Phi) = \Phi\} \tag{2.10}$$

The condition $F(\phi) \subseteq \phi$ ensures that filters only reduce the set of trees, instead of creating new ones. A parser for G/F is defined by the equality $\Pi(G/F)(w) = F(\Pi(G)(w))$. □

Definition 2.2.11 (Safe Filter)**.** *A filter F for grammar G is safe if it preserves the language of the grammar, i.e. if $w \in L(G)$ then $w \in L(G/F)$.* □

Thus, a filter is safe if it produces *at least one tree* for each sentence in the language of a grammar, i.e. if $w \in L(G)$ then $|F(\Pi(G)(w))| \geq 1$.

Definition 2.2.12 (Completely Disambiguating Filter)**.** *A filter F is completely disambiguating if all sentences are unambiguous, i.e. if $w \in L(G)$ then $|F(\Pi(G)(w))| \leq 1$.* □

Thus, a filter is completely disambiguating if it produces *at most one tree* for each sentence. Filters are often defined in *negative terms* by rejecting trees that are invalid in some sense. The following definition defines *subtree exclusion filters*, which reject trees that match a particular set of tree patterns.

Definition 2.2.13 (Subtree Exclusion Filter)**.** *Given a set Q of tree patterns, the subtree exclusion filter F^Q is defined by*

$$F^Q(\Phi) = \{t \in \Phi \mid \nexists t' \in sub(t) : \mathcal{M}(t', Q)\} \tag{2.11}$$

where $sub(t)$ denotes the set of all subtrees of t, including t itself. □

Definition 2.2.14 (Trees under Subtree Exclusion)**.** *Given a grammar G and a set of tree patterns Q, the family $T_X^Q(G)$ of well-formed trees under subtree exclusion are the smallest sets satisfying the following rules:*

$$\frac{a \in \Sigma \quad \neg\mathcal{M}(a, Q)}{a \in T_a^Q(G)} \tag{2.12}$$

$$\frac{\begin{array}{c} A.C = X_1...X_n \in P(G) \quad t = [A.C = t_1...t_n] \\ t_i \in T_{X_i}^Q(G) \text{ for } 1 \leq i \leq n \quad \neg\mathcal{M}(t, Q) \end{array}}{t \in T_A^Q(G)} \tag{2.13}$$

We denote with G^Q the grammar G under subtree exclusion with tree patterns Q. The language defined by G^Q is defined as $L^X(G^Q) = \{yield(t) \mid t \in T_X^Q(G)\}$. □

Lemma 2.2.15. *All trees produced by a grammar under subtree exclusion pass the subtree exclusion filter, i.e. $t \in T_X^Q(G) \iff t \in T_X(G) \wedge t \in F^Q(\{t\})$.*

Proof. By Definition 2.2.14 no subtree of $t \in T_X^Q(G)$ matches a pattern in Q. □

Lemma 2.2.16. *The language of a grammar filtered by a subtree exclusion filter is the same as the language of that grammar under subtree exclusion, i.e. $L(G/F^Q) = L(G^Q)$.*

Proof.

$$\begin{aligned} L_X(G/F^Q) &= \{w \in \Sigma^* \mid \exists \Phi \subseteq T_X(G),\ yield(\Phi) = \{w\},\ F^Q(\Phi) = \Phi\} \\ &= \{w \in \Sigma^* \mid \exists t \in T_X(G),\ yield(t) = w,\ F^Q(\{t\}) = \{t\}\} \\ &= \{w \in \Sigma^* \mid t \in T_X^Q(G),\ yield(t) = w\} \quad \text{(by Lemma 2.2.15)} \\ &= L_X(G^Q) \end{aligned}$$

□

Corollary 2.2.17. *A subtree exclusion filter for a set of patterns Q for a grammar G is safe if for each $w \in L(G)$ there is at least one $t \in T^Q(G)$ with $yield(t) = w$.*

Corollary 2.2.18. *A subtree exclusion filter for a set of patterns Q for a grammar G is completely disambiguating if $t_1, t_2 \in T^Q(G) \implies yield(t_1) \neq yield(t_2) \vee t_1 = t_2$*

2.2.7 *Expression Grammars*

In this chapter we consider the disambiguation of context-free grammars by means of associativity and priority declarations, which are not intended to solve all possible kinds of ambiguities in grammars, but rather to disambiguate *operators* in *expressions*. In the following sections we will consider the semantics of associativity and priority declarations for *expression grammars* of growing complexity.

Definition 2.2.19 (Basic Expression Grammars)**.** *The syntax of basic expressions is defined by basic expression grammars which have productions of the form:*

$$A.C = LEX \tag{2.14}$$
$$A.C = \triangleright\ A\ \triangleleft \tag{2.15}$$
$$A.C = A\ \oplus\ A \tag{2.16}$$
$$A.C = \blacktriangleright\ A \tag{2.17}$$
$$A.C = A\ \blacktriangleleft \tag{2.18}$$

such that A is a non-terminal, C is a constructor, and LEX is a (sequence of) lexical symbol(s). The symbols $\triangleright$, $\triangleleft$, $\blacktriangleright$, $\blacktriangleleft$, and $\oplus$ represent arbitrary lexical symbols corresponding to operator words, such that no operator word is a valid lexical symbol LEX.[6] □

```
context-free syntax
  Exp.Var       = ID
  Exp.Int       = INT
  Exp           = "(" Exp ")" {bracket}
  Exp.Add       = Exp "+" Exp
  Exp.Mul       = Exp "*" Exp
  Exp.Minus     = "-" Exp
  Exp.Lambda    = "\\" ID "." Exp
  Exp.Inc       = Exp "++"
  Exp.If        = "if" Exp "then" Exp
  Exp.IfElse    = "if" Exp "then" Exp "else" Exp
  Exp.Subscript = Exp "[" Exp "]"
  Exp.While     = "while" Exp "do" Exp "done"
  Exp.Match     = "match" Exp "with" PMatch
  Exp.Function  = "function" PMatch
  PMatch.Clause = ID "->" Exp
```

Figure 2.5 Example expression grammar.

Basic expressions consist of atomic (Equation 2.14) expressions, and infix (Equation 2.16), prefix (Equation 2.17), and postfix (Equation 2.18) operators.

[6] In general, lexical symbols *LEX* represent symbols of the grammar that are defined by lexical syntax, such as `INT` or `ID`.

For example, in the grammar of Figure 2.5, the `Exp.Int` and `Exp.Var` productions define atomic expressions, the `Exp.Add` and `Exp.Mul` productions define infix operators, the `Exp.Minus` and `Exp.Lambda` productions define prefix operators, and the `Exp.Inc` production defines a postfix operator. Note that distinguish operators from operator words, such that an operator may consist of a multiple operator words, such as '`\ID.`' in the production `Exp.Lambda`. Further, note that we conflate 'operators' and 'delimiters', the symbols that specify closed expressions (Equation 2.15) — e.g., the symbols `(` and `)` in the production annotated with `bracket`.

Basic expressions cover typical mathematical expressions. However, programming languages often feature more complicated expression forms with multiple and intercalated operator words, such as the `Exp.If` and `Exp.IfElse` productions in Figure 2.5, also known as *distfix* (or *mixfix*) operators [1].

Definition 2.2.20 (Distfix Grammars)**.** *The syntax of distfix expressions is defined by distfix expression grammars which have productions of the form:*

$$A.C = \blacktriangleright \;\; A \;\; \oplus_1 \;\; ... \;\; \oplus_k \;\; A \tag{2.19}$$

$$A.C = A \;\; \oplus_1 \;\; ... \;\; \oplus_k \;\; A \;\; \blacktriangleleft \tag{2.20}$$

$$A.C = A \;\; \oplus_1 \;\; ... \; A \;\; \oplus_k \;\; A \tag{2.21}$$

$$A.C = \triangleright \; A \;\; \oplus_1 \;\; ... \;\; \oplus_k \;\; A \; \triangleleft \tag{2.22}$$

where $\triangleright$, $\triangleleft$, $\blacktriangleright$, $\blacktriangleleft$, *and* $\oplus_i$ *stand for an arbitrary operator word that must be unique within the grammar. A context-free grammar containing only distfix and atomic expressions is called distfix grammar.* □

Distfix expressions can be classified as *infix distfix* (Equation 2.21), *prefix distfix* (Equation 2.19), and *postfix distfix* (Equation 2.20). The production in Equation 2.22 defines *closed distfix* expressions. For example, in the grammar of Figure 2.5, the productions `Exp.If`, `Exp.Subscript`, and `Exp.IfElse` define prefix distfix, postfix distfix and infix distfix expressions, respectively. The production `Exp.While` defines a closed distfix operator (with the operator words `while`, `do`, and `done`). The basic expression forms of Definition 2.2.19 are a subset of distfix expressions.

The distfix expression of Definition 2.2.20 are recursive in the same non-terminal. *Not* all productions for the non-terminal `Exp` in the grammar in Figure 2.5 define distfix expressions. The productions `Exp.Do` and `Exp.Function` compose multiple expressions using *indirect recursion*. Such productions are covered by the following extension of distfix grammars.

Definition 2.2.21 (Indirectly Recursive Distfix Grammars)**.** *Given* $A, B \in N$*, the syntax for indirect distfix expressions is defined by productions of the form:*

$$A.C = \blacktriangleright \;\; B_0 \;\; \oplus_1 \;\; ... \;\; \oplus_k \;\; B_k \tag{2.23}$$

$$A.C = B_0 \;\; \oplus_1 \;\; ... \;\; \oplus_k \;\; B_k \;\; \blacktriangleleft \tag{2.24}$$

$$A.C = B_0 \;\; \oplus_1 \;\; ... \;\; \oplus_k \;\; B_k \tag{2.25}$$

$$A.C = \triangleright \; B_0 \;\; \oplus_1 \;\; ... \;\; \oplus_k \;\; B_k \triangleleft \tag{2.26}$$

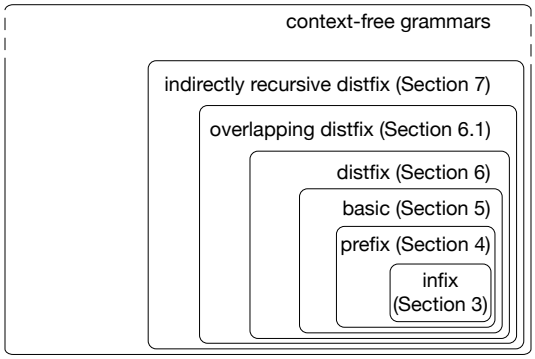


Figure 2.6 Expression grammars according to their complexity.

where $B_i \underset{lm\ G}{\overset{*}{\Rightarrow}} A\ \gamma$ *or* $B_i \underset{lm\ G}{\overset{*}{\Rightarrow}} \alpha\ A$ *for any* $0 \leq i \leq k$*, and* $\triangleright$*,* $\triangleleft$*,* $\blacktriangleright$*,* $\blacktriangleleft$*, and* $\oplus_i$ *stand for arbitrary operator words consisting of a lexical symbol. A grammar defining indirectly recursive distfix expressions is called indirectly recursive distfix grammar.* □

In this chapter we are interested in the semantics of associativity and priority rules for expression grammars. More specifically, we are interested in the following questions. (1) Is there a safe and complete semantics of disambiguation rules for arbitrary expression grammars (that also captures our intuition of these rules)? We will show that this is the case as long as such grammars only have harmless overlap. (2) Is it possible to determine for a *specific* expression grammar whether a set of rules fully disambiguates that grammar? We will show this is indeed possible. (3) How can we efficiently implement this disambiguation semantics? We will discuss several techniques. In the rest of the chapter we address these questions, where we consider expression grammars of increasing complexity as illustrated by Figure 2.6. It turns out that the complicated case of *deep priority conflicts* already surfaces in *prefix expression grammars* that just combine infix and prefix operators.

2.2.8 *Embedded Expression Grammars*

The grammars of programming languages are not restricted to expression grammars for a single expression non-terminal, but rather consist of many syntactic categories such as (top-level) declarations, functions, statements, and expressions. We want to apply the results of this chapter to full blown programming language grammars. In particular, we want to determine whether a set of disambiguation rules is complete, and we want to use the implementation techniques for these disambiguation rules. This is possible by observing that programming language grammars compose one or more expression grammars (for different expression non-terminals) with 'glue' productions (that are typically non-ambiguous) and lexical productions (that use different disam-

```
lexical syntax
  ID  = [a-zA-Z][a-zA-Z0-9]*
  INT = [0-9]+
  ID  = "if" {reject}
  ID  = "class" {reject}
lexical restrictions
  ID  -/- [a-zA-Z0-9]
  INT -/- [0-9]
context-free syntax
  Class.Class = "class" ID "{" Mem* "}"
  Mem.Method  = Type ID "(" Arg* ")" "{" Stmt* "}"
  Stmt.If     = "if" "(" Exp ")" Stmt
  Stmt.Expr   = Expr ";"
  Expr.Int    = INT
  Expr.Var    = ID
  Expr        = "(" Expr ")" {bracket}
  Expr.Add    = Expr "+" Expr {left}
  Expr.Mul    = Expr "==" Expr {non-assoc}
  Expr.Call   = Expr "." ID "(" {Exp ","}* ")"
context-free priorities
   Expr.Call > Expr.Add > Expr.Eq
```

Figure 2.7 Syntax definition for subset of Java with with embedded expression grammar.

biguation techniques). For example, consider the grammar in Figure 2.7 that defines a Java-like language with classes, methods, statements, expressions, identifiers, and integer constants. The definitions for lexicals `ID` and `INT` use lexical disambiguation rules — follow restrictions for longest match and reject productions for reserved words — that are different from the rules for expressions, and not further considered in this chapter. Embedded in this grammar is the expression grammar for `Expr` consisting of the productions for variables, constants, parentheses, addition, equality, and method calls. To determine that the disambiguaton rules are complete, we can consider just the productions for this embedded expression grammar. To implement the disambiguation rules, we can apply the techniques that we develop in this chapter to the grammar as whole, with the guarantee that a total set of disambiguation rules solves all ambiguities in the expression grammar. In Section 2.9.1 we discuss the extraction of expression grammars in more detail.

2.3 INFIX EXPRESSION GRAMMARS

We start our investigation with *infix expression grammars*, i.e., grammars that only define infix (and closed) operators. We introduce associativity and priority rules and their intuitive meaning, and we survey existing techniques for the interpretation of such rules.

Definition 2.3.1 (Infix Expression Grammars). *An infix grammar is an expression grammar with atomic, closed, and infix expressions.* □

2.3.1 *Declarative Disambiguation Rules*

```
context-free syntax
  Exp.Add = Exp "+" Exp {left}
  Exp.Sub = Exp "-" Exp {left}
  Exp.Mul = Exp "*" Exp {left}
  Exp.Pow = Exp "^" Exp {right}
  Exp.Mul = Exp "==" Exp {non-assoc}
  Exp.Var = ID
  Exp     = "(" Exp ")" {bracket}
context-free priorities
   Exp.Pow > Exp.Mul >
  {left: Exp.Add Exp.Sub} > Exp.Eq
```

Figure 2.8 SDF3 grammar with disambiguation declarations.

To disambiguate expression grammars, reference manuals often include a table defining the priority and associativity of its operators. Some syntax definition formalisms incorporate these concepts, enabling users to declaratively specify how to disambiguate grammars. While the basic intention of syntax of priority and associativity declarations does not vary much between formalisms, the definition of their semantics might be quite different, as we will see in the remainder of this section. Below, we informally describe the declarative disambiguation constructs of SDF3, which is based on the disambiguation rules of its predecessors from the SDF family of syntax formalisms (cf. Section 2.11).

The SDF family of syntax definition formalisms defines associativity and priority declarations on *productions* rather than on *operator* symbols. For example, consider the SDF3 grammar in Figure 2.8. The `left`, `right`, `non-assoc` production annotation on a production indicates that it is *left associative*, *right associative*, or *non-associative* with respect to itself. For example, the `left` annotation on `Exp.Add` implies the relation `Exp.Add left Exp.Add`, and declares that addition is left associative; e.g. the sentence `a + b + c` has the following parse trees:

```
[Exp.Add = [Exp.Add = a + b] + c]
[Exp.Add = a + [Exp.Add = b + c]]
```

The first of these trees corresponds to the left associative interpretation of the operator, the second to the right associative interpretation. Hence, declaring the production to be left associative, defines the first tree to be the correct parse tree of the sentence.

Associativity groups such as {`left`: `Exp.Add Exp.Sub`} indicate the mutual relation between the set of productions in the group (`Exp.Add left Exp.Sub`

and `Exp.Sub left Exp.Add`), that is, addition is left associative with respect to subtraction and vice-versa. For example, the sentence `a + b - c` has trees:

```
[Exp.Add = a + [Exp.Sub = b - c]]
[Exp.Sub = [Exp.Add = a + b] - c]
```

of which the second is the correct parse tree according to the associativity of the operators.

The priority rule $p_1 > p_2$ indicates that the (group of) production(s) p_1 has higher priority than the (group of) production(s) p_2. For instance, the priority `Exp.Mul > {left: Exp.Add Exp.Sub}` indicates that multiplication has higher priority than addition (`Exp.Mul > Exp.Add`) and subtraction (`Exp.Mul > Exp.Sub`), whereas the priority `{left: Exp.Add Exp.Sub} > Exp.Eq` indicates that both addition and subtraction have higher priority than equality expressions. Priority rules are transitive and irreflexive, whereas associativity rules are symmetric and non-transitive. For example, the sentence `a + b * c` has trees:

```
[Exp.Add = a + [Exp.Mul = b * c]]
[Exp.Mul = [Exp.Add = a + b] * c]
```

of which the first is correct according to the priority of the operators.

We have sketched the interpretation of associativity and priority declarations, appealing to the common intuitition about these notions. But what is the precise semantics of these declarations? And how does that extend to more complicated expression grammars? In the rest of this section we study existing techniques for interpreting and implementing disambiguation declarations, which *indirectly* define the intuition about the meaning of associativity and priority in terms of other mechanisms. We end with a semantics based on subtree exclusion, which directly corresponds to the informal explanation above.

2.3.2 *Grammar Rewriting*

To disambiguate an expression grammar, one can rewrite it to an unambiguous grammar by introducing a non-terminal for each priority level, and by guaranteeing that productions are either left or right recursive. For example, Figure 2.9 shows the result of disambiguating the grammar from Figure 2.2. The grammar directly encodes the fact that addition, subtraction and multiplication are left associative and that multiplication has higher priority than addition and subtraction. Using this grammar, the string `a + b * c` has only a single parse tree, as shown in Figure 2.10.

Directly encoding priority and associativity in the grammar has three major drawbacks. First, the grammar loses its direct mapping to the abstract syntax, due to the additional non-terminals and productions to encode priority levels. For example, the parse tree from Figure 2.10 directly corresponds to the following abstract syntax tree:

```
Add(Term(Fact(Var("a"))),
    Mul(Term(Fact(Var("b"))), Fact(Var("c"))))
```

```
context-free syntax
  Exp.Add    = Exp "+" Term
  Exp.Sub    = Exp "-" Term
  Exp.Term   = Term
  Term.Mul   = Term "*" Factor
  Term.Fact  = Factor
  Factor.Var = ID
  Factor     = "(" Exp ")" {bracket}
```

Figure 2.9 Encoding precedence and associativity in the grammar productions.

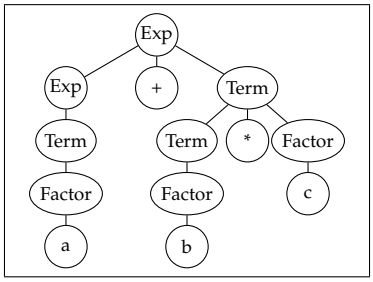


Figure 2.10

Such abstract syntax tree encodings typically make a custom mapping to abstract syntax necessary. Second, the grammar obscures the conceptual notions of associativity and priority by encoding it in grammar productions. Third, the approach does not scale. While the encoding is fairly simple for infix expressions, it becomes complex when considering other expression forms such as low priority prefix operators (e.g. let expressions in functional languages) and dangling else, which require duplication of productions to be encoded in the grammar.

To support declarative specification of associativity and priority, various techniques implement declarative disambiguation by translating a concise context-free grammar extended with disambiguation constructs into an unambiguous grammar [1, 5]. However, none of these approaches handles *all* types of ambiguities in expression grammars.

2.3.3 *Operator Grammars*

Operator grammars are a class of grammars that represent expressions in programming languages [11]. An operator grammar has two specific properties: no production right-hand side is the empty string, nor does it have two adjacent non-terminals. Our definition of infix expression grammars also satisfies these requirements, but operator grammars are more expressive than infix expression

Relation	Meaning
$a \lessdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \gtrdot b$	a "takes precedence over" b

Figure 2.11 Precedence relations in operator precedence grammars.

grammars, allowing other operators, such as prefix or postfix operators.

Operator grammars can be parsed by an easy-to-implement parsing technique called *operator precedence parsing*. This technique defines three disjoint precedence relations $\lessdot$, $\doteq$, and $\gtrdot$, between certain pairs of terminals indicating lower priority, same priority level, and higher priority, as shown in Figure 2.11. These relations dictate how to parse expressions because they can be used to delimit sentential forms, indicating the extent of each expression, and which one should have higher priority. Parsing works by changing the input to encode the precedence between each terminal, such that the parser can easily recognize which trees to construct by traversing the input. While seemingly related, the precedence relation in an operator grammar is not the same as associativity and priority. Encoding the latter in the former is non-trivial, and the resulting precedence table cannot be read as a description of associativity and priority [11].

2.3.4 *Disambiguation by Shift/Reduce Conflict Resolution*

Disambiguation rules in YACC are defined by associativity and priority directives on the *tokens* corresponding to operators [64]. The grammar in Figure 2.12 is an infix expression grammar written in YACC. The `%left` directive indicates that the following operators are left associative. Furthermore, the lines are listed by increasing priority, i.e., the operator `*` has higher priority than the operators `+` and `-`. Operators on the same line have the same priority.

The semantics of YACC's disambiguation constructs relies on its implementation using LR parser generation. From an ambiguous expression grammar such as the grammar in Figure 2.12, YACC generates an LR parse table contains conflicts as shown in Figure 2.13. These conflicts indicate that there are ambiguous interpretations of the input or that the parser needs more lookahead. If the conflict is caused by an ambiguity involving operators in the priority declaration, YACC selects one alternative over another based on the priority declarations, producing a deterministic parser as a result.

For example, when parsing a sentence `a + b + c` and reaching the conflict state 9 after parsing the variable `b`, as shown by the configuration in Figure 2.14a, YACC's parser prefers the reduce action `r2` over the shift action `s5` in the conflict in the column `+`. This results in the sentence being interpreted as `(a + b) + c`, instead of `a + (b + c)`, i.e., the addition operation is interpreted as left associative. However, when parsing a sentence `a + b * c` and reaching the conflicting state 9 after parsing `b`, as shown by the configuration in Figure 2.14b, the parser prefers the shift action `s4` in the column `*`, resulting

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

Expr
     : Expr '+' Expr
     | Expr '-' Expr
     | Expr '*' Expr
     | '(' Expr ')'
     | IDENTIFIER
     ;
```

Figure 2.12 YACC grammar for arithmetic expressions.

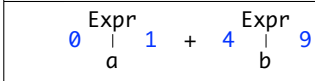
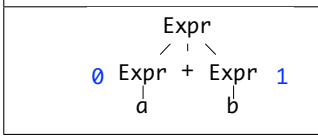
State	Action							GOTO
	*	+	-	(	)	ID	$	E
0				s2		s3		1
1	s4	s5	s6				acc	
...								
9	s4/r2	s5/r2	s6/r2		r2		r2	
...								

Figure 2.13 SLR(1) parse table for the grammar in Figure 2.12.

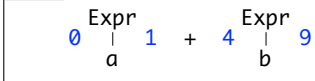
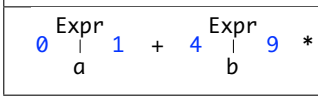
in the sentence being interpreted as `a + (b * c)`, implementing the higher priority of the multiplication over the addition.

The YACC semantics for disambiguation directives does not carry over to other parsing algorithms. In particular, since the selection of a parse action is based on the next token, this solution is not suitable for scannerless parsing, where the lookahead token is the next character, which might be layout (whitespace and comments) [130]. YACC's semantics of selecting a particular action over another may also produce unexpected results for more complex ambiguities such as dangling suffix, as we will discuss in Section 2.6.2. Furthermore, a semantics that depends on a particular parsing algorithm is not portable, and limits grammars to some deterministic subset, such as LL or LR, of the set of all context-free grammars. Finally, limiting grammars to a deterministic subset inhibits grammar (language) composition, since only the full class of context-free grammars is closed under composition. That is, composing the grammars of two languages may require modifying the grammars in order to solve parse table conflicts.

For example, when parsing a sentence `a + b + c` and reaching the conflict state 9 after parsing the variable `b`, as shown by the configuration in Figure 2.14a, YACC's parser prefers the reduce action `r2` over the shift action `s5` in the conflict in the column `+`. This results in the sentence being interpreted as `(a + b) + c`, instead of `a + (b + c)`, i.e., the addition operation is interpreted as left associative. However, when parsing a sentence `a + b * c` and

Stack	Remaining Input	Action
	+ c	r2
	+ c	s4

(a) Conflict between addition operators.

Stack	Remaining Input	Action
	* c	s6
	c	s3

(b) Conflict between addition and multiplication.

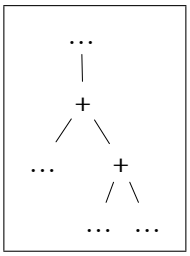
Figure 2.14 Solving parse table conflicts based on disambiguation rules.

reaching the conflicting state 9 after parsing `b`, as shown by the configuration in Figure 2.14b, the parser prefers the shift action `s4` in the column `*`, resulting in the sentence being interpreted as `a + (b * c)`, implementing the higher priority of the multiplication over the addition.

2.3.5 *Tree Automata*

Another approach to describe disambiguation of expression grammars involves tree automata [4, 39]. Disambiguation works by expressing invalid patterns using tree automata, and intersecting the automata with a context-free grammar. The intersection produces a context-free grammar that avoids invalid patterns. For instance, Figure 2.15 shows a specification that captures invalid patterns corresponding to the associativity of addition expressions. Figure 2.15b shows the specification of the invalid patterns in Figure 2.15a using a regular tree expression. Figure 2.15c shows the automaton produced by this specification. The intuition is that when traversing to the right side of an addition (transition $+2$ to state $+A$), an addition is forbidden (there is no transition from $+A$ that involves an addition). This automaton is later intersected with the original grammar, producing a grammar that does not allow the violating pattern.

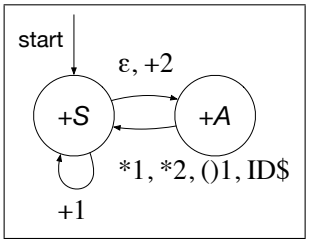
This approach allows expressing a variety of transformations on context-free grammars. However, it does not provide a semantics for associativity and priority, nor does it provide a convenient and concise notation for expressing grammar disambiguations for language engineers. A semantics *can* be defined by defining a translation from associativity and priority declarations to regular



(a) Invalid pattern.

```
(any* .
        +(_, +(_, _)))
```

(b) Specification of invalid pattern.



(c) Tree automaton.

Figure 2.15 Disambiguation using Tree Automata.

tree expressions. But establishing the *correctness* of such a translation requires a semantics for these declarations in the first place. Furthermore, the tree automata approach (by design) does not guarantee safety or completeness of disambiguation.

2.3.6 *Subtree Exclusion*

In Section 2.3.1, we informally explained disambiguation as the selection of a particular tree among the possible trees in an ambiguity. In the preceding subsections, we have seen several *indirect* definitions of the semantics of disambiguation declarations. Following the earlier work of Thorup [123] and Klint and Visser [74], we provide a *direct* semantics of associativity and priority rules as *subtree exclusion*, which is independent of a particular parsing algorithm and enables reasoning about the *safety* and *completeness* of a disambiguation mechanism. Earlier definitions of subtree exclusion did not (safely and completely) cover the full class of expression grammars, as we do in this chapter. To understand the issues and how they are addressed, we first study the definition by Visser [126] of the semantics of disambiguation in SDF2.

To understand the shape of ambiguities in infix expression grammars, Table 2.16 shows all possible tree shapes constructed by an infix expression grammar. From the table, it should be clear that only the combination of infix productions causes ambiguities. Thus, in order to disambiguate an infix expression grammar, we need to make a choice for one of the trees in the middle cell, for each pair of productions. The following definition defines the semantics of associativity and priority declarations by defining the tree patterns that they are *conflict* with.

Definition 2.3.2 (SDF2 Semantics: Priority Conflict Patterns)**.** *Given a grammar G extended with disambiguation rules PR defining symmetric, non-transitive relations* `right`, `left`, `non-assoc`*, and irreflexive, anti-symmetric, and transitive*[7] *relation $>$ over the infix*[8] *productions of G such that there is at most one relation between each*

[7] SDF2 also supports specification of relations with a *dot* (.>), as non-transitive variant of $>$.

[8] The original definition of Visser [126] does not make this restriction, leading to unsafe disambiguation.

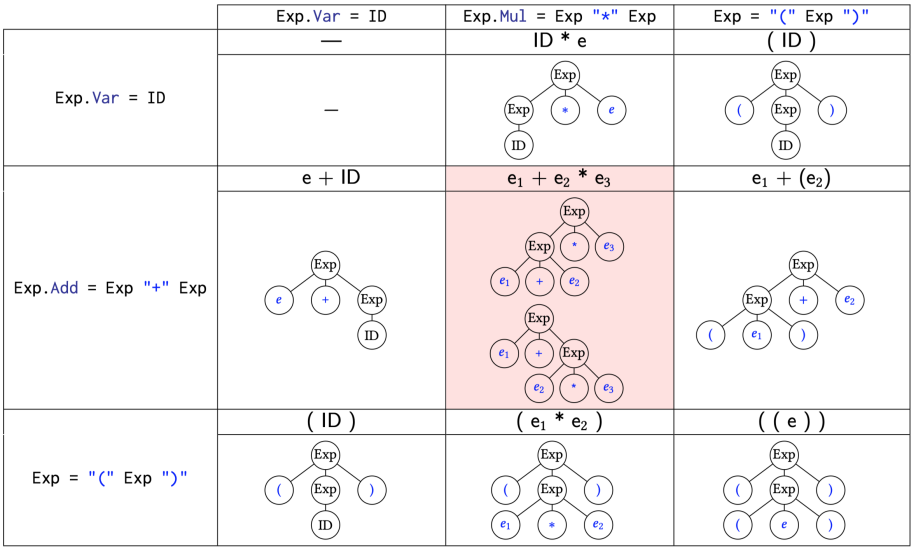


Figure 2.16 Examples of all direct combinations of closed, infix, and atomic expressions in infix expression grammars.

pair of productions, the set of priority conflict patterns Q_G is defined as follows:

$$\frac{A.C_1 > A.C_2 \in PR}{[A.C_1 = \alpha[A.C_2 = \beta]\gamma] \in Q_G} \tag{2.27}$$

$$\frac{A.C_1 \texttt{ right } A.C_2 \in PR}{[A.C_1 = [A.C_2 = \beta]\gamma] \in Q_G} \tag{2.28}$$

$$\frac{A.C_1 \texttt{ left } A.C_2 \in PR}{[A.C_1 = \alpha[A.C_2 = \beta]] \in Q_G} \tag{2.29}$$

$$\frac{A.C_1 \texttt{ non-assoc } A.C_2 \in PR}{[A.C_1 = [A.C_2 = \beta]\gamma] \in Q_G} \tag{2.30}$$

$$\frac{A.C_1 \texttt{ non-assoc } A.C_2 \in PR}{[A.C_1 = \alpha[A.C_2 = \beta]] \in Q_G} \tag{2.31}$$

A relation $A.C_1$ R $A.C_2$ is mutually exclusive, e.g., if $A.C_1$ `left` $A.C_2 \in PR$, then $A.C_1 > A.C_2 \notin PR$. With Q_G we construct the subtree exclusion filter F^{Q_G}. Thus, the semantics of a grammar G with disambiguation declarations Pr(G) is the disambiguated grammar G/F^{Q_G}. □

The set of conflict patterns Q_G captures the trees that are invalid according to the disambiguation rules in the grammar. The priority rule $A.C_1 > A.C_2$ forbids the tree constructed by $A.C_2$ to occur as a direct descendant of the tree constructed by $A.C_1$. The rule $A.C_1$ `right` $A.C_2$ forbids the tree $A.C_2$ to occur as the direct descendant at the *leftmost position* of tree $A.C_1$. Similarly,

the rule $A.C_1$ `left` $A.C_2$ forbids the tree $A.C_2$ to occur as a direct descendant at the *rightmost position* of the tree $A.C_1$. Finally, the rule $A.C_1$ `non-assoc` $A.C_2$ forbids the tree $A.C_2$ to occur as a direct descendant at the *rightmost and leftmost positions* of the tree $A.C_1$.

Let us consider some concrete examples of associativity and priority. The priority declaration `Exp.Mul > Exp.Add` generates the following pattern in Q_G according to Equation 2.27:

$$\frac{\texttt{Exp.Mul} > \texttt{Exp.Add} \in PR}{\texttt{[Exp.Mul = [Exp.Add = Exp + Exp] * Exp]} \in Q_G}$$

The pattern `[Exp.Mul = [Exp.Add = Exp + Exp] * Exp]` defines that any tree that contains an addition as the leftmost child of a multiplication is invalid and should be rejected. Therefore, the only valid tree for the sentence `a + b * c` is

```
[Exp.Add = a + [Exp.Mul = b * c]]
```

since the other tree in the ambiguity

```
[Exp.Mul = [Exp.Add = a + b] * c]
```

matches the conflict pattern above. Note that the pattern indicates that the addition is forbidden as the direct descendant of a multiplication at any position. Hence, the pattern `[Exp.Mul = Exp * [Exp.Add = Exp + Exp]]` is also created by the same priority declaration.

The associativity declaration in the production `Exp.Add = Exp + Exp {left}` generates the following conflict pattern according to Equation 2.29:

$$\frac{\texttt{Exp.Add left Exp.Add} \in PR}{\texttt{[Exp.Add = Exp + [Exp.Add = Exp + Exp]]} \in Q_G}$$

The pattern defines left associativity by ruling out the right-associative variant. Thus, for a sentence `a + b + c`, the only valid tree is

```
[Exp.Add = [Exp.Add = a + b] + c]
```

and the other tree in the ambiguity

```
[Exp.Add = a + [Exp.Add = b + c]]
```

is invalid, since it matches the conflict pattern.

2.3.7 *Safe and Complete Disambiguation*

The semantics from Definition 2.3.2 directly defines how to disambiguate expressions by filtering parse trees. We would like to know whether the disambiguation of a grammar with disambiguation rules is *safe* and *completely disambiguating* as defined in Section 2.2.6. Safety guarantees that the language of the underlying context-free grammar is preserved, i.e. each sentence in the language of the grammar is also in the language of the disambiguated grammar. Completely disambiguating means that all sentences in the disambiguated grammar have at most one parse tree. We first consider safety.

Lemma 2.3.3 (Subtree Exclusion is Safe)**.** *Given an infix expression grammar G and a set Q of priority conflict patterns generated by disambiguation rules (not including* `non-assoc` *) for G, if $w \in L(G)$ then there is a $t \in T^Q(G)$ such that $yield(t) = w$.*

Proof. By induction on the length of sentences in $L(G)$.

(Base case) If a is a lexeme then $a \in T^Q_a(G)$ since disambiguation rules do not exclude lexemes.

(Inductive case) Assume that $u, v \in L(G)$ and that there are $t_1, t_2 \in T^Q_A(G)$ such that $yield(t_1) = u$, $yield(t_2) = v$, then there are two cases:

(1) If $A.C = \triangleleft\ A\ \triangleright$ is a closed production in G, then $\triangleleft\ u\ \triangleright \in L(G)$ and $[A.C = \triangleleft\ t_1\ \triangleright] \in T^Q_A(G)$, since there is no priority conflict pattern that matches this tree. (Note that the original definition of Visser [126] does not restrict priority relations to infix productions. Via Equation 2.27 a priority relation $A.C > A.C'$ for some production $A.C' = \alpha$ in the grammar would lead to rejecting a tree $[A.C = \triangleleft\ [A.C' = ...]\ \triangleright]$, and hence the corresponding sentence.)

(2) If $A.C = A\ \oplus\ A$ is an infix production in G, then $u \oplus v = w \in L(G)$. Now we need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. By induction $v = yield(t_1)$ and $v = yield(t_2)$ such that $t_1, t_2 \in T^Q(G)$. We consider the following cases:

- If t_1 and t_2 are lexemes or closed expressions then $t = [A.C = t_1 \oplus t_2] \in T^Q(G)$ since there are no disambiguation rules that apply.
- If $t_1 = [A.C_1 = t_{11} \otimes t_{12}]$ with yield $u_{11} \otimes v_{12}$ and $t_2 = [A.C_2 = \triangleleft t_{21} \triangleright]$ with yield $\triangleleft w_{21} \triangleright$. Take $t = [A.C = [A.C_1 = t_{11} \otimes t_{12}]\ \oplus\ [A.C_2 = \triangleleft t_{21} \triangleright]]$ as the obvious candidate as tree for w. If $A.C_1 > A.C$ then $t \in T^Q(G)$ since it does not match a conflict pattern (since there are no other disambiguation relations between the productions). On the other hand, if $A.C > A.C_1$ then t matches a conflict pattern and therefore $t \notin T^Q(G)$. However, the reordering $t' = [A.C_1 = t_{11} \otimes [A.C = t_{12}\ \oplus\ [A.C_2 = \triangleleft t_{21} \triangleright]]]$ has the same yield and does *not* have a priority conflict, therefore $t' \in T^Q(G)$. If t_2 is a lexeme, or the disambiguation relation is `left` , `right` , the proof works analogously.
- The proof works analogously when t_1 is a lexeme or closed expression and t_2 is an infix expression.
- When both t_1 and t_2 are infix expressions we have to consider more cases, but the reasoning is analogous: by the fact that there is at most one disambiguation relation between each pair of operators, we can always construct a non-conflicted tree for the sentence by re-ordering the sub-expressions of t_1 and t_2. □

Next we consider the completeness of disambiguation. Clearly, if a grammar does not declare disambiguation rules for some/any productions in the grammar, it will not be completely disambiguating. In general, for arbitrary context-free grammars, it is undecidable whether a grammar is ambiguous or not. However, we establish that constructing a *total set of disambiguation rules*, which defines a disambiguation relation between each pair of (relevant) productions, guarantees complete disambiguation of an expression grammar.

Definition 2.3.4 (Total Set of Disambiguation Rules for Infix Expression Grammars)**.** *A set of disambiguation rules PR for an infix expression grammar G is total for a non-terminal A:*

- *If for any pair of productions* $A.C_1 = A\ op_1\ A \in P(G)$*, and* $A.C_2 = A\ op_2\ A \in P(G)$*, such that* $A.C_1 \neq A.C_2$*, either* $A.C_1\ R\ A.C_2 \in PR$ *or* $A.C_2\ R\ A.C_1 \in PR$ *where* $R \in \{>, \texttt{right}, \texttt{left}\}$*.*
- *If* $A.C = A\ op\ A \in P(G)$ *then* $A.C\ R'\ A.C \in PR$ *where* $R' \in \{\ \texttt{right}, \texttt{left}, \texttt{non-assoc}\}$*.*

Lemma 2.3.5 (Subtree Exclusion is Completely Disambiguating)**.** *Given an infix expression grammar G and a set Q of priority conflict patterns generated by a total set of disambiguation rules for G, then all trees in* $T^Q(G)$ *have unique yields. That is, if* $t_1, t_2 \in T^Q(G)$ *and* $yield(t_1) = yield(t_2)$ *then* $t_1 = t_2$*.*

Proof. By induction on $T^Q(G)$.

(Base case) If a is a lexeme, then $a \in T^Q_a(G)$ and has a unique yield.

(Inductive case) Assume that $t_1, t_2 \in T^Q_A(G)$ and that their yields are unique.

(1) If $A.C = \triangleleft\ A\ \triangleright$ is a closed production in G, then $t = [A.C = \triangleleft\ t_1\ \triangleright] \in T^Q_A(G)$, since there is no priority conflict pattern that matches this tree, and the fact that each constructor uniquely identifies a production, by uniqueness of t_1, t is also unique.

(2) If $A.C = A\ \oplus\ A$ is an infix production in G, since each constructor uniquely identifies a production, that is the only way we can construct the tree $t = [A.C = t_1\ \oplus\ t_2]$. Now we need to demonstrate that if $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$. We consider the following cases:

   - If t_1 and t_2 are lexemes or closed expressions then $t \in T^Q(G)$ since there are no disambiguation rules that apply. By uniqueness of t_1 and t_2 and non-overlap of productions, there are no other ways to construct a tree with the same yield as t.
   - If $t_1 = [A.C_1 = t_{11} \otimes t_{12}]$ with yield $u \otimes v$ and $t_2 = [A.C_2 = \triangleleft t_{21} \triangleright]$ with yields $\triangleleft w \triangleright$ then $t = [A.C = [A.C_1 = t_{11} \otimes t_{12}]\ \oplus\ [A.C_2 = \triangleleft t_{21} \triangleright]]$ with yield $u \otimes v \oplus \triangleleft w \triangleright$. By totality of disambiguation rules,

we have that there is a disambiguation relation between $A.C$ and $A.C_1$. If $A.C > A.C_1$ then t matches a conflict pattern and therefore $t \notin T^Q(G)$. If $A.C_1 > A.C$ then t does not match a conflict pattern (since there are no other disambiguation relations between the productions). The only other tree with the same yield is $t' = [A.C_1 = t_{11} \otimes [A.C = t_{12} \oplus [A.C_2 = \triangleleft t_{21} \triangleright]]] \in T^Q(G)$. However, t' *does* have a priority conflict and therefore $t' \notin T^Q(G)$. If the disambiguation relation is `left`, `right`, or `non-assoc`, the proof works analogously.

- The proof works analogously when t_1 is a lexeme or a closed expression and t_2 is an infix expression.
- When both t_1 and t_2 are infix expressions, we have to consider more cases since all combinations of disambiguation relations between the three productions need to be considered, but the reasoning is the same; by totality there are relations between all three productions, and therefore at most one tree is selected. □

Theorem 2.3.6. *Disambiguation of an infix expression grammar using a total set of disambiguation rules (not including `non-assoc`) is safe and completely disambiguating.*

Proof. Assume that G is an infix expression grammar and R a total set of disambiguation rules for G. Let Q be the set of priority conflict patterns for R according to Definition 2.3.2. By Lemma 2.3.3 we have that if $w \in L(G)$ then there is a $t \in T^Q(G)$ such that $yield(t) = w$. By Corollary 2.2.17 we have that F^Q is a safe disambiguation filter. By Lemma 2.3.5 we have that if $t_1, t_2 \in T^Q(G)$ then $yield(t_1) \neq yield(t_2) \vee t_1 = t_2$. By Corollary 2.2.18 we have that F^Q is completely disambiguating. □

2.3.8 *Explicit Unsafety*

Safety preserves the language of the underlying grammar. We have proven that disambiguation with associativity and priority is safe, but excluded `non-assoc` from that proof, because in fact this construct is *not* safe. That is, declaring production as non-associative removes sentences from the language of a grammar.

That sounds like an error, but in some cases, unsafe filters are actually desired. For example, consider the equality comparisons in the grammar of Figure 2.17 and the sentence `a == b == c`, which is in the language of the grammar. The parse trees for the sentence are:

```
[Exp.Eq = a == [Exp.Eq = b == c]]
[Exp.Eq = [Exp.Eq = a == b] == c]
```

Since the production `Exp.Eq` is annotated with the disambiguation directive `non-assoc`, it generates the following conflict patterns:

$$\frac{\texttt{Exp.Gt non-assoc Exp.Gt} \in PR}{\texttt{[Exp.Eq = [Exp.Gt = Exp == Exp] == Exp]} \in Q_G}$$

```
context-free syntax
  Exp.Add = Exp ">" Exp {non-assoc}
  Exp.Sub = Exp "<" Exp {non-assoc}
  Exp.Mul = Exp "==" Exp {non-assoc}
  Exp.Var = ID
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}
context-free priorities
  {non-assoc: Exp.Gt Exp.Lt Exp.Eq}
```

Figure 2.17 SDF3 grammar containing non-associative expressions.

$$\frac{\texttt{Exp.Gt non-assoc Exp.Gt} \in PR}{\texttt{[Exp.Eq = Exp == [Exp.Eq = Exp == Exp]]} \in Q_G}$$

These conflict patterns match both of the trees in the ambiguity, which are therefore rejected. Hence, the filter F^{Q_G} is unsafe, when considering `non-assoc` declarations. While this approach forbids constructing ambiguous sentences by rejecting the sentence altogether, it does allow that language users use explicit disambiguation to choose one of the parse trees. For example, users may still use parentheses to specify the priority of the operator, writing the expression as either `(a == b) == c` or `a == (b == c)`, which are both syntactically valid.

Treating non-assoc as an unsafe filter in this fashion leads to a parse error. This does not provide a very useful experience for the programmer, since it invokes parse error recovery for a program that has a perfectly fine shape. Thus, we propose to treat non-associativity as a semantic error rather than as a syntactic error, as follows. We define the righ-associative parse as a conflict, as before:

$$\frac{A.C_1 \texttt{ non-assoc } A.C_2 \in PR}{[A.C_1 = \alpha[A.C_2 = \beta]] \in Q_G} \tag{2.32}$$

However, rather than making the left associative parse a conflict as well, we flag it as a warning using the separate pattern set Q_G^W:

$$\frac{A.C_1 \texttt{ non-assoc } A.C_2 \in PR}{[A.C_1 = [A.C_2 = \beta]\gamma] \in Q_G^W} \tag{2.33}$$

Thus, non-assoc productions are treated as left associative, allowing programs with violations to be parsed, but occurrences of direct combinations of non-associative operators are flagged and produce a warning or error in the user interface. In the rest of the chapter we will treat non-associativity in this manner, i.e. define additional rules for the Q^W set, but abstract from its further treatment.

```
context-free syntax
  Exp.Add    = Exp "+" Exp {left}
  Exp.Lambda = "\\" ID "." Exp
  Exp.Minus  = "-" Exp
  Exp.Var    = ID
  Exp        = "(" Exp ")" {bracket}
context-free priorities
  Exp.Minus > Exp.Add > Exp.Lambda
```

Figure 2.18 Example prefix expression grammar.

2.4 PREFIX EXPRESSION GRAMMARS

In this section, we consider the extension of infix expression grammars with prefix expressions. Consider the grammar in Figure 2.18 with the `Exp.Minus` and `Exp.Lambda` productions, which define unary minus and lambda abstraction for prefix expressions, such as `- e` and `\x. e`. Since prefix operators are only right recursive, they do not require associativity annotations. However, in the absence of a priority specification, the combination of prefix and infix operators may be ambiguous as is the case with the sentence `- a + b`, which has the following parse trees:

```
(1) [Exp.Minus = - [Exp.Add = a + b]]
(2) [Exp.Add = [Exp.Minus = - a] + b]
```

How does the semantics of disambiguation rules from Definition 2.3.2 extend to prefix operators? First consider the case of prefix minus, which usually has higher priority than infix operators for arithmetic expressions, i.e. `Exp.Minus > Exp.Add`, with the following conflict pattern:

$$\frac{\texttt{Exp.Minus} > \texttt{Exp.Add} \in PR}{\texttt{[Exp.Minus = - [Exp.Add = Exp + Exp]]} \in Q_G}$$

Thus for the `- a + b` sentence, tree (2) above is selected, since tree (1) matches the conflict pattern. The sentence `a + - b` has only one parse tree

```
(3) [Exp.Add = a + [Exp.Minus = - b]]
```

which is valid since `Exp.Minus` > `Exp.Add`.

2.4.1 *The SDF2 Semantics is Unsafe for Prefix Grammars*

Next we consider the productions `Exp.Add` and `Exp.Lambda`. While the lambda operator is a prefix operator, it typically is given low priority (`Exp.Add > Exp.Lambda`) so that it is not necessary to enclose its body expression in parentheses. Definition 2.3.2 generates the following conflict patterns for this priority declaration:

$$\frac{\texttt{Exp.Add} > \texttt{Exp.Lambda} \in PR}{\texttt{[Exp.Add = [Exp.Lambda = \ ID . Exp] + Exp]} \in Q_G}$$

$$\frac{\texttt{Exp.Add} > \texttt{Exp.Lambda} \in PR}{\texttt{[Exp.Add = Exp + [Exp.Lambda = \ ID . Exp]]} \in Q_G}$$

The sentence `\x . a + b` has the following parse trees:

```
(4) [Exp.Add = [Exp.Lambda = \ x . a] + b]
(5) [Exp.Lambda = \ x .  [Exp.Add = a + b]]
```

Since tree `(4)` matches the first pattern, a filter based on this rule selects tree `(5)`. Similarly to the minus case, the sentence `a + \x. b` has only one parse tree:

```
(6) [Exp.Add = a + [Exp.Lambda = \ x . b]]
```

However, unlike the minus case, this tree matches the second pattern generated by the disambiguation rule, thereby *rejecting* the sentence! Thus, the filter F^{Q_G} is *unsafe* for this grammar.

In summary, the SDF2 semantics is safe for prefix operators with high priority, but not for prefix operators with low priority, which are prevalent in functional languages such as ML, with language constructs such as lambda, let, and case expressions.

2.4.2 *Safe Semantics*

We define a *new semantics* for disambiguation rules that is *safe* for low priority prefix operators, naming it after the SDF3 formalism in which it was introduced:

Definition 2.4.1 (SDF3 Semantics: Safe Priority Conflict Patterns)**.** *Given a grammar G extended with priority declarations Pr(G) defining symmetric, non-transitive relations* `right`, `left`, `non-assoc`, *and* `non-nested`, *and irreflexive, anti-symmetric, and transitive relation* $>$ *over the productions in* $P(G)$ *of G, the set of conflict patterns* Q_G^{safe} *and warning patterns* Q_G^W *are derived as follows:*

$$\frac{A.C_1 > A.C_2 \in PR}{[A.C_1 = [A.C_2 = \alpha A]\gamma] \in Q_G^{safe}} \tag{2.34}$$

$$\frac{A.C_1 > A.C_2 \in PR}{[A.C_1 = \alpha[A.C_2 = A\gamma]] \in Q_G^{safe}} \tag{2.35}$$

$$\frac{A.C_1 \texttt{ right } A.C_2 \in PR}{[A.C_1 = [A.C_2 = A\beta_2 A]\beta_1 A] \in Q_G^{safe}} \tag{2.36}$$

$$\frac{A.C_1 \texttt{ left } A.C_2 \in PR}{[A.C_1 = A\beta_1[A.C_2 = A\beta_2 A]] \in Q_G^{safe}} \tag{2.37}$$

$$\frac{A.C_1 \texttt{ non-assoc } A.C_2 \in PR}{[A.C_1 = A\beta_1[A.C_2 = A\beta_2 A]] \in Q_G^{safe}} \tag{2.38}$$

$$\frac{A.C_1 \texttt{ non-assoc } A.C_2 \in PR}{[A.C_1 = [A.C_2 = A\beta_2 A]\beta_1 A] \in Q_G^{W}} \tag{2.39}$$

$$\frac{A.C_1 \texttt{ non-nested } A.C_2 \in PR \quad \neg(\alpha_i \Rightarrow^* A\gamma)}{[A.C_1 = \alpha_1[A.C_2 = \alpha_2 A]] \in Q_G^W} \tag{2.40}$$

A tree $t \in T(G)$ has a priority conflict, if $\mathcal{M}(t, Q_G^{safe})$. A filter $F^{Q_G^{safe}}$ implements this semantics, thus, the new semantics of a grammar G with priority declarations Pr(G) is the disambiguated grammar $G/F^{Q_G^{safe}}$. □

Definition 2.3.2 is unsafe because it is too indiscriminate, deriving patterns matching parse trees of sentences that are not ambiguous. For example, the semantics of priority was defined as

$$\frac{A.C_1 > A.C_2 \in PR}{[A.C_1 = \alpha[A.C_2 = \beta]\gamma] \in Q_G}$$

rejecting any production with lower priority as a child *at any position* of a production with higher priority, which derives the problematic conflict pattern for `Exp.Lambda` as child of `Exp.Add`. In Definition 2.4.1, the new rules for priority only reject subtrees that are right-recursive in a left-recursive position (2.34) and that are left-recursive in a right-recursive position (2.35). Since a prefix expression is not left-recursive, it is no longer rejected as child in a right-recursive position, *even if the parent production has higher priority*. Thus, the only conflict pattern for `Exp.Lambda` and `Exp.Add` is now:

$$\frac{\texttt{Exp.Add} > \texttt{Exp.Lambda} \in PR}{\texttt{[Exp.Add = [Exp.Lambda = \ ID . Exp] + Exp]} \in Q_G^{safe}}$$

such that tree (6) above is no longer rejected. This semantics of priority also covers the postfix expressions, which we will introduce in Section 2.5. Similarly, the rules for associativity in Definition 2.4.1 are only defined between proper infix productions that are left- *and* right-recursive. (Or rather, *infix-like* operators, since the βs in these productions also match the 'extended infix' operators of distfix productions we will encounter later.)

In Definition 2.3.2 we have introduced the new disambiguation rule *non-nested*, which is the analogue of *non-assoc* for prefix operators. While the combination of two prefix operators is not ambiguous, we may sometimes want to force parentheses for such expressions. For example, combining prefix operators such as `assert e`, and `lazy e` in OCaml [84] leads to expressions such as `assert lazy e`, which is unambiguous, but may also be confusing to programmers considering the syntax of function application. By declaring the operators to be mutually *non-nested*, requires such an expression to be written as `assert (lazy e)`. As in Section 2.3.8, we treat this case as a warning pattern instead of a conflict pattern, so that violations can be treated semantically rather than generating a syntax error.

The filter $F^{Q_G^{safe}}$ is safe for prefix expression grammars, as we will show formally below. However, while the SDF2 semantics was indiscriminate in rejecting sentences, it turns out that the new semantics does not discriminate enough, making it incomplete.

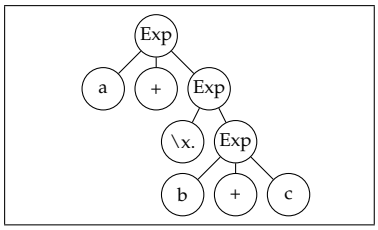


Figure 2.19

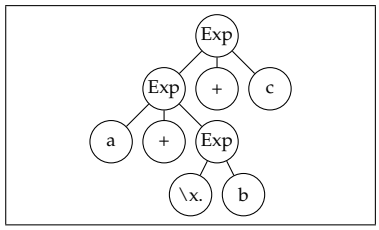


Figure 2.20

2.4.3 *Deep Priority Conflicts*

The elegance of the SDF2 semantics (if we put aside its unsafety for a moment) is that it covers the semantics of associativity and priority with just five rules,[9] defining patterns containing direct combinations of productions into so called *shallow priority conflicts*. By addressing the safety of the SDF2 semantics, we expose a new pattern of ambiguity that cannot be captured with shallow patterns.

Consider again the grammar of Figure 2.18 with priority rule `Exp.Add > Exp.Lambda`. As discussed in the previous section, the ambiguity in the sentence `\x. a + b` is addressed by the SDF3 semantics in Definition 2.4.1. Moreover, the expression `a + \x. b` is still valid, since its tree does not match any conflict pattern generated from this priority rule. However, the expression `a + \x. b + c` is now ambiguous, since parsing this sentence produces the trees in Figures 2.19 and 2.20. With the SDF2 semantics, both trees match the conflict pattern according to Equation 2.27:

$$\frac{\texttt{Exp.Add} > \texttt{Exp.Lambda} \in PR}{\texttt{[Exp.Add = Exp + [Exp.Lambda = \ ID . Exp]]} \in Q_G}$$

Thus, with the SDF2 semantics, one needs explicit parentheses to obtain either of the trees[10]: `a + (\x. b + c)` or `a + (\x. b) + c`. With the SDF3 semantics, the sentence is not rejected, but the ambiguity is not solved, since neither of the trees match the conflict pattern:

$$\frac{\texttt{Exp.Add} > \texttt{Exp.Lambda} \in PR}{\texttt{[Exp.Add = [Exp.Lambda = \ ID . Exp] + Exp]} \in Q_G^{safe}}$$

Since `Exp.Add > Exp.Lambda`, the addition to the right of the lambda expression should extend as much as possible. That is, the preferred tree is the one in Figure 2.19. However, it is not possible to filter the invalid tree of Figure 2.20 using a shallow conflict pattern, since the lambda expression does not occur as a direct left child of the top-most addition, occurring nested as the right-most descendant of another addition. In fact, any infix expression that

[9] In the original definition of Visser [131] they are factored in just three rules.

[10] Substitute function application for addition to get a phrase that may be more semantically meaningful.

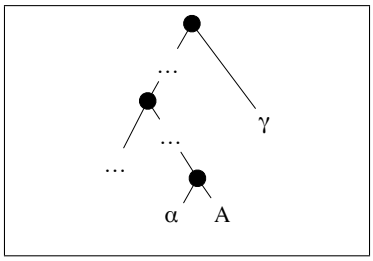


Figure 2.21 Conflict pattern to solve an ambiguity due to a lower priority prefix operator.

has higher priority than addition, and may occur as leftmost descendant of the addition, allowing an `Exp.Lambda` as its rightmost descendant (considering priorities), will produce an ambiguity that cannot be captured by shallow conflict patterns. Since such ambiguities involve patterns of unbounded depth, we call them *deep* priority conflicts. Figure 2.21 sketches the general shape of parse trees with a deep priority conflict.

2.4.4 *Filtering Deep Priority Conflicts by Subtree Exclusion*

To capture deep priority conflicts, we define a matching function that recursively checks for subtrees that match a certain pattern at arbitrary nesting depth.

Definition 2.4.2 (Rightmost Deep Matching)**.** *Given a grammar G, a tree $t \in T(G)$, and pattern $q \in TP(G)$, then $\mathcal{D}^{rm}(t, q)$ — t right-most deeply matches q — if t matches q, or if any subtree in the right-most branch of t_i matches the sub-pattern q_i:*

$$\frac{\forall 1 \leq i \leq n : \mathcal{M}^{rm}(t_i, q_i)}{\mathcal{D}^{rm}([A.C = t_1 ... t_n], [A.C = q_1 ... q_n])} \tag{2.41}$$

$$\frac{\mathcal{M}(t, q)}{\mathcal{M}^{rm}(t, q)} \tag{2.42}$$

$$\frac{\mathcal{M}^{rm}(t_n, q)}{\mathcal{M}^{rm}([A.C = t_1 ... t_n], q)} \tag{2.43}$$

If Q^{rm} is a set of rightmost patterns, $\mathcal{D}^{rm}(t, Q^{rm})$ if there is some $q \in Q^{rm}$ such that $\mathcal{D}^{rm}(t, q)$. □

With the rightmost deep matching function, we can specify patterns that capture trees that contribute a deep prefix operator ambiguity. For example, consider the tree t from Figure 2.20, and a rightmost pattern q, specified as follows:

```
t : [Exp.Add = [Exp.Add = a + [Exp.Lambda = \ x . b]] + c]
q : [Exp.Add = [Exp.Lambda = \ ID . Exp] + Exp]
```

Clearly we have that $\mathcal{M}(t,q)$ does not hold, since the pattern q does not directly match the tree t, particularly because t_1 does not match q_1. However, by Equations 2.41 and 2.43, the tree t right-most deeply matches the pattern q since the rightmost subtree of t_1 matches q_1,:

$$\frac{\dfrac{\mathcal{M}^{rm}(\texttt{[Exp.Lambda = \ x . b],[Exp.Lambda = \ ID . Exp]})}{\mathcal{M}^{rm}(\texttt{[Exp.Add = a + [Exp.Lambda = \ x . b]],[Exp.Lambda = \ ID . Exp]})}}{\mathcal{D}^{rm}(t,q)}$$

With this definition of deep matching, we can formally define the patterns for deep priority conflicts and the subtree exclusion disambiguation filter based on those patterns.

Definition 2.4.3 (Rightmost Deep Priority Conflict Patterns)**.** *Given a grammar G extended with priority declarations $Pr(G)$ defining an irreflexive, transitive relation $>$ over the productions in $P(G)$, the set Q_G^{rm} containing rightmost conflict patterns over G is the smallest set of patterns such that:*

$$\frac{A.C_2 > A.C_1 \in PR \quad \alpha \overset{*}{\not\Rightarrow}_G A\beta}{[A.C_2 = [A.C_1 = \alpha A]\gamma] \in Q_G^{rm}} \tag{2.44}$$

The set $TP^{Q^{rm}}(G) = \{q \in TP(G) \mid \mathcal{D}^{rm}(q, Q^{rm})\}$ is the set of all tree patterns that match a deep priority conflict pattern. We write $Q_G = Q_G^{safe} \cup TP^{Q^{rm}}(G)$ for the conflict patterns of a grammar G combining the safe shallow and deep priority conflict patterns. □

Equation 2.44 describes a deep priority conflict arising from a deep prefix operator ambiguity, i.e., the conflict involves an infix expression with higher priority ($A.C_2 = A\gamma$)[11] and a prefix operator ($A.C_1 = \alpha A$) with lower priority, as illustrated by our example with lambda and addition operators. The restriction $\alpha \overset{*}{\not\Rightarrow}_G A\beta$ prevents the creation of the conflict pattern if $A.C_1$ defines an infix operator, since lower precedence infix operators do not cause a deep priority conflict, as discussed previously.

2.4.5 *Overlapping Operators*

Associativity and priority rules solve ambiguities that are created by different permutations of the *same* productions. Some ambiguities in prefix expression grammars cannot be solved by priority and associativity declarations since they involve trees constructed with different productions. Such grammars are *inherently ambiguous*.

Definition 2.4.4 (Inherently Ambiguous Grammars)**.** *A grammar G is inherently ambiguous if there exist two different trees $t_1, t_2 \in T(G)$, such that $yield(t_1) = yield(t_2)$, and t_1 has been constructed with a different set of productions than t_2.* □

[11] We use a more general production with the sentential form γ to be able to generalize this equation to postfix operators, shown in Section 2.5.

```
context-free syntax
  Exp.Minus     = "-" Exp
  Exp.Plus      = "+" Exp
  Exp.PlusMinus = "+-" Exp
  Exp.Var       = ID
  Exp           = "(" Exp ")" {bracket}
```

Figure 2.22 Example of an inherently ambiguous prefix expression grammar.

```
lexical restrictions
  "+" -/- [\-]
context-free syntax
  Exp.Minus     = "-" Exp
  Exp.Plus      = "+" Exp
  Exp.PlusMinus = "+-" Exp
  Exp.Var       = ID
  Exp           = "(" Exp ")" {bracket}
```

Figure 2.23 Using lexical restrictions to disambiguate inherently ambiguous grammars.

For example, consider the grammar of Figure 2.22, and the productions `Exp.Plus`, `Exp.Minus`, and `Exp.PlusMinus`. Parsing the sentence `+-a` produces the following trees:

```
(1) [Exp.Plus = + [Exp.Minus = - a]]
(2) [Exp.PlusMinus = +- a]
```

Note that the tree `(1)` has been constructed with the productions `Exp.Plus`, `Exp.Minus`, whereas the tree `(2)` has been constructed with the production `Exp.PlusMinus`. The inherent ambiguity above occurs because the operators `+`, `-`, and `+-` *overlap*, i.e., they have character(s) in common.

Our goal is to provide a direct semantics to solve ambiguities due to operator priority and associativity. We would like to avoid grammars that are inherently ambiguous because they are out of the scope of priority and associativity disambiguation. These ambiguities occur when it is possible to construct a valid operator word (or a sequence of valid operator words) by combining two or more different operators in an expression grammar. In the case of prefix expression grammars, such ambiguities can always be solved by *lexical disambiguation*. As mentioned in Section 2.2.8, SDF2/3 uses *follow restrictions* to implement longest match on lexical constructs. In the example above, we can use follow restrictions to forbid the operator `+` to be followed by the operator `-`, since this would result in the operator `+-`, as shown in the grammar of Figure 2.23. Thus, using lexical disambiguation allows us to reject the tree `(1)`, disambiguating the sentence `+-a`. Of course, it is still possible to create that tree using the expression `+ - a`, i.e. by separating the operators with a space.

Below, we prove that the semantics in Definition 2.4.1 is safe and completely disambiguating for prefix expression grammars when extended with the deep priority conflict patterns shown above. First, we extend the notion of total set of disambiguation rules to also include rules for prefix operators.

Definition 2.4.5 (Total Set of Disambiguation Rules for Prefix Expression Grammars)**.** *A set of disambiguation rules PR for a prefix expression grammar G is total for a non-terminal A if it is total for all productions in G defining infix expressions, and for any pair of productions* $A.C_1 = op_1\ A \in P(G)$*, and* $A.C_2 = A\ op_2\ A \in P(G)$*, either* $A.C_1 > A.C_2 \in PR$ *or* $A.C_2 > A.C_1 \in PR$*.*

Lemma 2.4.6 (Subtree Exclusion for Prefix Expression Grammars is Safe)**.** *Given a prefix expression grammar G and set Q of shallow and deep priority conflict patterns generated by the disambiguation rules for G, if* $w \in L(G)$ *then there is a* $t \in T^Q(G)$*, such that* $yield(t) = w$*.*

Proof. By induction on the length of sentences in $L(G)$.

(Base case) If a is a lexeme then $a \in T^Q_a(G)$ since disambiguation rules do not exclude lexemes.

(Inductive case) Assume that $u, v \in L(G)$ and that there are $t_1, t_2 \in T^Q_A(G)$ such that $yield(t_1) = u$, $yield(t_2) = v$, then there are three cases:

(1) If $A.C = \triangleleft\ A\ \triangleright$ is a closed production in G, then $\triangleleft\ u\ \triangleright \in L(G)$ and $[A.C = \triangleleft\ t_1\ \triangleright] \in T^Q_A(G)$, since there is no priority conflict pattern that matches this tree.

(2) If $A.C = \blacktriangleright\ A$ is a prefix production in G, then $\blacktriangleright\ u = w \in L(G)$. We need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. By induction $u = yield(t_1)$ such that $t_1 \in T^Q(G)$. We consider the following cases:

- If t_1 is a lexeme, closed expression or another prefix expression, then $t = [A.C = \blacktriangleright\ t_1] \in T^Q(G)$, since there are no disambiguation rules that apply.
- If $t_1 = [A.C_1 = t_{11} \otimes t_{12}]$ with yield $u_{11} \otimes u_{12}$. Take $t = [A.C = \blacktriangleright\ [A.C_1 = t_{11} \otimes t_{12}]]$ as the candidate tree for w. If $A.C_1 > A.C$ then $t \in T^Q(G)$ since it does not match a conflict pattern (since there are no other disambiguation relations between the productions). On the other hand, if $A.C > A.C_1$ then t matches a conflict pattern and therefore $t \notin T^Q(G)$. However, the tree $t' = [A.C_1 = [A.C = \blacktriangleright\ t_{11}] \otimes t_{12}]$ has the same yield and does *not* have a priority conflict, therefore $t' \in T^Q(G)$.

(3) If $A.C = A \oplus A$ is an infix production in G, then $u \oplus v = w \in L(G)$. Now we need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. By induction $u = yield(t_1)$ and $v = yield(t_2)$ such that $t_1, t_2 \in T^Q(G)$. We consider the following cases:

- If $t_1 = [A.C_1 = \blacktriangleright\ t_{11}]$ the proof is analogous to the last case for the production $A.C = \blacktriangleright\ A$.
- If $t_1 = [A.C_1 = t_{11} \otimes t_{12}]$ with yield $u_{11} \otimes u_{12}$, t_2 a closed expression or lexeme, the analysis is analogous to the proof for the infix-infix case for Lemma 2.3.3, except for the case of a right-most deep match. If $A.C_1 > A.C$, the candidate tree is $t = [A.C = [A.C_1 = t_{11} \otimes t_{12}] \oplus t_2]$. If $\neg\mathcal{M}^{rm}(t, [A.C = [A.C_3 = \blacktriangleright\ t_{21}] \oplus t_{12}])$ for any prefix operator $A.C_3 = \blacktriangleright\ A$ then $t \in T^Q(G)$. However, if $\mathcal{M}^{rm}(t, [A.C = [A.C_3 = \blacktriangleright\ A] \oplus A])$ and $A.C > A.C_3$ then t has a deep priority conflict, i.e. it has the form $[A.C = [A.C_1 = t_{11} \otimes [..[A.C_3 = \blacktriangleright\ t_{121}]]] \oplus t_2]$ and $t \notin T^Q(G)$. However, the tree $t' = [A.C_1 = t_{11} \otimes [...[A.C_3 = \blacktriangleright\ [A.C = t_{121} \oplus t_2]]]]$ has the same yield and does *not* have a priority conflict, i.e. $t' \in T^Q(G)$.
- The remaining cases are also analogous to the cases in the proof of Lemma 2.3.3. □

Lemma 2.4.7 (Subtree Exclusion for Prefix Expression Grammars is Completely Disambiguating)**.** *Given a prefix expression grammar G and the set Q of shallow and deep priority conflict patterns generated by the total set of disambiguation rules for G, then all trees in $T^Q(G)$ have unique yields. That is, if $t_1, t_2 \in T^Q(G)$ and $yield(t_1) = yield(t_2)$ then $t_1 = t_2$.*

Proof. By induction on $T^Q(G)$.

(Base case) If a is a lexeme, then $a \in T^Q_a(G)$ and has a unique yield.

(Inductive case) Assume that $t_1, t_2 \in T^Q_A(G)$ and that their yields are unique.

(1) If $A.C_1 = \triangleleft\ A\ \triangleright$ is a closed production in G, then $t = [A.C_1 = \triangleleft\ t_1\ \triangleright] \in T^Q_A(G)$, since there is no priority conflict pattern that matches this tree, and that any sequence of operators can be disambiguated by lexical disambiguation, by uniqueness of t_1, t is also unique.

(2) If $A.C = \blacktriangleright\ A$ is a prefix production in G, we can construct the tree $t = [A.C = \blacktriangleright\ t_1]$. We need to demonstrate that if $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$.

- If t_1 is a lexeme or closed expression then $t \in T^Q(G)$ since there are no other ways to construct t by uniqueness of t_1 and since there are no disambiguation rules that apply.
- If $t_1 = [A.C_1 = t_{11} \otimes t_{12}]$ with yield $u_{11} \otimes v_{12}$, then $t = [A.C = \blacktriangleright\ [A.C_1 = t_{11} \otimes t_{12}]]$ with yield $\blacktriangleright\ u_{11} \otimes v_{12}$. If $A.C > A.C_1$ then t matches a conflict pattern and therefore $t \notin T^Q(G)$. If $A.C_1 > A.C$ then t does not match a conflict pattern (since there are no other disambiguation relations between the productions). The only other tree with the same yield is $t' = [A.C_1 = [A.C = \blacktriangleright\ t_{11}] \otimes t_{12}]$. However, t' *does* have a priority conflict and therefore $t' \notin T^Q(G)$ when $A.C_1 > A.C$.

(3) If $A.C = A \ \oplus \ A$ is an infix production in G, we can construct the tree $t = [A.C = t_1 \ \oplus \ t_2]$. We need to demonstrate that if $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$. We consider the following cases:

- If $t_1 = [A.C_1 = \blacktriangleright \ t_{11}]$ and $t_2 = [A.C_2 = \triangleleft t_{21} \triangleright]$, the analysis is analogous as the last case for the production $A.C = \blacktriangleright \ A$.
- If $t_1 = [A.C_1 = t_{11} \otimes t_{12}]$ with yield $u \otimes v$ and t_2 is a lexeme or closed expression with yield w then $t = [A.C = [A.C_1 = t_{11} \otimes t_{12}] \ \oplus \ t_2]$ with yield $u \otimes v \oplus \triangleleft w \triangleright$. The analysis is analogous to the proof for the infix-infix case of Lemma 2.3.5, except that we need to consider rightmost deep matching. If $\mathcal{M}^{rm}(t, [A.C = [A.C_3 = \blacktriangleright \ A] \oplus A])$ then t has the form $[A.C = [A.C_1 = t_{11} \otimes [..[A.C_3 = \blacktriangleright \ t_{121}]]] \oplus t_2]$. By totality of disambiguation rules, we have that there is a disambiguation relation between $A.C$ and $A.C_3$. If $A.C > A.C_3$, then t has a priority conflict, thus $t \notin T^Q(G)$. On the other hand, if $A.C_3 > A.C$, then t does not have a priority conflict. The other tree with the same yield is $t' = [A.C_1 = t_{11} \otimes [...[A.C_3 = \blacktriangleright \ [A.C = t_{121} \oplus t_2]]]]$, which *does* have a priority conflict and therefore $t \notin T^Q(G)$.
- The remaining cases are analogous to the cases in the proof of Lemma 2.3.5. □

Theorem 2.4.8. *Disambiguation of a prefix expression grammar using a total set of disambiguation rules is safe and completely disambiguating.*

Proof. Assume that G is a prefix expression grammar and R a total set of disambiguation rules for G. Let Q^{safe} be the set of priority conflict patterns for R according to Definition 2.4.1, and $TP^{Q^{rm}}$ be the set of deep priority conflict patterns for R according to Definition 2.4.3, such that $Q = Q^{safe} \cup TP^{Q^{rm}}$. By Lemma 2.4.6 we have that if $w \in L(G)$ then there is a $t \in T^Q(G)$ such that $yield(t) = w$. By Corollary 2.2.17 we have that F^Q is safe. By Lemma 2.3.5 we have that if $t_1, t_2 \in T^Q(G)$ then $yield(t_1) \neq yield(t_2) \vee t_1 = t_2$. By Corollary 2.2.18 we have that F^Q is completely disambiguating. □

2.5 POSTFIX EXPRESSIONS

In this section we extend the semantics for disambiguation rules to address expression grammars that contain *postfix expressions*. Expression grammars that contain only postfix and infix expressions are dual to prefix expression grammars. That is, expression grammars that contain postfix and infix expressions, but *do not* contain prefix expressions are susceptible to similar ambiguities and can also be affected by deep priority conflicts.

For example, the expression `a + b --` according to the grammar of Figure 2.24 is ambiguous with the following trees:

```
[Exp.Add = a + [Exp.Dec = b --]]
[Exp.Dec = [Exp.Add = a + b] --]
```

```
context-free syntax
  Exp.Add = Exp "+" Exp
  Exp.Dec = Exp "--"
  Exp.Var = ID
  Exp     = "(" Exp ")" {bracket}
```

Figure 2.24 Grammar containing a postfix expression.

According to Definition 2.4.1, a priority rule that states `Exp.Dec > Exp.Add` creates the pattern `[Exp.Dec = [Exp.Add = Exp + Exp] --]`, which matches the first tree, whereas a priority rule `Exp.Add > Exp.Dec` creates the pattern `[Exp.Add = Exp + [Exp.Dec = Exp --]]`, filtering the second tree. Similar to the combination of infix and postfix expressions, the sentence `a -- + b` involving an infix and a postfix expression can be parsed unambiguously with the tree:

```
[Exp.Add = [Exp.Dec = a --] + b]
```

Because the SDF3 semantics is safe, this tree is not captured by the priority `Exp.Add > Exp.Dec`. However, a deep priority conflict can occur when parsing the sentence `a + b -- * c`. Parsing this sentence produces the trees:

```
[Exp.Add = a + [Exp.Mul = [Exp.Dec = b --] * c]]
[Exp.Mul = [Exp.Dec = [Exp.Add = a + b] --] * c]
```

Note that only the second tree should be valid, since the addition to the left of the postfix expression should extend as much as possible. To disambiguate this sentence and solve the deep priority conflict we use a symmetric strategy to disambiguation of prefix operator deep ambiguities, applying *leftmost deep matching* using sets of *leftmost deep priority conflict patterns* according to the following definitions.

Definition 2.5.1 (Leftmost Deep Matching). *Given a grammar G, a tree $t \in T(G)$, and pattern $q \in TP(G)$, then $\mathcal{D}^{lm}(t, q)$ — t left-most deeply matches q — if t matches q, or if any subtree in the left-most branch of t_i matches the sub-pattern q_i:*

$$\frac{\forall 0 \leq i \leq n : \mathcal{M}^{lm}(t_i, q_i)}{\mathcal{D}^{lm}([A.C = t_1 ... t_n], [A.C = q_1 ... q_n])} \tag{2.45}$$

$$\frac{\mathcal{M}(t, q)}{\mathcal{M}^{lm}(t, q)} \tag{2.46}$$

$$\frac{\mathcal{M}^{lm}(t_1, q)}{\mathcal{M}^{lm}([A.C = t_1 ... t_n], q)} \tag{2.47}$$

If Q^{lm} is a set of leftmost patterns then $\mathcal{D}^{lm}(t, Q^{lm})$ if there is some $q \in Q^{lm}$ such that $\mathcal{D}^{lm}(t, q)$. □

Definition 2.5.2 (Leftmost Deep Priority Conflict Patterns)**.** *Given a grammar G extended with priority declarations $Pr(G)$ defining an irreflexive, transitive relation $>$ over the productions in $P(G)$, the set Q_G^{lm} containing leftmost conflicting patterns over G, respectively, is the smallest set of patterns such that:*

$$\frac{A.C_2 > A.C_1 \in PR \quad \gamma \overset{*}{\Rightarrow}_G \beta A}{[A.C_2 = \alpha[A.C_1 = A\gamma]] \in Q_G^{lm}} \tag{2.48}$$

The set $TP^{Q^{lm}}(G) = \{q \in TP(G) \mid \mathcal{D}^{lm}(q, Q^{lm})\}$ is the set of all tree patterns that match a deep priority conflict pattern. We write $Q_G = Q_G^{safe} \cup TP^{Q^{rm}}(G) \cup TP^{Q^{lm}}(G)$ for the conflict patterns of a grammar G combining the safe shallow and deep priority conflict patterns. □

2.5.1 *Basic Expression Grammars*

So far we have looked at the combination of prefix and infix expressions, and postfix and infix expressions separately. In this subsection we investigate ambiguities that may occur in *basic expression grammars*, i.e., expressions grammars that contain prefix, infix, and postfix operators.

Definition 2.5.3 (Basic Expression Grammars)**.** *A basic expression grammar is an expression grammar that contains only productions defining atomic, closed, prefix, infix, and postfix expressions.* □

According to Definition 2.5.3, infix and prefix expression grammars consist of simplified instances, i.e. subsets, of basic expression grammars. The grammar in Figure 2.25 is a basic expression grammar containing productions that define atomic (`Exp.Var`), closed (`Exp = "(" Exp ")"`), prefix (`Exp.Not` and `Exp.Minus`), infix (`Exp.Add`), and postfix (`Exp.Inc`) expressions.

Ambiguities in basic expression grammars do not only occur between prefix and infix, or postfix and infix operators, but may also occur between prefix and postfix operators. For example, parsing the expression `- a ++` using the grammar in Figure 2.25 produces the following trees:

```
(1) [Exp.Inc = [Exp.Minus = - a] ++]
(2) [Exp.Minus = - [Exp.Inc = a ++]]
```

According to the safe semantics from Definition 2.4.1, the priority `Exp.Inc > Exp.Minus` creates the following priority conflict:

$$\frac{\texttt{Exp.Inc} > \texttt{Exp.Minus} \in PR}{\texttt{[Exp.Inc = [Exp.Minus = - Exp] ++]} \in Q_G^{safe}}$$

Thus, our safe semantics can also be used to disambiguate this sentence, since the tree `(1)` is rejected because it matches this pattern. Note that the inverse priority creates the following pattern, which can be used to reject tree `(2)`:

$$\frac{\texttt{Exp.Minus} > \texttt{Exp.Inc} \in PR}{\texttt{[Exp.Minus = - [Exp.Inc = Exp ++]]} \in Q_G^{safe}}$$

```
context-free syntax
  Exp.Add   = Exp "+" Exp
  Exp.Not   = "!" Exp
  Exp.Minus = "-" Exp
  Exp.Inc   = Exp "++"
  Exp.Var   = ID
  Exp       = "(" Exp ")" {bracket}
context-free priorities
  Exp.Not > Exp.Inc > Exp.Minus > Exp.Add
```

Figure 2.25 Example of a basic expression grammar.

```
context-free syntax
  Exp.Min    = "-" Exp
  Exp.PosMin = Exp "-"
  Exp.Sub    = Exp "-" Exp {left}
  Exp.Var    = ID
  Exp        = "(" Exp ")" {bracket}
context-free priorities
  Exp.PosMin > Exp.Min > Exp.Sub
```

Figure 2.26 Inherently ambiguous grammar.

Deep matching may also be necessary to disambiguate particular sentences constructed by combining only prefix and postfix operators. For instance, consider the prefix operator `!` with higher priority, the sentence `! - a ++` contains an ambiguity that can only be solved by the deep priority conflict in Definition 2.4.3, created by the priority `Exp.Inc > Exp.Minus`. Lower priority postfix operators do not occur frequently in programming languages. However, when such operators do occur, the ambiguities they cause also require disambiguation by deep matching using the deep priority conflict patterns of Definition 2.5.2.

2.5.2 *Overlapping Operators*

In Section 2.4.5 we considered *inherently ambiguous prefix expression grammars*, discussing how lexical disambiguation can be used to address ambiguities in such grammars. However, that is not always the case. For example, consider the inherently ambiguous grammar in Figure 2.26. Parsing the sentence `a - - - b` produces the following trees:

```
[Exp.Sub = [Exp.PosMin = [Exp.PosMin = a -] -] - b]
[Exp.Sub = a - [Exp.Min = - [Exp.Min = - b]]]
[Exp.Sub = [Exp.PosMin = a -] - [Exp.Min = - b]]
```

This ambiguity cannot be solved by priority and associativity disambiguation, since it involves trees constructed with different productions. Furthermore, this ambiguity can not be solved by follow restrictions such as the ambiguity

discussed in Section 2.4.5, because operators overlap by having the same operator words. One solution to avoid such ambiguities is to forbid *all* overlap of operators, dramatically restricting the set of grammars we support. However, not all overlap is harmful, i.e. causes inherent ambiguities. We can check for harmful overlap by verifying the productions used to construct a tree, using this information to restrict the grammars we consider, avoiding inherently ambiguous grammars.

Definition 2.5.4 (Harmful Overlap)**.** *Given a function $prod(t)$ returning the set of productions used to construct a particular tree t, an expression grammar contains harmful overlap if there exist trees t_1 and t_2 such that $yield(t_1) = yield(t_2)$, and $prod(t_1) \neq prod(t_2)$, i.e., the overlap in the operators in the productions $prod(t_1)$ and $prod(t_2)$ is harmful.*

Lemma 2.5.5. *An expression grammar that contains only contains harmless overlap is not inherently ambiguous.*

Proof. By the definition of inherently ambiguous grammars (Definition 2.4.4) and harmful overlap (Definition 2.5.4).

□

We cannot claim a decision procedure for stating that a grammar contains only harmless overlap, since ambiguity of context-free grammars is an undecidable property. Instead, we describe an implementation in Section 2.9 that iteratively constructs parse trees, checking for harmful overlap. In the remainder of the chapter we only consider expression grammars with harmless overlap.

2.5.3 *Safe and Complete Disambiguation*

The safety and completeness of disambiguation of basic expression grammars follows the structure of the earlier proofs. We consider the interaction of prefix and postfix operators.

Definition 2.5.6 (Total Set of Disambiguation Rules for Basic Expression Grammars)**.** *A set of disambiguation rules PR for a basic expression grammar G is total for a non-terminal A if it is total for all productions in G defining infix and prefix expressions (Definition 2.4.5), and for any pair of productions $A.C_1 = A\ \gamma \in G$, and $A.C_2 = \alpha\ A \in G$, either $A.C_1 > A.C_2 \in PR$ or $A.C_2 > A.C_1 \in PR$.*

Lemma 2.5.7 (Subtree Exclusion for Basic Expression Grammars is Safe)**.** *Given a basic expression grammar G, and Q the set of safe, left-most, and right-most deep conflict patterns generated from the disambiguation rules of G, if $w \in L(G)$ then there is a $t \in T^Q(G)$, such that $yield(t) = w$.*

Proof. By induction on the length of sentences in $L(G)$.

(Base case) If a is a lexeme then $a \in T_a^Q(G)$ since disambiguation rules do not exclude lexemes.

(Inductive case) Assume that $u, v \in L(G)$ and that there are $t_1, t_2 \in T_A^Q(G)$ such that $yield(t_1) = u$, $yield(t_2) = v$, then there are four cases:

(1) If $A.C = \triangleleft\ A\ \triangleright$ is a closed production in G, then $\triangleleft\ u\ \triangleright \in L(G)$ and $[A.C = \triangleleft\ t_1\ \triangleright] \in T_A^Q(G)$, since there is no priority conflict pattern that matches this tree.

(2) If $A.C = \blacktriangleright\ A$ is a prefix production in G, then $\blacktriangleright\ u = w \in L(G)$. We need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. By induction $u = yield(t_1)$ such that $t_1 \in T^Q(G)$. We consider the following cases:

- If $t_1 = [A.C_1 = t_{11}\ \blacktriangleleft]$ with yield $u_{11}\ \blacktriangleleft$. Take $t = [A.C = \blacktriangleright\ [A.C_1 = t_{11}\ \blacktriangleleft]]$ as the candidate tree for w.
  First consider $A.C_1 > A.C$. If $\neg\mathcal{M}^{lm}(t, [A.C = \blacktriangleright\ [A.C_2 = t_{21} \lessdot]])$ for any postfix operator $A.C_2 = A \lessdot$, then $t \in T^Q(G)$ since it does not match a conflict pattern.
  If $\mathcal{M}^{lm}(t, [A.C = \blacktriangleright\ [A.C_2 = t_{21} \lessdot]])$ then t has the shape $[A.C = \blacktriangleright\ [A.C_1 = [[A.C_2 = t_{21} \lessdot]...]\ \blacktriangleleft]]$. If $A.C_2 > A.C$ then t does not have a priority conflict and $t \in T^Q(G)$. If $A.C > A.C_2$, then t has a deep priority conflict and $t \notin T^Q(G)$. However, the tree $t' = [A.C_1 = [[A.C_2 = [A.C = \blacktriangleright\ t_{21}] \lessdot]...]\ \blacktriangleleft]$ has the same yield and does not have a priority conflict, therefore $t' \in T^Q(G)$.
  On the other hand, if $A.C > A.C_1$, then t matches a conflict pattern and therefore $t \notin T^Q(G)$. However, the tree $t' = [A.C_1 = [A.C = \blacktriangleright\ t_{11}]\ \blacktriangleleft]$ has the same yield. We need to consider right-most deep matches similarly as above.
- The remaining cases are analogous to the proof for Lemma 2.4.6.

(3) If $A.C = \ A\ \blacktriangleleft$ is a postfix production in G, then $u\ \blacktriangleleft = w \in L(G)$. We need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. By induction $u = yield(t_1)$ such that $t_1 \in T^Q(G)$. We consider the following cases:

- If t_1 is a lexeme, closed expression, or a postfix expression, then $t = [A.C = t_1\ \blacktriangleleft] \in T^Q(G)$, since there are no disambiguation rules that apply.
- If $t_1 = [A.C_1 = \blacktriangleright\ t_{11}]$, then the analysis is similar to the first case for the production $A.C = \blacktriangleright\ A$.
- If $t_1 = [A.C = t_{11} \otimes t_{12}]$, with yield $u_{11} \otimes u_{12}$. Take $t = [A.C = [A.C_1 = t_{11} \otimes t_{12}]\ \blacktriangleleft]$ as the candidate tree for w. If $A.C_1 > A.C$ then $t \in T^Q(G)$ since it does not match a conflict pattern (and since there are no other disambiguation relations between the productions). On the other hand, if $A.C > A.C_1$ then t matches a conflict pattern and therefore $t \notin T^Q(G)$. However, the tree $t' = [A.C_1 = t_{11} \otimes [A.C = t_{12}\ \blacktriangleleft]]$ has the same yield and does *not* have a priority conflict, therefore $t' \in T^Q(G)$.

(4) If $A.C = A \oplus A$ is an infix production in G, then $u \oplus v = w \in L(G)$. Now we need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. By induction $u = yield(t_1)$ and $v = yield(t_2)$ such that $t_1, t_2 \in T^Q(G)$.

- If $t_2 = [A.C_2 = \ t_{11} \blacktriangleleft]$ the proof is analogous to the last case for the production $A.C = A \ \blacktriangleleft$.
- The remaining cases are analogous to the cases of the proof for Lemma 2.4.6. □

Lemma 2.5.8 (Subtree Exclusion for Basic Expression Grammars is Completely Disambiguating)**.** *Given a basic expression grammar G with only harmless overlap, and Q the set of safe, left-most, and right-most deep conflict patterns generated from a total set of the disambiguation rules for G, then all trees in $T^Q(G)$ have unique yields, i.e., if $t_1, t_2 \in T^Q(G)$ and $yield(t_1) = yield(t_2)$ then $t_1 = t_2$.*

Proof. By induction on $T^Q(G)$.

(Base case) If a is a lexeme, then $a \in T^Q_a(G)$ and has a unique yield.

(Inductive case) Assume that $t_1, t_2 \in T^Q_A(G)$ and that their yields are unique.

(1) If $A.C = \triangleleft\ A\ \triangleright$ is a closed production in G, then $t = [A.C_1 = \triangleleft\ t_1\ \triangleright] \in T^Q_A(G)$, since there is no priority conflict pattern that matches this tree, and by uniqueness of t_1, t is also unique.

(2) If $A.C = \blacktriangleright\ A$ is a prefix production in G, we can construct the tree $t = [A.C = \blacktriangleright\ t_1]$. We need to demonstrate that if $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$. We consider the following cases:

- If $t_1 = [A.C_1 = t_{11} \blacktriangleleft]$ with yield $u_{11} \blacktriangleleft$, we can construct the tree $t = [A.C = \blacktriangleright\ [A.C_1 = t_{11} \blacktriangleleft]]$ with yield $w = \blacktriangleright\ u_{11} \blacktriangleleft$.
  If $A.C > A.C_1$ then t matches a conflict pattern and therefore $t \notin T^Q(G)$.
  If $A.C_1 > A.C$, then t does not match a shallow conflict pattern.
  If $\neg\mathcal{M}^{lm}(t, [A.C = \blacktriangleright\ [A.C_2 = t_{31} \lessdot]])$ for any postfix operator $A.C_2 = A \lessdot$ then t does not have a conflict, and has a unique yield w, since the only other tree $[A.C_1 = [A.C = \blacktriangleright\ t_{11}] \blacktriangleleft]$ with the same yield has a priority conflict.
  If $\mathcal{M}^{lm}(t, [A.C = \blacktriangleright\ [A.C_2 = t_{31} \lessdot]])$, then $t = [A.C = \blacktriangleright\ [A.C_1 = [[A.C_2 = t_{21}]\lessdot]...] \blacktriangleleft]]$ and $t' = [A.C_1 = [[A.C_2 = [A.C = \blacktriangleright\ t_{21}]\lessdot]...] \blacktriangleleft]$ has the same yield. By totality of disambiguation rules, either $A.C > A.C_2$ and $t \notin T^Q(G)$, or $A.C_2 > A.C$ and $t' \notin T^Q(G)$.
- The remaining cases are analogous to the proof for Lemma 2.4.7.

(3) If $A.C = A \blacktriangleleft$ is a postfix production in G, we can construct the tree $t = [A.C = t_1 \blacktriangleleft]$. We need to demonstrate that *if* $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$. We consider the following cases:

- If t_1 is a lexeme, another postfix expression, or closed expression then $t \in T^Q(G)$ since there are no disambiguation rules that apply, and t has a unique yield since there are no other ways to construct t by uniqueness of t_1.
- If $t_1 = [A.C_1 = \blacktriangleright\ t_{11}]$, then the analysis is analoguous to the first case for the production $A.C = \blacktriangleright\ A$.
- If $t_1 = [A.C_1 = t_{11} \otimes t_{12}]$, with yield $u_{11} \otimes u_{12}$, then $t = [A.C = [A.C_1 = t_{11} \otimes t_{12}]\ \blacktriangleleft]$ with yield $w = u_{11} \otimes u_{12}\ \blacktriangleleft$. If $A.C > A.C_1$ then t matches a conflict pattern and therefore $t \notin T^Q(G)$. If $A.C_1 > A.C$ then t does not match a conflict pattern. The only other tree with the same yield is $t' = [A.C_1 = t_{11} \otimes [A.C = t_{12}\ \blacktriangleleft]]$, however $t' \notin T^Q(G)$, since $A.C_1 > A.C$.

(4) If $A.C = A\ \oplus\ A$ is an infix production in G, we can construct the tree $t = [A.C = t_1\ \oplus\ t_2]$. Similarly, we need to demonstrate that if $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$. We consider the following cases:

- If $t_2 = [A.C_2 =\ t_{11}\ \blacktriangleleft]$ the proof is analogous to the last case for the production $A.C = A\ \blacktriangleleft$.
- The remaining cases are analogous to the cases for the proof of Lemma 2.4.7. □

Theorem 2.5.9. *Disambiguation of a basic expression grammar G with only harmless overlap by a total set of disambiguation rules R is safe and completely disambiguating.*

Proof. Let Q be the set of safe, left-most, and right-most deep conflict patterns generated from R. By Lemma 2.5.7 and Corollary 2.2.17 we have that F^Q is safe. By Lemma 2.5.8 and Corollary 2.2.18 we have that F^Q is completely disambiguating. □

2.6 DISTFIX EXPRESSIONS

In Section 2.2.7, we defined distfix expressions, an extension of basic expressions in which productions can have multiple alternating non-terminals and operators. Since ambiguities only occur at the open recursive positions of productions (and not in closed positions), ambiguities in distfix grammars have the same shape as ambiguities in basic expression grammars. That is, distfix operators behave as if they were infix, prefix, or postfix operators.

Consider the grammar in Figure 2.27, which defines several distfix productions. The production `Exp.Let` defines a prefix distfix operator, the production `Exp.Cond` defines a infix distfix operator, the production `Exp.Subscript` defines a postfix distfix operator, and the production `Exp.While` defines a closed distfix operator.

Disambiguation for distfix expressions uses the same rules for safe shallow conflicts and deep priority conflicts that we established in the previous sections.

```
context-free syntax
  Exp.Add       = Exp "+" Exp {left}
  Exp.If        = "if" Exp "then" Exp
  Exp.Cond      = Exp "?" Exp ":" Exp {left}
  Exp.Subscript = Exp "[" Exp "]"
  Exp.While     = "while" Exp "do" Exp "done"
  Exp.Let       = "let" ID "=" Exp "in" Exp
  Exp.Var       = ID
  Exp           = "(" Exp ")" {bracket}
context-free priorities
  Exp.Subscript > Exp.Add > Exp.If >
  Exp.Cond  > Exp.Let
```

Figure 2.27 An example of a distfix grammar.

`let x = a [ b ] in c`	`[Exp.Let = let x = [Exp.Subscript = a [ b ]] in c] *`
`let x = a in b [ c ]`	`[Exp.Let = let x = a in [Exp.Subscript = b [ c ]] *` `[Exp.Subscript = [Exp.Let = let x = a in b] [ c ]]`
`let x = a in x ? c : d`	`[Exp.Let = let x = a in [Exp.Cond = x ? c : d]] *` `[Exp.Cond = [Exp.Let = x = a in x] ? c : d]`
`a + let x = b in c + d`	`[Exp.Add = a + [Exp.Let = let x = a in [Exp.Add = c + d]]] *` `[Exp.Add = a + [Exp.Add = [Exp.Let = let x = a in c] + d]]`
`while a do b ? c : d done`	`[Exp.While = while a do [Exp.Cond = b ? c : d] done] *`

Figure 2.28 Example sentences over the grammar of Figure 2.27 and their parse trees. Trees marked with * are selected according to the disambiguation rules.

The examples in Figure 2.28 demonstrate their application to sentences over the grammar of Figure 2.27. In the case of an ambiguity, an * marks the tree that is selected. Note that the portion of a production that is enclosed in operators behaves as closed expression, i.e. it cannot cause ambiguities with nested productions; see the first and last examples. Note that deep priority conflicts apply to low priority distfix operators as well, for example, in the sentence `a + let x = b in c + d` with priority `Exp.Add > Exp.Let`.

Theorem 2.6.1. *Disambiguation of a distfix expression grammar without overlap and with a total set of disambiguation rules is safe and completely disambiguating.*

Proof. By Lemmas 2.5.7 and 2.5.8, we have that disambiguating of a basic expression grammar using a total set of disambiguation rules is safe and completely disambiguating. Thus disambiguation for distfix grammars is safe and completely disambiguating in *the right-most and left-most recursive positions* of distfix expressions.

Since there are no conflict patterns that match on the internal recursive positions of a distfix production, disambiguation of distfix grammars is safe. (This is an important distinction from the SDF2 semantics from Definition 2.3.2, which is not safe in the closed positions of distfix productions.)

To show that no ambiguity occurs in the internal recursive positions of a distfix expression consider the tree $t = [A.C = ... \oplus_i t_i \oplus_{i+1} ...]$ with yield $w = ... \oplus_i u_i \oplus_{i+1} ....$ For any unique t_i with yield u_i, t is the only tree with yield w, since no operators overlap. Therefore, disambiguation of distfix grammars is completely disambiguating. □

2.6.1 *Overlapping Distfix Operators*

A requirement in the definition of distfix grammars by [1] is that its operators *must not* overlap. For instance, extending the grammar of Figure 2.27 with the productions

```
Exp.IfThenElse = "if" Exp "then" Exp "else" Exp
Exp.Plus       = "+" Exp
```

does not result in a distfix grammar according to Definition 2.2.20, since their operators overlap with the operators in the productions `Exp.If` and `Exp.Add`. However, this is a strong restriction on expression grammars, which excludes grammars with ambiguities that can be solved completely by priority and associativity declarations. For that reason, we relax this restriction on distfix grammars, to grammars with *harmless overlap* according to Definition 2.5.4. We call these grammars *overlapping distfix grammars*.

Definition 2.6.2 (Overlapping Distfix Grammars)**.** *An overlapping distfix grammars is a distfix expression grammar with only harmless overlap.*

Relaxing the restriction on overlap for distfix grammars, gives rise to new forms of ambiguity, besides the now familiar ambiguities that occur at left-most and right-most recursive positions. These new ambiguities are due to overlapping prefixes (or suffixes) of productions and cause dangling suffix (or prefix) ambiguities. The poster child of this form of ambiguity is the *dangling-else*. Consider the grammar in Figure 2.29. Parsing the sentence `if a then b + if c then d else e` produces the trees:

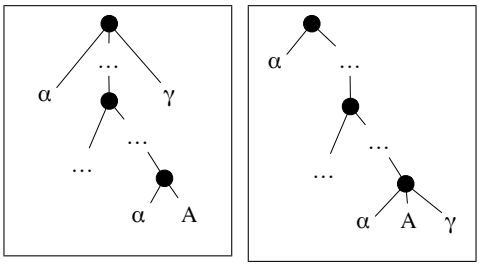
```
(1) [Exp.IfElse = if a then [Exp.Add = b + [Exp.If = if c
  then d]] else e]
(2) [Exp.If = if a then [Exp.Add = b + [Exp.IfElse = if c
  then d else e]]]
```

First note that the overlap in this grammar is harmless. The ambiguities it causes are permutations of the same productions, in contrast to inherent ambiguities consisting of trees composed from different productions.

The standard disambiguation policy is that the dangling "else" clause should belong to the nearest "if-then" clause. Hence, this sentence can be disambiguated by forbidding any if expression to be nested as the rightmost child of any other expression that occurs as "then" clause of an if-then-else expression. Nevertheless, since the conflict does not occur at the leftmost or rightmost position of a particular expression, dangling suffix ambiguities cannot be captured by any of the conflict patterns shown previously.

```
context-free syntax
  Exp.If     = "if" Exp "then" Exp
  Exp.IfElse = "if" Exp "then" Exp "else" Exp
  Exp.Add    = Exp "+" Exp {left}
  Exp.Var    = ID
  Exp        = "(" Exp ")" {bracket}
context-free priorities
  Exp.Add > Exp.IfElse > Exp.If
```

Figure 2.29 Grammar containing dangling suffix ambiguity.



(a) (b)

Figure 2.30 Rejecting a tree that matches the pattern (a), results in attaching γ to the closest α, whereas rejecting pattern (b) results in attaching γ to the furthest α.

2.6.2 *Disambiguating Overlapping Distfix Grammars*

Dangling suffix ambiguities are a common issue in grammars of programming languages. For example, the Java grammar adopts a grammar rewriting technique that solves the ambiguity, forbidding if statements to occur (deeply nested) inside if-then-else statements. YACC, on the other hand, uses the mechanism of preferring a particular action based on the lookahead symbol of the sentence being parsed. While this strategy works for enforcing that an else clause should attach to the closest if-then clause by *preferring* a shift in the presence of such conflict, *preferring a reduce* does not allow selecting the other tree in the ambiguity, but rejects the sentence altogether. We use (deep) priority conflicts to define the semantics for disambiguating expressions containing dangling suffix and dangling prefix ambiguities, enabling the selection of either permutation, as illustrated by the generic patterns in Figure 2.30.

Definition 2.6.3 (Dangling Prefix/Suffix Conflict Patterns)**.** *Given a grammar G with priority declarations defining transitive relations $>$ over the productions $A.C_1 = \alpha A$ and $A.C_2 = \alpha A \gamma$ in G, the set Q_G^{rm} is extended with the following dangling suffix conflict patterns over G:*

$$\frac{A.C_2 > A.C_1 \in PR}{[A.C_2 = \alpha[A.C_1 = A\gamma]\gamma] \in Q_G^{lm}} \tag{2.49}$$

$$\frac{A.C_1 > A.C_2 \in PR}{[A.C_1 = [A.C_2 = \alpha A \gamma] \gamma] \in Q_G^{lm}} \tag{2.50}$$

$$\frac{A.C_2 > A.C_1 \in PR}{[A.C_2 = \alpha [A.C_1 = \alpha A] \gamma] \in Q_G^{rm}} \tag{2.51}$$

$$\frac{A.C_1 > A.C_2 \in PR}{[A.C_1 = \alpha [A.C_2 = \alpha A \gamma]] \in Q_G^{rm}} \tag{2.52}$$

□

Note that Equations 2.49 and 2.50 represent patterns that match trees in ambiguities due to dangling prefix, and Equations 2.51 and 2.52 represent patterns that match trees in ambiguities due to dangling suffix. If we apply this definition to the priority rule `Exp.IfElse > Exp.If` of the grammar in Figure 2.29 we get the following right-most deep conflict pattern, which solves the ambiguity by pairing the closest if-then and else:

$$\frac{\texttt{Exp.IfElse} > \texttt{Exp.If} \in PR}{\texttt{[Exp.IfElse = if Exp then [Exp.If = if ...] else Exp]} \in Q_G^{rm}}$$

This pattern *deeply matches* tree `(1)` above, and therefore tree `(2)` is selected.

2.6.3 *Safe and Complete Disambiguation*

As shown previously, filters that address basic expression grammars are also able to address distfix grammars without overlap. Below we consider disambiguation for overlapping distfix grammars. We start by extending our definition of total set of disambiguation rules to enforce priority between productions involved in dangling prefix and dangling suffix ambiguities. Then we show that subtree exclusion for overlapping distfix grammars is safe and completely disambiguating.

Definition 2.6.4 (Total Set of Disambiguation Rules for Overlapping Distfix Grammars)**.** *A set of disambiguation rules PR for a distfix grammar G is total for a non-terminal A:*

- *If for any pair of productions* $A.C_1 = \alpha\ A \in G$ *and* $A.C_2 = A\ \gamma \in G$*, either* $A.C_1 > A.C_2 \in PR$ *or* $A.C_2 > A.C_1 \in PR$*.*
- *If two (not necessarily distinct) productions* $A.C_1 = A\ \beta_1\ A \in G, A.C_2 = A\ \beta_2\ A \in G$*, then either* $A.C_1\ R\ A.C_2 \in PR$ *or* $A.C_2\ R\ A.C_1 \in PR$*, where* $R \in \{>, \texttt{right}, \texttt{left}, \texttt{non-assoc}\}$*.*
- *If* $A.C_1 = \alpha\ A\ \gamma\ \in G$ *and* $A.C_2 = \alpha\ A \in G$*, or* $A.C_1 = \alpha\ A\ \gamma \in G$ *and* $A.C_2 = A\ \gamma \in G$*, either* $A.C_1 > A.C_2 \in PR$ *or* $A.C_2 > A.C_1 \in PR$*.*

Lemma 2.6.5 (Subtree Exclusion for Overlapping Distfix Expression Grammars is Safe)**.** *Given a distfix expression grammar G with only harmless overlap, and Q the set of safe, left-most, and right-most deep conflict patterns generated from a total set of the disambiguation rules for G, if $w \in L(G)$ then there is a $t \in T^Q(G)$, such that $yield(t) = w$.*

Proof. By induction on the length of sentences in $L(G)$.

(Base case) If a is a lexeme then $a \in T^Q_a(G)$ since disambiguation rules do not exclude lexemes.

(Inductive case) Assume that there are $u_i \in L(G)$ and that there are $t_i \in T^Q_A(G)$ such that $yield(t_i) = u_i$.

(1) If $A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i A \oplus_{i+1} ... \oplus_n A \blacktriangleleft$ is a closed distfix production in G, then $w = \blacktriangleright\ u_1 ... \oplus_i u_i \oplus_{i+1} ... \oplus_n u_n \blacktriangleleft \in L(G)$. We need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. Take $t = [A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i t_i \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$. By induction $u_i = yield(t_i)$ such that $t_i \in T^Q(G), \forall 1 \leq i \leq n$. We consider the following cases:

   – Let $q = [A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i [A.C_1 = \blacktriangleright\ A \oplus_1 ... \oplus_i A] \oplus_{i+1} ... \oplus_n A \blacktriangleleft]$. If $\neg \mathcal{M}^{rm}(t, q)$, then $t \in T^Q(G)$, since $A.C$ is a closed distfix expression. However, if $\mathcal{M}^{rm}(t, q)$ and $A.C > A.C_1$, then $t = [A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i [..[A.C_1 = \blacktriangleright\ t_{11} \oplus_1 ... \oplus_i t_{1i}]] \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$ has a deep priority conflict and $t \notin T^Q(G)$. However, the tree $t' = [A.C_1 = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i [..[A.C = \blacktriangleright\ t_{11} \oplus_1 ... \oplus_i t_{1i} \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]]]$ has the same yield and does *not* have a priority conflict, i.e., $t' \in T^Q(G)$.

   – Similarly, let $q = [A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i [A.C_2 = A \oplus_{i+1} ... \oplus_n A \blacktriangleleft] \oplus_{i+1} ... \oplus_n A \blacktriangleleft]$. if $\neg \mathcal{M}^{lm}(t, q)$, then $t \in T^Q(G)$. However, if $\mathcal{M}^{lm}(t, q)$ and $A.C > A.C_2$, $t = [A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i [[A.C_2 = t_{21} \oplus_{i+1} ... \oplus_n t_{2n} \blacktriangleleft]..] \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$ matches a deep priority conflict and $t \notin T^Q(G)$. However, the tree $t'' = [A.C_2 = [[A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i t_{21} \oplus_{i+1} ... \oplus_n t_{2n} \blacktriangleleft]..] \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$ has the same yield and does *not* have a priority conflict, thus, $t'' \in T^Q(G)$.

(2) The remaining cases involving the production $A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i A \oplus_{i+1} ... \oplus_n A$ defining a prefix distfix expression, the production $A.C = A \oplus_1 ... \oplus_i A \oplus_{i+1} ... A \blacktriangleleft$ defining a postfix distfix expression, and the production $A.C = A \oplus_1 ... \oplus_i A \oplus_{i+1} ... \oplus_n A$ defining an infix distfix expression are analogous to the case above. Note that when $t_i = t_1$ or $t_i = t_n$, i.e., the conflict occurs at the leftmost or rightmost tree, the analysis is analogous to the proof of Lemma 2.5.7. □

Lemma 2.6.6 (Subtree Exclusion for Overlapping Distfix Expression Grammars is Completely Disambiguating)**.** *Given a distfix expression grammar G with only harmless overlap, and Q the set of safe, left-most, and right-most deep conflict patterns generated from a total set of the disambiguation rules for G, then all trees in $T^Q(G)$ have unique yields, i.e., if $t_1, t_2 \in T^Q(G)$ and $yield(t_1) = yield(t_2)$ then $t_1 = t_2$.*

Proof. By induction on $T^Q(G)$.

(Base case) If a is a lexeme, then $a \in T^Q_a(G)$ and has a unique yield.

(Inductive case) Assume that $t_i \in T^Q_A(G)$ and that their yields are unique.

(1) If $A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i A \oplus_{i+1} ... \oplus_n A \blacktriangleleft$ is a closed distfix production in G, we can construct the tree $t = [A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i t_i \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$ with yield $w = \blacktriangleright\ u_1 ... \oplus_i u_i \oplus_{i+1} ... \oplus_n u_n \blacktriangleleft$. We need to demonstrate that if $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$. Consider the following cases:

- Let $q = [A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i [A.C_1 = \blacktriangleright\ A \oplus_1 ... \oplus_i A] \oplus_{i+1} ... \oplus_n A \blacktriangleleft]$. If $t = [A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i [..[A.C_1 = \blacktriangleright\ t_{11} \oplus_1 ... \oplus_i t_{1i}]] \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$, $A.C > A.C_1$, and $\mathcal{M}^{rm}(t, q)$ then t has a deep priority conflict and $t \notin T^Q(G)$. If $A.C_1 > A.C$, then $t \in T^Q(G)$. The tree $t' = [A.C_1 = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i [..[A.C = \blacktriangleright\ t_{11} \oplus_1 ... \oplus_i t_{1i} \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]]]$ is the only tree with the same yield, since by our definition of harmless overlap, no other productions can be used to derive w. However t' matches a priority conflict when $A.C_1 > A.C$ i.e., $t' \notin T^Q(G)$.
- Similarly, let $q = [A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i [A.C_2 = A \oplus_{i+1} ... \oplus_n A \blacktriangleleft] \oplus_{i+1} ... \oplus_n A \blacktriangleleft]$. If $t = [A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i [[A.C_2 = t_{21} \oplus_{i+1} ... \oplus_n t_{2n} \blacktriangleleft]..] \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$, $A.C > A.C_2$, and $\mathcal{M}^{lm}(t, q)$, t matches a deep priority conflict and $t \notin T^Q(G)$. If $A.C_2 > A.C$, then $t \in T^Q(G)$. The tree $t'' = [A.C_2 = [[A.C = \blacktriangleright\ t_1 \oplus_1 ... \oplus_i t_{21} \oplus_{i+1} ... \oplus_n t_{2n} \blacktriangleleft]..] \oplus_{i+1} ... \oplus_n t_n \blacktriangleleft]$ is the only tree with the same yield, since by our definition of harmless overlap, no other productions can be used to derive w. However t' matches a priority conflict when $A.C_2 > A.C$, thus, $t'' \notin T^Q(G)$.

(2) The remaining cases involving prefix distfix productions $A.C = \blacktriangleright\ A \oplus_1 ... \oplus_i A \oplus_{i+1} ... \oplus_n A$, the postfix distfix productions $A.C = A \oplus_1 ... \oplus_i A \oplus_{i+1} ... A \blacktriangleleft$, and infix distfix productions $A.C = A \oplus_1 ... \oplus_i A \oplus_{i+1} ... \oplus_n A$ are analogous to the case above. Note that when $t_i = t_1$ or $t_i = t_n$, i.e., the conflict occurs at the leftmost or rightmost tree, the analysis is analogous to the proof of Lemma 2.5.8. □

Theorem 2.6.7. *Disambiguation of a distfix expression grammar with harmless overlap using a total set of disambiguation rules is safe and completely disambiguating.*

Proof. As before by Lemma 2.6.5 and Lemma 2.6.6. □

2.7 INDIRECT RECURSION

In this section, we examine the extension of distfix expression grammars to *indirectly recursive* distfix expression grammars. Ambiguities involving indirectly recursive distfix expressions may also occur deeply nested in a tree, and require resolution using deep priority conflicts. For example, consider the

```
context-free syntax
  Exp.Add      = Exp "+" Exp {left}
  Exp.Function = "function" Match
  Match.Clause = ID "->" Exp
  Exp.Var      = ID
  Exp          = "(" Exp ")" {bracket}
context-free priorities
  Exp.Add > Exp.Function
```

Figure 2.31 An example of a grammar containing indirect recursion.

grammar of Figure 2.31, which models (simplified) pattern match clauses in OCaml using the non-terminal `Match`. The production `Match.Clause` uses the non-terminal `Exp` in its right-most position, making `Exp` and `Match` mutually recursive. Parsing the sentence `a + function p -> b + c`, produces the following parse trees:

```
(1) [Exp.Add = a + [Exp.Function = function [Match.Clause =
  p -> [Exp.Add = b + c]]]]
(2) [Exp.Add = [Exp.Add = a + [Exp.Function = function
  [Match.Clause = p -> b]]] + c]
```

This ambiguity is similar to the deep priority conflicts we saw before. The priority `Exp.Add > Exp.Function` indicates that tree `(2)` should be rejected and tree `(1)` should be selected. However, the priority does not generate the required deep priority conflict pattern, since the productions involved are not directly recursive, and hence do not match the rules in Definition 2.4.3. In general, the patterns we defined in Definitions 2.4.3, 2.5.2, and 2.6.3 are not applicable to indirectly recursive distfix expressions.

2.7.1 *Disambiguating Indirectly Recursive Distfix Grammars*

The following definition generalizes our previous semantic rules to indirectly recursive expression grammars, subsuming the rules for shallow and deep conflict patterns for directly recursive expression grammars.

Definition 2.7.1 (Indirectly Recursive Conflict Patterns)**.** *Given a distfix expression grammar G with indirect recursion and disambiguation rules PR defining symmetric, non-transitive relations* `right` *and* `left` [12]*, and irreflexive, anti-symmetric, and transitive relation $>$ over the productions of G, the sets Q_G^{lm} and Q_G^{rm} of left-most and right-most deep conflict patterns are the smallest sets such that:*

$$\frac{A.C_2 > A.C_1 \in PR \quad \alpha \overset{*}{\Rightarrow}_G \alpha' A}{[A.C_2 = [A.C_1 = \alpha]\gamma] \in Q_G^{rm}} \tag{2.53}$$

[12] The relation `non-assoc` is interpreted as left associative, and `non-assoc` and `non-nested` generate the symmetric patterns as warnings, which we leave out here.

$$\frac{A.C_1 > A.C_2 \in PR \quad \gamma \overset{*}{\Rightarrow}_G A\gamma'}{[A.C_1 = \alpha[A.C_2 = \gamma]] \in Q_G^{lm}} \tag{2.54}$$

$$\frac{A.C_2 > A.C_1 \in PR \quad \alpha' \overset{*}{\Rightarrow}_G \alpha A}{[A.C_2 = \alpha[A.C_1 = \alpha']\gamma] \in Q_G^{rm}} \tag{2.55}$$

$$\frac{A.C_1 > A.C_2 \in PR \quad \alpha' \overset{*}{\Rightarrow}_G \alpha A}{[A.C_1 = \alpha[A.C_2 = \alpha'\gamma]] \in Q_G^{lm}} \tag{2.56}$$

$$\frac{A.C_1 > A.C_2 \in PR \quad \gamma' \overset{*}{\Rightarrow}_G A\gamma}{[A.C_1 = \alpha[A.C_2 = \gamma']\gamma] \in Q_G^{lm}} \tag{2.57}$$

$$\frac{A.C_2 > A.C_1 \in PR \quad \gamma' \overset{*}{\Rightarrow}_G A\gamma}{[A.C_2 = [A.C_1 = \alpha\gamma']\gamma] \in Q_G^{rm}} \tag{2.58}$$

$$\frac{A.C_1 \ \texttt{right}\ A.C_2 \in PR \quad \beta_2 \overset{*}{\Rightarrow}_G A\beta A \quad \beta_1 \overset{*}{\Rightarrow}_G \alpha A}{[A.C_1 = [A.C_2 = \beta_2]\beta_1] \in Q_G^{rm}} \tag{2.59}$$

$$\frac{A.C_1 \ \texttt{left}\ A.C_2 \in PR \quad \beta_2 \overset{*}{\Rightarrow}_G A\beta A \quad \beta_1 \overset{*}{\Rightarrow}_G A\gamma}{[A.C_1 = \beta_1[A.C_2 = \beta_2]] \in Q_G^{lm}} \tag{2.60}$$

□

The conflict patterns above take into consideration indirect recursion, capturing ambiguities that can be solved by priority rules. The patterns in Equations 2.53 and 2.54 disambiguate prefix- and postfix-basic ambiguities involving indirectly recursive distfix expressions. The patterns in Equations 2.55 to 2.58 address dangling prefix and dangling suffix ambiguities, now considering indirect recursion. Note that because conflicts due to indirect recursion require deep matching, associativity declarations also contribute to indirectly recursive patterns, as shown by Equations 2.59 and 2.60.

For example, consider again the sentence `a + function p -> b + c` and the trees constructed when parsing it. Since `function Match` $\overset{*}{\Rightarrow}_G$ `function Pattern -> Exp`, the priority rule `Exp.Add > Exp.Function` now generates the following conflict pattern according to Equation 2.53:

$$\frac{\texttt{Exp.Add} > \texttt{Exp.Function} \in \text{PR}}{\texttt{[Exp.Add = [Exp.Function = function Match] + Exp]} \in Q_G^{rm}}$$

This pattern deeply matches the tree `(2)`, and therefore tree `(1)` is selected.

Consider now the grammar in Figure 2.32, containing a dangling suffix conflict involving indirectly recursive distfix expressions. Parsing a sentence `if a then if b then s1 else s2` creates the following trees:

```
(3) [CondStmt.IfElse = if a then [Stmt.Cond = [CondStmt.If =
  if b then s1]] else s2]
(4) [CondStmt.If = if a then [Stmt.Cond = [CondStmt.IfElse =
  if b then s1 else s2]]]
```

```
context-free syntax
  Stmt.Cond       = CondStmt
  CondStmt.If     = "if" Exp "then" Stmt
  CondStmt.IfElse = "if" Exp "then" Stmt "else" Stmt
  Exp.Var         = ID
context-free priorities
  CondStmt.IfElse > CondStmt.If
```

Figure 2.32 Grammar containing a dangling suffix conflict caused by indirect recursion.

Since `if Exp then Stmt` $\overset{*}{\Rightarrow}_G$ `if Exp then CondStmt`, the indirectly recursive interpretation of the priority rule `CondStmt.IfElse > CondStmt.If` generates the following pattern, according to Equation 2.55:

$$\frac{\texttt{CondStmt.IfElse} > \texttt{CondStmt.If} \in \text{PR}}{\texttt{[CondStmt.IfElse = if Exp then [CondStmt.If = ...] ...]} \in Q_G^{rm}}$$

Since this conflict pattern deeply matches tree `(3)`, a disambiguation filter constructed using this pattern rejects this tree, and tree `(4)` is selected.

Lemma 2.7.2 (Deep Conflicts Subsume Shallow Conflicts)**.** *The deep priority conflict patterns of Definition 2.7.1 include the shallow conflict patterns of Definition 2.4.1.*

Proof. By inspection, each of the rules for Q_G^{safe} is included in either Q_G^{rm} or Q_G^{lm}, and rightmost and leftmost deep matching includes shallow matching. □

Hidden recursion is another problem that may occur when we consider indirect recursion. Hidden recursion occurs when a non-terminal can derive the empty string, and hides another recursive non-terminal. We address hidden recursion by applying a grammar transformation to make the recursion explicit (Section 2.8.4).

2.7.2 *Nested Lists*

Programming language grammars often feature lists of expressions. For example, consider the actual parameters of a function call (typically a list of expressions separated by commas) or the bindings in a let expression. These particular examples probably do not involve ambiguities since the lists are closed within delimiters. However, other lists do give rise to ambiguities, in particular when they may be nested. In SDF3, the regular symbols A^+ and A^* represent shorthands for lists with one or more, or zero or more elements, respectively. For example, consider the grammar in Figure 2.33. The production `Exp.Do` (based on the Haskell do-notation) defines a construct with a list of one or more expressions as argument. The production `Exp.Function` defines

```
context-free syntax
  Exp.Do       = "do" Exp+
  Exp.Function = "function" Match+
  Match.Clause = Pattern "->" Exp
  Exp.Var      = ID
```

Figure 2.33 Grammar containing nested list ambiguities.

a function literal expression, which pattern matches its arguments with one or more pattern match clauses.

We consider the extension of distfix expression grammars with list symbols. SDF3 defines list symbols using the following production schemas[13]:

$A^*.Lst_A = A^+$
$A^*.Nil_A = \epsilon$
$A^+.App_A = A^+\ A$
$A^+.Sng_A = A$

Given the grammar of Figure 2.33 and the productions for list symbols, the sentence `do do a b` has the following parse trees:

```
(1) [Exp.Do = do [Exp+.App = [Exp+.Sng = [Exp.Do = do a]] b]]
(2) [Exp.Do = do [Exp+.Sng = [Exp.Do = do [Exp+.App = a b]]]]
```

An ambiguity involving nested lists occurs because the expression `b` can belong to the inner or outermost list. The standard resolution policy is to expand the inner list as much as possible, i.e., the *longest match*. When considering the standard disambiguation policy and the example above, the inner list should be longest match, thus only tree `(2)` should be produced. Conversely, rejecting tree `(2)` and producing tree `(1)` specifies the opposite, i.e., the *shortest match*.

Erdweg et al. [45] propose to use a post-parse filter that selects the alternative that represents the longest innermost list from all possible parse trees. While this solution works in principle, it does not scale to larger programs, since the number of ambiguities grows exponentially in the length of the list elements. Moreover, it only enables longest match disambiguation, since it lacks an alternative to implement the dual disambiguation based on shortest match. Therefore, instead of applying a post parse filter, we propose a solution based on deep priority conflicts.

Definition 2.7.3 (Nested List Conflict Patterns). *Given a grammar G containing productions of the form $A.C_1 = \alpha B^+$ such that $B \overset{*}{\Rightarrow}_G \alpha' A$, the set Q^{rm}_G is extended with the following conflict patterns over G with respect to longest match or shortest match lists B^+ (or B^*):*

$$\frac{A.C = \alpha B^+ \ \{\texttt{longest-match}\} \in PR \quad B \overset{*}{\Rightarrow}_G \alpha' A}{[B^+.App_B = [A.C = \alpha B^+]B] \in Q^{rm}_G} \tag{2.61}$$

[13] SDF3 also defines lists with separators $\{A\ s\}+$ and $\{A\ s\}*$. These are defined analogously.

$$\frac{A.C = \alpha B^+ \ \{\texttt{shortest-match}\} \in PR \quad B \overset{*}{\Rightarrow}_G \alpha' A}{[B^+.Sng_B = [B^+.App_B = B^+B]] \in Q^{rm}_G} \tag{2.62}$$

$$\frac{A.C_1 = B^+\gamma \ \{\texttt{longest-match}\} \in PR \quad B \overset{*}{\Rightarrow}_G \alpha' A}{[B^+.App_{A_1} = [A.C_1 = \alpha B^+]B] \in Q^{rm}_G} \tag{2.63}$$

$$\frac{A.C_1 = B^+\gamma \ \{\texttt{shortest-match}\} \in PR \quad B \overset{*}{\Rightarrow}_G A\gamma'}{[B^+.Cons_B = [B^+.App_{A_1} = B^+B]] \in Q^{rm}_G} \tag{2.64}$$

Equations 2.61 and 2.62 define the conflict patterns for productions with nested lists as postfixes, whereas Equations 2.63 and 2.64 define conflict patterns for productions with nested lists as prefixes. □

```
context-free syntax
  Exp.Do       = "do" Exp+ {longest-match}
  Exp.Function = "function" Match+ {shortest-match}
  Match.Clause = Pattern "->" Exp
  Exp.Var      = ID
```

Figure 2.34 Disambiguation rules to implement longest match and shortest match.

To express disambiguation by longest or shortest match, we extend SDF3 disambiguation rules with `longest-match` and `shortest-`**`match`** annotations to specify that a list is longest or shortest match, respectively. Consider the grammar in Figure 2.34, which contains disambiguation rules for both lists in the grammar of Figure 2.33. The annotation in the production `Exp.Do` creates the following conflict pattern:

$$\frac{\texttt{Exp.Do = "do" Exp}^+ \ \texttt{\{longest-match\}} \in \text{PR}}{\texttt{[Exp}^+\texttt{.App}_{\texttt{Exp}} \texttt{ = [Exp.Do = "do" Exp}^+\texttt{] Exp]} \in Q^{rm}_G}$$

Since this pattern deeply matches tree `(1)` above, only tree `(2)` is selected by a subtree exclusion filter that considers this pattern.

2.7.3 *Safe and Complete Disambiguation*

We extend the safety and completeness proofs to indirectly recursive distfix expression grammars.

Definition 2.7.4 (Total Set of Disambiguation Rules)**.** *A set of disambiguation rules PR for a grammar G is total for a non-terminal A,*

- *If for any pair of productions $A.C_1 = \alpha \in G$ and $A.C_2 = \gamma \in G$, such that $\alpha \overset{*}{\Rightarrow}_G \alpha' A, \gamma \overset{*}{\Rightarrow}_G A\gamma'$, either $A.C_1 > A.C_2 \in PR$ or $A.C_2 > A.C_1 \in PR$.*
- *If two (not necessarily distinct) productions $A.C_1 = \beta_1, A.C_2 = \beta_2 \in P(G)$, with $\beta_1 \overset{*}{\Rightarrow}_G A\beta_1' A$ and $\beta_2 \overset{*}{\Rightarrow}_G A\beta_2' A$, then either $A.C_1 \ R \ A.C_2 \in PR$ or $A.C_2 \ R \ A.C_1 \in PR$, $R \in \{>, \texttt{right}, \textbf{\texttt{left}}\}$.*

- *If* $A.C_1 = \alpha\gamma$ *and* $A.C_2 = \alpha$*, such that* $\alpha \stackrel{*}{\Rightarrow}_G \alpha'\ A$*, or* $A.C_1 = \alpha\gamma$ *and* $A.C_2 = \gamma$*, such that* $\gamma \stackrel{*}{\Rightarrow}_G A\ \gamma'$*, either* $A.C_1 > A.C_2 \in PR$ *or* $A.C_2 > A.C_1 \in PR$*.* □

Lemma 2.7.5 (Subtree Exclusion for Indirectly Recursive Distfix Expression Grammars is Safe)**.** *Given an indirectly recursive distfix expression grammar G with only harmless overlap, and the set Q of left-most, and right-most deep priority conflict patterns generated by the disambiguation rules for G, if* $w \in L(G)$ *then there is a* $t \in T^Q(G)$*, such that* $yield(t) = w$*.*

Proof. By induction on the length of sentences in $L(G)$.

(Base case) If a is a lexeme then $a \in T^Q_a(G)$ since disambiguation rules do not exclude lexemes.

(Inductive case) Assume that $u_i \in L(G)$ and that there are $t_i \in T^Q_{B_i}(G)$ such that $yield(t_i) = u_i$, for any $0 \leq i \leq n$, then there are the following cases:

(1) If $A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_n B_n$ is an indirectly recursive distfix production in G, i.e., $B_i \stackrel{*}{\Rightarrow}_G \alpha A$, for any $0 \leq i \leq n$, then $\blacktriangleright\ u_1 ... \oplus_n u_n = w \in L(G)$. We need to demonstrate that there is a $t \in T^Q(G)$ such that $yield(t) = w$. Take $t = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_n t_n]$. By induction $u_i = yield(t_i)$ such that $t_i \in T^Q(G), \forall 0 \leq i \leq n$. Consider the following cases:

- Consider $B_i \stackrel{*}{\Rightarrow}_G \alpha A$ and $i \neq n$, and let $q = [A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_i [A.C_1 = \blacktriangleright\ B_0 \oplus_1 ... \oplus_i B_i] \oplus_{i+1} ... \oplus_n B_n]$. If $\neg\mathcal{M}^{rm}(t, q)$ then $t \in T^Q(G)$, since both $A.C$ and $A.C_1$ define indirectly recursive prefix distfix expressions. However, if $\mathcal{M}^{rm}(t, q)$ and $A.C > A.C_1$, then $t = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_i [..[A.C_1 = \blacktriangleright\ t_{10} \oplus_1 ... \oplus_i t_{1i}]] \oplus_{i+1} ... \oplus_n t_n]$ has a deep priority conflict and $t \notin T^Q(G)$. However, the tree $t' = [A.C_1 = \blacktriangleright\ t_0 \oplus_1 ... \oplus_i [..[A.C = \blacktriangleright\ t_{10} \oplus_1 ... \oplus_i t_{1i} \oplus_{i+1} ... \oplus_n t_n]]]$ has the same yield and does *not* have a conflict, thus $t' \in T^Q(G)$. A similar analysis can be applied if $B_i \stackrel{*}{\Rightarrow}_G A\gamma$, with $q' = [A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_i [A.C_2 = B_i \oplus_{i+1} ... \oplus_n B_n] \oplus_{i+1} ... \oplus_n B_n]$, and $\mathcal{M}^{lm}(t, q')$.
- Consider $B_i \stackrel{*}{\Rightarrow}_G \alpha A$ and $i = n$, and let $q = [A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_n [A.C_1 = B'_0 \otimes_1 ... \otimes_n B'_n]]$. If $\neg\mathcal{M}^{rm}(t, q)$ and $A.C_1 > A.C$, then $t \in T^Q(G)$. However, if $\mathcal{M}^{rm}(t, q)$, $B'_0 \stackrel{*}{\Rightarrow}_G A\gamma$, and $A.C > A.C_1$, then $t = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_n [..[A.C_1 = t_{10} \otimes_1 ... \otimes_n t_{1n}]]]$ has a deep priority conflict, and $t \notin T^Q(G)$. On the other hand, the tree $t' = [A.C_1 = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_n [..t_{10}]] \otimes_1 ... \otimes_n t_{1n}]$ has the same yield and does not have a conflict, thus $t' \in T^Q(G)$. The analysis is similar if $A.C_1 = B'_0 \otimes_1 ... \otimes_n B'_n \blacktriangleleft$, and $B'_0 \stackrel{*}{\Rightarrow}_G A\gamma$.

(2) The analysis is analogous for the other indirectly recursive distfix productions. □

Lemma 2.7.6 (Subtree Exclusion for Indirectly Recursive Distfix Expression Grammars is Completely Disambiguating)**.** *Given an indirectly recursive distfix expression grammar G and the set Q of left-most, and right-most deep priority conflict patterns generated from a total set of disambiguation rules for G, then all trees in $T^Q(G)$ have unique yields. That is, if $t_1, t_2 \in T^Q(G)$ and $yield(t_1) = yield(t_2)$ then $t_1 = t_2$.*

Proof. By induction on $T^Q(G)$.

(Base case) If a is a lexeme, then $a \in T^Q_a(G)$ and has a unique yield.

(Inductive case) Assume that $t_i \in T^Q_{B_i}(G)$, such that $B_i \stackrel{*}{\Rightarrow}_G \alpha A$ or $B_i \stackrel{*}{\Rightarrow}_G A\gamma$ and that their yields are unique. Then there are the following cases:

(1) If $A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_n B_n$ is an indirectly recursive distfix production in G, i.e., $B_i \stackrel{*}{\Rightarrow}_G \alpha A$, for any $0 \leq i \leq n$, then we can construct the tree $t = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_n t_n]$ with yield $\blacktriangleright\ u_1 ... \oplus_n u_n = w$. We need to demonstrate that *if* $t \in T^Q(G)$ then there is no tree $t' \in T^Q(G)$ such that $t' \neq t$ and $yield(t) = yield(t')$. Consider the following cases:

- Consider $B_i \stackrel{*}{\Rightarrow}_G \alpha A$ and $i \neq n$, and let $q = [A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_i [A.C_1 = \blacktriangleright\ B_0 \oplus_1 ... \oplus_i B_i] \oplus_{i+1} ... \oplus_n B_n]$. If $t = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_i [..[A.C_1 = \blacktriangleright\ t_{10} \oplus_1 ... \oplus_i t_{1i}]] \oplus_{i+1} ... \oplus_n t_n]$, $A.C > A.C_1$, and $\mathcal{M}^{rm}(t, q)$, then $t \notin T^Q(G)$. If $A.C_1 > A.C$, $t \in T^Q(G)$. The tree $t' = [A.C_1 = \blacktriangleright\ t_0 \oplus_1 ... \oplus_i [..[A.C = \blacktriangleright\ t_{10} \oplus_1 ... \oplus_i t_{1i} \oplus_{i+1} ... \oplus_n t_n]]]$ is the only tree with the same yield, since by our definition of harmless overlap, no other productions can be used to derive w. However, when $A.C_1 > A.C$, $t' \notin T^Q(G)$. A similar analysis can be applied if $B_i \stackrel{*}{\Rightarrow}_G A\gamma$ and we consider $q' = [A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_i [A.C_2 = B_i \oplus_{i+1} ... \oplus_n B_n]$, such that $\mathcal{M}^{lm}(t, q')$.
- Consider $B_i \stackrel{*}{\Rightarrow}_G \alpha A$ and $i = n$, and let $q = [A.C = \blacktriangleright\ B_0 \oplus_1 ... \oplus_n [A.C_1 = B'_0 \otimes_1 ... \otimes_n B'_n]]$, with $B'_0 \stackrel{*}{\Rightarrow}_G A\gamma$. If $t = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_n [..[A.C_1 = t_{10} \otimes_1 ... \otimes_n t_{1n}]]]$, $A.C > A.C_1$, and $\mathcal{M}^{rm}(t, q)$, then $t \notin T^Q(G)$. However, if $A.C_1 > A.C$, then $t \in T^Q(G)$. The tree $t' = [A.C_1 = [A.C = \blacktriangleright\ t_0 \oplus_1 ... \oplus_n [..t_{10}]] \otimes_1 ... \otimes_n t_{1n}]$ is the only tree with the same yield, since by our definition of harmless overlap, no other productions can be used to derive w. However, when $A.C_1 > A.C$ then $t' \notin T^Q(G)$. The analysis is similar if $A.C_1 = B'_0 \otimes_1 ... \otimes_n B'_n \blacktriangleleft$, and $B'_0 \stackrel{*}{\Rightarrow}_G A\gamma$.

(2) The analysis is analogous for the other indirectly recursive distfix productions. Note that we only guarantee that the trees in the embedded expression grammar are unique. Since the other non-terminals may have arbitrary productions, we cannot guarantee that subtree exclusion is completely disambiguating when considering the other subtrees. □

Theorem 2.7.7. *Disambiguation of an indirectly recursive distfix expression grammar using a total set of disambiguation rules is safe and completely disambiguating.*

Proof. As before by Lemma 2.7.5 and Lemma 2.7.6. □

2.8 GRAMMAR TRANSFORMATION

In the previous sections we presented a semantics for disambiguation using priority and associativity rules that can be applied to expression grammars of increasing complexity. In this section we define a canonical implementation of this semantics by means of a grammar transformation.

2.8.1 *Transformation to Contextual Grammars*

The main strategy of our grammar transformation is to create copies of existing productions using *contextual non-terminal symbols*. Such symbols act as filter by forbidding particular trees to be nested as the rightmost or leftmost subtrees. We formally define contextual symbols and contextual productions as follows.

Definition 2.8.1 (Contextual Symbols and Productions)**.** *Given a context-free grammar G, we define a contextual symbol as a non-terminal ${}^{lm}A^{rm} \in N_{ctx}(G)$, with $A \in N(G)$ and lm and rm sets of constructors for productions in $P(G)$ named contextual tokens. We may omit empty sets of contextual tokens, that is, ${}^{lm}A = {}^{lm}A^{\emptyset}$, $A^{rm} = {}^{\emptyset}A^{rm}$, and $A = {}^{\emptyset}A^{\emptyset}$. A contextual production is a production with contextual symbols. The function $d({}^{lm}A^{rm}) = A$ drops the contextual annotation of a contextual non-terminal and is lifted to productions, grammars, and parse trees by point-wise application to all their non-terminals.* □

The first step of the grammar transformation consists in marking non-terminals in the body of productions to indicate which productions may not occur (deeply nested) in those positions.

Definition 2.8.2 (Marking Symbols)**.** *Given a context-free grammar G, and sets of leftmost patterns Q_G^{lm} and rightmost patterns Q_G^{rm}, the basic contextual grammar $P(G_{ctx})$ contains for each production $[A.C = X_1 \dots X_n] \in P(G)$, a contextual production $[A.C = {}^{lm_1}X_1{}^{rm_1} \dots {}^{lm_n}X_n{}^{rm_n}]$ where*

$$lm_i = \{X_i.C_i \mid [A.C = X_1..X_{i-1}\ [X_i.C_i = \gamma]\ X_{i+1}..X_n] \in Q_G^{lm}\}$$
$$rm_i = \{X_i..C_i \mid [A.C = X_1..X_{i-1}\ [X_i.C_i = \gamma]\ X_{i+1}..X_n] \in Q_G^{rm}\}$$

□

Figure 2.35 illustrates the result of this transformation applied to the expression grammar with addition, subtraction, and multiplication with their standard disambiguation rules. (We have left out the non-terminal prefix of the constructors for readability.) Note how the disambiguation rules are expressed by the contextual non-terminal symbols. For example, according to the priorities and the conflict patterns created, the multiplication cannot have an addition as its child, and the right-most child of an addition cannot be an addition or subtraction.

```
context-free syntax
  Exp.Add = Exp "+" {Add,Sub}Exp
  Exp.Sub = Exp "-" {Add,Sub}Exp
  Exp.Mul = Exp{Add,Sub} "+" {Mul,Add,Sub}Exp
  Exp.Var = ID
```

Figure 2.35 Basic contextual transformation for an expression grammar containing addition, subtraction, and multiplication.

This first step of the transformation marks the positions where certain productions are (deeply) forbidden. The next step of the transformation, dubbed *contextual closure*, generates the productions that define the contextual non-terminal symbols thus introduced, and propagates contextual tokens to rightmost and leftmost recursive positions, repeating the process for the contextual symbols created in the propagation. Figure 2.36 shows the result of applying the contextual closure to the grammar of Figure 2.35.

Definition 2.8.3 (Contextual Closure)**.** *Given a basic contextual grammar G_{ctx} derived according to Definition 2.8.2 for a grammar G, the closure G_{ctx}^{cls} of G_{ctx} is the smallest grammar such that $G_{ctx} \subseteq G_{ctx}^{cls}$ and:*

$$\frac{{}^{lm}A^{rm} \in N(G_{ctx}^{cls}) \quad A.C \notin \{lm \cup rm\} \qquad A.C = {}^{lm'}B^{rm'}\ \gamma \in P(G_{ctx}) \quad \gamma \overset{*}{\not\Rightarrow}_G \beta A \quad B \overset{*}{\Rightarrow}_G A\gamma'}{{}^{lm}A^{rm}.C = {}^{lm' \cup lm}B^{rm'}\gamma \in P(G_{ctx}^{cls})} \tag{2.65}$$

$$\frac{{}^{lm}A^{rm} \in N(G_{ctx}^{cls}) \quad A.C \notin \{lm \cup rm\} \qquad A.C = \alpha\ {}^{lm'}B'^{rm'} \in P(G_{ctx}) \quad \alpha \overset{*}{\not\Rightarrow}_G A\beta \quad B' \overset{*}{\Rightarrow}_G \alpha' A}{{}^{lm}A^{rm}.C = \alpha\ {}^{lm'}B'^{rm' \cup rm} \in P(G_{ctx}^{cls})} \tag{2.66}$$

$$\frac{{}^{lm}A^{rm} \in N(G_{ctx}^{cls}) \quad A.C \notin \{lm \cup rm\} \qquad A.C = {}^{lm_1}B^{rm_1}\ \beta\ {}^{lm_2}B'^{rm_2} \in P(G_{ctx}) \quad B \overset{*}{\Rightarrow}_G A\gamma' \quad B' \overset{*}{\Rightarrow}_G \alpha' A}{{}^{lm}A^{rm}.C = {}^{lm_1 \cup lm}B^{rm_1}\ \beta\ {}^{lm_2}B'^{rm_2 \cup rm} \in P(G_{ctx}^{cls})} \tag{2.67}$$

$$\frac{{}^{lm}A^{rm} \in N(G_{ctx}^{cls}) \quad A.C \notin \{lm \cup rm\} \qquad A.C = \beta \in P(G_{ctx}) \quad \beta \overset{*}{\not\Rightarrow}_G A\gamma \quad \beta \overset{*}{\not\Rightarrow}_G \alpha A}{{}^{lm}A^{rm}.C = \beta \in P(G_{ctx}^{cls})} \tag{2.68}$$

□

Note that the grammar in Figure 2.36 describes the same language as the unambiguous expression/term/factor (ETF) grammar of Figure 2.9 in Section 2.3.2. We can obtain the ETF grammar by renaming the (equivalent) contextual symbols $\texttt{Exp}^{\{A,S\}}$, $^{\{A,S\}}\texttt{Exp}$, and $^{\{A,S\}}\texttt{Exp}^{\{A,S\}}$ as the non-terminal

```
context-free syntax
  Exp.A = Exp "+" {A,S}Exp                          // E = E "+" T
  Exp.S = Exp "-" {A,S}Exp                          // E = E "-" T
  Exp.M = Exp{A,S} "+" {M,A,S}Exp                   // E = T "*" F
  Exp.V = ID                                        // E = ID

  Exp{A,S}.M = Exp{A,S} "+" {M,A,S}Exp              // T = T "*" F
  Exp{A,S}.V = ID                                   // T = ID

  {A,S}Exp.M = {A,S}Exp{A,S} "+" {M,A,S}Exp         // T = T "*" F
  {A,S}Exp.V = ID                                   // T = ID

  {A,S}Exp{A,S}.M = {A,S}Exp{A,S} "+" {M,A,S}Exp{A,S} // T = T "*" F
  {A,S}Exp{A,S}.V = ID                              // T = ID

  {M,A,S}Exp{A,S}.V = ID                            // F = ID

  {M,A,S}Exp.V = ID                                 // F = ID
```

Figure 2.36 Contextual closure for the expression grammar of Figure 2.35 (we use `A` for `Add`, `S` for `Sub`, `M` for `Mul`, and `V` for `Var`). As shown by the comments, there is a correspondence with an unambiguous ETF grammar.

`Term` (`T`) and the contextual symbols ${}^{\{M,A,S\}}\texttt{Exp}^{\{A,S\}}$, ${}^{\{M,A,S\}}\texttt{Exp}$ as `Factor` (`F`). The set of the resulting productions correspond exactly to ETF productions as shown in the comments in Figure 2.36, except that our contextual grammar transformation preserves the shape of the abstract syntax trees, since productions with a single non-terminal on the right-hand side are inlined.

2.8.2 *Disambiguating Deep Priority Conflicts*

The example above shows how to use contextual grammars to disambiguate an infix expression grammar. However, such grammars are the simplest expression grammars, containing only *shallow priority conflicts*. When we consider both shallow and deep priority conflicts, the resulting grammar can be even larger due to the number of extra contextual symbols created when calculating the closure. For instance, consider the grammar in Figure 2.37a containing a dangling else ambiguity, and a lower priority prefix operator. Figure 2.37b shows the respective contextual grammar constructed by marking the (deep) priority conflicts from the disambiguation rules. Note that the contextual grammar encodes both shallow conflicts, (e.g. forbidding an addition to be a direct child of a multiplication), and deep priority conflicts (since the then-branch of the if-then-else cannot be an if-expression, nor the leftmost child of an addition can be if- and if-then-else expressions).

Calculating the contextual closure for the grammar in Figure 2.37b results

```
context-free syntax
  Exp.Mul = Exp "*" Exp {left}
  Exp.Add = Exp "+" Exp {left}
  Exp.IfE = "if" Exp "then" Exp "else" Exp
  Exp.If  = "if" Exp "then" Exp
  Exp.Var = ID
context-free priorities
  Exp.Mul > Exp.Add > Exp.IfE > Exp.If
```

(a)

```
context-free syntax
  Exp.Mul = Exp{If,IfE,Add} "*" {Mul,Add}Exp
  Exp.Add = Exp{If,IfE} "+" {Add}Exp
  Exp.IfE = "if" Exp "then" Exp{If} "else" Exp
  Exp.If  = "if" Exp "then" Exp
  Exp.Var = ID
```

(b)

Figure 2.37 Basic contextual transformation for a grammar with dangling else.

in a rather unwieldy grammar as shown in Figure 2.38. We obtain a large grammar as result due to the number of (deep) conflicts, which lead to many additional contextual symbols created by the contextual closure. One alternative is to restrict the generation of contextual grammars, excluding shallow conflicts from the transformation, and leaving their resolution to be performed during parse table generation, as we show in Section 2.9.3. If we only consider deep priority conflicts, we obtain the contextual grammars in Figure 2.39a. Note that the resulting contextual grammar from the contextual closure is considerably smaller when compared to the grammar of Figure 2.38. However, this grammar still contains duplicated productions to encode different precedence levels of operators that could not be eliminated even if we used a similar strategy as we used for ETF grammars, by trying to find equivalent contextual symbols and remove redundant productions. In Section 2.9.4 we present a technique based on data-dependent parsing that does not generate any duplicated productions, only marking contextual symbols and using them to disambiguate programs at parse-time.

2.8.3 *Termination and Correctness*

Below we discuss two properties of our canonical implementation of the semantics for disambiguation of priority and associativity in expression grammars we defined previously: *termination* and *correctness*.

Lemma 2.8.4 (Contextual Closure is Terminating)**.** *The transformation of Definition 2.8.3 is terminating.*

```
context-free syntax
  Exp.M              = Exp{I,IE,A} "*" {M,A}Exp
  Exp.A              = Exp{I,IE} "+" {A}Exp
  Exp.IE             = "if" Exp "then" Exp{I} "else" Exp
  Exp.I              = "if" Exp "then" Exp
  Exp.V              = ID

  Exp{I,IE,A}.M      = Exp{I,IE,A} "*" {M,A}Exp{I,IE,A}
  Exp{I,IE,A}.V      = ID
  {M,A}Exp.IE        = "if" Exp "then" Exp{I} "else" Exp
  {M,A}Exp.I         = "if" Exp "then" Exp
  {M,A}Exp.V         = ID
  Exp{I,IE}.M        = Exp{I,IE,A} "*" {M,A}Exp{I,IE}
  Exp{I,IE}.A        = Exp{I,IE} "+" {A}Exp{I,IE}
  Exp{I,IE}.V        = ID
  {A}Exp.M           = {A}Exp{I,IE,A} "*" {M,A}Exp
  {A}Exp.IE          = "if" Exp "then" Exp{I} "else" Exp
  {A}Exp.I           = "if" Exp "then" Exp
  {A}Exp.V           = ID
  Exp{I}.M           = Exp{I,IE,A} "*" {M,A}Exp{I}
  Exp{I}.A           = Exp{I,IE} "+" {A}Exp{I}
  Exp{I}.IE          = "if" Exp "then" Exp{I} "else" Exp{I}
  Exp{I}.V           = ID
  {A}Exp{I,IE}.M     = {A}Exp{I,IE,A} "*" {M,A}Exp{I,IE}
  {A}Exp{I,IE}.V     = ID
  {A}Exp{I,IE,A}.M   = {A}Exp{I,IE,A} "*" {M,A}Exp{I,IE,A}
  {A}Exp{I,IE,A}.V   = ID
  {M,A}Exp{I}.IE     = "if" Exp "then" Exp{I} "else" Exp{I}
  {M,A}Exp{I}.V      = ID
  {A}Exp{I}.M        = {A}Exp{I,IE,A} "*" {M,A}Exp{I}
  {A}Exp{I}.IE       = "if" Exp "then" Exp{I} "else" Exp{I}
  {A}Exp{I}.V        = ID
  {M,A}Exp{I,IE,A}.V = ID
  {M,A}Exp{I,IE}.V   = ID
```

Figure 2.38 Contextual closure of expression grammar with dangling else (we use `M` for `Mul`, `A` for `Add`, `I` for `If`, `IE` for `IfE`, and `V` for `Var`).

Proof. The basic transformation that generates the grammar G_{ctx} does not add any productions to G. The contextual closure adds productions for the contextual symbols in G_{ctx}, and for contextual symbols that arise as a result of computing the closure. Thus, the computation of the contextual closure terminates when no new productions can be added. Since the number of contextual symbols for an expression grammar is bounded by the set of constructors (productions) in the original grammar, the number of productions

```
context-free syntax
  Exp.Mul = Exp{If,IfE} "*" Exp
  Exp.Add = Exp{If,IfE} "+" Exp
  Exp.IfE = "if" Exp "then" Exp{If} "else" Exp
  Exp.If  = "if" Exp "then" Exp
  Exp.Var = ID
```

(a)

```
context-free syntax
  Exp.Mul = Exp{If,IfE} "*" Exp
  Exp.Add = Exp{If,IfE} "+" Exp
  Exp.IfE = "if" Exp "then" Exp{If} "else" Exp
  Exp.If  = "if" Exp "then" Exp
  Exp.Var = ID

  Exp{If,IfE}.Mul = Exp{If,IfE} "*" Exp{If,IfE}
  Exp{If,IfE}.Add = Exp{If,IfE} "+" Exp{If,IfE}
  Exp{If,IfE}.Var = ID

  Exp{If}.Mul = Exp{If,IfE} "*" Exp{If}
  Exp{If}.Add = Exp{If,IfE} "+" Exp{If}
  Exp{If}.IfE = "if" Exp "then" Exp{If} "else" Exp{If}
  Exp{If}.Var = ID
```

(b)

Figure 2.39 Basic contextual transformation and closure for a grammar with dangling else considering only deep conflict patterns.

added by the closure transformation is finite. □

The number of productions in the contextual closure is bound by the number of productions in the original expression grammar, since there is a finite number of contextual symbols ${}^{lm}A^{rm}$. In the worst-case scenario, if an expression grammar contains ctx different contextual tokens, and n non-terminals, the contexual grammar G_{ctx} contains at most $n * (2^{ctx} - 1)$ contextual symbols—each non-terminal generates a contextual symbol with a different subset of contextual tokens. Thus, in the worst case, the number of additional productions in the contextual grammar G_{ctx}^{cls} after calculating the contextual closure is $p * n * (2^{ctx} - 1)$, where p is the number of productions of the original expression grammar. This number is obtained by generating duplicated productions for each new contextual symbol. Since the number of states in the parse table also depends on the shape of each production, it is not possible to determine the effect on the number of states of a parse table constructed from a contextual grammar, as we will see in Section 2.10.

Theorem 2.8.5 (Correctness of Grammar Transformation)**.** *Given a context-free grammar G, and sets of leftmost patterns Q_G^{lm} and rightmost patterns Q_G^{rm}, let $Q = TP^{Q^{lm}} \cup TP^{Q^{rm}}$, let $G' = G_{ctx}^{cls}$, then $T^Q(G) = d(T(G'))$. That is, the grammar produced by the contextual closure has the same parse trees as the grammar under subtree exclusion.*

Proof. We can show that the trees in $T^Q(G)$ and $T(G')$ are equivalent by a direct correspondence between the definitions of subtree exclusion and contextual grammars. It is straightforward to see that contextual productions embody conflict patterns, since the initial marking of the grammar indicates the positions at which productions may not occur according to the conflict patterns. Furthermore, the complete fixpoint algorithm defined by the contextual closure in Definition 2.8.3 constitutes an implementation of the deep matching function from Definitions 2.4.2 and 2.5.1:

- Equations 2.41 and 2.45 correspond to the condition ${}^{lm}A^{rm} \in N(G_{ctx}^{cls})$ in Equations 2.65-2.68, i.e., deep matching is considered for each contextual symbol, since contextual symbols encode conflict patterns.
- Equations 2.42 and 2.46 correspond to the condition $A.C \notin \{lm \cup rm\}$ in Equations 2.65-2.68. The condition $\mathcal{M}(t, q)$ in a filter that uses deep matching states that if a tree t matches a conflict pattern, $t \notin T^Q(G)$. Similarly, if the condition fa holds, no production for the contextual symbol ${}^{lm}A^{rm}$ is created, thus $t \notin T(G')$.
- Finally, Equation 2.43 corresponds to the propagation of contextual tokens to the right-most symbol in Equations 2.66 and 2.67, whereas Equation 2.47 correspond to the propagation of contextual tokens to the left-most symbol in Equations 2.65 and 2.67. Since the contextual tokens are recursively propagated to the right-most and left-most positions of contextual productions, they are a direct implementation of right-most and left-most deep matching. That is, if $t \notin T^Q(G)$ because $\mathcal{M}^{rm}(t, q)$ or $\mathcal{M}^{lm}(t, q)$ according to a filter by subtree exclusion, then $t \notin T^(G')$ according to a contextual grammar. Note that Equation 2.68 ensures that all trees that *do not* (deeply) match a conflict pattern can be constructed by the contextual grammar. □

To see the equivalence of the trees produced by a grammar under subtree exclusion and a contextual grammar, consider the priorities in the grammar of Figure 2.37a, and contextual grammar of Figure 2.39. Typical matching is implemented by forbidding a contextual symbol to derive the productions in its set of contextual tokens. For example, a tree t in the original grammar G should be filtered because it matches a pattern `[Exp.Mul = [Exp.If = if Exp then Exp] * Exp]`) created by the priority rule `Exp.Mul > Exp.If`!. This tree is also filtered by the contextual grammar G' since the contextual token `If` is included in the contextual symbol representing the left operand of `Exp.Mul`, forbidding the corresponding subtree to be an if-expression.

Deep matching is implemented by propagating the contexts to the left-most (Equation 2.65), right-most (Equation 2.66), or left-most and right-most non-terminals (Equation 2.67) of the derived productions. For example, when constructing the production `Exp{If,IfE}.Add` for the contextual symbol `Exp{If,IfE}`, we pass the contextual tokens `If` and `IfE` to the right-most `Exp` non-terminal. Thus, if a tree $t \notin T^{Q}(G)$ such that $\mathcal{M}^{rm}(t,$ `[Exp.Add = [Exp.If = if Exp then Exp] + Exp]`$)$, the tree t cannot be produced by the contextual grammar. Finally, Equation 2.68 guarantees that any production that does not cause deep priority conflicts, i.e. is not recursive, is simply duplicated. This is the case for the production `Exp{If,IfE}.Var`.

Implementing disambiguation of deep priority conflicts as a canonical grammar transformation (even when considering only deep conflicts) comes with a major drawback. For grammars of expression-based languages with many deep priority conflicts, such as OCaml, the grammar can get about three times bigger [118]. To avoid the blow-up in productions caused by the grammar transformation above, we show a different approach to implement contextual filters based on data-dependent grammars in Section 2.9.4.

2.8.4 *Recursive Productions with Nullable Symbols*

Since recursive productions may cause priority conflicts, we also consider the case where the recursion is hidden by nullable symbols, using contextual grammars to *expose* productions that are indirectly recursive. For example consider the grammar in Figure 2.40a. The production `Exp.Add = OptLabel Exp "+" Exp` is still left recursive, but only when the non-terminal `OptLabel` derives the empty string. Therefore, to apply the same strategy and solve the conflicts that may occur due to nullable symbols, we can rewrite this grammar to enforce the cases where the non-terminal will *always* derive the empty string, i.e., we "inline" the productions for the optional symbol, creating two different productions. With this new grammar, we apply the same analysis described in the paper, detecting deep priority conflicts. The grammar in Figure 2.40b illustrates the post-processed contextual grammar in which leftmost recursion in the production `Exp.Add` is exposed, allowing us to solve the deep conflict involving the two addition productions that share the same suffix.

2.9 IMPLEMENTATION

In this section we discuss other implementations of our semantics for disambiguating expression grammars. We start by detailing how to detect expression grammars and how to check for harmful overlap. Next, we recall disambiguation of shallow conflicts, as implemented in SDF2. Finally, we present an implementation of disambiguation of deep priority conflicts using data-dependent parsing, which does not require calculating a contextual closure, resulting in a grammar with the same number of productions as the original grammar.

```
context-free syntax
  Exp.Add   = Label Exp "+" Exp {left}
  Exp.Int   = INT
  Label.NoL =
  Label.L   = ID ":"
```

(a)

```
context-free syntax
  Exp.Add   = Label{NoL} Exp "+" Exp
  Exp.Add   = Label{L} Exp{L} "+" Exp
  Exp.Int   = INT
  Label.NoL =
  Label.L   = ID ":"

  Label{L}.NoL =
  Label{NoL}.L  = ID ":"
  Exp{L}.Add    = OptLabel{L} Exp{L} "+" Exp{L}
  Exp{L}.Int    = INT
```

(b)

Figure 2.40 (a) SDF3 grammar containing an deep priority conflict hidden by a nullable symbol. (b) Rewritten contextual grammar that solves the conflict.

2.9.1 *Detecting Expression Grammars*

As mentioned in Section 2.2.8, the techniques described in this paper target the subset of context-free grammars that define expressions. To apply these techniques to traditional context-free grammars we define an algorithm that detects and captures embedded expression grammars. We use the shape of productions that define expressions to identify the productions that contribute to priority and associativity ambiguities. We construct the sets of productions that may cause associativity and priority ambiguities by looking at (possibly indirectly) recursive productions of a particular non-terminal. These sets can be constructed as follows.

Definition 2.9.1 (Expression Grammar Extraction)**.** *Given a context-free grammar G, the set of expression productions $EXP(G)(A)$ can be constructed as the smallest set of productions such that:*

$$\frac{A.C = \beta \quad \beta \overset{*}{\Rightarrow}_G \alpha\ A\ \gamma}{A.C \in EXP(G)(A)} \tag{2.69}$$

$$\frac{A.C = \alpha\ B\ \gamma \quad B \overset{*}{\Rightarrow}_G \alpha'\ A\ \gamma'}{EXP(G)(B) \subset EXP(G)(A)} \tag{2.70}$$

□

```
context-free syntax
  Stmt.Assign = ID "=" Expr
  Stmt.If     = "if" "(" Expr ")" Stmt
  Stmt.IfElse = "if" "(" Expr ")" Stmt "else" Stmt
  Call.Call   = ID "(" {Expr ","}+ ")"
  Expr.Mul    = Expr "*" Expr {left}
  Expr.Add    = Expr "+" Expr {left}
  Expr.Int    = INT
  Expr        = "(" Expr ")" {bracket}
context-free priorities
 Expr.Mul > Expr.Add, Stmt.IfElse > Stmt.If
```

Figure 2.41 Grammar containing embedded expression grammars.

Totality Check By extracting the productions of expression grammars, we are also able to check whether a set of disambiguation rules is total for that expression grammar, warning about missing rules that may lead to ambiguities.

For example, in the grammar in Figure 2.41, the productions `Expr.Mul` and `Expr.Add` define an embedded expression grammar, whereas the productions `Stmt.If` and `Stmt.IfElse` define a different expression grammar, with the non-terminal `Stmt`. We can then use both sets of productions to verify if the set of disambiguation rules in this grammar is total. For instance, the set of disambiguation rules for the first expression grammar is total. However, when considering the second expression grammar, we can warn the language developer that a priority rule between the productions `Stmt.If` and `Stmt.IfElse` is missing, and that the grammar is ambiguous.

Harmless Overlap Another benefit of extracting embedded expression grammars is using expression productions to detect whether an expression grammar contains harmful overlap. We implement this analysis as described in the pseudocode of Figure 2.42. We start by first isolating productions $p_1, ..., p_n$ that have *any* overlap at all. Then, we exhaustively construct trees (tree patterns) using these productions, checking if there are multiple trees with the same yield. If there exist two trees t_1 and t_2 with the same yield such that the productions used to construct t_1 ($prod(t_1)$) are different than the productions used to construct t_2 ($prod(t_2)$), we flag that the grammar contains harmful overlap. This analysis is performed on demand, limited by a particular depth $maxDepth$, since the number of trees to consider is infinite. In practice, the majority of grammars for existing programming languages only have harmless overlap. (Which is to be expected since harmful overlap causes inherent ambiguities.)

2.9.2 *Binding-Time of Disambiguations*

Disambiguation may occur at different stages when using a generalized parser. Figure 2.43 highlights the different stages in which disambiguation may occur using SDF2 in combination with a scannerless generalized LR parser (SGLR) [131]. First, the grammar is transformed to a normal form representing

```
function HARMFUL-OVERLAP(Set<Production> p_1,...,p_n, maxDepth)
  List<TreePattern> trees = the patterns for trees t_i with root p_i
  List<TreePattern> oldTrees = trees
  Integer depth = 1
  while (depth < maxDepth) do
    List<TreePattern> constructedTrees
    for each tree pattern t_i in oldTrees do
      constructedTrees += expand recursive non-terminal
                          tree pattern t_i with p_1,...,p_n
    end for
    if ∃ t_1,t_2 and yield(t_1) = yield(t_2) and prod(t_1) ≠ prod(t_2) then
      flag harmful overlap between prod(t_1) and prod(t_2)
    end if
    oldTrees = constructedTrees
    depth = depth + 1
  end while
end function
```

Figure 2.42 Pseudocode for the function to detect harmful overlap.

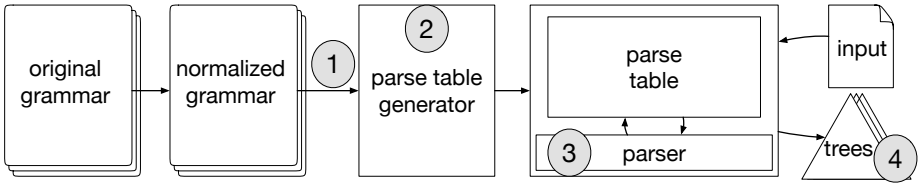


Figure 2.43 Parsing a program with SGLR and the times when disambiguation might occur.

a completely explicit character level grammar. This transformation explicates, among others, the distinction between lexical and context-free constructs, the definition of regular expression symbols (optionals and lists), and the treatment of layout (whitespace and comments). The normalized grammar is then used as input to a parse table generator, which produces a parse table that drives the parser when parsing an input program. We identify four different stages in this process in which disambiguation can be applied: (1) before parse table generation, (2) during parse table generation, (3) during parsing, and (4) after parsing.

The canonical interpretation of a disambiguation filter is as a post-parse filter [74]. For example, Erdweg et al. [45] define a post-parse implementation of longest match disambiguation for lists. A generalized parser produces (a compact representation of) all possible parse trees (a parse forest), from which a filter selects the correct disambiguation. However, in the case of ambiguities due to operator priority and associativity, the number of trees (when unpacking the parse forest) may grow exponentially with the number of sub-expressions in an expression.

```
context-free syntax
  Expr.Mul = Exp "*" Exp {left}
  Expr.Add = Exp "+" Exp {left}
  Expr.Var = ID
context-free priorities
  Exp.Mul > Exp.Add
```

Figure 2.44 Simple expression grammar.

A disambiguation filter provides a direct semantics of disambiguation that is independent of other implementation concerns. However, for many applications, in particular for the disambiguation of expression grammars, it does not provide a direct route to an implementation with low complexity (and good performance). Thus, we are interested in alternative binding times of disambiguation filters that ensure such characteristics. Concretely, for associativity and priority, the question is what the most effective binding time is. It would seem that earlier in the pipeline is always better, but that turns out not to be the case since grammar transformations may unnecessarily blow up the grammar. In the rest of this section we discuss several implementation techniques. In the next section we evaluate their efficacy.

2.9.3 *Disambiguation of Shallow Conflicts during Parser Generation*

In SDF2, the semantics in Definition 2.3.2 is implemented at stage 2, during parse table generation [126]. The states of an LR parse table consist of sets of items of the form `[A.C = α • β]` consisting of a production with a dot (•) somewhere on the right hand side, indicating how much of the production has been already processed by the parser. The *closure* of an item set adds all items that are predicted by an item with the dot before a non-terminal. For example, the closure of an item set with the item

```
[Exp.Mul = Exp "*" • Exp]
```

for the grammar in Figure 2.44 also contains the items

```
[Exp.Add = • Exp "+" Exp]
[Exp.Mul = • Exp "*" Exp]
[Exp.Var = • ID]
```

Disambiguation of LR tables is achieved by not predicting items that would cause a priority conflict. For example, the `left` annotation in the production `Exp.Mul` creates the pattern `[Exp.Mul = Exp * [Exp.Mul = Exp * Exp]]`, forbidding a `Exp.Mul` to occur as the rightmost child of another `Exp.Mul`. Thus when computing the closure of a state with the item `[Exp.Mul = Exp "*" • Exp]`, we do not create the item `[Exp.Mul = • Exp "*" Exp]`, forbidding a multiplication to be a direct rightmost child of another multiplication. Similarly, the priority rule `Exp.Mul > Exp.Add` creates the patterns `[Exp.Mul = [Exp.Add = Exp + Exp] * Exp]` and `[Exp.Mul = Exp * [Exp.Add = Exp + Exp]]`, forbidding any `Exp.Add` to be a direct child of

`Exp.Mul`. Thus, when computing the closure of `[Exp.Mul = Exp "*" • Exp]` (or of `[Exp.Mul = • Exp "*" Exp]`), we do not create the item `[Exp.Add = • Exp "+" Exp]`, implementing this semantics.

Filtering predictions is not sufficient to rule out all shallow priority conflicts. While a production may not be predicted by one item, it could still be predicted by another. To address these situations, the standard SLR table generation algorithm is adapted to compute the GOTO function with *productions* instead of non-terminals. For example, consider the item set consisting of the following productions.

```
[Exp.Add = • Exp "+" Exp]
[Exp.Mul = • Exp "*" Exp]
[Exp.Var = • ID]
```

A goto with the `Exp` non-terminal leads to the item set:

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

The standard goto does not take into account the priority of the reduction that gives rise to the goto. For example, a reduction of an addition expression would still give rise to a state containing the item `[Exp.Mul = Exp • "*" Exp]`, while that would clearly lead to a priority conflict. Thus, the SDF2 parser generation performs gotos with productions, taking conflict patterns into account. For example, the goto from the state above with production `Exp.Add` leads to the state

```
[Exp.Add = Exp • "+" Exp]
```

Since `[Exp.Mul = [Exp.Add = Exp + Exp] * Exp]` is a priority conflict.

This method can only be used to implement disambiguation of *shallow conflicts*, since it only forbids direct descendants of a tree to be a particular production. For SDF3, we have adapted this method to the safe semantics in Definition 2.4.1, using an approach based on data-dependent grammars to disambiguate deep priority conflicts.

2.9.4 *Data-Dependent Parsing*

Data-dependent grammars [63] extend context-free grammars allowing parameterized non-terminals, variable binding, and arbitrary computation at parse time. Purely data-dependent grammars are powerful enough to disambiguate priority conflicts at parse time [8]. However, using pure data-dependent grammars may result in a performance penalty in the generated parser. For that reason, we present a lightweight form of data-dependency, proposed by Souza Amorim, Steindorfer, and Visser [120], aimed at solving deep priority conflicts without requiring any variable bindings nor arbitrary computations at parse time, using a scannerless generalized LR parser (SGLR) [130] in combination with contextual grammars.

We leverage the analysis that constructs the contextual productions from Definition 2.8.2, but without duplicating the productions for contextual symbols

```
context-free syntax
  Exp.Add = Exp{Exp.If} "+" Exp {left}
  Exp.If  = "if" Exp "then" Exp
  Exp.Int = INT
  Exp     = "(" Exp ")" {bracket}
context-free priorities
  Exp.Add > Exp.If
```

Figure 2.45 Data-dependent Contextual grammar with addition and if expressions.

using the contextual closure from Definition 2.8.3. Thus, the final contextual grammar has the same shape and number of productions as the original grammar. The grammar itself is still ambiguous, but the contextual tokens present in the grammar can be used to solve deep priority conflicts at parse time. For example, consider the grammar in Figure 2.45, which consists of a data-dependent contextual grammar containing if and addition expressions.

To perform disambiguation at parse time, we propagate information about leftmost and rightmost subtrees, using this information to enforce the constraints imposed by a contextual symbol. As SGLR builds parse trees bottom-up, we propagate the information about productions that were used to construct the possibly nested leftmost and rightmost nodes of a tree. Each node of the parse tree of the adapted data-dependent SGLR parser contains two additional sets that indicate the productions used to construct its leftmost and rightmost (nested) subtrees, respectively. For every node, the set representing the leftmost contextual tokens is the union of the production used to construct the current node with the leftmost set of the leftmost direct child. Similarly, the set representing the rightmost contextual tokens is the union of the production used to construct the node itself with the rightmost set of contextual tokens of the rightmost direct child. Finally, we ensure that only productions that can cause deep priority conflicts are added to the sets of contextual tokens, since the number of tokens propagated is often significantly lower than the total number of productions. For example, the tree in Figure 2.46a for the sentence `a + if b then c` was parsed using the data-dependent contextual grammar of Figure 2.45.

To perform disambiguation at parse time, we propagate information about left-most and right-most subtrees, using this information to enforce the constraints imposed by a contextual symbol. As SGLR builds parse trees bottom-up, we propagate the information about productions that were used to construct the possibly nested leftmost and rightmost nodes of a tree. Each node of the parse tree of the adapted data-dependent SGLR parser contains two additional sets that indicate the productions used to construct its left-most and right-most (nested) subtrees, respectively. For every node, the set representing the left-most contextual tokens is the union of the production used to construct the current node with the left-most set of the left-most direct child. Similarly, the set representing the right-most contextual tokens is the union of the pro-

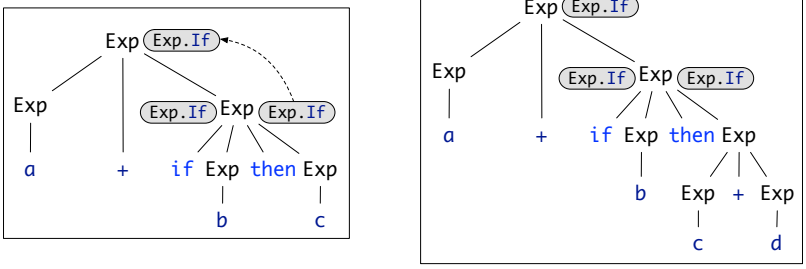


Figure 2.46 (a) Propagating contextual tokens when building parse trees. (b) Invalid tree constructed when parsing the sentence `a + if b then c + d`.

duction used to construct the node itself with the right-most set of contextual tokens of the right-most direct child. Finally, we ensure that only productions that can cause deep priority conflicts are added to the sets of contextual tokens, since the number of tokens propagated is often significantly lower than the total number of productions. For example, the tree in Figure 2.46a for the sentence `a + if b then c` was parsed using the data-dependent contextual grammar of Figure 2.45.

The algorithm for the data-dependent scannerless generalized LR parser requires only a few changes to the original SGLR algorithm of Visser [130]. The algorithm needs to propagate contextual tokens corresponding to the productions used to construct the left-most and right-most (possibly nested) subtrees (`t.LeftmostTokens` and `t.RighmostTokens`),[14] and to check the constraints when performing reduce actions. Figure 2.47 shows the pseudocode from Souza Amorim, Steindorfer, and Visser [120] corresponding to the modified methods of the original SGLR algorithm.

Using this algorithm, parsing the sentence `a + if b then c + d` produces only a single tree. When propagating the contextual tokens `Exp.If`, the right-most context of the tree corresponding to the left operand of an addition in Figure 2.46b is in the set of contextual tokens of the left operand in the production for addition ($\texttt{Exp}^{\texttt{Exp.If}}$), in data-dependent contextual grammar of Figure 2.45. Thus, the reduce action to construct this tree is not performed and this tree is rejected. Note that because we leverage contextual grammars, we are able to address all deep priority conflicts shown previously.

In Section 2.10, we show that an optimized implementation of this algorithm, using bitsets and bitwise operations produces near-zero parsing overhead while maintaining the size original grammar.

[14] In the original SGLR algorithm, creating a parse tree node consisted simply of applying a production to the trees collected when calculating the path for the reduce action. In the data-dependent algorithm, the sets of left-most and right-most subtrees need to be updated by propagating the information from the rightmost and leftmost direct subtrees.

```
1  function DO-REDUCTIONS(Stack st, Production A.C = X1...Xn)
2    for each path from stack st to stack st0 of length n} do
3      List<Tree> [t1,...,tn] = the trees from the links in the path from
       st to st0
4      for each Xi such that Xi is a contextual symbol lmXrm do
5         if ti.LeftmostTokens ∩ lm ≠ ∅ or ti.RightmostTokens ∩ rm ≠ ∅
       then
6           return
7         end if
8      end for
9      REDUCER(st0, goto(state(st0), A.C = X1...Xn), A.C = X1...Xn, [t1,...,tn])
10   end for
11 end function
```

```
1  function CREATE-TREE-NODE(Production A.C = X1...Xn, List<Tree>
       [t1,...,tn])
2    Tree t = [A.C = t1,...,tn]
3    t.LeftmostTokens = t1.LeftmostTokens ∪ A.C
4    t.RightmostTokens = tn.RightmostTokens ∪ A.C
5    return t
6  end function
```

Figure 2.47 Pseudocode for the modified *DO-REDUCTIONS* and *CREATE-TREE-NODE* methods from the original SGLR, in the implementation of the data-dependent SGLR.

2.10 EVALUATION

In this section we evaluate our approach to disambiguate associativity and priority conflicts. We have implemented the grammar transformation as part of a new parser generator for SDF3 and integrated it into the Spoofax Language Workbench [68]. The transformation produces a full contextual grammar based on the contextual symbols and contextual productions derived from the conflicting patterns presented earlier. The LR parse table generator has been adapted to the safe one-level rules of Definition 2.4.1. We also evaluate the performance of a data-dependent parser using contextual grammars without duplication of productions, comparing its performance to a parser that uses a grammar produced by the grammar transformation approach.

We have investigated how our approach scales to real-world languages. We have used SDF3 syntax definitions for five different languages: Tiger, a language for education purposes [15]; IceDust, a domain-specific language for data modeling [57]; Jasmin,[15] an assembly language for Java; Java 1.8 [55]; and OCaml 4.05 including some of its extensions.[16] All the specifications are in a repository on GitHub.[17]

[15] http://jasmin.sourceforge.net/about.html
[16] https://caml.inria.fr/pub/docs/manual-ocaml-4.05/
[17] https://github.com/MetaBorgCube/declarative-disamb

	Original Grammar	Normalized Grammar		
	Productions	Productions	States (Unsafe)	States (Safe)
Tiger	101	281	731	709
IceDust	155	401	1733	1864
Jasmin	388	1023	3174	3174
Java	479	1207	5138	5127
OCaml	648	1632	7121	6493

Table 2.1 For each language, the number of productions of the original grammar, and productions and states considering the normalized grammar, using the SDF2 semantics (unsafe) and SDF3 semantics (safe).

	Expression Grammar		Contextual Grammar		
	Quantity	Largest Size	Productions		States
			Deep Conflicts	Total	
Tiger	1	19	13	334	750
IceDust	1	32	29	712	2093
Jasmin	0	0	0	1023	3174
Java	2	53	49	1739	4624
OCaml	7	56	65	4282	42116

Table 2.2 For each language, the number of expression grammars, the size of the largest expression grammar, the number of productions with deep priority conflicts, the size of the contextual grammar, and the number of states of the parse table constructed using the contextual grammar.

We compare the unsafe SDF2 implementation with the new safe SDF3 implementation. For this purpose we use the exact same source grammars since the only difference is the *interpretation* of priority declarations.

Complexity Table 2.1 lists for each language the size of the source SDF3 grammar in number of productions, the size of the normalized grammar in number of productions, and the number of states produced by the parser generator (with safe and unsafe treatment of priorities). Table 2.2 shows the number of expression grammars in each language, the size of the largest expression grammar in number of productions, the number of productions causing deep conflicts, the number of productions of the transformed contextual grammar, and the number of states of the parse table (with safe treatment of priorities) for the contextual grammar. Some observations about the aspects that influence the complexity of our approach, based on these tables:

- Grammar normalization integrates lexical syntax and context-free syntax to create explicit character-level grammars. It generates productions for regular expressions, generates productions for literals, and injects layout between symbols of context-free productions. Normalization is orthogonal to the new treatment of priorities. To compare the unsafe

(SDF2) approach and the safe approach, the size of normalized grammar and the size of the parse tables should be compared.

- Most languages have a single embedded expression grammar. However, some languages have multiple embedded expression grammars, defining recursive productions using different non-terminals. OCaml for example, has many embedded expression grammars of sizes varying from 5 productions to 56 productions. Java, on the other hand, has only 2 expression grammars, one for expression productions and another one for statements (defining if and if-else statements).

- The increase in size of the contextual grammar depends on the number of productions with deep conflicts and the number of productions of the conflicting non-terminals. For a language without expressions such as Jasmin, our approach does not add any extra production. While the Java grammar is one of the largest in our benchmark, it does not have many productions for expressions, nor many low priority prefix/suffix operators. Therefore, only a few productions need to be duplicated when generating the contextual productions.

- Regarding number of states: Because the semantics of SDF2 is unsafe, in some cases it unnecessarily splits states, rejecting unambiguous sentences. Note that for Java, the number of states for the contextual grammar is *lower* than the table with a normalized grammar. Contextual symbols might contribute to decreasing the number of states, as some symbols should derive fewer productions, even when considering the additional contextual productions. Note also that in general the safe semantics produces fewer states than the unsafe SDF2 semantics, except for Java and IceDust. In those cases, many splits in the unsafe scenario may lead to already existing states. Note that the number of states generated for a data-dependent contextual grammar is the same generated by the normalized grammar considering the safe semantics, i.e., the transformation to generate a data-dependent contextual-grammar does not generate additional productions nor states. Nevertheless, data-dependent contextual grammars can still be used to address disambiguation of deep priority conflicts, whereas the parse table constructed from the normalized grammar using only the safe semantics cannot, since the safe semantics only addresses shallow conflicts.

- On the other hand, the size of the parse table *grows* as a function of the number of contextual productions and the number of related expression productions. Our parse table generator produces a large number of states for OCaml because of the high number of deep priority conflicts combined with the large size of the contextual grammar.

Correctness For all the languages in Table 2.1, we have written tests that cover syntactic features of the language, i.e., tests that exercise valid and invalid fragments for each non-terminal defined in the grammar. We have

also collected full programs, including programs that purposely stress deep priority conflicts. In total we have 112 tests for Tiger; 180 tests for Jasmin; 67 tests for IceDust; 10749 tests for Java; and 3431 tests for OCaml. We have run these test sets with parsers based on the SDF2 and new SDF3 approaches. The parsers agree on most test cases, but the SDF2 parser fails on programs with deep conflicts. It either filters too much, rejecting unambiguous programs due to unsafety, or filters too little, as the semantics is incomplete, producing ambiguous trees. In all tests involving valid programs, parsing the program using our safe semantics combined with disambiguation of deep conflicts using contextual grammars produced one single final tree. Finally, the Java and OCaml grammars have also been used in an experiment described by Souza Amorim, Steindorfer, and Visser [118] to measure the number of deep priority conflicts that occur in real programs (which are used in our test sets). In this experiment, real Java and OCaml programs were extracted from the top 10 trending repositories on GitHub, to provide a better understanding of how often deep conflicts occur in practice. The experiment showed that for OCaml, up to 17% of the source files used in the benchmark contain deep priority conflicts.

Usability Our approach automatically detects deep priority conflicts and correctly disambiguates the grammar based on the original (SDF2) priority declarations. For example, the grammar in the specification for Java 1.8 is unambiguous, but encodes priority and associativity by means of extra non-terminals, which makes it larger and harder to read. Rewriting the grammar to use priorities, results in a more concise grammar and adds a deep priority conflict involving lambda expressions. With SDF2 such a conflict would be detected only when running into a specific case that covers it. When using contextual grammars, the conflict is automatically solved using the same grammar and the same priority rules.

All the benchmark languages (except for Jasmin) support if and if-else expressions or statements. The Java Specification solves the problem by duplicating productions for the conflict and using an additional non-terminal *StatementNoShortIf*. Instead of rewriting the grammar manually, we capture this problem with a contextual production that is automatically generated based on the obvious priorities between productions.

Longest Match We compared our results with the solution for the longest-match implemented by Erdweg et al. [45] using post-parse disambiguation filters, which do not scale, since ambiguities can grow exponentially. We tested SGLR with the grammar in Figure 2.48 using the SDF2 and SDF3 parser generators and the test program $(\textit{alpha INT})^N$, where N is the number of repetitions of *alpha INT*. For $N = 17$, SGLR with a post-parse filter takes 51s to produce a single tree, whereas using a contextual grammar produced by grammar transformation, it takes 1ms. By solving ambiguities at the grammar level, the impact on parse time is the same as that of a deterministic parser.

Parsing Performance To measure the performance of a parser that disambiguates deep priority conflicts, we use the corpus of the top-10 trending

```
context-free syntax
  Term.Alpha = "alpha" Term+ {longest-match}
  Term.Int   = Int
```

Figure 2.48 SDF3 defining a longest-match list.

Language	Data Set	Disambiguation	Time (seconds)	Speedup	Cost
Java	with conflicts	data-dependent	0.18 ± 0.00	1.29 x	—
		rewriting	0.23 ± 0.00	1.00 x	—
Java	without conflicts	data-dependent	270.64 ± 1.28	1.73 x	1.02 x
		rewriting	467.20 ± 4.03	1.00 x	1.77 x
		none	264.20 ± 2.36	—	1.00 x
OCaml	with conflicts	data-dependent	80.60 ± 1.48	1.54 x	—
		rewriting	123.75 ± 1.02	1.00 x	—
OCaml	without conflicts	data-dependent	89.82 ± 0.51	1.46 x	1.01 x
		rewriting	130.71 ± 0.55	1.00 x	1.48 x
		none	88.58 ± 0.98	—	1.00 x

Table 2.3 Benchmark Results when parsing the OCaml and Java Corpus.

OCaml and Java projects on GitHub mentioned previously, containing 9935 Java and 3296 OCaml real-world source files. The corpus was qualitatively analyzed by Souza Amorim, Steindorfer, and Visser [118], listing the types of priority conflicts each file from the projects contains. For both languages we partitioned the files into two groups according to their analysis results: files that are free of deep priority conflicts (and therefore can be parsed by parsers without sophisticated disambiguation mechanisms), and files that contain deep priority conflicts. We then compare disambiguation by grammar transformation against data-dependent disambiguation, measuring the parse time of all programs from a particular group.

Column *Cost* shows how the parser's performance is affected by supporting the disambiguation of deep priority conflicts. The cost measurements were performed solely for the data sets that are guaranteed to be free of deep priority conflicts, since we use a parser without deep priority conflict disambiguation as a baseline. The results show that the cost of disambiguation with data-dependency is between 1 % (OCaml) and 2 % (Java). Column *Speedup* of Table 2.3 shows the performance improvements of data-dependent disambiguation over disambiguation via grammar rewriting (baseline). In all tested configurations, data-dependent disambiguation speeds-up from 1.29 x to 1.73 x, reducing batch parse times considerably. E.g., parse time for the conflict-free Java corpus reduced from 467.20 s to 270.64 s.

Threats to Validity Even though we extensively tested our grammars with code fragments and real programs, we have not compared the resulting abstract syntax trees with the abstract syntax tree produced by the "official" parser of each language to verify the correctness of our syntax definitions. However, in the tests and programs that contained ambiguities due to priority and associativity, we did observe the unsafety and incompleteness characteristics of the SDF2 semantics. Furthermore, for those tests we observed that our safe semantics combined with disambiguation of deep conflicts using contextual grammars can successfully solve ambiguities from shallow and deep conflicts.

2.11 RELATED WORK

Throughout the paper we have referred to work that inspired our approach for solving deep priority conflicts using contextual filtering. Here we highlight the differences with our work and discuss other approaches related to declarative disambiguation, such as ambiguity detection, disambiguation by grammar transformation, and post-parse disambiguation.

Post-Parse Disambiguation Filters Klint and Visser [74] present *disambiguation filters*, which select trees from a set of trees for a sentence, as a unifying approach to the formalization of disambiguation techniques for context-free grammars. Among the case studies, the paper formalizes priority declarations using one-level sub-tree exclusion patterns, which were later used in the design of SDF2 [127, 131]. The paper introduces higher-order tree patterns with an application to the definition of the dangling else ambiguity.

Priority Rule Recovery Bouwers, Bravenboer, and Visser [23] present an approach for recovering priority rules from grammars in which these are encoded in the grammar and for comparing the priority rules of different grammars for a language, possibly defined in different formalisms. To that end they derive sets of tree patterns from a grammar and establish which tree patterns it accepts. The information can be used to migrate grammars between different styles of disambiguation and/or between grammar formalisms. The approach does only consider fixed size tree patterns, and does not cover deep priority conflicts. It would be interesting to consider the extension of the approach to deep priority conflicts.

Disambiguation by Grammar Transformation In this paper, we have given a semantics of priority declarations as disambiguation filters on sets of parse trees, which we then interpreted as a grammar transformation. This fits in a longer tradition of approaches using grammar transformation to realize disambiguation of grammars.

Solving the dangling else ambiguity by transforming the grammar is as old as the last author. Abrahams [2] describes how to encode dangling else in a grammar for *Algol-like conditional statements*, by duplicating productions in order to distinguish open and closed conditional statements.

Thorup [123] proposes filters that capture deep conflicts such as dangling suffix, using labels to exclude infinite sets of forbidden sub-parse trees. The

approach constructs a grammar that encodes the subtrees that should be excluded, removing grammar symbols that correspond to such trees. However, it is not clear how to construct infinite sets of forbidden patterns considering other operators that may still contribute to a dangling suffix ambiguity (e.g., in the program `if` e_1 `then` e_2 `+ if` e_3 `then` e_4 `else` e_5).

Aasa [1] proposes grammar rewriting to solve priority and associativity conflicts. They handle deep priority conflicts involving prefix and postfix operators by creating different non-terminals that do not derive trees that could result in an ambiguity. However, the approach does not consider harmless overlap or indirect recursion, requiring that productions do not have overlapping prefixes or suffixes, and have a single non-terminal, meaning that it cannot be used to solve dangling prefix and dangling suffix conflicts, for example.

Afroozeh et al. [5] use grammar transformation to produce a disambiguated grammar from an ambiguous one. Motivated by SDF2 semantics, they propose a semantics that is safe. However, their semantics for disambiguation of ambiguities in expression grammars is still not complete, as it does not cover dangling prefix and suffix ambiguities, and nested list ambiguities. Furthermore, our implementation leverages SDF2 parse table generation techniques to solve one-level conflicts [128], avoiding unnecessary changes to the grammar, whereas Afroozeh et al. [5] also transform the grammar to solve one-level conflicts. Methodologically, Afroozeh et al. [5] use derivations to reason about ambiguities in expression grammars and define disambiguation as a grammar transformation. This tangles reasoning about subtree exclusion and grammar transformation, making it harder to understand. A contribution of this paper is to use tree patterns to characterize (deep) priority conflict patterns and thus provide a semantics for priority rules independent of a grammar transformation. The grammar transformation of Section 2.8.1 is just one possible implementation of our semantics.

While these techniques directly interpret associativity and priority declarations, Adams and Might [4] propose tree automata to express disambiguation policies. The intersection of a context-free grammar with a regular tree automaton yields a grammar that does (not) admit the tree patterns captured by the automaton. This approach applies to a variety of disambiguation problems as demonstrated in the paper. The technique is not safe and complete by construction. That is, a tree automaton may reject sentences that are not ambiguous and it is not guaranteed to address all ambiguities. Directly expressing associativity and priority rules as automata requires enumerating all combinations of operators, which requires a number of automata that is quadratic in the number of operators. It seems possible to use tree automata as an intermediate representation to compile priority rules to. The generic grammar intersection technique would then be used to drive the grammar transformation, instead of the specialized transformation from Section 2.8.1. The instantiation of each conflict pattern to the productions and priority rules of a grammar would give rise to an automaton. Soundness of the translation would rely on soundness of the individual automata and the generic composition.

Disambiguation during Parse Table Generation Interpreting disambiguation declarations as grammar transformations provides a principled definition of their semantics independent of particular parsing algorithms. That is, the resulting grammars can be used with arbitrary parsing techniques (providing the grammar class matches). Alternatively, priority rules can be interpreted during parser *generation*, to avoid an expensive grammar transformation step, or to make use of special properties of the parser generator.

An approach to eliminate ambiguities for LR parsers is to handle conflicts that occur in the parse table. Aho, Johnson, and Ullman [9] present an approach to eliminate shift/reduce conflicts based on the priority of operator tokens. The method is applied to resolve all conflicts in the parse table to make the parser deterministic. YACC's [64] disambiguation mechanisms are similar, and use the operator priority indicated in the grammar to define which action should be preferred when handling a shift/reduce conflict. The disadvantage of this approach is that it lacks a principled semantics such as those based on grammar transformations, which precludes its generalization to a larger class of grammars.

A disambiguation filter is applied *after* parsing, which makes it independent of particular parsing algorithms. Partial evaluation of the composition of parser generator, parser, and disambiguation filter can lead to optimized implementations of disambiguation. Visser [126] presents a case study of this idea by optimizing the composition of one-level subtree exclusion filters with the parsing schemata [111] for Earley and LR parsing to arrive at the application of disambiguation during parser generation. To realize disambiguation, goto transitions are applied per production instead of per non-terminal. Thus, priorities can be taken into account when computing the itemset of the target state; a goto with production `[Exp.Add = Exp + Exp]` will not include the result of shifting the item `[Exp.Mul = Exp * • Exp]` in the target item set if `Exp.Mul > Exp.Add`. This parser generation technique is not limited to LR grammars and generalizes to character-level grammars [130]. In this paper we adapt the technique by using *safe* one-level subtree exclusion patterns during parse table generation for a contextual grammar in which deep priority conflicts are resolved through grammar transformation.

Dynamic Disambiguation Instead of resolving ambiguities after parsing, as a grammar transformation, or during parser generation, ambiguities can also be resolved *during* parsing.

Parsing Expression Grammars (PEGs) [50] is a grammar formalism that deals with ambiguities by not allowing them in the first place, such that the productions for a non-terminal are tried according to the order they have been defined. If an alternative fails, the parser efficiently backtracks by memoizing intermediate results. However, the PEG approach is not a real solution, since it *hides* ambiguities behind an ordered choice. Furthermore, PEGs do not support left recursion, thus the grammar needs to be rewritten to eliminate left recursive rules. Laurent and Mens [83] present an implementation of parsing expression grammars that supports left recursion, and implements operator priority. Disambiguation is implemented via a global parse state that captures

the priority level of operators declaratively defined in the grammar. However, their specification does not address deep priority conflicts.

Afroozeh and Izmaylova [8] propose a dynamic solution to solve operator priority and associativity that relies on data-dependent grammars. Data-dependent grammars extend context-free grammars with computations, variable bindings and constraints that can be checked at parse time. A number indicating the priority level of a rule is assigned to each left or right recursive non-terminal, and passed around when parsing. A rule might fail to parse if it violates a constraint defined as a pre- or postcondition when parsing a symbol. Even though their approach does not require additional productions to solve most ambiguities, it relies on dynamically evaluated follow restrictions (parse-time lookahead) that bypass layout to solve dangling prefix and suffix, and nested list ambiguities.

Other Classes of Ambiguities and Ambiguity Detection Not all ambiguities in context-free grammars can be characterized in the same manner as ambiguities in expression grammars. Indeed, ambiguity of context-free grammars is an undecidable property. Several techniques have been proposed to detect ambiguities in context-free grammars, using conservative language approximations [19, 107, 24]. We focus on particular types of ambiguities in *expression grammars*. We are able to detect such ambiguities using total sets of disambiguation rules, i.e., by checking if priority rules are missing in the syntax definition.

A prominent class of ambiguities in programming language grammars is in grammars for layout sensitive languages. In such languages, the absence of punctuation such as end of statement delimiters (semicolon) causes ambiguities, which is disambiguated by making layout (newlines, indentation) significant. The approach to the presentation of such languages is similar to that for operator expressions. The language is defined through an ambiguous context-free grammar. A separate specification defines disambiguation. Examples of such approaches are the layout constraints of Erdweg et al. [45] and the indentation-sensitive grammars of Adams [3]. An important difference is that such disambiguation rules are often not safe in the sense used in this paper; some programs are rejected (e.g. if some expression is 'off-side') even though they can be parsed using the underlying context-free grammar.

2.12 CONCLUSION

Syntax definitions for programming languages should have a high-level semantics that is independent of particular parser implementations [69]. Specification of the associativity and priority rules separately from the productions of a grammar contributes to that goal. However, that requires a sound and complete implementation-independent formal semantics of such rules. Formulating such a semantics has been an elusive goal for three decades.

In this paper we have defined the *first direct* semantics of associativity and priority rules that is *safe* and *complete*. The semantics is *safe* since it does not

reject unambiguous parse trees (except where that is explicitly intended using the *non-assoc* directive).

The semantics is *complete* since it disambiguates all ambiguities in different types of expression grammars, including deep priority conflicts, dangling prefix and suffix ambiguities, and nested list ambiguities (assuming a total disambiguation relation). The semantics is *direct* since it defines disambiguation in terms of sub-tree exclusion patterns, independently of a particular parsing algorithm or grammar transformation.

While a complete disambiguation declaration requires specification of a disambiguation relation (roughly) between each pair of productions, the approach admits concise specification through priority chains that are transitively closed and declaration of groups of productions that are mutually (left or right) associative [58]. Furthermore, the approach supports grammar modularity by not requiring a linear ranking of productions. Since the semantics does not rely on lookahead symbols, it naturally extends to character-level grammars, which in turn support language composition [131, 31]. Our evaluation shows that our approach supports concise specification of disambiguation in real world grammars such as that of Java 1.8 and OCaml.

A (pilot) study of priority and associativity disambiguation in the grammars of real-world languages indicates that expression productions (and their combinations) requiring the extended soundness and completeness criteria are used in practice in programs [118]. A larger investigation is needed to establish whether programmers actually understand the disambiguation rules of the language they program in (or whether they trust that readers of their code understand those rules). For example, redundant (non-disambiguating) use of parentheses may indicate this is not the case.

In this paper we have provided an implementation of the declarative semantics as a *grammar transformation* producing unambiguous context-free grammars using contextual labels to distinguish the roles of symbols in (duplicated) productions without affecting the underlying tree structure. While this demonstrates that the associativity and priority rules provide a conservative extension of context-free grammars, the size of the grammar (and the size of the resulting parse tables) may grow significantly. (This is also observed in other approaches relying on grammar transformation for expression grammar disambiguation.) For example, the grammar for OCaml, which combines a large expression sub-grammar with a large number of deep priority conflicts, produces a 3x increase of (the states of) the parse table. We also show that contextual symbols can be interpreted using a light-weight data-dependent extension of the Generalized-LR parsing algorithm (propagating contextual symbols bottom-up) with near-zero overhead without requiring the generation of productions defining contextual non-terminals [120]. Thus, demonstrating that adopting the semantics defined in this paper does not have to come at the expense of increased parser generation time or parse time.

In the future we would like to mechanize the proofs shown in this paper with the help of an automatic theorem prover (e.g., Coq [18]), and also investigate a data-dependent approach to disambiguation of shallow priority conflicts.

Deep Priority Conflicts in the Wild: A Pilot Study

3

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 chapter 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.

3.1 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"* [62]. 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 [69].

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 [54] 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 [55] 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 [64] or SDF2 [131] 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 completeness, recent proposals rely either on grammar-to-grammar transformation techniques [5, 116], or they rely on data-dependent formalisms [8] that deflect performance overhead to run-time. On one hand, grammar-to-grammar transformations have the advantage to output (well-researched) pure context-free grammars. On the other hand, they blow up the resulting grammar by extensively duplicating productions,[1] which may result in large LR parse tables.

In this chapter, we aim to investigate the efficiency and usefulness of grammar-to-grammar transformations for solving (deep) priority conflicts. We address the *efficiency* issue by inspecting how much of the resulting grammars are used and respectively, how much of their parse tables are exercised. To reason about *usefulness*, we investigate to what extent deep priority conflicts occur in real code and whether conflicts are explicitly disambiguated, looking into declarative disambiguation from the programmers' perspective. In particular we empirically address the following research questions concerning coding practices:

RQ1 To what extent do deep priority conflicts in declarative language specifications occur in real-world programs?

RQ2 How do deep priority conflicts impact the efficiency of declarative disambiguation techniques that rely on grammar transformations?

RQ3 To what extent do programmers use brackets for disambiguation of priority conflicts explicitly?

We study the aforementioned research questions for declarative context-free grammars of two programming languages — OCaml and Java — that inherently feature deep priority conflicts. In a pilot study, we empirically examine the top 10 trending open-source projects of each language on GitHub.

[1] Grammar transformations create copies of original productions, modifying only specific non-terminals.

Contributions We performed an empirical pilot study that investigates deep priority conflicts *"in the wild"*. In particular:

- We contribute a research method for measuring deep priority conflicts and for obtaining coverage metrics.
- We provide initial results on the frequency and circumstances under which deep priority conflicts occur.
- We present insights about explicit disambiguation of deep and shallow priority conflicts using brackets.

With our pilot study, we provide an indication on how much deep priority conflicts are an issue when parsing real-world code. We investigate the causes of deep priority conflicts, and provide guidance for subsequent studies.

The remainder of this chapter is organized as follows. In Section 3.2 we provide background on (deep) priority conflicts and declarative disambiguation. Section 3.3 develops the research method necessary to empirically reason about deep priority conflicts. Section 3.4 presents the results of our empirical pilot study. We discuss threats to validity in Section 3.5. Finally, we present related work in Section 3.6, before concluding.

3.2 A PRIMER ON DECLARATIVE DISAMBIGUATION

Safe and complete disambiguation is a precondition for precisely reasoning about deep priority conflicts. Thus, we discuss the necessary background on the nature of (deep) priority conflicts, declarative disambiguation of such conflicts, and explain associated safety and completeness properties.

What is a priority conflict? Context-free grammars allow to declaratively and concisely define the syntax of a programming language. E.g., the following example defines productions rules for an expression grammar supporting integer literals, addition and multiplication:

```
Exp.Add = Exp "+" Exp
Exp.Mul = Exp "*" Exp
Exp.Int = INT
```

Although this grammar describes the basic syntax of arithmetic expressions properly, it fails to mention that multiplication binds stronger than addition, or that such operators are left associative. As a result, the input string `1 + 2 * 3` is ambiguous, because it could be parsed or interpreted as either `(1 + 2) * 3` or `1 + (2 * 3)`. Generalized parsers [124, 131, 6] typically derive all possible derivations and capture them in so-called ambiguity nodes:

```
AmbiguityList(
  Mul(Add(Int("1"), Int("2")), Int("3")),
  Add(Int("1"), Mul(Int("2"), Int("3")))
)
```

The ambiguity node represents a variable-length list of alternative interpretations, in our case for the string `1 + 2 * 3`.

How to declaratively disambiguate priority conflicts? In order to designate unambiguous parse interpretations, reference manuals of programming languages traditionally describe precedence and associativity relationships among a language's operators in supplementary tables. Syntax definition formalisms translate these tables into declarative constructs for determining the correct parse when combining such operators [64, 58, 131]. More recent context-free grammar formalisms directly integrate precedence and associativity using associativity attributes and priority relations, such as:

```
Exp.Mul = Exp "+" Exp {left}
Exp.Mul = Exp "*" Exp {left}

Exp.Mul > Exp.Add
```

These declarations, written in SDF3 [133] syntax, specify that addition and multiplication are left associative, and that multiplication binds stronger than addition. Technically, associativity attributes and priority relations define patterns used by a parse tree filter [74] to prohibit conflicts to occur. The filter defined using the priority and the precedence attribute in the example prohibits an addition to occur as a direct child of a multiplication, or additions (multiplications) to occur as a direct rightmost child of another additions (multiplications), respectively. Even though this approach is enough to support the operator precedence and associativity of many programming languages, it is not *safe* nor *complete* [5].

What is safe and complete disambiguation? Ideally, a parser is assumed to deterministically produce exactly one valid parse tree for any valid input string of a language. When considering concise but ambiguous grammars, declarative disambiguation shifts the responsibility of resolving ambiguities due to operator precedence and associativity from the language engineer to the parser generator. *Safe disambiguation* denotes that valid inputs strings are not rejected by the parser, i.e., if an input string belongs to the language covered by the grammar, then the parser should produce at least one tree. *Complete disambiguation* states that the declarative priority relations specified together with the grammar can disambiguate all combinations of operators, i.e., for any input constructed combining the operators from the grammar, at most one tree is produced.

What is a deep priority conflict? Most priority conflicts can be solved by looking at the direct expansions of the symbols within a production, ruling out trees containing invalid patterns. Such conflicts will henceforth be called *shallow* priority conflicts. The previous example, of multiplication binding stronger than addition, is such a shallow conflict, as the priority relation states that an addition cannot be a direct child of a multiplication. In contrast, conflicts that cannot be solved via parent-child relations of productions are henceforth referred to as *deep* priority conflicts. Deep priority conflicts can occur arbitrarily nested due to indirections (e.g., intermediate productions) that hide directly

conflicting productions. In the following, we will discuss the three types of deep priority conflicts by example.[2]

Deep Priority Conflict #1: Operator-Style. To illustrate deep conflicts with operator precedence,[3] we add if-else-expressions to our example expression grammar:

```
Exp.IfElse = "if" Exp "then" Exp "else" Exp

Exp.Add > Exp.IfElse
```

The declarative disambiguation rule on the last line specifies that addition binds stronger than if-else-expressions. Yet, parsing the string `1 + if e then 2 else 3 + 4` could produce two different interpretations:

```
1 + if e then 2 else (3 + 4)
(1 + if e then 2 else 3) + 4
```

If we consider disambiguation of shallow conflicts, the priority declaration `Exp.Add > Exp.IfElse` states that an if-else-expression cannot occur as a direct child of an addition. Safe disambiguation guarantees that an if-else-expression can still occur as the right child of an addition, i.e., the string `1 + if e then 2 else 3` should not be rejected as it is unambiguously accepted by the grammar. In general, operator-style conflicts may occur whenever nesting prefix operators and post-fix operators[4] of different precedences, and their precedence cannot be checked with a parent-child relation due to indirections. In our example, a deep conflict occurs because the if-else-expression can still occur as left child of the addition, hidden by another addition. When writing `Exp.Add > Exp.IfElse`, one would like to indicate that any addition to the right of the if-else-expression should always have higher precedence. That is, the correct interpretation should be only the first one: `1 + if e then 2 else (3 + 4)`.

Deep Priority Conflict #2: Dangling Else. To illustrate dangling-else conflicts, we add if-expressions without else-branches to our running example, the expression grammar:[5]

[2] The formalization of deep priority conflicts in Chapter 2 includes a description of the symmetric versions of the conflicts presented in this chapter.

[3] The notion of *operators* has been extended to sentential forms in recursive productions. E.g., `"if" Exp "then" Exp "else"` is a prefix operator in a production `Exp.IfElse = "if" Exp "then" Exp "else" Exp`.

[4] Infix operators are considered both as prefix, and post-fix. A deep priority conflict does not occur between two infix operators, since in this case, the conflict can be disambiguated with filters based on one-level tree patterns.

[5] The if-else-expression was copied from the previous listing to emphasize that both if-variants share the same prefix.

```
Exp.If     = "if" Exp "then" Exp
Exp.IfElse = "if" Exp "then" Exp "else" Exp

Exp.IfElse > Exp.If
```

The dangling-else conflict, which is present in grammars of many programming languages, arises when an if-expression is nested inside an if-clause of another if-else-expression, such as in the string `if e1 then if e2 then 3 else 4`. This input string results in two possible parses:

```
if e1 then (if e2 then 3 else 4)
if e1 then (if e2 then 3) else 4
```

The else-branch could be either connected to the first or the second if-expression. The root cause of the dangling-else conflict is that two productions of the same non-terminal share a common prefix, with the smaller production being right-recursive. The disambiguation rule `Exp.IfElse > Exp.If` indicates that an else-branch must be connected to the closest if-expression (cf. first interpretation). Note that even though the ambiguity above could be solved as a shallow conflict, dangling else conflicts are also deep conflicts, as the inner if-expression could be nested inside another expression.

Deep Priority Conflict #3: Longest Match. Longest match conflicts are caused when nesting lists of the same symbols inside each other. For example, consider the built-in match-expression of a language such as OCaml that has the following form:

```
Exp.Match    = "match" Exp "with" Pattern+
Pattern.Case = "|" Pattern "->" Exp
```

An ambiguity arises when a case-clause of a match-expression has an inner match-expression with multiple case-clauses. In that situation, the parser cannot distinguish to which expression the subsequent case-clauses are connected to:

```
match value with
| pattern -> result
| pattern -> match value with
             | pattern -> result
             | pattern -> result


match value with
| pattern -> result
| pattern -> match value with
             | pattern -> result
| pattern -> result
```

The standard disambiguation of this conflict consists of expanding the list of case-clauses of the inner match-expression as much as possible, i.e., producing the longest match. Thus, in the previous example the first interpretation is correct.

3.3 REASONING ABOUT DEEP PRIORITY CONFLICTS

In the previous section, we presented examples of three common types of (deep) priority conflicts that may arise in declarative context-free grammar specifications. This section incrementally introduces a method that we subsequently use to measure and analyze declarative disambiguation of deep priority conflicts in practice.

Although declarative disambiguation is widely used in syntax definition formalisms such as SDF, only recently, shortcomings concerning safe and complete disambiguation were reported [5]. That raises the question, why those shortcomings remained undetected for more than a decade? If and to what extent are deep priority conflicts indeed an issue in real-world grammars and programs? Not much is known about deep priority conflicts *in the wild*. In Section 3.3.1 we will discuss *contextual grammars* to quantify RQ1: *To what extent do deep priority conflicts in declarative language specifications occur in real-world programs?*

One may question the usefulness of declarative disambiguation techniques, as no research has been performed into deep priority conflicts in real-world settings. In particular, solving deep priority conflicts with grammar transformations has a cost attached, potentially resulting in large context-free grammars with many productions that are not used in practice. In Section 3.3.2 we will discuss *lazy parse table generation* to quantify RQ2: *How do deep priority conflicts impact the efficiency of declarative disambiguation techniques that rely on grammar transformations?*

Aside from the technical limitation, deep priority conflicts that are inherent to grammars of programming languages may impact common programming practice. One may ask if programmers need to be aware of the notion of deep priority conflicts, and if programmers are exposed to limitation of the disambiguation techniques respectively? Programmers can usually fall back to explicit disambiguation with brackets, in case the precedence rules are not clear or the parser is unable to parse an input string due to ambiguities. In Section 3.3.3 we will discuss a method for detecting explicit disambiguation to quantify RQ3: *To what extent do programmers use brackets for disambiguation of priority conflicts explicitly?*

3.3.1 *Contextual Grammars*

As mentioned in Section 3.2, generalized parsers produce a parse forest containing ambiguity nodes corresponding to the possible interpretations of a program. Ideally, a filter should be able to select only one correct interpretation and return it as result. Filtering ambiguities that arise from operator precedence and associativity after parsing is not practical though, as the number of ambiguities can grow exponentially with the number of operators in an expression. Thus, priority conflicts should preferably be solved either at parser generation time or at parse time.

To solve deep priority conflicts, we use a technique based on *contextual grammars*. This approach consists of a grammar transformation that generates additional productions forbidding *deep* conflicting patterns before parser generation. Furthermore, this technique enables precise measurements of the number of deep conflicts by only duplicating the productions that contribute to solving a certain conflict. For example, the contextual grammar to solve the operator-style conflict described in Section 3.2, has the following form:

```
Exp.Add = Exp^{IfElse} "+" Exp
Exp.IfElse = "if" Exp "then" Exp "else" Exp
Exp.Int = INT

Exp^{IfElse}.Add = Exp^{IfElse} "+" Exp^{IfElse}
Exp^{IfElse}.Int = INT
```

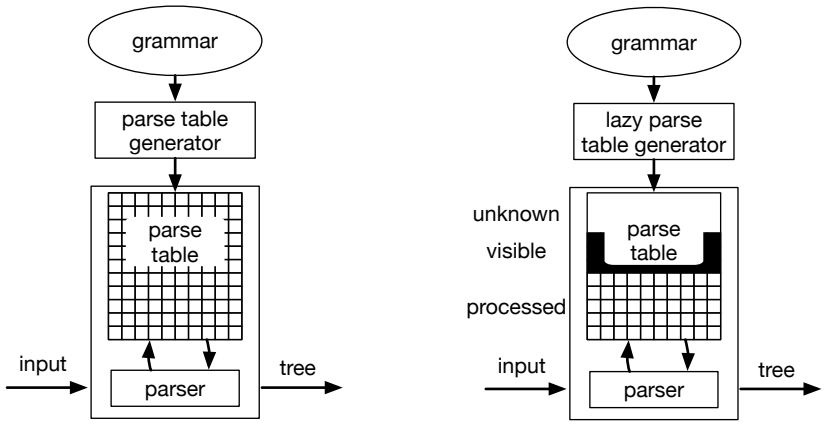
The contextual symbol $\texttt{Exp}^{\{\texttt{IfElse}\}}$ indicates that any expression derived by this symbol cannot have an if-else-expression as its rightmost child. This semantics is implemented when duplicating the productions of the non-terminal `Exp`, passing the context `IfElse` to the respective rightmost symbols and forbidding the contextual symbol $\texttt{Exp}^{\{\texttt{IfElse}\}}$ to derive an `IfElse` production itself.

To count the number of deep priority conflicts of a specific type, we use a contextual grammar that solves all-but-one type of priority conflict. For example, to measure the number of operator style conflicts, we use a contextual grammar $\mathbf{G}_{\{\mathbf{DE},\mathbf{LM}\}}$ as a contextual grammar $\mathbf{G}$ that solves dangling else and longest match conflicts, i.e., $\mathbf{G}_{\{\mathbf{DE},\mathbf{LM}\}}$ does *not* contain the additional productions to solve operator-style conflicts. Thus, parsing programs using this grammar produces an ambiguity whenever this program contains an operator-style conflict. Similarly, the contextual grammars $\mathbf{G}_{\{\mathbf{OS},\mathbf{LM}\}}$, which solves only operator-style and longest match conflicts, and $\mathbf{G}_{\{\mathbf{OS},\mathbf{DE}\}}$, which solves only operator-style and dangling else conflicts, can be used to detect dangling else and longest match conflicts, respectively. To guarantee that all ambiguities that arise in a program are related to deep priority conflicts, we use a contextual grammar $\mathbf{G}_{\{\mathbf{OS},\mathbf{DE},\mathbf{LM}\}}$that solves all conflicts, verifying that the same program parses unambiguously.

3.3.2 *Lazy Parse Table Generation*

Transformation to contextual grammars can produce large grammars when considering languages containing a large number of deep priority conflicts and many different productions that refer to conflicting symbols. We derive contextual grammars from SDF3 [133] syntax definitions, using them in combination with a scannerless generalized LR parser (SGLR) [131]. In this case, the number of states in the generated parse tables can also grow considerably, because states are split to handle each possible interpretation of a conflict.

Since we suspect that a considerable portion of states generated from contextual productions is not used in practice, we adopted *lazy parse table generation* [59]. With that technique, the parser generates parse states on demand



(a) Conventional Table Generator (b) Lazy Table Generator

Figure 3.1 Table-driven parser with a conventional and a lazy parse table generator, as presented in [59].

and as a result, only those states actually needed for parsing a program or a series of programs are generated. In addition to improving the performance of parse table generation, this technique provides an alternative to measure parse table coverage of a program or corpus of programs.

A common scenario for most table-driven parsers is described in Figure 3.1a. A (complete) parse table is generated from the grammar, and a generic parser reads the actions in the parse table to process the input. The table stays the same as long as the grammar has no changes, but whenever compiling a modified grammar, the table generator produces a new full parse table containing all processed states. That is, for larger grammars with many states, generating and loading a large parse table in which only a few states are used, is inefficient.

In a lazy parser generation scenario, whenever the parser requests a state, a lazy generator either processes a new state or returns an already processed state to the parser. Processing a state may create actions that refer to unprocessed states, making them visible. The processed and visible states from all previous parses are cached until the grammar has been changed, and the subsequent parses of the same program do not have any impact on parser generation time. Thus, if most of the programs do not exercise the full grammar, the parser can be regenerated without a big penalty in performance as parser generation time is amortized over parsing many input programs. Figure 3.1b illustrates the scenario of a table-driven parser in combination with a lazy table generator. Note that all states that are not processed or visited remain unknown.

Applying the conventional SDF3 parser generator and a lazy parser generator to the contextual grammar presented before (replacing `Exp`$^{\texttt{\{IfElse\}}}$ by `Exp1`) produces a parse table defined by the automaton of Figure 3.2. The complete automaton is generated by the conventional parse table generator,

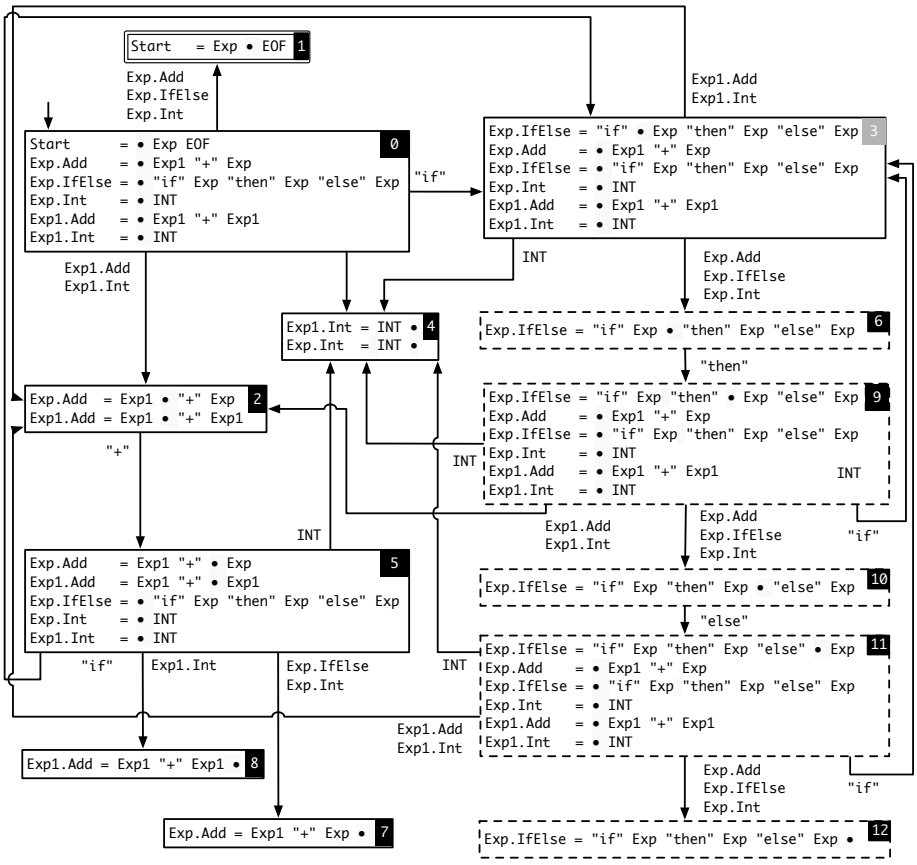


Figure 3.2 States generated by the conventional and lazy table generation algorithms when applied to the contextual grammar of Section 3.3.1. The states with striped lines are the ones still unknown by the lazy generator after parsing $1 + 2$. Note that state 3 is visible but has not been processed yet.

whereas only the highlighted states are processed when using the lazy table generator to parse the program $1 + 2$.

We measure the coverage of contextual grammars by the number of all states visible and processed in the parse tables generated by our lazy generator. We use two different tables: a fresh parse table to parse each separate program and a different table that accumulates the number of (visible and processed) states when parsing all the programs for each corpus. We also measure the number of productions that have been used considering both scenarios.

3.3.3 *Explicit Disambiguation*

Programmers might use brackets to explicitly specify the precedence of operators when writing a program. For example, when writing a program `(1 + 2) * 3`, a programmer uses brackets to explicitly state that in this expression the addition should have higher precedence over the multiplication. However, writing a program `1 + (2 * 3)`, does not change the actual precedence of the operators, since the multiplication already has higher precedence over the addition. In the last case, the brackets are redundant as they do not change the shape of the resulting abstract syntax tree (AST).

Programmers might also use brackets to disambiguate deep priority conflicts explicitly. For example, the brackets in the following two expressions result in ASTs that would be forbidden by the original contextual grammar, if we consider the same expressions without brackets:

```
(1 + if e then 2 else 3) + 4
1 + (if e then 2 else 3) + 4
```

In both expressions, the brackets specify a different operator precedence from the one defined in the grammar. To measure the number of brackets used for explicit disambiguation, we investigate the unambiguous ASTs produced by the contextual grammar $\mathbf{G}_{\{\mathbf{OS},\mathbf{DE},\mathbf{LM}\}}$. Because brackets do not appear explicitly in the AST, we added an attribute to AST nodes to indicate whether a node is surrounded by brackets.

First, we collect all nodes that have a bracket attribute, calculating the total number of the (pairs of) brackets present in the program. Then, we navigate through the program's AST searching for conflicting patterns, as such patterns are forbidden by the grammar and can only occur inside brackets. We remove the bracket nodes found this way from the initial list, counting the ones that disambiguate deep conflicts and the ones that disambiguate shallow conflicts. The remaining bracket nodes are marked as redundant.

Note that brackets can disambiguate a shallow and a deep conflict at the same time, as illustrated in the example below:

```
1 + (2 + if e then 2 else 3) + 4
```

The brackets are used to disambiguate a deep conflict involving an addition and an if-else-expression, and a shallow conflict which states that the addition inside the brackets should be right associative with respect to the outer one. In cases where brackets disambiguate deep and shallow conflicts, we opted to consider the brackets used in these cases to disambiguate only a deep priority conflict.

3.4 EVALUATION

In the previous section, we devised measurement techniques that enable empirical investigation into how deep priority conflicts occur in practice. In this section, we are applying our method to answer to what extent deep priority conflicts do actually occur in real programs. This pilot study specifically focuses

on the syntax of two programming languages — OCaml and Java — that have inherently different attributes.

The OCaml syntax is for the most part expression-oriented, and a large number of deep priority conflicts originate from the expression part. We have used the grammar from the OCaml reference manual, which contains all three types of deep priority conflicts that were discussed in Section 3.2: operator-style, dangling else and longest match.

In contrast to OCaml, Java is a predominantly statement-oriented programming language. Dangling-else conflicts may apply to if-statements, while expressions are the main subject to operator-style priority conflicts. The Java grammar does not contain longest-match constructs.

Based on the inherently different syntaxes of the two languages, we present our hypotheses of the expected results, grouped according to the research questions.

Hypotheses for RQ1: The first research question is related to the number of conflicts that occur in real programs:

H1 We expect more ambiguities triggered by the expression-oriented grammar of OCaml than by Java's grammar.

H2 The majority of OCaml deep conflicts are longest-match, because many expressions can have pattern matches.

H3 Deep priority conflicts are expected to be sparse in Java programs, as most priority conflicts are shallow.

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

Hypothesis for RQ2: The second research question considers the efficiency of grammar transformation approaches to solve deep priority conflicts. Our hypothesis is based on the coverage of contextual grammar productions (and parse table states respectively) that are used to solve deep conflicts.

H5 For both languages we expect that only a minor part of grammar productions and parse table states is exercised, even after parsing all programs in the corpus.

Hypotheses for RQ3: The third research question is concerned with explicit disambiguation. Our expectations are:

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

H7 The majority of the brackets in Java and OCaml are necessary for disambiguating shallow priority conflicts.

3.4.1 *Experimental Setup*

We directly transcribed the declarative context-free grammar of the OCaml version 4.04 reference manual[6] to SDF3. The natural and ambiguous OCaml grammar contains 1793 productions.[7] The original Java SE 8 reference grammar[8] encodes conflict resolution in the grammar itself. To make deep priority conflicts of Java detectable with our method, we have replaced the syntax for expressions by a natural (and ambiguous) syntax, defining operator precedence and associativity by means of SDF3 priorities. The resulting Java grammar contains 1327 productions.

In our pilot study we examine the top 10 trending open-source projects on GitHub for each language.[9] Two out of the top 10 OCaml projects were misclassified by GitHub, i.e., not containing any OCaml file at all. We removed the misclassified projects and added the subsequent projects from the list. Furthermore, we had to clean one project in order to avoid data duplication. The *bucklescript* repository duplicated the whole *ocaml* project into a subfolder. We removed the subfolder from *bucklescript*, because the *ocaml* project itself is part of our pilot study corpus of OCaml projects.

3.4.2 *Results of the OCaml Case Study*

Our pilot study corpus contains 3296 OCaml source files, from which 95.9% (i.e., 3161 files) were successfully parsed with our grammar while 4.1% (i.e., 135 files) could not be parsed due to language extensions that we do not support.[10]

Tables 3.1 and 3.2 summarize our findings with respect to occurrences of deep priority conflicts and bracket usage considering each project in the OCaml corpus. Table 3.1 presents the number of affected files of each project, the number of deep priority conflicts found and how frequent each type occurs. Table 3.2 shows information about brackets usage, highlighting the number of brackets that have been used for disambiguation in each project. Note that the remaining percentage of brackets for each project corresponds to redundant brackets. In the following paragraphs we discuss data points from the aforementioned table.

Affected Files 530 OCaml files (i.e., 16.8% of all files) contained deep priority conflicts. Concerning the individual categories, the most frequent priority conflict type was longest match (in 356 files), followed by operator-style conflicts (in 278 files) and dangling-else conflicts (in 7 files).

When looking at combinations of categories, our data shows that 79.6% of files with deep priority conflicts contained conflicts of just a single category, whereas 19.8% mixed two conflict categories, and 0.6% contained three categories.

[6] `http://caml.inria.fr/pub/docs/manual-ocaml/language.html`

[7] We consider the number of productions after SDF3 normalization.

[8] `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`

[9] `https://github.com/trending/` accessed on May 19, 2017.

[10] The grammar currently does not support some language extensions defined in `http://caml.inria.fr/pub/docs/manual-ocaml/extn.html`.

Project	Affected Files	Deep Priority Conflicts			
		Total	Operator Style	Dangling Else	Longest Match
FStar	6 / 160 (3.8%)	6	33.3%	0.0%	66.7%
bincat	5 / 26 (19.2%)	26	57.7%	0.0%	42.3%
bucklescript	85 / 885 (9.6%)	305	50.2%	1.3%	48.5%
coq	158 / 417 (37.9%)	441	35.4%	0.5%	64.2%
flow	52 / 305 (17.0%)	117	36.8%	0.0%	63.2%
infer	33 / 234 (14.1%)	52	23.1%	0.0%	76.9%
ocaml	112 / 909 (12.3%)	275	28.4%	0.7%	70.9%
reason	4 / 36 (11.1%)	14	7.1%	0.0%	92.9%
spec	4 / 40 (10.0%)	5	100.0%	0.0%	0.0%
tezos	71 / 149 (47.7%)	416	79.3%	0.0%	20.7%
All	530 / 3161 (16.8%)	1657	48.0%	0.5%	51.5%

Table 3.1 Overview of Deep Priority Conflicts in OCaml Corpus.

Project	Disambiguation with Brackets	
	Deep Conflicts	Shallow Conflicts
FStar	607 (1.7%)	12487 (35.7%)
bincat	28 (0.8%)	2735 (81.3%)
bucklescript	924 (2.1%)	29238 (65.3%)
coq	1039 (1.3%)	56083 (69.9%)
flow	278 (1.6%)	13374 (78.9%)
infer	376 (2.5%)	10720 (70.6%)
ocaml	737 (1.6%)	35010 (77.4%)
reason	25 (1.5%)	1194 (73.2%)
spec	15 (1.0%)	1293 (86.1%)
tezos	130 (1.6%)	6969 (84.3%)
All	4159 (1.6%)	169103 (67.1%)

Table 3.2 Overview of Bracket Usage in OCaml Corpus.

Deep Priority Conflicts In total, we discovered 1657 deep priority conflicts in all files. From these, 48% are operator-style conflicts, 51.5% are longest match and only 0.5% are dangling else. When looking at the number of conflicts per project, six projects contained more longest match conflicts, whereas four projects contained a majority of operator-style conflicts. Three out of ten projects had dangling else conflicts. On a per file basis, the maximum number of conflicts observed were 38 operator-style conflicts, 21 longest match conflicts, and 2 dangling-else conflicts.

Disambiguation with Brackets We observed a total number of 248830 pairs of brackets. From these, 31.3% are redundant, i.e., removing them does not affect the resulting tree. The remaining 68.7% of the brackets account for resolving priority conflicts (67.1% shallow and 1.6% deep conflicts).

Discussion When looking at the files with most conflicts, we found that the most common patterns of operator-style conflicts have the following form:

```
exp1 op fun param -> exp2 op2 exp3
exp1 op function pattern -> exp2 op2 exp3
```

In most cases, `op` and `op2` are user-defined operators such as `>>?` and `>>=?`, whereas `fun` and `function` are function definitions in the OCaml language.

For most dangling else conflicts, we noticed that the problematic `if` was hidden by an inner `let` expression, such as:

```
if exp1 then let binding1 in
             if exp2 then exp3
             else exp4
```

One interesting remark is that all dangling else conflicts were indented in a way that is consistent with how the conflict is solved by the contextual grammar.

Longest match conflicts did not follow a unique form. However, occasionally pattern-matching constructs that contain `match`, `function`, or `try` expressions caused conflicts of the following form:

```
begin function (e, info) -> match e with
  | pattern1
  | pattern2
end
```

In the example above, the indentation does not clarify whether `pattern2` belongs to `function (e, info)` or `match e`.

3.4.3 *Results of the Java Case Study*

From 9935 Java source files we successfully parsed 97.3% with our grammar. Manual inspection of the failing 2.7% files (i.e., 268 files) revealed that they indeed had syntax errors. All these files contained only snippets of Java code and belonged to a *testData* folder from the *kotlin* repository. Table 3.3 presents the information about deep priority conflicts and bracket usage in Java, for each project we studied.

Project	Affected Files	Disamb. with Brackets	
		Deep Conflicts	Shallow Conflicts
Matisse	0 / 41 (0.0%)	0 (0.0%)	33 (94.3%)
RxJava	0 / 1469 (0.0%)	0 (0.0%)	398 (78.3%)
aurora-imui	0 / 55 (0.0%)	0 (0.0%)	57 (74.0%)
gitpitch	0 / 45 (0.0%)	0 (0.0%)	1 (1.6%)
kotlin	0 / 3854 (0.0%)	0 (0.0%)	4892 (53.3%)
leetcode	0 / 94 (0.0%)	0 (0.0%)	30 (44.8%)
litho	0 / 510 (0.0%)	0 (0.0%)	297 (66.3%)
lottie-android	0 / 109 (0.0%)	0 (0.0%)	134 (87.6%)
spring-boot	2 / 3444 (0.06%)	0 (0.0%)	630 (55.4%)
vlayout	0 / 46 (0.0%)	0 (0.0%)	285 (76.2%)
All	2 / 9667 (0.02%)	0 (0.0%)	6757 (56.1%)

Table 3.3 Deep Priority Conflicts and Bracket Usage in Java.

Affected Files In total, only 2 Java files from the corpus contained deep priority conflicts.

Deep Priority Conflicts The pilot study revealed in total two operator-style conflicts involving lambda expressions. The conflicts adhered to the following form:

```
(CastType) () -> exp1 == exp2
```

Lambda expressions have lower priority than expressions for equality comparison (==). In turn, cast expressions have the highest priority amongst the three operators in the previous example. Incomplete disambiguation would allow two different interpretations:

```
((CastType) () -> exp1) == exp2
(CastType) () -> (exp1 == exp2)
```

Our experimental setup made this conflict measurable with context-free grammars that use declarative disambiguation.

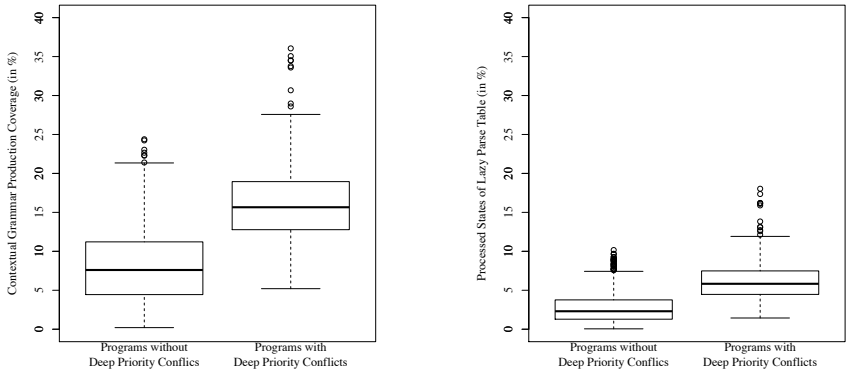
Explicit Disambiguation with Brackets The total number of observed pairs of brackets was 12049. None of these brackets actually avoided deep priority conflicts, however 56.08% of brackets account for resolving shallow priority conflicts. The remaining 43.92% of brackets are redundant.

3.4.4 *Grammar and Parse Table Coverage Statistics*

Table 3.4 lists the statistics for parsing the corpus with the transformed contextual grammars and resulting parse tables.

	Grammar		Parse Table		
	# Prod.	Used	# States	Lazy Expansion	
				Proc.	Visible
OCaml	3420	59.3%	20200	36.4%	45.8%
Java	1916	54.8%	4674	49.0%	56.8%

Table 3.4 Grammar and Parse Table Coverage Statistics.



(a) Grammar Coverage (b) Lazy Parse Table Coverage

Figure 3.3 Comparison of Grammar and Parse Table Coverage between OCaml Programs with / without Deep Priority Conflicts.

OCaml The transformed contextual grammar, capable of resolving all three categories of conflicts that we investigate, expands to 3420 productions. In terms of grammar coverage, parsing all files together exercised 59.3% of the productions.

From the grammar productions, a lazy parse table was generated that could extend to 20200 states, i.e., the number of states that a conventional SDF3 parse table generator would produce. From the possible number of states, 36.4% of the states were processed during parsing, and 45.8% of the states were visible, using the lazy parse table generator.

When looking at individual files, the mean coverage observed was 9.5% (range 0.2–36.1%). Figure 3.3a splits coverage data between programs free of deep priority conflicts and programs exhibiting deep priority conflicts. We observed that programs free of deep conflicts use on average 8.1% of contextual productions per file, while programs with deep conflicts use, in average, 16%. Figure 3.3b shows the corresponding data for processed states of the lazy parse table that was generated from the contextual grammar. We observed that programs free of deep priority conflicts exercise on average 2.7% of all possible states compared to 6.17% of programs that do have deep conflicts.

Java The transformed contextual grammar consists of 1916 productions. Parsing the whole Java corpus exercised 54.8% of the productions. When looking at individual files, the mean coverage measured was 12.95% (range 0.52–34.08%).

The corresponding full parse table has 4674 states. Parsing all Java files with the lazy parse table generator resulted in 49% processed states and 56.8% visible states. Due to the low number of deep priority conflicts found in Java, we omitted separate statistics for programs with and without deep priority conflicts.

3.4.5 *Recapitulation of Hypotheses*

Based on the pilot study results reported in previous subsections, we do conclude:

Confirmation of Hypothesis 1: Parsing the OCaml corpus with OCaml's expression-oriented grammar triggered deep priority conflicts in about one out of six files. In contrast, parsing the Java corpus resulted in total in only two deep priority conflicts.

Confirmation of Hypothesis 2: With 51.5%, the majority of deep priority conflicts were of type longest match. To our surprise, with 48%, operator-style conflicts were almost as frequent as longest match conflicts.

Confirmation of Hypothesis 3: Deep priority conflicts are sparse when parsing Java with grammars that use declarative disambiguation. Deep conflicts exclusively occurred in the context of lambda expressions.

Rejection of Hypothesis 4: We did not expect that deep priority conflicts would occur in about one out of six files when parsing OCaml, nor that *all* projects would have files with deep priority conflicts. With a frequency of 16.8% those conflicts can be considered common case, requiring support for (declarative) disambiguation.

Confirmation of Hypothesis 5: The results indicate that there is indeed a high cost attached to declarative disambiguation with grammar transformation techniques. On a per-file basis, our expectations were met. E.g., parsing OCaml files yielded a mean coverage of contextual productions of 12.95% (range 0.52–34.08%). Contrary to our intuition, parsing all files exercised more than 50% of the contextual productions but processed slightly less than 50% of the parse table states in both languages.

Confirmation of Hypotheses 6 and 7: Brackets are more excessively used in OCaml than in Java. In both languages brackets are mainly used to disambiguate (shallow) priority conflicts. However, 1.6% of bracket pairs in OCaml are used to disambiguate deep priority conflicts, suggesting that language users are exposed to deal explicitly with deep conflicts, and also that they rely on the disambiguation policy of the language for disambiguating deep priority conflicts.

Considering both languages, even though only up to 17% of the files contained deep priority conflicts, such conflicts do occur, and there is a need for supporting efficient disambiguation in combination with readable, concise, but inherently ambiguous context-free grammars. One may question whether it is a good language design practice to allow deep priority conflicts to occur in the first place, due to the problems they cause.

If we consider the efficiency of grammar transformation techniques to solve deep priority conflicts, we can conclude that there is room for improvement. Producing an unambiguous grammar that disallows deep conflicts results in many duplicate productions that do not seem to be used, even after parsing a considerable number of programs. This conclusion should lead to follow-up studies that can improve the efficiency of these grammar transformations.

From the programmer's point of view, deep priority conflicts can be even more confusing, as it is necessary to *really* understand the operator precedence specified with the grammar. Deep priority conflicts could even contribute to the decision of whether to use or learn a language, if we consider this extra burden imposed on novice programmers. Therefore, now that we have an indication that deep priority conflicts occur in real code, we can ask: are programmers aware of such conflicts?

From our study, we observed that programmers use brackets relatively often, but that up to 40% of the brackets were redundant. Therefore, we may ask if programmers use redundant brackets for readability or because they did not fully understand the precedence of the language. Considering this aspect, language design may also play a role in how often programmers need to use brackets explicitly. Future empirical studies could lead to more insights that connect language design, declarative disambiguation and how both of them affect programmers.

3.5 THREATS TO VALIDITY

In our pilot study, we have decided to examine the number of occurrences of deep priority conflicts by the number of ambiguities that occur when we *turn off* the solutions for such conflicts. Because ambiguities may occur nested within each other, this number corresponds to an under-approximation of the number of deep priority conflicts.

As we mentioned before, a program `1 + if e then 2 else 3 + 4` contains an ambiguity that cannot be solved by a parent-child relation on the addition and if-then-else expressions, causing a deep conflict. Such conflict is captured by our approach via the top-level ambiguity node of the whole expression. However, any operator-style conflict that occurs inside the expression `e` will not be counted, since it is hidden inside the top-level ambiguity of the outer expression.

Furthermore, we do not validate the ASTs produced when parsing each program in our test suite. In order to test the correctness of our grammars, we have used Java and OCaml program snippets that compare expected ASTs with the (unambiguous) ASTs produced when parsing such programs with

contextual grammars. These tests stress syntactic elements of each language, including cases of deep conflicts.

When counting brackets that disambiguate deep conflicts, we do not include brackets that would produce the same AST if removed from the program. For example, in the program `1 + (if e then 2 else 3 + 4)` the brackets disambiguate a deep conflict, choosing explicitly how the program should be interpreted. However, when removing the brackets and parsing the same program with a contextual grammar that solves operator-style conflicts, the same AST is produced, i.e., the brackets are redundant.

Operator-style conflicts in Java only occur due to lambda expressions. However, these expressions are relatively new in the Java language, being introduced in Java SE 8. It may be the case that lambda expressions are not yet used frequently by programmers or are only used in newer projects, which could explain the low number of deep priority conflicts we found in Java programs.

Finally, the size of our empirical study could hinder the conclusions we draw in this chapter. However, we still believe that this study can give significant insights to the research questions we raised.

3.6 RELATED WORK

In this section we highlight previous work on parsing and related work on empirical studies (based on corpus analysis).

3.6.1 *Safe and Complete Disambiguation*

Grammar-to-grammar transformations. Afroozeh et al. [5] proposed a new semantics for SDF2 priorities [74] that is safe and complete. The approach consists of rewriting the grammar, duplicating the productions for the non-terminals such that shallow and deep conflicting patterns cannot be produced. Even though this approach produces an unambiguous context-free grammar as result, the size of the resulting grammar can be quite large for languages containing many conflicts. Furthermore, the approach does not handle dangling else nor longest match conflicts.

Contextual grammars [116] is a grammar transformation that extends the approach from Afroozeh et al. [5]. The grammar productions are only duplicated to handle deep conflicts, and the technique also solves dangling else and longest match ambiguities. However, the resulting contextual grammars can still have many duplicated productions for languages with many deep conflicts.

Data-dependent grammars. A dynamic solution to safe and complete disambiguation is presented by Afroozeh and Izmaylova [8]. Instead of producing pure and unambiguous context-free grammars, this approach consists of producing a data-dependent grammar that checks for priority conflicts at parse time. Priorities are automatically translated into data-dependent productions that passes the precedence levels of the expression currently being parsed and checks whether it produces a priority conflict. This solution does not handle

longest match nor dangling else conflicts, and it is used with a top down parser that supports data-dependency tracking.

3.6.2 *Lazy Parser Generation*

IPG. The incremental parser generator IPG [59] was developed with the purpose of speeding-up parser generation in a highly interactive environment. At the early stages of language design, the language's syntax is constantly being changed, invalidating the current parse table. In incremental table generation, only the parts of a partial table that are affected by a change in the grammar are reconstructed at parse time. We have only adopted the lazy generation of IPG, i.e., when the grammar changes, our generator starts from scratch with an empty table. Lazy parse table generation allowed us to measure the coverage of contextual grammars.

ANTLR. A top-down approach to lazy parser generation is used in the Adaptive LL(*) parsing algorithm [97]. ANTLR 4 generates ALL(*) parsers that *adapts* to the input sentences presented to it at parse time. The parser dynamically constructs a prediction automaton that matches the lookahead of input being parsed deciding which production rule to expand. Intermediate results are memoized, i.e., the parser incrementally constructs the prediction automaton by need.

3.6.3 *Corpus Analysis and Grammar Coverage*

Empirically studying source code allows researchers to learn from real-world programs, for example to uncover coding practices [60, 82] or to evaluate the performance of tools and analyses. Studies do either reuse existing corpuses of various sizes, or construct corpuses that are suitable for answering their research questions. E.g., Landman et al. refuted common knowledge about the correlation of two source code metrics [81] by constructing and analyzing large corpuses.[11]

In contrast to large scale empirical studies, we conducted a pilot study for getting an intuition of how deep priority conflicts occur *in the wild*. The outcomes of this pilot study pinpoint and characterize real-world issues that arise with deep priority conflicts, guiding future large scale studies.

Context-dependent branch coverage [79] can give more insights on the coverage of contextual grammars. In our experiment, we adopted a rather simplistic approach by counting the number of productions used by the parser. However, our approach also includes information about the coverage of lazily generated tables in order to investigate the efficiency of grammar transformations to solve deep priority conflicts.

[11] One Java corpus containing 17.6M methods that are spread out over 1.7M files, and one C corpus containing 6.3M functions extracted from 462K files.

3.6.4 *Code Readability and Programming Style*

Buse and Weimer [33] define source code readability as *"as a human judgment of how easy a text is to understand"* and proposed a metric for measuring readability. Stefik and Siebert [121] and Sedano [110] provide an overview on empirical studies concerning code readability and programming style.

Instead of focusing on the human judgement perspective, we investigated deep priority conflicts that may hamper unambiguous parsing of source code. We studied how often brackets are used for readability purpose or to disambiguate (deep) priority conflicts. There is empirical evidence that if-statements that remove parentheses and braces are easier to comprehend by novice programmers [121], however such syntaxes are more likely to trigger (deep) priority conflicts.

3.7 CONCLUSION AND FUTURE WORK

We have presented an experiment to analyze deep priority conflicts in real-world programs. Our experiment uses contextual grammars to produce unambiguous grammars that solve deep conflicts. By turning off the generation of productions to solve each type of conflict, we were able to categorize and count occurrences of each type. Furthermore, we used lazy parse table generation to investigate the efficiency of contextual grammars when solving deep conflicts. We have also looked into explicit disambiguation, i.e., counting and analyzing the number of brackets in each program.

Our experiment indicates that deep conflicts do occur often in real programs. However, when looking into the efficiency of grammar transformations to solve deep conflicts, we observed that many productions and parse states resulting from the grammar transformation are not used after parsing corpuses of real-world programs.

We also observed that a large percentage of deep conflicts are explicitly disambiguated by brackets, which suggests that the default precedence of the language does not correspond to how programmers typically use the language constructs. The analysis of explicit disambiguation also gave us the insight that brackets are not used exclusively for disambiguation purposes. Further investigation is necessary to understand the actual intention of the programmer when using redundant brackets.

As future work, we propose extending the case study to other languages, i.e., including other SDF3 grammars, and to consider larger corpuses. Based on our findings, we propose further investigation on grammar transformation techniques used for declarative disambiguation, analyzing the reasons for the lack of coverage, and aiming to improve their efficiency when solving deep priority conflicts.

Towards Zero-Overhead Disambiguation of Deep Priority Conflicts

4

ABSTRACT

Context-free grammars are widely used for language prototyping and implementation. They allow formalizing the syntax of domain-specific or general-purpose programming languages concisely and declaratively. However, the natural and concise way of writing a context-free grammar is often ambiguous. Therefore, grammar formalisms support extensions in the form of declarative disambiguation rules to specify operator precedence and associativity, solving ambiguities that are caused by the subset of the grammar that corresponds to expressions.

Ambiguities with respect to operator precedence and associativity arise from combining the various operators of a language. While shallow conflicts can be resolved efficiently by one-level tree patterns, deep conflicts require more elaborate techniques, because they can occur arbitrarily nested in a tree.

Current state-of-the-art approaches to solving deep priority conflicts come with a severe performance overhead. In this chapter, we present a novel low-overhead implementation technique for disambiguating deep associativity and priority conflicts in scannerless generalized parsers with lightweight data-dependency. By parsing a corpus of popular open-source repositories written in Java and OCaml, we found that our approach yields speedups of up to 1.73x over a grammar rewriting technique when parsing programs with deep priority conflicts — with a modest overhead of 1% to 2% when parsing programs without deep conflicts.

4.1 INTRODUCTION

Context-free grammars have been established as the main formalism for concisely describing the syntax of programming languages (e.g., in reference manuals). Yet, context-free grammar definitions still cause problems when used to generate parsers in practice. On the one hand, a parser generator may expect a deterministic grammar that fits a certain grammar class, such as LL or LR. On the other hand, natural and concise context-free grammars may be inherently ambiguous, more specifically when considering the subset of the grammar that defines expressions and operators.

Mainstream parser generators, such as YACC [64], extend their grammar formalism with declarative constructs for disambiguation, allowing users to specify the precedence and associativity of operators. In order to not compromise performance, under the hood, YACC's translation of disambiguation

rules highly depends on specific characteristics of LR parsing technology — exploiting LR shift/reduce conflicts— rather than building upon a generalized solution. Furthermore, YACC does not support the composition of modular grammar fragments (of potentially different languages).

In contrast, the SDF2 syntax definition formalism [131] allows modular and composable language specifications, providing mechanisms for declaratively specifying operator precedence and associativity [74, 126]. The semantics for SDF2 disambiguation is parsing independent, but it only addresses ambiguities that are caused by so called *shallow conflicts*, i.e., conflicts that can be solved by checking whether a certain parse node is a direct descendant of another node in the parse tree. However, some ambiguities that occur in expressions can only be solved by checks in the final tree of unbounded depth [5, 116]. Such ambiguities are caused by *deep priority conflicts*.

Several approaches have been proposed to solve deep priority conflicts. Many of these approaches are based on grammar transformations and thus are parser independent [5, 4, 116]. However, those techniques typically result in large unambiguous grammars, which may impact on the performance of the parser and be somewhat inefficient, as considerable parts of the grammar are not exercised at runtime, even after parsing many programs [118].

Alternative solutions of so called data-dependent grammars [8] postpone solving of priority conflicts to parse time. Data-dependent grammars are context-free grammars, extended with arbitrary computations, parameters, variable binding, and constraints that can be evaluated at parse time [63]. Data dependent grammars that address disambiguation of priority conflicts can be generated from a context-free grammar with disambiguation constructs, but they are fairly complex and arguably hard to read and understand. The additional complexity comes from the bindings and constraints, and by the fact that data can be propagated "downwards" and "upwards" at parse time, when building the parse tree. Finally, data-dependent grammars have not yet been generalized to solve frequent types of deep priority conflicts such as longest match or the well known dangling else problem.

This chapter proposes a different solution to disambiguate deep priority conflicts at parse time based on *data-dependent contextual grammars*. Our approach relies on lightweight data dependency that does not require arbitrary computation nor variable bindings at parse time, and instead expresses disambiguation in terms of set-algebraic operations that can be implemented scoped and efficiently. The contributions of the chapter are:

- We define a lightweight data-dependent extension of the scannerless generalized LR parsing algorithm for disambiguating deep priority conflicts.
- We show that this data-dependent extension yields the same disambiguation as contextual grammars, an approach that solves deep priority conflicts through grammar rewriting.
- We compare the performance of disambiguation strategies and show that our lightweight data-dependent disambiguation is up to 1.73 x faster

when parsing programs with deep priority conflicts and has very low overhead for programs without deep priority conflicts.

We implemented our solution as a modified parser generator for SDF3 [133], which supports modular and composable syntax definitions. Furthermore, we evaluated our approach using OCaml, an expression-based language that contains many deep priority conflicts; and Java, a statement-based language that contains a small number of deep priority conflicts. Given that deep priority conflicts may occur in about one in five real-world programs for OCaml [118], we provide a technique that supports efficient disambiguation of such conflicts, showing that for programs without conflicts, our approach has a modest overhead of 1% to 2% on parsing time.

The chapter is organized as follows: Section 4.2 details background information on declarative disambiguation and deep priority conflicts. Section 4.3 describes data-dependent contextual grammars. Next, in Section 4.4 we evaluate our approach by comparing to disambiguation techniques that rely on grammar transformations. Finally, we discuss related work in Section 4.5, before concluding.

4.2 DISAMBIGUATING PRIORITY CONFLICTS

We start by presenting the notation for grammars, grammar productions and parse trees that will be used throughout this chapter. The remainder of this section then discusses background on the issue of deep priority conflicts and a grammar rewriting technique, contextual grammars [116], that addresses the resolution of such conflicts.

4.2.1 *Notation*

Grammars. A context-free grammar G can be formally defined as a tuple (Σ, N, P) with the set Σ representing the terminal symbols; the set N consisting of the non-terminal symbols defined in the grammar; and the set P representing the productions. When not mentioned, we adopted the letter A to represent arbitrary non-terminals; the letter X to represent a symbol from $\Sigma \cup N$; and Greek letters α, β or γ to represent a symbol in $(\Sigma \cup N)^*$, also known as sentential forms.

Productions. We use the same notation for productions as in the syntax definition formalism SDF3 [133]. A production in a grammar G has the form `A = α` or `A.C = α`, where `C` represents a constructor. SDF3 productions may have constructors to specify the name of the abstract syntax tree node constructed when imploding the parse tree. A non-terminal and a constructor uniquely identify a production, i.e., a production `A.C = α` may also be referred as `A.C`. Note that SDF3 constructors are orthogonal to our approach.

Parse Trees. A production `A.C = X1 ... Xn` may be used to construct a tree of form `[A.C = t1 ... tn]`, with the subtree `ti` being defined by the symbol `Xi`. Explicit subtrees are indicated by their productions using nested square

brackets, whereas arbitrary subtrees and terminal elements that occur as leaves do not require brackets. For example, the tree `[Exp.Add = e1 + [Exp.Mul = e2 * e3]]` constructed with a production `Exp.Add = Exp "+" Exp` has an arbitrary subtree e_1 as its leftmost subtree, and a rightmost explicit subtree defined by the production `Exp.Mul = Exp "*" Exp`, with arbitrary subtrees e_2 and e_3.

4.2.2 *Background on Deep Priority Conflicts*

In context-free grammars of programming languages, ambiguities are often caused by the subset of the language that contains expressions and operators. To address this issue, grammar formalisms used in practice support the specification of declarative disambiguation rules to define operator precedence and associativity among the grammar productions. E.g., in SDF3, a grammar production can have an annotation —either `left`, `right`, or `non-assoc`— to specify its associativity. SDF3 also supports context-free priorities, which form a partial order, defining a priority relation between productions. E.g., the disambiguation rule `Exp.Add > Exp.If` defines that addition has a higher priority than conditional expressions.

The most common ambiguities from expression grammars involve the direct combination of operators with different priorities. These ambiguities are caused by so-called *shallow priority conflicts* and can be efficiently solved by subtree filtering [74], i.e., disallowing certain kinds of trees to occur as a direct child of others.

A small but complicated-to-solve subset of ambiguities is caused by *deep priority conflicts*. Unlike shallow conflicts, deep priority conflicts cannot be filtered by observing the direct parent-child relationship of nodes within a parse tree. Deep priority conflicts can occur arbitrarily nested (i.e., in unbounded depth) in a parse tree. In general, deep conflicts can occur when a low-priority operator shadows a nested higher priority operator on the left- or rightmost edges along a sub-tree [5, 118]. Deep priority conflicts are commonly found in the expression parts of grammars of ML-like languages. A recent empirical pilot study suggests that up to 17 % of OCaml source files originating from popular projects on Github do contain deep priority conflicts [118], raising the question how such conflicts can be disambiguated efficiently.

Deep priority conflicts are categorized in three classes, according to their nature [116]:

Operator-Style Conflicts. Operator-style conflicts involve two operators: 1) a prefix operator[1] with lower priority, and 2) a postfix or infix operator with higher priority.[2] Figure 4.1 contains a minimal grammar example, illustrating

[1] We consider the definition of operators used in [116]: prefix operators are defined by *right* recursive productions, postfix operators by *left* recursive productions and infix operators by productions that are both *left* and *right* recursive.

[2] Operator-style conflicts may also involve lower priority postfix operators, but these are uncommon, as postfix operators usually have higher priority in most programming languages.

an operator-style conflict. In our case, it involves an addition expression that has higher priority than the conditional expression. Parsing the example sentence on line 13 causes an ambiguity due to a deep priority conflict and yields two possible interpretations (lines 17 and 19). In the example, the first instance represents the supposedly correct interpretation, since the addition to the left of the conditional expression extends as far as possible. The second and incorrect interpretation cannot be filtered by checking the direct parent-child relation of parse nodes, since the first addition expression shadows that the conditional (prefix operator with lower-priority) occurs indirectly nested at the rightmost position of the second addition (infix operator with higher-priority).

```
1  context-free syntax
2
3    Exp.If  = "if" "(" Exp ")" Exp
4    Exp.Add = Exp "+" Exp {left}
5    Exp.Int = INT
6
7  context-free priorities
8
9    Exp.Add > Exp.If
10
11 causes conflict in sentence
12
13   e1 + if(e2) e3 + e4
14
15 with interpretations
16
17   e1 + if(e2) (e3 + e4)
18
19   (e1 + if(e2) e3) + e4
```

Figure 4.1 Operator-Style Conflict

Dangling-Else Conflicts. Dangling-else describes a pattern for a deep priority conflict involving two productions that share the same prefix or suffix, where the shorter production is (left or right) recursive. Figure 4.2 illustrates a conflict involving the `Exp.If` and `Exp.IfElse` productions. For the sentence on line 9, a parser cannot decide where the else branches should be connected. Note that the first interpretation (line 13) is supposedly the correct one, where the else branches are connected to the closest if-expressions.

Longest-Match Conflicts. Another type of deep priority conflict involves indirectly nested lists [116]. The example grammar in Figure 4.3 defines `Exp.Match` expressions ending with a list of patterns. However, match expressions can themselves occur at the end of a pattern. E.g., for the sentence in line 9, the parser cannot decide whether the pattern p_2 belongs to the list of the `match` e_1 (cf. line 15) or the list of the `match` e_2 expression (cf. line 13). The first inter-

```
context-free syntax

  Exp.If     = "if" "(" Exp ")" Exp
  Exp.IfElse = "if" "(" Exp ")" Exp "else" Exp
  Exp.Int    = INT

causes conflict in sentence

  if(e1) if(e2) e3 else if(e4) e5 else e6

with interpretations

  if(e1) (if(e2) e3 else (if(e4) e5 else e6))

  if(e1) (if(e2) e3 else (if(e4) e5)) else e6

  if(e1) (if(e2) e3) else (if(e4) e5 else e6)
```

Figure 4.2 Dangling-Else Conflict

pretation should be preferred if the list construct (`Pat+`) itself follows longest match.

```
context-free syntax

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

causes conflict in sentence

  match e1 with id -> match e2 with p1 p2

with interpretations

  match e1 with id -> (match e2 with p1 p2)

  match e1 with id -> (match e2 with p1) p2
```

Figure 4.3 Longest-Match Conflict

4.2.3 *Disambiguating Deep Priority Conflicts with Contextual Grammars*

Many disambiguation approaches achieve independence from a particular parsing technology by relying on grammar rewriting (i.e., transforming an ambiguous context-free grammar into a context-free grammar that does not contain any priority conflicts). In the following, we discuss the disambiguation approach of contextual grammars [116] as a representative example for

```
1  context-free syntax
2
3    Exp.If  = "if" "(" Exp ")" Exp
4    Exp.Add = Exp{Exp.If} "+" Exp {left}
5    Exp.Int = INT
6
7    Exp{Exp.If}.Add = Exp{Exp.If} "+" Exp{Exp.If} {left}
8    Exp{Exp.If}.Int = INT
9
10 context-free priorities
11
12   Exp.Add > Exp.If
13
14 uniquely parses sentence
15
16   e1 + if(e2) e3 + e4
17
18 with interpretation
19
20   e1 + if(e2) (e3 + e4)
```

Figure 4.4 Contextual grammar that solves an operator-style deep priority conflict involving if and addition expressions in ML-like languages.

rewriting-based disambiguation strategies, since our contribution builds upon it. (An extensive discussion and comparison of related work on the subject of disambiguation can be found in Section 4.5).

Contextual grammars [116] are context-free grammars that can be used to solve deep priority conflicts. Under the hood, contextual grammars express invalid parse-tree patterns with respect to the disambiguation rules defined in the grammar. These patterns can be deeply matched to filter trees that would cause an ambiguity. The deep pattern matches do not occur at parse-time, but rather are implemented as a grammar transformation. A "black-list" of forbidden patterns, represented by so-called *contextual tokens* drives the recursive rewriting algorithm, restricting which parse trees a production may produce along the (leftmost or rightmost) positions of sub-trees.

Recursive Rewriting by Example. Figure 4.4 illustrates an example for a contextual grammar that solves the operator-style conflicts of Figure 4.1. A grammar transformation recursively rewrites the grammar and adds new productions for the symbol `Exp{Exp.If}` (lines 7– 8). The rewriting propagates the contextual tokens to all leftmost and rightmost non-terminals of the newly added production. By using the tokens to create new non-terminal symbols that implement filters, the rewriting avoids the construction of invalid trees.

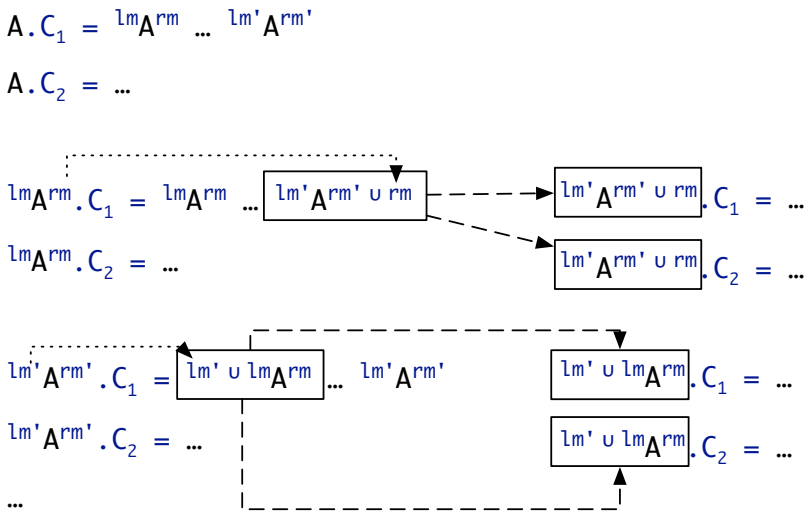


Figure 4.5 Generalized recursive propagation of contextual tokens in contextual grammars.

According to this grammar, the example sentence in line 16 now can be unambiguously parsed as shown in line 20. The invalid sentence `(e1 + if(e2) e3) + e4` cannot be parsed anymore, because the addition `… + e4` that is parsed with `Exp.Add` must not have an if-expression on the rightmost position of the first addition `e1 + if(e2) e3`.

One issue with the rewriting is that the propagation of constraints might result in many additional productions in the final contextual grammar, as productions of the original grammar need to be duplicated (recursively) for each new contextual symbol. While the previous example is relatively concise, the general case is not.

Recursive Rewriting in General. Formally, a contextual symbol ${}^{lm}A^{rm}$ is a regular non-terminal A that is uniquely identified by the tuple (lm, A, rm), where lm and rm are sets containing contextual tokens. For brevity we omit lm or rm respectively when the set is empty. The set lm stores unique references to productions that are not allowed to occur in any, possibly deeply nested, leftmost node of the tree defined by the contextual symbol ${}^{lm}A^{rm}$. Similarly, the productions referenced in the set rm cannot be used to construct any, possibly deeply nested, rightmost node of the tree defined by ${}^{lm}A^{rm}$. Finally, the tree for the symbol A itself cannot be constructed using any of the productions referenced in lm and rm.

Figure 4.5 highlight the general case of how productions are recursively rewritten. The starting point are the first two productions `A.C1` and `A.C2`, where the production `A.C1` contains deep priority conflicts, as indicated by the contextual symbols on the right-hand side of the production. For each

unique contextual symbol, the productions for the symbol `A` need to be duplicated excluding the productions in the sets *lm* and *rm*, while propagating the contextual tokens accordingly. In the second pair of rules in the grammar, new productions are created for the symbol ${}^{lm}A^{rm}$ assuming that $\mathtt{c}_1$ and $\mathtt{c}_2$ are not in the sets *lm* and *rm*, and the set *rm* is propagated to the rightmost symbol of that rule (cf. dotted arrow). When propagating the set containing the rightmost contextual tokens *rm* to the rightmost symbol of this rule, a new unique contextual symbol ${}^{lm'}A^{rm' \cup rm}$ is generated, causing a ripple effect: new productions need to be recursively generated for the new symbol as well (cf. the dashed arrows). The same ripple effect might occur for the symbol ${}^{lm' \cup lm}A^{rm'}$ and for any other unique contextual symbol resulting from the propagation of contexts.

Contextual grammars can correctly disambiguate all previously discussed conflicts, however at a high cost. For expression-based languages with many deep priority conflicts —such as OCaml— the grammar can get about three times bigger [116]. In the case of contextual grammars, the duplication is necessary to solve deep conflicts, however, many productions are not exercised in practice, even after parsing a large set of programs [118]. The duplication directly introduces a performance penalty, causing larger parse tables, and longer parse times in practice.

Our aim is to avoid the blow-up in productions caused by grammar transformations, without giving up the correctness properties guaranteed by contextual grammars. In the next section we illustrate how the underlying concepts of contextual grammars can be repurposed to disambiguate deep priority conflicts efficiently at parse time.

4.3 DATA-DEPENDENT CONTEXTUAL GRAMMARS

In this section, we focus on declarative disambiguation techniques that are more general than, for example, YACC's approach, in order to support modular and composable syntax definitions. In particular, we illustrate how low-overhead disambiguation can be implemented in SDF3 [133] with a scannerless generalized LR parser (SGLR) [130].

Figure 4.6 highlights the different stages in the context of parser generation using SDF3 and parsing in SGLR. First, a *normalized*[3] SDF3 grammar is first transformed by recursive rewriting into a contextual grammar, which contains additional productions to remove deep priority conflicts (cf. Section 4.2.3). Second, the parse table generator produces a parse table given the contextual grammar, solving shallow priority conflicts directly when constructing the table, by filtering *goto*-transitions according to the priorities specified in the grammar [126]. Afterwards, the SGLR parser uses the generated parse table for processing arbitrary input programs. The parser may use other disambiguation

[3]SDF3 grammars are normalized to handle lexical and context free syntax declarations, derive additional productions for symbols that represent lists or optionals, insert optional layout in between context-free symbols, and expand priority groups and chains.

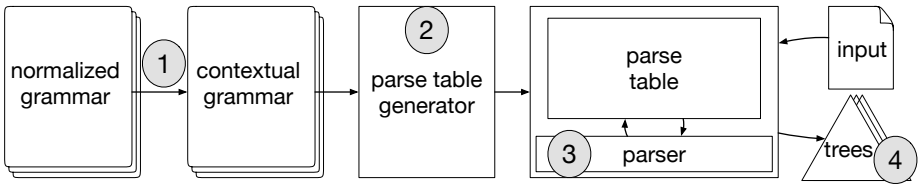


Figure 4.6 Parsing a program with a scannerless generalized parser, and the times when disambiguation might occur.

mechanisms at parse time to address, for example, ambiguities in the lexical syntax using reject productions [130]. The SGLR parser returns a compact representation of a parse forest that contains all trees that were derived when parsing an input program. As a final step, a (post-parse) disambiguator may still remove invalid trees from the parse forest according to given constraints.

According to Figure 4.6, we can identify four different stages when disambiguation of priority conflicts can occur: (1) before parse table generation, (2) at parser generation, (3) at parse time, and (4) after parsing. Post-parse disambiguation is conceptually the most expensive approach, since ambiguities can grow exponentially with the size of the input [45]. Disambiguation should preferably occur in the first three identified stages of disambiguation (i.e., avoiding the construction of invalid trees beforehand). Nevertheless, disambiguating early in the pipeline does not necessarily guarantee the best performance either, as we will discuss next.

When to Disambiguate (Deep) Priority Conflicts? Disambiguating priority conflicts by grammar rewriting occurs before parse table generation. Rewriting techniques have the advantage that the remainder of the parser generation and parsing pipeline can operate oblivious of priority conflicts. Especially for resolving deep priority conflicts, grammar rewriting unfortunately adds many productions for forbidding conflicting patterns and may result in large grammars that negatively impact performance.

Disambiguating conflicts during parse table generation does not require any grammar rewriting and can be achieved by modifying the LR parse table generator. In a scannerless parser, disambiguation at parse table generation can only resolve shallow priority conflicts, requiring that deep priority conflicts are addressed earlier or later [116].

As noted previously, using post-parse disambiguation filters to solve priority conflicts can be inefficient in practice, because the number of ambiguities in expressions can grow exponentially with the size of an expression. Hence, a post-parse filter would have to traverse a large number of trees in the parse forest to filter invalid trees.

According to the reasoning above, in the next sections we will explore a solution to disambiguate deep priority conflicts efficiently at parse time, while keeping the disambiguation of shallow conflicts when generating the parse table.

Data-dependent grammars [63] extend context-free grammars allowing parameterized non-terminals, variable binding, evaluation of constraints, and arbitrary computation at parse time. Data-dependent grammars can be translated into stack-based automata, i.e., push-down automata with environments to track data-dependent parsing states. For example, consider the productions:

```
Iter(n).Conc  = [n >= 1] Iter(n - 1) A
Iter(n).Empty = [n == 0] ϵ
```

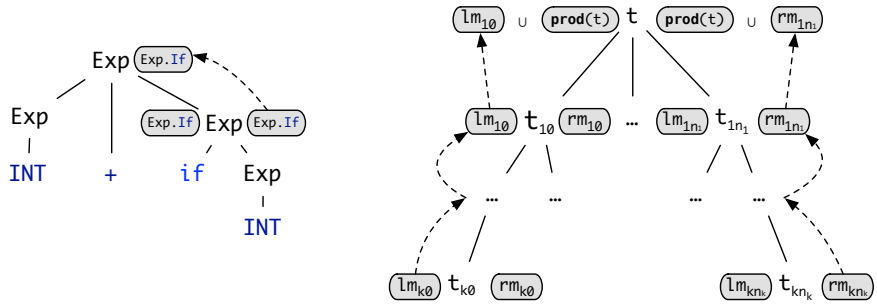
In the example above, the non-terminal `Iter` is parameterized by an integer `n`, which indicates the length of the iteration over the non-terminal `A`. The constraint `[n >= 1]` is checked before trying to parse `Iter(n - 1) A`, i.e., if $n \geq 1$ the first production is used, otherwise, the second.

Purely data dependent grammars are powerful enough to disambiguate priority conflicts of grammars for programming languages at parse time [8]. They allow resolution of possible priority conflicts in the grammar by means of constraints that forbid the creation of invalid trees. Nevertheless, just relying on data-dependency might negatively impact the performance of the parser, especially when parsing files that are free of priority conflicts. For that reason, we selectively use a lightweight form of data-dependency to solely solve deep priority conflicts, without requiring variable bindings or arbitrary computations at parse time.

Leveraging Data-Dependency to Avoid Duplicating Productions. Contextual grammars (cf. Section 4.2.3) can be treated as pure context-free grammars, if we consider that every unique contextual symbol specifies a new non-terminal. Transforming a contextual grammar into a context-free one, occurs by duplicating productions (recursively) for each unique contextual symbol.

Without duplicating the productions, a contextual grammar would have exactly the same shape and number of productions as the original grammar, since contextual symbols consist of essentially annotated non-terminals that originate from an analysis phase. Without rewriting, the grammar itself is still ambiguous, but the inferred contextual tokens that occur in the grammar can be reused to solve deep priority conflicts at parse time.

Bottom-up Constraint Aggregation instead of Top-Down Rewriting. Instead of propagating the constraints in the form of contextual tokens in the grammar productions, which may result in new contextual symbols and consequently new productions, we propagate the data to which the constraints are applied at parse time. Since the SGLR parser constructs trees bottom-up, we propagate the information about the productions used to construct the possibly nested leftmost and rightmost nodes of a tree bottom-up as contextual tokens during tree construction. Each node of the parse tree of the adapted data-dependent SGLR parser contains two additional sets that indicate the productions used to construct its leftmost and rightmost (nested) subtrees, respectively. For every node, the set representing the leftmost contextual tokens is the union of the the production used to construct the current node with the leftmost set of the



(a) Propagation by Example

(b) Schema of Generic Propagation

Figure 4.7 Propagation of contextual tokens in contextual grammars with data-dependency.

```
1  context-free syntax
2
3    Exp.If  = "if" Exp
4    Exp.Add = Exp{Exp.If} "+" Exp {left}
5    Exp.Int = INT
6
7  context-free priorities
8
9    Exp.Add > Exp.If
```

Figure 4.8 A (truncated) succinct syntax for a data-dependent contextual grammar with if-expressions, to be used for illustrating concise parse tree examples.

leftmost direct child. Similarly, the set representing the rightmost contextual tokens is the union of the production used to construct the node itself with the rightmost set of contextual tokens of the rightmost direct child. Note that only productions that can cause deep priority conflicts are added to the sets of contextual tokens; the number of tokens propagated is significantly lower than the total number of productions, even for highly ambiguous grammars. (The largest number of contextual tokens —33— where required for OCaml, which contains a highly ambiguous expression grammar.)

Data-Dependent Contextual Token Propagation by Example. The tree in Figure 4.7a for the sentence `INT + if INT` was parsed using the data-dependent contextual grammar of Figure 4.8. Since `Exp.If` is the only production that appears in the contextual tokens in the grammar, it is the only token that needs to be propagated upwards. Because the if-expression occurs as a direct right subtree of the addition, only its rightmost set of contextual tokens is propagated upwards, discarding the leftmost set of tokens.

```
1  function DO-REDUCTIONS(Stack st, Production A.C = X_1...X_n)
2    for each path from stack st to stack st_0 of length n} do
3      List<Tree> [t_1,...,t_n] = the trees from the links in the path from
       st to st_0
4      for each X_i such that X_i is a contextual symbol lm X^rm do
5         if t_i.LeftmostTokens ∩ lm ≠ ∅ or t_i.RightmostTokens ∩ rm ≠ ∅
       then
6           return
7         end if
8      end for
9      REDUCER(st_0, goto(state(st_0), A.C = X_1...X_n), A.C = X_1...X_n, [t_1,...,t_n])
10   end for
11 end function
```

```
1  function CREATE-TREE-NODE(Production A.C = X_1...X_n, List<Tree>
       [t_1,...,t_n])
2    Tree t = [A.C = t_1,...,t_n]
3    t.LeftmostTokens = t_1.LeftmostTokens ∪ A.C
4    t.RightmostTokens = t_n.RightmostTokens ∪ A.C
5    return t
6  end function
```

Figure 4.9 Pseudocode for the modified *DO-REDUCTIONS* and *CREATE-TREE-NODE* methods from the original SGLR, in the implementation of the data-dependent SGLR.

Data-Dependent Contextual Token Propagation in General. Consider the tree schema indicating the propagation of possible contextual tokens of Figure 4.7b. Assuming that the tree has depth k, the tokens will be propagated bottom-up through the leaves until reaching the root t. However, for a leaf node t_{k0}, its set of contextual tokens consist only of $\mathbf{prod}(t_{k0})$ (the production used to construct t_{k0}). As we will discuss in Section 4.3.3, we only propagate contextual tokens that occur in the contextual grammar, i.e., if $\mathbf{prod}(t_{k0})$ cannot cause a deep priority conflict, the set is in fact, an empty set. Thus, besides limiting the propagation to the depth of the trees being constructed, for grammars with few conflicts, only a small amount of data is actually propagated when constructing the tree.

Customization of Parsing Algorithm. The algorithm for the data-dependent scannerless generalized LR parser requires only a few changes in the original SGLR algorithm shown in [130]. More specifically, the algorithm needs to propagate contextual tokens corresponding to the productions used to construct the leftmost and rightmost (possibly nested) subtrees (`t.LeftmostTokens` and `t.RighmostTokens`),[4] and to check the constraints when performing reduce

[4]In the original SGLR algorithm, creating a parse tree node consisted simply of applying a production to the trees collected when calculating the path for the reduce action. In the data-dependent algorithm, the sets of leftmost and rightmost subtrees need to be updated by propagating the information from the rightmost and leftmost direct subtrees.

actions. We show the pseudocode for the modified methods of the original SGLR in Figure 4.9. Note that because we leverage the analysis done by contextual grammars, our data-dependent SGLR algorithm can solve the same types of deep priority conflicts that can be solved by regular contextual grammars, i.e., operator-style, dangling else and longest match. Furthermore, because we propagate the data representing possible conflicts at parse time, and enforce the constraints when performing a reduce operation, the grammar does not require modifications that increase its number of productions.

4.3.2 *Scannerless Generalized LR Parsing with Data-Dependent Disambiguation*

To illustrate how our implementation of a data-dependent SGLR performs disambiguation at parse time, consider the scenario when parsing the input `INT + if INT + INT`, which contains an operator-style deep priority conflict, using the data-dependent contextual grammar shown previously. After parsing `INT + if INT`, the parser reaches a state with a shift/reduce conflict in the parse table shown in configuration (I) from Figure 4.10. Before this point, SGLR performs actions according to the parse table to construct the single stack shown in this configuration, with the links between states (represented by the boxes) containing the trees that have been created so far, or the terminal symbols that have been shifted.

Note that in this first configuration, when reaching the conflict in the parse table, a parser that uses the disambiguation mechanism from YACC (see Section 4.5) can make the decision of which action to take based on the next input token. In this case, the parser would choose shifting over reducing because the next token in the input is `+` and the addition has higher priority over the if expression. However, this approach of looking at the next input token does not extend to scannerless parsers or character-level grammars, since the parser operates on characters, and the character `+` might be preceded by layout, or could be the prefix of a different operator.

Thus, instead of making a decision at the configuration (I), a generalized parser such as SGLR performs both actions in pseudo-parallel, producing an ambiguity if both actions lead to a successful parse. First, SGLR performs all possible reduce actions, which may result in the creation of different stacks. That is, the parser continues by forking a new stack, adding a link to the original one, creating a graph structured stack. This occurs at the configuration (II), as a reduce action has been performed to construct the tree:

```
[Exp.If = if [Exp.Int = INT]]
```

Because the production used to construct the tree appears in the rightmost set of contextual tokens of the contextual symbol $\texttt{Exp}^{\{\texttt{Exp.If}\}}$ in the grammar, the tree is constructed with the sets of leftmost and rightmost tokens containing `Exp.If`.

As the parser can still perform another reduce action from configuration (II), it does so, reaching configuration (III). Using the tree for the if expression, SGLR creates the following tree, by reducing using an `Exp.Add` production:

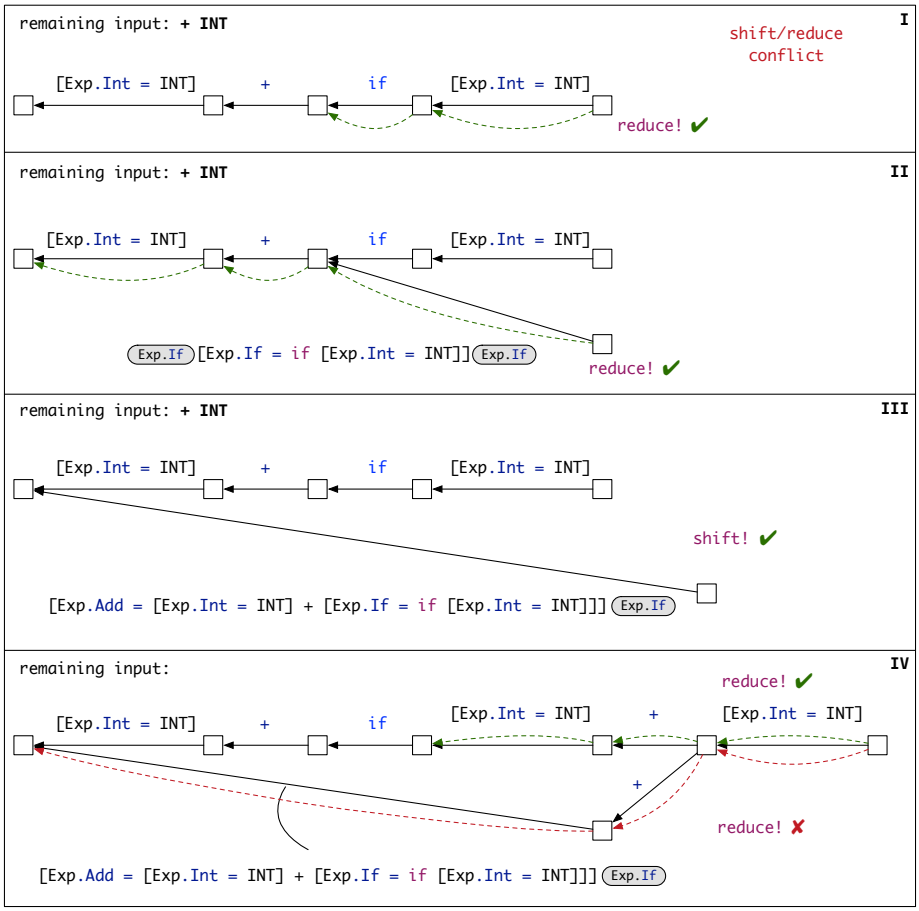


Figure 4.10 The configurations of SGLR when solving a deep priority conflict when parsing program `INT + if INT + INT`.

```
[Exp.Add = [Exp.Int = INT] + [Exp.If = if [Exp.Int = INT]]]
```

Note that since the tree for the if expression is used as the rightmost tree when applying the reduce action, and this tree has a non-empty set of rightmost contextual tokens, the set is propagated when creating the tree for the addition.

After shifting the additional symbols from the input, and performing a reduce action that creates the last `[Exp.Int = INT]` tree, the parser reaches configuration (IV). At this point, there is no other symbol to shift, so only reduce actions are left to be performed. When reducing using an `Exp.Add` production, there are two possible paths from the state at the top at the stack. The first path, at the top of the graph, creates a tree corresponding to the addition of two integers, which does not contain any deep priority conflict. The second path, at the bottom, contains a conflict since the set of rightmost

tokens for the first tree intersects with the rightmost set of contextual tokens for the contextual symbol $\texttt{Exp}^{\{\texttt{Exp.If}\}}$ in the `Exp.Add` production. Thus, the data-dependent SGLR uses this information to forbid the reduce action on the path at the bottom. By doing that, it produces only a single tree, solving the deep priority conflict.

4.3.3 *Performance Optimizations*

As shown in the algorithm for the data-dependent SGLR, the operations necessary to perform disambiguation of deep priority conflicts consist of set-algebraic operations such as union (for data propagation) and intersection (for constraint checking). To optimize our solution, we first map all productions that occur in the sets of contextual tokens of contextual symbols from the grammar to a bitset, limiting the amount of data that needs to be propagated. Thus, if a certain production belongs to a set of tokens, its corresponding bit is set to 1, or to 0, otherwise. Using this approach, constraint checking and data propagation can be achieved at a very low cost by performing bitwise operations on these bitsets. With such optimization we were able to achieve near zero-overhead when comparing our data-dependent approach and the original SGLR, for programs that do not contain deep priority conflicts, as we will show in the next section.

4.4 EVALUATION

In this section, we evaluate our approach of declarative disambiguation for solving deep priority conflicts, by applying it to a corpus of real programs. We are interested in answering the following research questions:

RQ1 For files that do not contain deep priority conflicts, how much overhead is introduced by data-dependent contextual grammars?

RQ2 For files that do contain deep priority conflicts, how do data-dependent contextual grammars perform when solving such conflicts, in comparison to related work?

In order to tackle the aforementioned research questions, it is essential to partition a data set in files that are free of deep priority conflicts, and files that are known to have deep priority conflicts. We re-use a corpus of the top-10 trending OCaml and Java projects on Github. The corpus was qualitatively analyzed by Souza Amorim, Steindorfer, and Visser [118], listing the types of priority conflicts each file from the projects contains. For both languages we partitioned the files into two groups according to their analysis results: files are free of deep priority conflicts (and therefore can be parsed by parsers without sophisticated disambiguation mechanisms), and files that contain deep priority conflicts. Table 4.1 lists the projects contained in the corpus, the total number of source files contained in those project, and the (relative) number of files with deep priority conflicts for OCaml and Java, respectively. Based on the research questions listed before, we can formulate our hypotheses:

Table 4.1 Deep Priority Conflicts in OCaml and Java Corpus.

OCaml Project	Affected Files	Java Project	Affected Files
FStar	6 / 160 (3.8%)	Matisse	0 / 41 (0.0%)
bincat	5 / 26 (19.2%)	RxJava	0 / 1469 (0.0%)
bucklescript	85 / 885 (9.6%)	aurora-imui	0 / 55 (0.0%)
coq	158 / 417 (37.9%)	gitpitch	0 / 45 (0.0%)
flow	52 / 305 (17.0%)	kotlin	0 / 3854 (0.0%)
infer	33 / 234 (14.1%)	leetcode	0 / 94 (0.0%)
ocaml	112 / 909 (12.3%)	litho	0 / 510 (0.0%)
reason	4 / 36 (11.1%)	lottie-android	0 / 109 (0.0%)
spec	4 / 40 (10.0%)	spring-boot	2 / 3444 (0.06%)
tezos	71 / 149 (47.7%)	vlayout	0 / 46 (0.0%)
All	530 / 3161 (16.8%)	All	2 / 9667 (0.02%)

H1 Due to our lightweight data-dependent disambiguation, we expect single-digit percentage overhead when parsing files that do not contain deep priority conflicts.

H2 Since disambiguation by grammar transformation produces up to three times bigger [118] grammars, we expect our lightweight data-dependent disambiguation to perform significantly better (i.e., higher double-digit percentage improvements).

4.4.1 *Experiment Setup*

The benchmarks were executed on a computer with 16 GB RAM and an Intel Core i7-6920HQ CPU with a base frequency of 2.9 GHz and a 8 MB Last-Level Cache. The software stack consisted of Apple's macOS operating system version 10.13.1 (17B48) and an Oracle's Java Virtual Machine (version 8u121).

To obtain statistically rigorous performance numbers, we adhere to best practices for (micro-)benchmarking on the Java Virtual Machine (JVM) as, for example, discussed in Georges, Buytaert, and Eeckhout [52] and Kalibera and Jones [67]. We measure the execution time of batch-parsing the corpus of OCaml and Java sources with the Java Microbenchmarking Harness (JMH), which is a framework to overcome the pitfalls of (micro-)benchmarking. Since the batch-parsing execution times are expected to be in terms of minutes — rather than microbenchmarks that execute in milliseconds— we configured JMH to perform 15 *single-shot* measurements: i.e., forking a fresh virtual machine 15 times and measuring the total batch-parsing time including cold startup.

For executing the benchmarks, we disabled CPU frequency scaling, disabled background processes as much as possible, and fixed the virtual machine heap

Language	Data Set	Disambiguation	Time (seconds)	Speedup	Cost
Java	with conflicts	data-dependent	0.18 ± 0.00	1.29 x	—
		rewriting	0.23 ± 0.00	1.00 x	—
Java	without conflicts	data-dependent	270.64 ± 1.28	1.73 x	1.02 x
		rewriting	467.20 ± 4.03	1.00 x	1.77 x
		none	264.20 ± 2.36	—	1.00 x
OCaml	with conflicts	data-dependent	80.60 ± 1.48	1.54 x	—
		rewriting	123.75 ± 1.02	1.00 x	—
OCaml	without conflicts	data-dependent	89.82 ± 0.51	1.46 x	1.01 x
		rewriting	130.71 ± 0.55	1.00 x	1.48 x
		none	88.58 ± 0.98	—	1.00 x

Table 4.2 Benchmark Results when parsing the OCaml and Java Corpus.

sizes to 10 GB for benchmark execution. The benchmark setup was tested and tuned to yield accurate measurements with relative errors of typically less than 2 % of the execution time. We report the measurement error as Median Absolute Deviation (MAD), which is a robust statistical measure of variability that is resilient to small numbers of outliers.

4.4.2 *Experiment Results*

The results of our experiment are illustrated in Table 4.2. We first report the precision of the individual data points. For all the data points, the relative measurement errors are in the range of 1.0 % to 4.1 % with a median error of 1.6 %; the absolute amounts are printed in the table next to the benchmark runtimes (cf. column *Time (seconds)*).

Cost of Disambiguating Deep Priority Conflicts (Hypothesis H1). Column *Cost* shows how the parser's performance is affected by supporting the disambiguation of deep priority conflicts. The cost measurements were performed solely for the data sets that are guaranteed to be free of deep priority conflicts, since we use a parser without deep priority conflict disambiguation as a baseline. The results show that the cost of disambiguation with data-dependency is between 1 % (OCaml) and 2 % (Java), supporting Hypothesis H1. Note that the result for OCaml is not statistically significant, i.e., the 1 % difference may as well be in the margin of error. For the Java case, the result is statistically significant, however the error intervals are very close and almost overlap. We conclude that Hypothesis H1 is supported by our experiment: the cost of declarative disambiguation is clearly below 10 %.

Data-Dependent Disambiguation versus Grammar Rewriting (Hypothesis H2). Column *Speedup* of Table 4.2 shows the performance improvements of data-

dependent disambiguation over disambiguation via grammar rewriting (baseline). In all tested configurations, data-dependent disambiguation speeds-up from 1.29 x to 1.73 x, reducing batch parse times considerably. E.g., parse time for the conflict-free Java corpus reduced from 467.20 s to 270.64 s. We conclude that Hypothesis H2 is supported by our experiment: data-dependent disambiguation outperforms disambiguation via grammar rewriting as discussed in Souza Amorim, Haudebourg, and Visser [116] and Adams and Might [4].

4.4.3 *Threats to Validity*

To counter internal threats to validity, we properly tested the data-dependent implementation and assured that it produces abstract syntax trees identical to the contextual grammars. For the data sets that are guaranteed to be free of deep priority conflicts, we also assured that the resulting parse trees are identical to the trees from the corresponding non-disambiguating grammar. In all scenarios, we checked that each resulting parse tree is indeed free of ambiguities, cross-validating the findings from the empirical pilot study [118] that accompanies the corpus.

To counter external threats to validity, we carefully designed and implemented our approach to use a lightweight form of data-dependency selectively, solely disambiguating deep priority conflicts. The delta to a baseline SGLR parser without support for disambiguation of deep conflicts is minimal: it requires the addition of a few lines of code, as shown in Figure 4.9. Therefore, we are confident that the observed cost of 1 % to 2 % for disambiguating deep priority conflicts remains steady, even when using different or larger data sets. We are also confident that the significance of the performance improvement remains clearly observable regardless of the used data sets, because it is commonly known that grammar rewriting blows-up the grammars and the resulting parse tables [118], negatively impacting parsing performance. Nevertheless, the size and choice of our corpus arguably remains an external threat to validity.[5]

4.5 RELATED WORK

In the following section, we highlight previous work on disambiguation of conflicts that arise from the declarative specification of operator precedence and associativity in context-free grammars, grouped by the phase *when* disambiguation happens.

4.5.1 *Disambiguation by Grammar Rewriting*

Ambiguities that arise from operator precedence and associativity can be avoided by rewriting the grammar to an unambiguous one. In the following,

[5] For the Java data set with conflicts, we would even assume that performance further improves with a larger data set. Unlike the other data sets, the current Java data set with conflicts consists of only two files, because deep conflicts are scarce in Java. The small data set is disadvantaged, because we compare batch-parsing time including cold startup time.

we list declarative disambiguation techniques that try to automatically derive unambiguous grammars.

Aasa [1] proposes a grammar rewriting technique that addresses priority conflicts by generating new non-terminals with explicit precedence levels, forbidding the construction of trees that could cause a conflict. The approach addresses shallow conflicts as well as deep conflicts of type operator-style. Due to a restriction —productions may not have overlapping prefixes or suffixes— it cannot solve dangling-else conflicts.

Thorup [123] presents a grammar transformation algorithm that constructs an unambiguous grammar, given an ambiguous grammar, and a set of counterexamples (i.e., illegal parse trees). The resulting grammar encodes the parse trees bottom-up, while removing grammar symbols that correspond to illegal trees. Thorup's approach specifically supports dangling-else, but does not generalize the construction of counterexamples to capture arbitrary deep priority conflicts.

Conceptually similar to Thorup's idea, Adams and Might [4] propose a grammar rewriting solution, where invalid patterns are expressed using tree automata [39]. Each type of conflict should be expressed as a tree automaton, representing the pattern of the counterexample. Intersecting the counterexample automata with the original context-free grammar yields an unambiguous grammar as result. The authors address all conflicts shown in this chapter, with the exception of longest match.[6]

Afroozeh et al. [5] describe a *safe* semantics for disambiguation by grammar rewriting that only excludes trees that are part of an ambiguity. While their semantics does cover shallow priority conflicts and deep priority conflicts of type operator-style, it addresses neither dangling-else nor longest match.

Contextual grammars [116] generalize the approach by Afroozeh et al. [5], by supporting a *"safe"* semantics for disambiguation of arbitrary deep priority conflicts. The authors analyze and address the root causes of deep priority conflicts. Their grammar analysis yields as a result, combinations of conflicting productions that may rise to a deep priority conflict. The authors show that deep conflicts can only occur in specific paths in the parse trees. Illegal patterns are conceptually described as deep pattern matches, and implemented by means of recursive grammar rewriting that forbids invalid trees to be constructed. Rewriting is used solely for solving deep priority conflicts; disambiguation of shallow conflicts happens at parse table generation.

All related work mentioned above suffers from the same performance issues: large unambiguous grammars as a result of recursive rewriting, with even larger parse tables that have a low-coverage of parsing states when parsing programs [118]. By contrast, our lightweight data-dependent disambiguation technique avoids grammar transformations and is able to reuse LR parse tables of grammars that do not solve deep priority conflicts, resulting in high parse table coverage and a low overhead of 1 % to 2 % for disambiguation. By reusing the grammar analysis results of contextual grammars, but implementing our mechanism for disambiguation via lightweight data-dependency, we are able

[6]Due to the expressivity of tree automata, we assume that longest match could be supported.

to handle arbitrary deep priority conflicts, including deep conflicts caused by indirect recursion.

4.5.2 *Disambiguation at Parser Generation*

Instead of changing the original grammar, some techniques perform disambiguation of priority conflicts at parser generation time. Even though these solutions do not require changing the productions of the original grammar, they might still be restricted to a certain parser, e.g., by depending on specific characteristics of a parsing technique, or by requiring modifications in the parser generation algorithm.

YACC [64] resolves ambiguities concerning operator precedence and associativity by solving shift/reduce conflicts that occur in LR parse tables. Even though the decision is made dynamically depending on the current lookahead token, the grammar has to specify the default action to take in the conflicting state, given the precedence of the operators involved in the conflict specified in the grammar. This technique has two major drawbacks. First, users have to reason in terms of shift/reduce conflicts, and annotate the grammar if a shift action should be preferred over reduce, or vice versa, to resolve conflicts. Second, YACC's disambiguation does not apply to scannerless parsers, since it requires knowing the lookahead token for decision making. More specifically, YACC's disambiguation does not apply to any parser that relies on character-level grammars [103, 104, 131], which are useful to avoid issues when composing grammars of different languages [31, 30].

Klint and Visser [74] propose a semantics for declarative disambiguation based on disambiguation filters. This semantics has been implemented by SDF2, so invalid tree patterns can be constructed from SDF2 priority declarations [127], and used in a disambiguation filter. The implementation relies on a custom LR parse table generator, as SDF2 parse tables encode goto transitions between states using productions, forbidding transitions that could construct an invalid tree according to the tree patterns. Because this semantics only targets conflicts by checking a parent-child relation in a tree, this solution is not able to solve deep priority conflicts.

4.5.3 *Disambiguation while Parsing*

Ambiguities from operator precedence may also be addressed at parse time. Such approaches have the advantage of using the original (or a slightly modified) grammar as input, but require adaptations of the parsing algorithm that may cause overhead when parsing programs that are free of priority conflicts.

Afroozeh and Izmaylova [8] introduce a solution for disambiguating priority conflicts on the basis of a full-fledged data-dependent grammar formalism. The authors implemented their approach in a generalized LL parser named Iguana [6]. Iguana addresses shallow priority conflicts and operator-style deep conflicts using data-dependent grammars, but the approach does not extend to dangling-else nor longest match. Given the experimental setup described

in their paper [8], it is not possible to assess the overhead of solving deep priority conflicts, because no analysis has been performed on programs free of deep priority conflicts. When solving the shallow conflicts present in the Java 7 grammar, Afroozeh and Izmaylova's approach causes, on average, 5% overhead compared to the unambiguous Java grammar that directly encodes precedence and associativity. In contrast, in this chapter we measure the cost of disambiguating deep priority conflicts by parsing programs that are known to be free of deep priority conflicts. Our lightweight data-dependent solution has negligible overhead to solve deep priority conflicts, furthermore, it is able to address more types of deep priority conflicts than Iguana.

The ALL(*) parsing algorithm of ANTLR [97] also handles operator precedence dynamically by means of semantic predicates. Because top-down parsers cannot handle left-recursive rules, the grammar is first rewritten to eliminate direct recursion using a technique known as *precedence climbing* [38]. Next, semantic predicates that are evaluated at parse time may filter invalid trees according to the order in which productions are defined in the grammar. The predicates are interwoven in the grammar productions representing the constraints to avoid producing invalid trees. In our case, the constraints are encoded in contextual non-terminals, as they indicate the trees that a non-terminal should not produce as its leftmost or rightmost child. Furthermore, we assume that the ANTLR solution to disambiguate deep conflicts have a bigger impact on performance than our lightweight data-dependency, more specifically when parsing programs without conflicts, as it uses similar techniques to data-dependent grammars.

Finally, Erdweg et al. [45] implemented a disambiguation strategy at parse time for SGLR, to support layout-sensitive languages. The disambiguation mechanism consists of propagating information about layout when constructing the parse trees, and enforcing constraints that are defined as attributes of productions in SDF grammars. While we only propagate information about leftmost and rightmost subtrees, their approach needs to propagate line and column positions of all terminal symbols that were used to construct a tree. In our case, we express constraints using sets of contextual symbols, checking them using set-algebraic operations. By using an optimized bitset representation, our approach achieves near zero-overhead when disambiguating deep priority conflicts. In contrast, Erdweg et al. state that their layout-sensitive parsing approach is practicable with an average slowdown of 1.8 x compared to a layout-insensitive solution. The authors mix enforcing constraints at parse-time and post-parse, compared to our solution that solely disambiguates deep priority conflicts at parse time.

4.6 CONCLUSIONS

In this chapter, we presented a novel low-overhead implementation technique for disambiguating deep priority conflicts with data-dependency. The approach was implemented in a scannerless generalized LR parser, and evaluated by benchmarking parsing performance of a corpus of popular Java and OCaml

projects on Github. Results show that our data-dependent technique cuts down the cost of disambiguating deep priority conflicts to 1 % to 2 %, improving significantly over contextual grammar rewriting strategies that have an overhead of 48 % to 77 %, as shown in Section 4.4. By using data-dependency selectively for just solving deep priority conflicts, we were able to reuse the (compact) LR parse tables of grammars that do not disambiguate deep conflicts, avoiding the typical problems of parse-table blowup of grammar rewriting strategies. Overall, we showed that declarative disambiguation can indeed be solved with almost no cost.

Part II

Declarative Syntax Definition

Declarative Specification of Layout-Sensitive Languages

5

ABSTRACT

In layout-sensitive languages, the indentation of an expression or statement can influence how a program is parsed. While some of these languages (e.g., Haskell and Python) have been widely adopted, there is little support for software language engineers in building tools for layout-sensitive languages. As a result, parsers, pretty-printers, program analyses, and refactoring tools often need to be handwritten, which decreases the maintainability and extensibility of these tools. Even state-of-the-art language workbenches have little support for layout-sensitive languages, restricting the development and prototyping of such languages.

In this chapter, we introduce a novel approach to declarative specification of layout-sensitive languages using *layout declarations*. Layout declarations are high-level specifications of indentation rules that abstract from low-level technicalities. We show how to derive an efficient layout-sensitive generalized parser and a corresponding pretty-printer automatically from a language specification with layout declarations. We validate our approach in a case-study using a syntax definition for the Haskell programming language, investigating the performance of the generated parser and the correctness of the generated pretty-printer against a corpus of 22191 Haskell files.

5.1 INTRODUCTION

Layout-sensitive (also known as indentation-sensitive) languages were introduced by Landin [80]. The term characterizes languages that must obey certain *indentation rules*, i.e., languages in which the indentation of the code influences how the program should be parsed. In layout-sensitive languages, alignment and indentation are essential to correctly identify the structures of a program. Many modern programming languages including Haskell [86], Python [100], Markdown [56] and YAML [20] are layout-sensitive. To illustrate how layout can influence parsing programs in such languages, consider the Haskell program in Figure 5.1, which contains multiple *do*-expressions:

In Haskell, all statements inside a *do*-block should be aligned (i.e., should start at the same column). In Figure 5.1, we know that the statement on line 7 (`guessValue x`) belongs to the inner *do*-block solely because of its indentation. If we modify the indentation of this statement, aligning it with the statements in the outer *do*-block, the program would have a different interpretation, looping indefinitely.

```
1 guessValue x = do
2   putStrLn "Enter your guess:"
3   guess <- getLine
4   case compare (read guess) x of
5     EQ -> putStrLn "You won!"
6     _  -> do putStrLn "Keep guessing."
7              guessValue x
```

Figure 5.1 *Do*-expressions in Haskell.

While layout-sensitive languages are widely used in practice, their tools are often handwritten, which prevent their adoption by language workbenches or declarative language frameworks. State-of-the-art solutions for declarative specification of layout-sensitive languages extend context-free grammars to automatically generate layout-sensitive parsers from a language specification, but are limited by their usability, performance and tooling support. For example, Adams [3] proposes a new grammar formalism called *indentation-sensitive context-free grammars* to declaratively specify layout-sensitive languages. However, this technique requires modifying the original symbols of the context-free grammar and, as result, may produce a larger grammar in order to specify certain indentation rules. Erdweg et al. [45] propose a less invasive solution using a generalized parser, requiring only that productions of a context-free grammar are annotated with *layout constraints*. In these constraints, language engineers are required to encode indentation rules, such as alignment or *Landin's offside rule*,[1] at a low-level of abstraction, that is, by comparing lines and columns. In both solutions, parsing may introduce a large performance overhead.

Both approaches ignore an essential tool in a language workbench: *pretty-printers*. Pretty-printers play an important role since they transform trees back into text. This transformation is crucial to developing many of the features provided by a language workbench, such as refactoring tools, code completion, and source-to-source compilers. Deriving a layout-sensitive pretty-printer from a declarative language specification is challenging as the pretty-printer must be *correct*, i.e., the layout used to pretty-print the program must not change the program's meaning.

In this chapter, we propose a novel approach to declaratively specifying layout-sensitive languages. We take a holistic approach by considering a domain-specific language to specify common indentation rules of layout-sensitive languages that is (a) general enough to support both parsing and pretty-printing, and (b) lets the user express indentation rules without resorting to low-level constraints in terms of lines and columns.

We make the following contributions.

[1] Landin introduced the *offside rule*, enforcing that in a program of a layout-sensitive language, all the subsequent lines of certain structures of the language should be "further to the right" than the first line of the corresponding structure. If the tokens of the subsequent lines occur further to the left than the first line, they are *offside*, and the structure is invalid.

- We define a domain-specific notation that concisely captures common patterns for indentation rules that occur in layout-sensitive languages (Section 5.3).
- We discuss our implementation of a layout-sensitive generalized parser with efficient support for parse-time disambiguation of layout constraints (Section 5.4).
- We present an algorithm for deriving correct layout-sensitive pretty-printers from grammars with layout declarations (Section 5.5).
- We evaluate the performance and correctness of our solution on a benchmark introduced by Erdweg et al. [45], exercising 22191 Haskell files (Section 5.6).

We cover related work in Section 5.7, discussing future work in Section 5.8, and concluding in Section 5.9.

5.2 BACKGROUND

In this section, we motivate our work on declarative specification of layout-sensitive languages by providing an overview of *layout constraints* [45], enumerating their shortcomings when used in a language workbench.

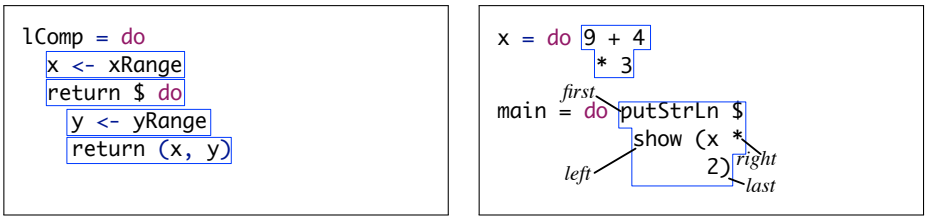
5.2.1 *Layout Constraints*

In layout-sensitive languages, indentation and alignment define the *shape* of certain structures of the language and the relationship between these shapes, such that for the structures to be valid, their shape must adhere to certain rules. A shape can be constructed as a box, with boundaries around the non-layout tokens that constitute the structure.

For example, consider the code from Figures 5.2a and 5.2b. In Figure 5.2a, because the list of statements inside a do-expression should be aligned, each shape indicating a single statement of the list must start at the same column. Similarly, if we consider that each statement in the do-expressions from Figure 5.2b should obey the offside rule, if the statement spans multiple lines, it must have a shape similar to ⌜⌟ (but not □ or ⌞⌝, for example).

Layout constraints can be used as annotations in productions of context-free grammars to enforce specific shapes into the source code of abstract syntax trees. Each tree exposes its shape given the location of four tokens in its original source code: `first`, `last`, `left`, and `right`—called *token selectors*—as shown in Figure 5.2b. The token selectors `first` and `last` access the position of the first and last tokens in a tree, respectively. The selector `left` selects the leftmost non-whitespace token that is not on the same line as the first token, whereas the selector `right` selects the rightmost non-whitespace token that is not on the same line as `last`.

Together with token selectors, a layout constraint may also refer to a specific indentation element of the source code—called *position selectors*—`line` and



(a) (b)

Figure 5.2 Boxes used to highlight the shape of subtrees in *do*-expressions.

`col`, which yield the token's line and column offsets, respectively. For example, a layout constraint `layout(x.left.col > x.first.col)` indicates that the subtree at position `x` should follow Landin's offside rule.

Note that constraints may also mention multiple subtrees in an annotated production, defining the relative position of these subtrees. That is the case in the constraint used to indicate that all statements inside a do-expression should be aligned, i.e., `layout(x.first.col == y.first.col)`. Finally, note that constraints may also be combined using the boolean operators *and* (`&&`), and *or* (`||`), and a constraint `ignore-layout` can be used deactivate layout validation locally.

5.2.2 *Tools for Layout-Sensitive Languages*

While layout constraints can be used to generate layout-sensitive parsers, there has been little adoption of such specifications by tools such as language workbenches.

Language workbenches enable agile development and prototyping of programming languages by generating an integrated development environment (IDE) from a language specification [51]. Therefore, one of the requirements for language specifications of layout-sensitive languages is related to the *usability* of the specifications, i.e., they must be declarative, concise and easy to use. Furthermore, when using an IDE, language users expect rapid feedback from the editor when editing their programs. Hence, the *performance* of the tools generated from a language specification is another important concern when using a language workbench to develop layout-sensitive programming languages. Finally, language workbenches go beyond parsing and code generation, providing many different features to language users, such as refactorings and code completion. Thus, another concern when developing a layout-sensitive language using a language workbench consists of specifying a *pretty-printer*, which transforms the abstract syntax tree of a program back into source code.

Below, we discuss the shortcomings of layout constraints against these requirements.

Usability. Layout constraints require annotating context-free productions to indicate how the source code corresponding to subtrees should be indented.

However, they are rather verbose and low-level, since they involve comparing lines and columns of tokens of different subtrees.

Parsing Performance. Generating tools from a language specification increases maintainability and extensibility, but usually comes with a penalty in performance. For example, Erdweg et al. [45] reported an overhead of about 80% when using a layout-sensitive generalized LR parser that uses layout constraints to disambiguate Haskell programs.

Pretty-printing. Layout constraints can be used to generate parsers, but it is not clear how to use them to automatically derive pretty-printers. One of the challenges when generating a pretty-printer for a layout-sensitive language is that the pretty-printer must be correct, i.e., pretty-printing a program should not change its meaning.

In the remainder of this chapter, we show how we tackle each of these concerns, such that language designers can develop layout-sensitive languages using tools such as language workbenches.

5.3 LAYOUT DECLARATIONS

To improve the usability of declarative specifications for layout-sensitive languages, we introduce *layout declarations*: high-level annotations in productions of a context-free grammar that enforce indentation rules on a specific node of the abstract syntax tree. Layout declarations abstract over token and position selectors, and provide a concise specification for most common indentation rules: *alignment* and *indentation* of constructs, and the *offside* rule. We also equip layout declarations with *tree selectors*, allowing them to be more readable than when using the position of the subtree involved in a declaration.

5.3.1 *Tree Selectors*

When writing the original layout constraints, one must use the position of the subtree in a production to enforce a constraint over this subtree. However, when reading and writing layout constraints, we want to avoid counting terminals and non-terminals in the production to identify to which tree the constraint applies.

Layout declarations allow the specification of constraints using *tree selectors*. Tree selectors may consist not only of the number of the subtrees, but also *literals* and *labeled non-terminals* that occur in the production. A labeled non-terminal is a non-terminal preceded by a label and a colon. Labels must be unique within a production, and if a literal occurs multiple times in the same production, then they must be referred by its position. For example, consider the following productions, written using SDF3 [133] syntax:[2]

[2]SDF3 productions have the form: `A.C = X1 X2 ... Xn {annos}`, where the symbol `A` represents a non-terminal, X_i represents either a terminal or a non-terminal, and the constructor `C` indicates the name of the node in the abstract syntax tree when imploding the parse tree. The list of annotations inside brackets `annos` can be used for different purposes, such as operator precedence disambiguation or to specify layout constraints.

```
Exp.Seq = exp1:Exp ";" exp2:Exp ";"
Exp.Add = exp:Exp "+" exp:Exp {left}
```

In the first production, the first `Exp` subtree might be referred in a layout declaration by its position (1) or by the label `exp1`. Considering the same production, the literals `";"` must be referred by their position, as they occur multiple times in the production. In the second production, the literal `"+"`, can be referred using the literal itself, as it is unique within the production. Finally, note that the underlined label in the second production is invalid, because the same label is used on the first `Exp` non-terminal.

5.3.2 *Alignment*

A common rule in layout-sensitive languages requires that certain structures must be aligned in the source code. For instance, as shown previously, all statements in a *do*-block of a Haskell program must be aligned, i.e., they must start at the same column. To express this indentation rule using layout constraints, one may use the following productions:

```
Exp.Do           = "do" StmtList
StmtList.Stmt    = Stmt
StmtList.StmtSeq = Stmt StmtList
  {layout(1.first.col == 2.first.col)}
```

Instead of using low-level concepts such as token and position selectors, we propose using high-level layout declarations `align` or `align-list` to indicate alignment of structures in the source code. A declaration `layout(align ref t)` enforces that a tree indicated by the tree selector `t` should start in the same column as the tree indicated by the `ref` tree selector, used as reference. Consider the example below, which uses an `align` declaration to indicate that the tail of the list `StmtList` should be aligned with the head of the list:

```
Exp.Do           = "do" StmtList
StmtList.Stmt    = Stmt
StmtList.StmtSeq = head:Stmt tail:StmtList
  {layout(align head tail)}
```

In SDF, lists may be represented by specific non-terminals (A+ or A*), which instructs the parser to flatten the tree structure corresponding to the list when constructing the abstract syntax tree. However, using layout constraints require explicitly defining productions for lists, which breaks this abstraction. The layout declaration `align-list` can be applied to list non-terminals to indicate that all elements in a list should start at the same column. Thus, one may write:

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

to indicate that the statements in the list should be aligned.

Semantics We define translation rules from layout declarations that describe alignment to layout constraints using token and position selectors. Consider the tree selectors `x` and `y`, the function *pos*(`t`), which obtains the position of a subtree indicated by selector `t`, the function *rename*(`X`, `Y`), which locally renames a non-terminal `X` to a non-terminal `Y`, and the following equations:

$$\frac{\texttt{align x y} \quad \textit{pos}(\texttt{x}) = \texttt{x'} \quad \textit{pos}(\texttt{y}) = \texttt{y'}}{\texttt{y'.first.col == x'.first.col}} \tag{5.1}$$

$$\frac{\texttt{align-list x} \quad \texttt{x} \text{ is a tree selector for } \texttt{A+} \text{ (or } \texttt{A*})}{\textit{rename}(\texttt{A+},\texttt{A'+})} \tag{5.2}$$

$$\texttt{A'+ = A'+ A layout(1.first.col == 2.first.col)}$$

Note that in Equation 5.2, using `align-list` enforces the layout constraint on the list `A+` (or `A*`), which could affect all occurrences of the list in the grammar. Therefore, we first locally rename this non-terminal `A+` to a non-terminal `A'+`, restricting the alignment declaration to the particular list in the production annotated with `align-list`. In Equation 5.1, on the other hand, the layout declaration can be directly translated to the layout constraint involving token and position selectors.

5.3.3 *Offside Rule*

As mentioned before, the offside rule is a common indentation rule applied in layout-sensitive languages. This rule requires that any character in the subsequent lines of a certain structure occur in a column that is further to the right than the column where the structure starts. For example, consider the following productions, which contains a layout constraint that requires that the source code for the `OffsideStmt` tree obey the offside rule:

```
Exp.Do = "do" Stmt
Stmt.OffsideExp = Exp
   {layout(1.left.col > 1.first.col)}
```

According to this rule, the expression in the following statement is invalid, since the second line starts at a column that is to the left of the column where the statement inside the do-expression starts:

```
do 18 + 8
* 3
```

In fact, any statement in which the multiplication sign is at the left of the digit `1` is invalid. By contrast, a valid program that satisfies the offside rule is:

```
do 18 + 8
    * 3
```

Instead of using layout constraints, one may use the `offside` layout declaration to achieve the same effect:

```
Stmt.OffsideExp = exp:Exp
  {layout(offside exp)}
```

The offside layout declaration can also be used to specify the relationship between the leftmost column of subsequent lines of a tree, and the initial column of another tree. For example, consider the following productions:

```
Exp.Do = "do" stmt:Stmt
  {layout(offside "do" stmt)}
Stmt.ExpStmt = Exp
```

With this declaration, the subsequent lines of `Stmt` should be in a column to the right of the column where the literal `do` starts. For example, even if we do not consider the offside rule for the inner statement, the following program is still invalid:

```
do 18 + 8
* 3
```

as the symbol `*` occurs at the same column as the keyword `do`, i.e., it is *offside*.

Semantics We show the semantics of layout declarations `layout(offside t)` and `layout(offside ref t)` by defining a translation into layout constraints using tokens and position selectors. Consider the following equations with `x` and `y` as tree selectors:

$$\frac{\texttt{offside x} \quad pos(\texttt{x}) = \texttt{x'}}{\texttt{x'.left.col > x'.first.col}} \tag{5.3}$$

$$\frac{\texttt{offside x y} \quad pos(\texttt{x}) = \texttt{x'} \quad pos(\texttt{y}) = \texttt{y'}}{\texttt{y'.left.col > x'.first.col}} \tag{5.4}$$

In Equation 5.3, the declaration `offside x` specifies that the `left` token of the tree `x` should be in a column further to the right than its `first` token. Similarly, in Equation 5.4, the `offside` declaration between the trees `x` and `y` specifies that the `left` token of `y` should be in a column further to the right than the `first` token of `x`.

5.3.4 *Indentation*

Another common pattern in layout-sensitive languages is to enforce indentation between subtrees. That is, a subtree should have its first token at a column to the right of the column of the first token of another subtree. Consider for example, the following productions:

```
Exp.Do = "do" stmt:Stmt
  {layout(indent "do" stmt)}
Stmt.ExpStmt = Exp
```

The declaration in the first production indicates that the statement should start further to the right than the `do` keyword. Thus, this declaration invalidates the following program:

```
do
18 + 8 * 3
```

On the other hand, the following program obeys the declaration, as the expression statement starts further to the right, when compared to the `do` keyword:

```
do 18 + 8 * 3
```

Similar to the `indent` layout declaration, the declaration `newline-indent` allows enforcing that a target subtree should start at a column further to the right than another subtree. Moreover, the latter declaration also enforces that the target subtree starts at a line below the line where the reference subtree ends. Thus, when considering this layout declaration, the program presented previously would also be invalid. A valid program would then be:

```
do
 18 + 8 * 3
```

Semantics The `indent` and `newline-indent` declarations are rewritten into layout constraints involving token and position selectors. Consider `x` and `y` tree selectors and the following equations:

$$\frac{\texttt{indent x y} \quad pos(\texttt{x}) = \texttt{x'} \quad pos(\texttt{y}) = \texttt{y'}}{\texttt{y'.first.col > x'.first.col}} \tag{5.5}$$

$$\frac{\begin{array}{c}\texttt{newline-indent x y} \quad pos(\texttt{x}) = \texttt{x'} \quad pos(\texttt{y}) = \texttt{y'} \\ \texttt{y'.first.col > x'.first.col \&\&} \\ \texttt{y'.first.line > x'.last.line}\end{array}}{} \tag{5.6}$$

Note that the layout declaration `newline-indent` requires a conjunction between two constraints involving the columns of the `first` tokens of both trees referenced by `x` and `y`, and the line of the `last` token of the tree `x` and the line of the `first` token of the tree `y`.

5.4 LAYOUT-SENSITIVE PARSING

Parsing layout-sensitive languages is difficult because these languages cannot be straightforwardly described by traditional context-free grammars. Such languages require counting the number of whitespace characters in addition to keeping track of nesting, which requires context-sensitivity. Therefore, most parsers for layout-sensitive languages rely on some ad-hoc modification to a handwritten parser. For example, the Python language specification describes a modified scanner that preprocesses the token stream, generating **newline**, **indent** and **dedent** tokens to keep track of when the indentation changes. Meanwhile, Python's grammar assumes these tokens to enforce the indentation rules of the language. In Haskell, an algorithm that runs between the lexer and parser converts implicit layout into explicit semicolons and curly braces to determine how the structures should be parsed by a traditional context-free grammar.

Because modifications to the parser vary from language to language, they are hard to implement when deriving a parser from a declarative language specification. Therefore, in this section, we propose a solution similar to Erdweg

et al.'s, which consists of deriving a scannerless generalized layout-sensitive LR parser (SGLR) from a language specification. Our algorithm improves on Erdweg et al.'s implementation by performing parse-time disambiguation of layout constraints, in contrast to post-parse disambiguation.

5.4.1 *Layout-Sensitive SLGR*

In theory, traditional context-free grammars can be used to generate a generalized parser for layout-sensitive languages. Since the parser produces a parse forest containing all possible interpretations of the program, this forest can then be traversed, such that only the trees that obey the indentation rules of the language are produced as result.

In practice this approach does not scale, since ambiguities caused by layout can grow exponentially [45], making it infeasible to traverse all trees in a parse forest produced when parsing a program of a layout-sensitive language. Thus, disambiguation of layout constraints at parse time should be preferred over post-parse disambiguation [126, 69].

We propose an implementation of a scannerless generalized LR parser (SGLR), that rejects trees that violate layout constraints at parse time. Our implementation calculates position information (line and column offsets for starting and ending positions) for token selectors (`first`, `last`, `left`, and `right`), propagating this information when building the trees bottom up, and using this information to evaluate layout constraints. The main difference between our implementation and the one proposed by Erdweg et al. [45] is that we evaluate all layout constraints at parse time, when building the subtrees, whereas in Erdweg et al.'s implementation, disambiguation using `left` and `right` constructs is performed after parsing (we discuss their implementation in more detail in Section 5.7).

Position Information The first modification we propose to add layout-sensitivity to the original SGLR algorithm [130] is to add position information to every tree node. That is, each node of the parse tree should contain the line and column at which it begins, and the line and column at which it ends. This information can be obtained from the parser, since it keeps track of the position in the source code when it starts and finishes parsing a structure. Besides that, our algorithm also calculates the position information for the `left` and `right` tokens of every tree node. We present the algorithm that constructs parse tree nodes in Figure 5.3.

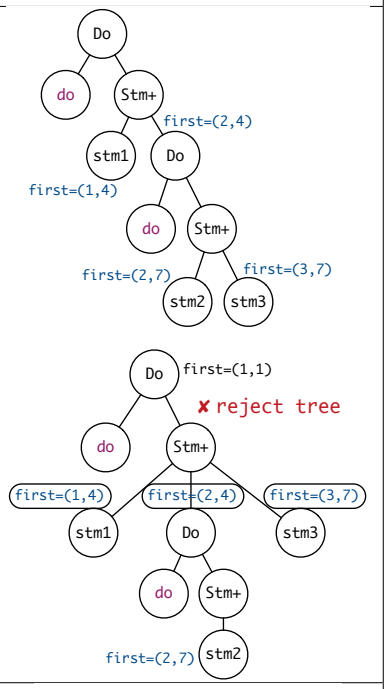
The algorithm propagates position information about token selectors based on the subtrees of the tree node being constructed. The method CREATE-TREE-NODE takes as arguments the production being applied, the list of trees that represent the subtrees of the node being created, and two `Position` variables `beginPos` and `curPos`, indicating the line and column where the tree starts and the line and column where the parser is currently at, respectively. The algorithm first constructs a tree node `t` given its list of subtrees, as shown in line 2. In lines 3 and 4, the information about the `first` and `last` tokens of `t` are assigned to the current node given the arguments `beginPos` and `endPos`.

```
1  function CREATE-TREE-NODE(Production A.C = X1 ... Xn,
       List<Tree> [t1, ..., tn], Position beginPos, Position
       curPos)
2    Tree t = [A.C = t1, ..., tn]
3    t.first = beginPos
4    t.last  = curPos
5    t.left  = null
6    t.right = null
7
8    // calculate left and right
9    foreach(ti in t) {
10     // should not consider layout
11     if (isLayout(ti))
12       continue
13     if (ti.left != null)
14       t.left = leftMost(t.left, ti.left)
15     if (ti.first.line > t.first.line)
16       t.left = leftMost(t.left, ti.first)
17
18     if (ti.right != null)
19       t.right = rightMost(t.right, ti.right)
20     if (ti.last.line < t.last.line)
21       t.right = rightMost(t.right, ti.last)
22   }
23   return t
24 end function
25
26 function leftMost(p1, p2) {
27   if (p1 == null || p1.col > p2.col)
28     return p2
29   else return p1
30 end function
31
32 function rightMost(p1, p2) {
33   if (p1 == null || p1.col < p2.col)
34     return p2
35   else return p1
36 end function
```

Figure 5.3 Pseudocode for the modified *CREATE-TREE-NODE* method from the original SGLR and the auxiliary functions *leftMost* and *rightMost*, in the implementation of the layout-sensitive SGLR.

The remainder of the algorithm computes the information about `left` and `right`. The algorithm calculates the position information about `left` by

Source Code	Layout Constraints	Trees

`do stm1`
`  do stm2`
`  stm3`
	`Exp.Do = "do" stmts:Stmt+`
	`  {layout(align-list stmts)}`
		

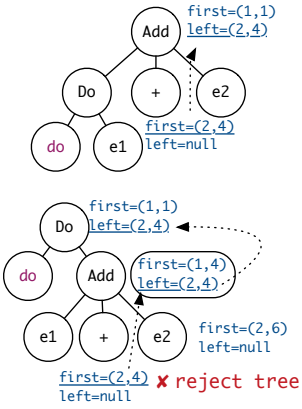
`do e1` `  + e2`	`Stmt.OffsideExp = exp:Exp` `  {layout(offside exp)}`	

Figure 5.4 Example of how our algorithm for a layout-sensitive SGLR constructs trees and applies layout constraints.

processing the list of subtrees, as its value should be the leftmost value (the one in the lowest line, and lowest column), when considering the `left` tokens of all subtrees that do not represent layout (line 14). However, if any subtree starts in a line that is below the line where `t` starts, the algorithm updates the `left` token of `t` accordingly (line 16). A similar strategy is applied to calculate the information about the `right` token.

Enforcing Layout Constraints The layout-sensitive SGLR algorithm works by rejecting trees that violate the layout constraints defined in a production using the information collected in the algorithm of Figure 5.3. A layout constraint is enforced at parse time when executing reduce actions in the parser, i.e., in the function DO-REDUCTIONS [130]. In layout-sensitive SGLR, a reduction is performed only when a production does not define a layout constraint, or when the layout constraint it defines is satisfied.

For example, the trees in Figure 5.4 indicate how the parser constructs tree nodes and verifies layout constraints. For the first program, the layout constraint states that the statements must be aligned. Therefore, since the second tree for this program does not satisfy this constraint, the tree is rejected as the parser does not perform the reduce action to construct it. In the second program, we can see how the information about the `left` is propagated. Similarly to the first example, the first tree constructed when parsing this program is the only one produced by the parser, since the second tree violates the offside rule.

5.4.2 *Propagation of `left` and `right` at Parse Time*

In the algorithm presented in Section 5.4.1, we propagate position information about `left` and `right` while building the parse tree. However, this approach may not produce the correct result in all scenarios. For example, consider a parse forest containing two different parse trees. Suppose that the source code for each tree in the parse forest is indicated by the programs below, where the symbols $\star$ represent actual characters in the program, and – represents a comment:

```
*****                    *****
 ****                    ----
  ***                     ***
```

Considering that both programs start at column 1, in the first tree, the `left` token is at column 2, whereas in the second tree, `left` is actually at column 3, because part of its source code is a comment. Thus, it is unclear what is the actual value for `left` when considering the parse forest, i.e., both trees simultaneously.

While this could be a problem when propagating position information about `left` and `right` tokens, and applying layout constraints at parse time, we believe that this scenario does not occur often in practice. As an alternative solution, we adapt our SGLR algorithm to fall back to post-parse disambiguation in such cases.

5.5 LAYOUT-SENSITIVE PRETTY-PRINTING

A pretty-printer is a tool that transforms an abstract syntax tree back into text. Pretty-printers are key components of language workbenches. For example, they can be used by other tools such as refactoring tools and code completion, or when defining source-to-source compilers. A lack of pretty-printing support

effectively prevents the adoption of language workbenches for layout-sensitive languages.

Pretty-printing programs in a layout-sensitive language is not an easy task. Because the layout in the source code identifies how the code should be parsed, the pretty-printer needs to be designed such that the meaning of the original program does not change after it is pretty-printed. Thus, in general, a pretty-printer is *correct* if the same abstract syntax tree is produced when parsing both the original and the pretty-printed programs. More formally, if we consider a program p and parsing and pretty-printing as two functions `parse` and `prettyPrint`, the following equation must hold:

$$\texttt{parse}(p) = \texttt{parse}(\texttt{prettyPrint}(\texttt{parse}(p)))$$

In this section we propose a technique to derive a correct pretty-printer based on a language specification containing layout declarations. We use strategies to apply modifications to the pretty-printed program, such that each layout declaration is considered while performing a top-down traversal in an intermediate representation of the abstract syntax tree.

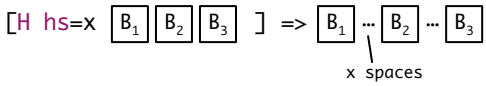
5.5.1 *From Trees to Boxes*

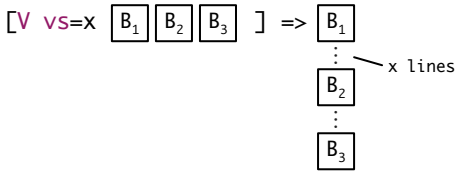
A naive implementation of a pretty-printer consists of printing the program separating each token by a single whitespace. However, it is easy to see that for a layout-sensitive language that enforces alignment, our naive pretty-printer would produce an invalid result as the pretty-printed program would not contain any newlines.

Manipulating this string directly to fix the layout according to the indentation rules of the language is also not ideal, as we lose the information about the structure of the program and the layout declarations encoded in the abstract syntax tree. Therefore, in order to produce an abstract representation of a program that takes into account the program structure and its layout, we use the Box language [25, 26] as an intermediate representation.

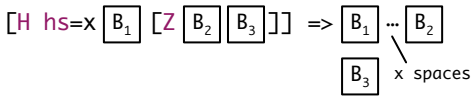
Boxes provide a structured representation of the pretty-printed text. Each node in the abstract syntax tree can be translated into a box, with its subtrees recursively translated into sub-boxes. The most basic boxes are string boxes, which can be composed (nested) using composition operators. Our approach considers three different composition operators in the Box language: vertical composition (`V`), horizontal composition (`H`) and z-composition (`Z`) [132].

The horizontal composition operator concatenates a list of boxes into a single line, whereas the vertical composition operator concatenates a list of boxes putting each box into a different line, starting at the same column. Each operator optionally takes an integer `hs` or `vs` as parameter to determine the number of spaces or empty lines separating each box, respectively. To illustrate, consider the examples below:





The z-composition operator places its boxes vertically on separate lines resetting the indentation of all boxes after the first to 0. Thus, for those boxes, the indentation from surrounding boxes is ignored and they start at the left margin. For example, if we combine the horizontal operator and the z-composition operator, we obtain the following output:



Boxes can be easily converted into text by recursively applying the box operators, as shown by the examples. Therefore, instead of manipulating the string produced by the pretty-printer, we manipulate boxes to enforce the layout declarations from the language specification.

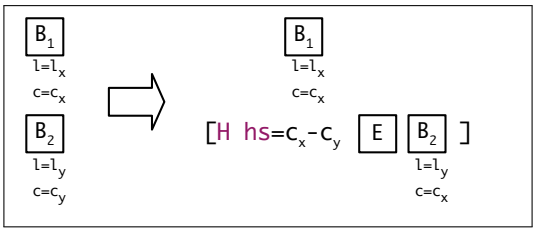
5.5.2 *Applying Layout Declarations to Boxes*

Boxes provide information about the layout of the program, retaining the structure of the abstract syntax tree. In order to apply layout declarations to the boxes generated from pretty-printing a tree, each box should also contain its relative line and column positions in the pretty-printed program. For example, consider the following Haskell program, pretty-printed from a naive pretty-printer, as discussed previously:

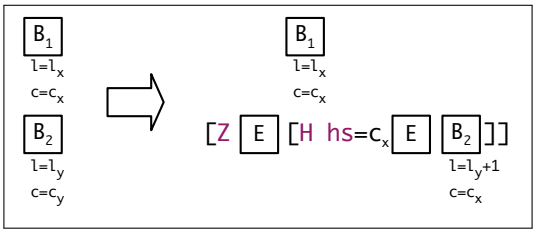
```
x = do s1 s2
```

One possible box representation for this program is:

To apply layout declarations to this program, we attach the relative line and column positions in the source code to the box (indicated by `l` and `c` in the diagram below). Furthermore, since boxes are created from the nodes in the abstract syntax tree, we also attach to the boxes the layout declarations from the corresponding node in the abstract syntax tree. Assuming that s_1 ends at column x, our pretty-printer produces the following boxes:

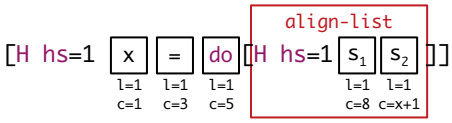


(a) $c_x > c_y$

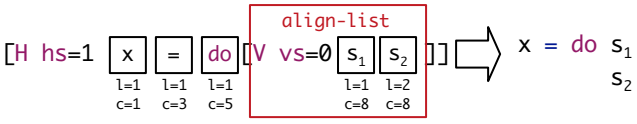


(b) $c_y > c_x$

Figure 5.5 Manipulating boxes to apply a layout declaration that enforces alignment between the boxes B_1 and B_2.



Transforming this box into a string and parsing that string results in a syntax error, since the statements inside the do-expression do not start at the same column. To ensure correct use of layout in the pretty-printed string, we apply a *layout fixer* that traverses the boxes and fixes the indentation where necessary. In this case, when considering an `align-list` layout declaration, the layout fixer replaces the inner horizontal operator by a vertical operator producing the following boxes and pretty-printed program:



which satisfies the layout declaration.

We adopt a similar strategy for adapting the boxes for the remaining layout declarations. For a layout declaration `align x y`, the left-most column of a box B_2 corresponding to the tree indicated by `y` should be equal to the left-most column of a box B_1 from the reference tree `x`. To satisfy this layout declaration,

if B_2 starts at a column to the left of the starting column of B_1, our layout fixer wraps B_2 in a horizontal operator, using an empty box (a box `E` containing the empty string), setting `hs` as the number of spaces necessary to align the two boxes. For the case where B_2 starts at a column further to the right than the left-most column where B_1 starts, the layout fixer uses a combination of a z-operator and a horizontal operator to skip to the next line, adding the indentation necessary to align both boxes. Both scenarios are illustrated in Figure 5.5. Note that empty boxes allow indenting other boxes (using the horizontal operator) or moving them to a new line (using the z-operator).

The same strategy can be used for the layout declarations `indent x y`, and `newline-indent x y`, setting the horizontal box such that the boxes are not aligned, but that the left-most column of B_2 is to the right of the left-most column of B_1, enforcing a z-operator whenever it is necessary to print the text into another line.

For offside declarations, we apply a slightly different approach. Because an offside declaration requires that the boxes in the subsequent lines should be further to the right than the column where the structure starts, we verify the operands of the z-operator. That is, for all boxes that move to a newline due to a z-operator and violate the offside rule, we use horizontal composition with an empty box to indent them such that the offside rule is satisfied.

We apply these strategies in a top-down traversal of the boxes that represent the original program. This approach produced satisfactory results when considering the Haskell programs in our benchmark as we will discuss in Section 5.6.2.

5.5.3 *Layout Declarations for Pretty-printing*

In this chapter, we focus primarily on the correctness of a generated pretty-printer, but pretty-printing the program in a single line, adding newlines only to enforce layout declarations may not produce a *pretty*-printer. In layout-sensitive languages, concepts such as alignment, indentation and even the offside rule contribute to make the pretty-printed code prettier, i.e., more readable. However, these are *not sufficient* to determine a pretty layout. For example, consider the following production defining an if-else construct, with layout declarations to enforce the alignment of the *then* (`T`) and *else* (`E`) branches:

```
S.IfElse = "if" E "then" T:S "else" E:S {layout(align T E)}
```

A pretty-printed program using this production and the layout fixing algorithm looks like:

```
if e1 then s1 else
           if e2 then s2 else
                      s3
```

While this program is correct according to the layout declaration, one may say it is not pretty, as its layout may not make the program more readable, specially if we would consider writing programs with nested if-else statements.

The declarations from Section 5.3 are always enforced when parsing the program. However, for constructs that are not layout-sensitive, we could use a more flexible approach, using declarations only to produce better pretty-printers. Thus, we introduce *pretty-printing layout declarations*, which are similar to the previous ones, but are used *only* for pretty-printing. Layout declarations for pretty-printing start with the prefix `pp-`, and are ignored by the parser.

With pretty-printing layout declarations, the language designer can generate prettier pretty-printers, but still allow flexible layout when parsing the program. For example, consider the same production as the one shown previously, with additional pretty-printing layout declarations:

```
S.IfElse = "if" E "then" T:S "else" E:S
  {layout(pp-newline-indent "if" T && pp-align "if" "else" &&
  align T E)}
```

Applying the pretty-printer generated from this production into the same program, produces:

```
if e1 then
 s1
else
 if e2 then
  s2
 else
  s3
```

Note that the pretty-printed program using only the `align` declaration would also be accepted by the same parser defined by the production above, since the additional layout declarations are used only for pretty-printing.

To provide more flexibility to language designers regarding indentation sizes and newlines, we also introduce the declaration `pp-newline-indent-by(x)` and `pp-newline(x)`. The declaration `pp-newline-indent-by(x)` is a variation of the declaration `pp-newline-indent`, such that it is possible to specify the number of spaces (using the integer `x`) that pretty-printer must consider when indenting the program. The declaration `pp-newline(x) t`, on the other hand, enforces that the tree `t` starts on a newline, indented by `x` spaces from the enclosing indentation.[3]

For instance, if instead we use the layout declaration `layout(pp-newline(1) T && pp-newline "else")` on the same production, it is possible to construct a pretty-printer that produces the following program:

```
if e1 then
 s1
else if e2 then
 s2
else s3
```

[3] If `x = 0`, the declaration `pp-newline` can be used instead.

5.6 EVALUATION

In this section we evaluate our approach for generating a parser and a pretty-printer from a grammar containing layout declarations. We are interested in answering the following research questions.

RQ1 How parse-time disambiguation of ambiguities due to layout affects the performance of a generalized parser?

RQ2 What is the accuracy of our layout fixer when pretty printing files of a layout-sensitive language?

RQ3 How easy is it to specify a layout-sensitive language?

In order to answer the these research questions, we generate a parser and a pretty-printer derived from a declarative specification for Haskell containing layout declarations. We apply both generated parser and pretty-printer to 22191 Haskell programs from the Hackage[4] repository, using the benchmark described in [45]. We used the files in the same benchmark to provide a fair comparison between the performance of our parser and their implementation.

In order to measure the performance overhead of the layout-sensitive parser, we use a pretty-printer tool, part of the `haskell-src-exts` package[5], which has an option to pretty-print programs using only explicit grouping (brackets and semicolons). We also preprocess files using the C preprocessor part of the Glasgow Haskell Compiler (GHC) supporting additional extensions to increase the coverage of files. The diagram in Figure 5.6a illustrates the process we adopted.

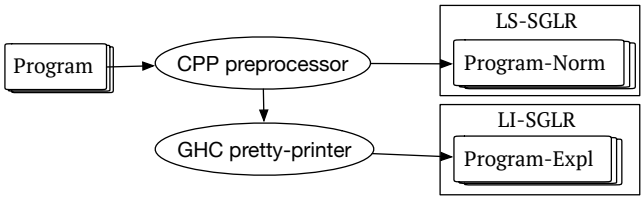
To measure the performance of our layout-sensitive parser (`LS-SGLR`) on the original program, we first apply the C preprocessor, applying the parser to the `Program-Norm` file. Similarly, we measure the performance of an implementation of SGLR without support for layout-sensitive disambiguation (`LI-SGLR`) on a program that contains brackets and semicolons to explicitly delimit structures (`Program-Expl`), using the pretty-printer from the `haskell-src-exts` package. We then compare the performance of both parsers to verify the overhead of using the layout-sensitive features of our implementation.

To measure the correctness of the pretty-printer generated using our approach, we use the process described in Figure 5.6b. First, we preprocess the file using the C preprocessor, generating the file `Program-Norm`. Next, we parse this file and pretty-print its abstract syntax tree using our pretty-printer to generate a new program `Program-PP`. Finally, we parse this file comparing its tree with the tree originated from the file `Program-Norm`.

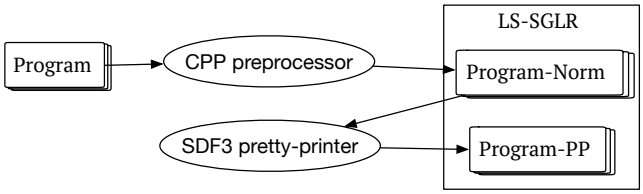
We measure how easy it is to specify a layout-sensitive language by counting the total number of layout declarations used in the grammar.

[4] `http://hackage.haskell.org`

[5] `http://hackage.haskell.org/package/haskell-src-exts`



(a) Evaluating the performance of the parser.



(b) Evaluating the correctness of the pretty-printer.

Figure 5.6 Evaluation setup.

5.6.1 *Experimental Setup*

We executed the benchmarks on a computer with 16GB RAM, and an Intel Core i7 CPU with a base frequency of 2.7GHz and a 6MB Last-Level Cache. The software consisted of Apple's macOS version 10.13.5 (17F77) and Oracle's Java Virtual Machine version 1.8.0_102.

We measured the execution time of batch-parsing the corpus of Haskell programs using the Java Microbenchmarking Harness (JMH), which is a framework to overcome the pitfalls of (micro-)benchmarking. When executing the benchmarks, we disabled background processes as much as possible, fixing the virtual machine heap size to 8 GB. We configured JMH to perform 5 warmup iterations, and 10 measurements, calculating the average time of each execution. We use the same settings to test the correctness of the pretty-printer, however, instead of using JMH, we simply compare Java objects corresponding to the abstract syntax tree of the programs `Program-Norm` and `Program-PP`.

5.6.2 *Experiment Results*

Performance of the Parser The results showing the *parse-time* of the LS-SGLR parser on programs with original layout, and the original SGLR parser on programs with explicit layout are presented in Table 5.1. Overall, we measured the overhead of our layout-sensitive parser to be 1.72x. This compares to 1.80x for Erdweg et al.'s implementation. Because Haskell programs may still require an additional post-parse disambiguation to disambiguate longest-match constructs [86, 45], we suspect that part of this overhead is caused by

Parser	Data Set	Time (seconds)	Overhead
`LS-SGLR`	`Program-Norm`	638.05 ± 1.96	1,72x
`LI-SGLR`	`Program-Expl`	370.26 ± 0.68	—

Table 5.1 Benchmark results when executing our LS-SGLR parser on programs containing their original layout, and the LI-SGLR parser on programs containing explicit layout.

Parser	Data Set	Time (seconds)	Overhead
`LS-SGLR`	`Program-Norm`	239.79 ± 0.90	1.53x
`LI-SGLR`	`Program-Expl`	156.37 ± 0.56	—

Table 5.2 Benchmark results when considering a subset of 14830 programs that do not have longest-match ambiguities.

this additional disambiguation step, since programs with explicit layout do not present such ambiguities. For this reason, we also ran the same experiment on programs that *do not* contain longest-match ambiguities (14830 programs), measuring the overhead of disambiguating only ambiguities due to layout. As shown in Table 5.2, for such programs our parser presented an overhead of 1.5x.

Correctness of the pretty-printer When executing our pretty-printer, we verified that only 6 out of 22191 programs produced incorrect results (0.03 %). Because of the low number of cases, we investigated these programs manually and verified that because we apply our layout-fixer using a top-down traversal, a ripple effect when fixing a layout declaration may disrupt parts of the source code that have been previously fixed.

Language specification The SDF3 grammar for Haskell used in our experiments contains 473 productions. It was necessary to annotate 34 productions to specify the indentation rules for Haskell. In total, we added 43 layout declarations, being 10 `offside`, 1 `align`, 5 `align-list`, 19 `indent`, and 8 `ignore-layout` declarations. Note that some productions required multiple declarations.

5.6.3 *Threats to Validity*

A threat to external validity, with respect to the generality of our results, is that we used only Haskell in our benchmarks. Despite not being able to generalize our results beyond Haskell programs, we believe that Haskell has indentation rules that are similar to other layout-sensitive languages. We have also tried our approach on a syntax definition for a subset of Python. However, because we do not cover the entire language, we could not parse many real-world programs, and decided to not include it in our benchmarks.

Another threat to the validity of our results concerns the correctness of our parser. To tackle this issue, we verified that the abstract syntax trees

we obtained from our parser and the trees from the implementation done by Erdweg et al. were equal. Erdweg et al. checked the correctness of their parser by comparing it with to the parser from GHC. Since they obtained positive results from that comparison, we believe that our parser is also correct.

5.7 RELATED WORK

In this section, we highlight previous work on layout-sensitive parsers and generating pretty-printers from a declarative specification, presenting the main differences with our work, and discussing how prior work inspired us.

5.7.1 *Layout-Sensitive Parsing*

As we mentioned previously, our approach to derive a layout-sensitive parser from a declarative specification was inspired by the work by Erdweg et al. [45]. Their parser performs post-parse disambiguation to avoid splitting parse states that were already merged when finding an ambiguity, which would degrade the performance of the parser. We eliminate the necessity of merging these states by propagating the information about token selectors at parse time, preventing invalid trees from being constructed in the first place, and improving the performance of our parser as it is not necessary to disambiguate such trees after parsing.

Indentation-sensitive context-free grammars (IS-CFGs) [3], can be used to generate LR(k) or GLR layout-sensitive parsers. In IS-CFGs each terminal is annotated with the column at which it occurs in the source code, i.e., its indentation, and each non-terminal is annotated with the minimum column at which it can occur. To express alignment of constructs, an IS-CFG requires additional productions, which are generated automatically from certain non-terminals. We opted for not modifying the original grammar, only requiring that productions are annotated with layout declarations. While our approach is based on a scannerless generalized parser, we obtained similar performance results to a layout-sensitive LR(k) parser generated from an IS-CFG when considering Haskell programs with longest-match ambiguities. Finally, it is not clear how to automatically derive a pretty-printer from an IS-CFGs, whereas we provided a mechanism to derive a pretty-printer from a specification with layout declarations.

Afroozeh and Izmaylova [7] use data-dependent grammars [63] to generate a layout-sensitive parser. They propose high-level declarations such as `align` and `offside` that are translated into equations, which are evaluated during the execution of a generalized LL parser. In our work, we opted to leave the grammar intact and have layout declarations as annotations on productions. In contrast, their declarations are intermingled with the non-terminals in productions, which decreases readability. Finally, their approach also requires propagating data "upwards" and "downwards" when building tree nodes, whereas we propagate data only upwards.

Brunauer and Mühlbacher [32] propose another approach to declaratively specify layout-sensitive languages using a scannerless parser. They modify the non-terminals of the grammar to include integers as parameters, which are mixed with the grammar productions to indicate the number of spaces that must occur within certain productions. However, these changes have a detrimental effect on the readability and on the size of the resulting grammar. We opted to abstract over details such as number of spaces, columns, and lines, by using high-level layout declarations.

5.7.2 *Pretty-printing*

Many solutions have been proposed to integrate the specification of a parser and a pretty-printer [101, 132, 22]. However, none of these solutions is aimed at generating layout-sensitive parsers *and* pretty-printers using the same specification. For instance, the syntax definition formalism SDF3 [132] allows the specification of a parser and a default pretty-printer to be combined by using *template productions*. Template productions are similar to regular productions, but the indentation inside the template is considered only when pretty-printing the program. Thus, they are similar to our layout declarations for pretty-printing as they do not enforce any restriction with respect to layout while parsing. However, when using templates in combination with layout declarations to generate layout-sensitive parsers, any inconsistency between the templates and the declarations might result in an incorrect pretty-printer.

Different approaches have been proposed to derive prettier [138], and correct-by-construction [40] pretty-printers. However, these approaches are aimed at traditional programming languages, and might require further adaptations to be applied to layout-sensitive languages. Finally, none of these approaches allow a specification of the pretty-printer that can be derived from the context-free grammar. We use layout declarations as annotations to context-free productions to indicate how structures should be pretty-printed such that the pretty-printed program obeys the indentation rules of the language. Furthermore, our pretty-printing layout declarations enable customizing the generated pretty-printer such that it also produces prettier results.

5.8 FUTURE WORK

As future work we plan to apply our techniques to more layout-sensitive languages, examining their indentation rules to observe how our generated parser and pretty-printer behave in other scenarios. We also would like to investigate different strategies to apply our layout fixer, finding alternatives that do not cause a ripple effect when applying (pretty-printing) layout declarations, as it may produce incorrect results. Furthermore, we would like to study the integration between our pretty-printing layout declarations and other syntax definition formalisms that enable declarative specification of both parser and pretty-printer, such as SDF3.

Another aspect to consider is preservation of comments when pretty-printing layout-sensitive programs. Currently, our pretty-printer discards comments altogether, but ideally, comments should be preserved while maintaining the correctness of the pretty-printer. Preservation of comments in transformations is challenging even for traditional languages, and most approaches rely on heuristics [66, 85].

Finally, we propose a more in-depth analysis of SGLR mechanisms to disambiguate longest-match constructs. As shown by our experiment, such ambiguities are responsible for a considerable fraction of the overhead of our parser for Haskell. It would also be interesting to study how layout-sensitive and longest-match disambiguation are related to operator precedence disambiguation [8, 119].

5.9 CONCLUSION

In this chapter, we presented an approach to support declarative specifications of layout-sensitive languages. We tackled the main issues that prevent the adoption of these languages in tools such as language workbenches: usability, performance and tool support. We introduced layout declarations, providing language designers with a high-level specification language to declare indentation rules of layout-sensitive languages. Furthermore, we described a more efficient implementation of a scannerless layout-sensitive generalized LR parser based on layout declarations. Finally, we presented strategies to derive a correct pretty-printer, which produced the correct result for almost all of the programs in our benchmark. Overall, we believe that our work can be used to facilitate the development and prototyping of layout-sensitive languages using tools such as language workbenches.

Principled Syntactic Code Completion

6

ABSTRACT

Principled syntactic code completion enables developers to change source code by inserting code templates, thus increasing developer efficiency and supporting language exploration. However, existing code completion systems are ad-hoc and neither complete nor sound. They are not complete and only provide few code templates for selected programming languages. They also are not sound and propose code templates that yield invalid programs when inserted. This chapter presents a generic framework that automatically derives complete and sound syntactic code completion from the syntax definition of arbitrary languages. A key insight of our work is to provide an explicit syntactic representation for incomplete programs using placeholders. This enables us to address the following challenges for code completion separately: (i) completing incomplete programs by replacing placeholders with code templates, (ii) injecting placeholders into complete programs to make them incomplete, and (iii) introducing lexemes and placeholders into incorrect programs through error-recovery parsing to make them correct so we can apply one of the previous strategies. We formalize our framework and provide an implementation in Spoofax.

6.1 INTRODUCTION

Code completion, also known as content completion or content assist, is an editor service that proposes and performs expansion of the program text. Code completion helps the programmer to avoid misspellings and acts as a guide to discover language features and APIs. Most mainstream integrated development environments (IDEs) provide some form of code completion and industrial studies indicate that code completion is one of the most frequently used IDE services [12].

There are two classes of code completion: syntactic and semantic. Syntactic code completion considers the syntactic context at the cursor position and proposes code templates for syntactic structures of the language. For example, most IDEs for Java support syntactic code completion with class and method templates. Semantic code completion also uses the cursor position to propose templates, but by applying semantic analysis to the program, the IDE can propose code templates or identifiers that do not violate the language's name binding or typing rules. For example, in this case IDEs for Java may suggest variables or methods that are visible in the current scope and have the expected type at the cursor position.

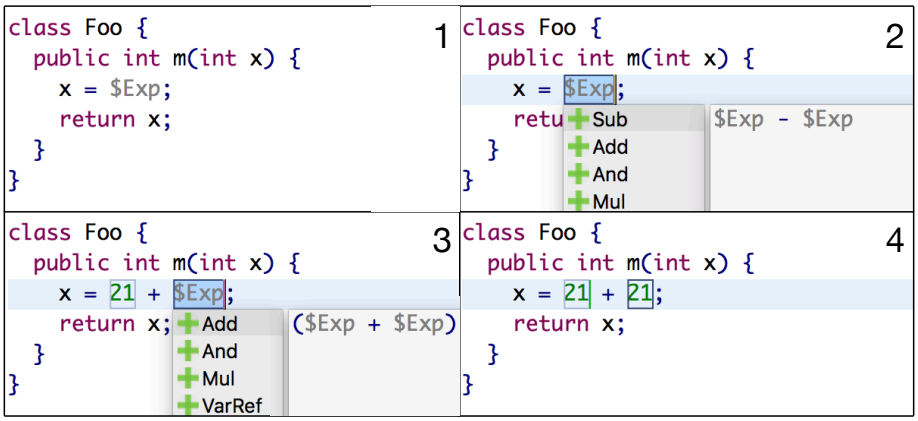


Figure 6.1 (1) Incomplete program with explicit placeholders. (2) Triggering completion for a placeholder. (3) After selecting a proposal, showing completions for nested placeholders. (4) Completing a nested placeholder by typing.

In our work, we focus on *syntactic code completion*. Even for mainstream languages in mainstream IDEs, syntactic code completion is often ad-hoc and unreliable. Specifically, most existing services for syntactic code completion are incomplete and only propose code templates for selected language constructs of a few supported languages, thus inhibiting exploring the language's syntax. Moreover, most existing services are unsound and propose code templates that yield syntax errors when inserted at the cursor position.

To address these shortcomings, we present a generic code-completion framework that derives sound and complete syntactic code completion from syntax definitions. From the syntax definition, we derive code templates and applicability conditions for them to ensure soundness. To support completeness and propose all language structures, we represent incomplete program text explicitly using placeholders that we automatically introduce into the syntax definition. This allows our code templates to yield incomplete programs that can be subsequently completed.

Figure 6.1 illustrates our use of placeholders in a Java-like program. The first box shows an incomplete program with an expression placeholder. The program is syntactically correct since we introduce the placeholder as part of the language. The second box shows that placeholders give rise to completion proposals, which may themselves be incomplete (contain placeholders). After selection of a proposal, the developer can expand or textually replace the placeholders inserted by the template.

In addition to enabling step-wise code completion, explicit placeholders allow us to address two important practical challenges of code completion: inferring completion opportunities in complete program texts and generating completion proposals while recovering from syntax errors. Complete programs do not contain placeholders, yet code completion can be useful for adding list

elements or optional constructs. For example, we may want to use syntactic code completion to add modifiers like `public` to a method or to add statements to a method's body. Instead of developing such support for complete programs directly, we provide our solution using placeholder inference (to make the program incomplete) followed by regular syntactic code completion of the inferred placeholder.

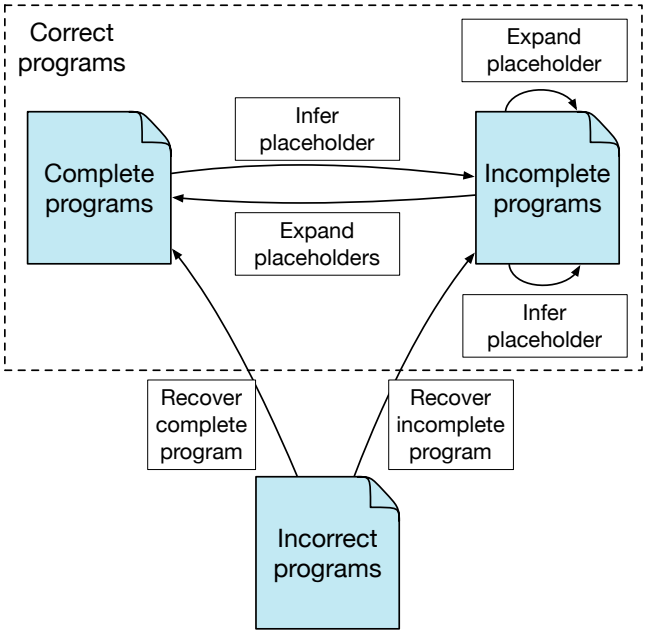


Figure 6.2 Separation of concerns in code completion.

Incorrect programs contain syntax errors but are important to support because incorrect programs occur frequently during development. Again, instead of developing syntactic code completion for incorrect programs directly, we decompose this activity. We use error-recovery parsing [65, 41] to insert lexemes into the program text. However, since we made placeholders part of the language, an error-recovering parser will also consider placeholders for insertion, thus yielding incomplete programs. A developer can select one of multiple alternative recoveries and can use regular syntactic code completion for placeholders in the selected recovered program. Figure 6.2 shows all transitions between complete, incomplete, and incorrect programs.

We present a formalization of our completion framework and the involved algorithms. We describe completeness, formally define soundness, and prove soundness for our algorithms. We also implemented our framework as part of the Spoofax language workbench [68], which we used to derive syntactic code completion for a subset of Java containing classes, methods, statements and expressions, Pascal, and IceDust [57], a domain-specific language for data modeling.

In summary, we make the following contributions:

- An analysis of syntactic code completion in IDEs and language workbenches, revealing completeness and soundness issues (Section 6.2).
- A sound and complete approach for completing incomplete programs by rewriting placeholders (Section 6.3).
- An algorithm for inferring placeholders, yielding support for expanding complete programs (Section 6.4).
- An extension of error-recovery parsing for inserting placeholders, yielding syntactic code completion for incorrect programs (Section 6.5).
- Throughout, we develop a formal framework for reasoning about syntactic code completion and we describe how we realized the algorithms in Spoofax.

6.2 STATE OF THE ART OF SYNTACTIC COMPLETION

In this section, we motivate our work on syntactic code completion by presenting examples collected from state-of-the-art IDEs and language workbenches. We observe that state-of-the-art solutions are unsound, incomplete, language-dependent, and do not support incorrect programs well.

Soundness Existing IDEs and language workbenches often propose unsound completions that yield syntax errors when inserted. For example, as shown in Figure 6.3, Eclipse largely ignores the syntactic context at the cursor position and proposes the insertion of an *else*-block without a corresponding *if*-statement, yielding a syntax error after insertion.
IntelliJ provides code templates that are sensitive to the syntactic and typing context, but may yield syntax errors nonetheless. For example, the live template `lst` inserts an expression for fetching the last element of an array, but yields the invalid fragment `[.length - 1]` when no array is available in the current scope. Language workbenches have similar issues and also propose invalid completions. For example, as shown in Figure 6.4, Xtext [43] only proposes the next keyword, but not a complete *def*-template.

Completeness IDE developers define code templates manually. As a consequence, the set of available templates is limited. Many language constructs are not available through syntactic code completion, and changes to a language are often not reflected in the code templates. For example, the proposal list of Eclipse 4.5.2 shown in Figure 6.3 does not provide a code template for *try-with-resource* statements.

Besides missing templates, existing IDEs and language workbenches also have no way to represent partial completions that users can subsequently complete to form complex constructs. Instead, existing systems always generate complete programs with concrete "dummy" constructs as subexpressions. For example, as shown in Figure 6.5, Eclipse's proposal for constructing and storing

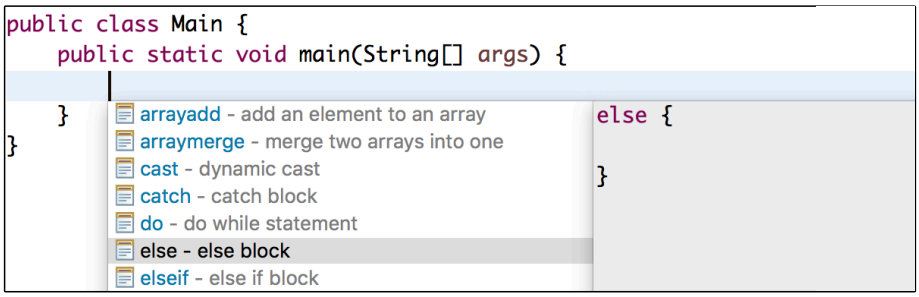


Figure 6.3 Eclipse: Unsound completion yields syntax error.

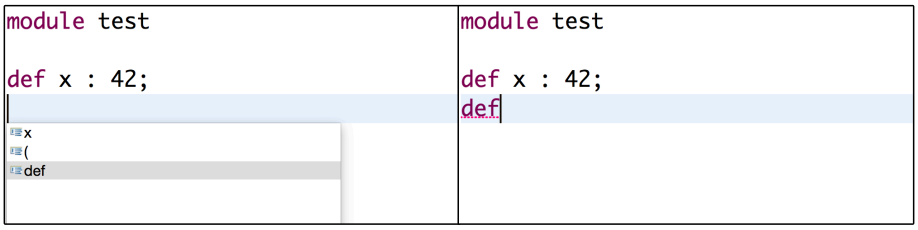


Figure 6.4 Xtext: Unsound completion yields syntax error.

a new object yields a complete program using `type` as a "dummy" class name and `name` as a "dummy" variable name.

If the developer leaves the IDE's completion mode, the "dummy" constructs become part of the program. This inhibits partial completions such as for assignment statements `type name = exp`, where `exp` will be interpreted as a variable reference rather than as a placeholder for arbitrary expressions.

Incorrect Programs In current IDEs and language workbenches, error-recovery parsing and code completion are largely orthogonal. For example, Eclipse uses error-recovery parsing to identify the syntactic context at the cursor position and compute corresponding proposals. However, Eclipse does not actually offer support for recovering from the syntax error itself. Similarly, code completion and error recovery are orthogonal in IntelliJ and the previous version of the Spoofax language workbench [68, 65]. Instead, error recovery should yield a list of alternative recovery proposals for the user to select from. If the user selects an incomplete program as recovery, the user can continue to expand the program in subsequent code completion steps.

Summary Based on this discussion, we derive the following requirements for principled syntactic code completion:

- Proposals need to be *sound* such that code completion does not introduce syntax errors.

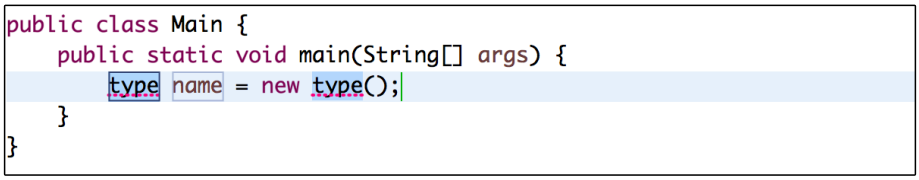


Figure 6.5 Eclipse: Completion with “dummy” constructs.

- Proposals need to be *complete*, meaning that code templates exist for all language constructs and that developers can use iterative code completion.
- Code completion should propose *recoveries* for incorrect programs and allow the iterative completion of recovered programs.

In the remainder of this chapter, we present a generic framework for syntactic code completion that satisfies these requirements. Our framework is language-independent and automatically derives principled code-completion support from a language’s syntax definition.

6.3 COMPLETION BY REWRITING PLACEHOLDERS

In most editors, programs in an incomplete state are incorrect, as they contain syntax errors indicating missing elements in the source code. In this section, we present a formal model for syntactic code completion for a subset of incomplete programs where missing elements correspond to entire structures from the language. We introduce a valid representation for this subset, representing these structures by explicit placeholders. As the programs in the subset are syntactically correct considering our representation, our framework models sound and complete syntactic code completion deriving completion proposals as placeholder expansions. Finally, we present an instantiation of our formal model, describing our implementation of syntactic code completion in Spoofax.

6.3.1 *Representing Complete and Incomplete Programs*

We consider abstract syntax trees as our primary program representation. We define representations for complete and incomplete programs as terms over a signature Σ:

Definition 1 (Signature). *A signature $\Sigma = \langle S, C \rangle$ is a pair consisting of a set of sorts $s \in S$ and a set of constructor declarations $(c : s_1 \times \ldots \times s_n \rightarrow s_0) \in C$ with zero or more arguments and all $s_i \in S$. The set of sorts must contain the predefined sort* LEX $\in S$ *for representing lexemes.*

Given a signature $\Sigma = \langle S, C \rangle$, we define the well-formed terms T_Σ over Σ as follows:

Definition 2 (Well-formed terms)**.** *For each sort $s \in S$, the set of well-formed terms T_Σ^s of sort s is the least set such that*

$$\frac{s \text{ is a string literal}}{s \in T_\Sigma^{\mathsf{LEX}}} \tag{6.1}$$

$$\frac{(c : s_1 \times \ldots \times s_n \to s) \in C \quad t_i \in T_\Sigma^{s_i} \quad \forall 1 \leq i \leq n}{c(t_1, \ldots, t_n) \in T_\Sigma^s} \tag{6.2}$$

The family T_Σ of well-formed terms is then defined as

$$T_\Sigma = (T_\Sigma^s)_{s \in S}.$$

Well-formed terms represent complete programs over Σ. For example, given a signature for a statement in an imperative programming language, the term

```
Assign("x", Add(Int("21"), Int("21")))
```

represents a complete program.

We represent incomplete structures in programs by means of explicit placeholders. We introduce an explicit placeholder $\$s$ for each sort s as a nullary constructor:

Definition 3 (Placeholders)**.** *Given a signature $\Sigma = \langle S, C \rangle$, we define placeholders as the set of nullary constructor declarations*

$$S^{\$} = \{\$s : s \mid s \in S\}$$

and the placeholder-extended signature

$$\Sigma^{\$} = \langle S, C \cup S^{\$} \rangle.$$

Well-formed terms over an extended signature $\Sigma^{\$}$ represent incomplete programs. For example, the term

```
Assign("x", $Exp)
```

represents an incomplete program, where we use the placeholder `$Exp` of sort `Exp` instead of a concrete argument term for `Assign`. According to our definition, every complete program is also an incomplete program. However, a program is properly incomplete if $t \in T_{\Sigma^{\$}}$ and $t \notin T_\Sigma$, that is, t contains at least one placeholder.

6.3.2 *Terms with Source Regions*

The goal of our formalization is to provide a formal framework for syntactic code completion. Since code completion is sensitive to the cursor position in the source code, we need to extend our representation of terms with source regions. This will later enable us to map the cursor position to the corresponding subterm.

A term's source region identifies the region of the original source file to which the term corresponds. Later, when we use code completion to synthesize terms, we will also need empty source regions that have no correspondence in the source file.

Definition 4 (Source region)**.** *A source region r is an interval $[m, n] = \{x \in \mathbb{N} \mid m \leq x \leq n\}$ starting at character offset m and ending at character offset n. We define $r_1 < r_2$ to mean $\forall x_1 \in r_1.\ \forall x_2 \in r_2.\ x_1 < x_2$.*

Note that $\emptyset$ denotes an empty source region of the source file. We use $\emptyset$ to denote the source region of synthesized terms. Note furthermore that $r_1 < r_2$ expresses that r_1 precedes r_2 and the two regions may not touch and not overlap. Finally, the empty region is not affected by the ordering, $\emptyset < r$ and $r < \emptyset$ for all r.

We define an augmented set of well-formed terms that associates a source region with each subterm:

Definition 5 (Well-formed terms with source regions)**.** *For each sort $s \in S$, the set of well-formed terms with source regions $T_\Sigma^{R,s}$ of sort s is the least set such that*

$$\frac{s \text{ is a string literal}}{s^r \in T_\Sigma^{R,\mathsf{LEX}}} \tag{6.3}$$

$$\frac{\begin{array}{c}(c : s_1 \times \ldots \times s_n \to s) \in C \\ \forall\, 1 \leq i \leq n : t_i^{r_i} \in T_\Sigma^{R,s_i} \\ \forall\, 1 \leq i < j \leq n : r_i < r_j \\ r_1 \cup \cdots \cup r_n \subseteq r\end{array}}{c(t_1^{r_1}, \ldots, t_n^{r_n})^r \in T_\Sigma^{R,s}} \tag{6.4}$$

The family T_Σ^R of well-formed terms with source regions is then defined as

$$T_\Sigma^R = (T_\Sigma^{R,s})_{s \in S}.$$

The first two preconditions of Equation 6.4 ensure that the terms in T_Σ^R are well-formed as before. The latter two preconditions ensure that the annotated source regions are well-formed. That is, the region of each left-sibling precedes the region of each right-sibling and the region of a parent term includes the regions of all its subterms. The well-formedness of source regions allows us to efficiently navigate within terms to identify the subterm corresponding to a cursor position. Finally, note that $T_{\Sigma^\$}^R$ denotes the set of terms of incomplete programs with source regions.

6.3.3 *Completing Incomplete Programs*

We are now ready to define code completion for incomplete programs where we replace explicit placeholders by proposed terms. We divide the definition of code completion into three functions replace, propose, and complete. Function replace takes a term t^r and replaces its subterm u^p by term v. We use source regions r and p to navigate in t and to uniquely identify subterm u.

Definition 6 (Function replace)**.**

$$\mathsf{replace}(t^r, u^p, v) = \begin{cases} v^{\emptyset}, & \text{if } t^r = u^p \\ \mathsf{replace}(t_i^{r_i}, u^p, v), & \text{if } t^r \neq u^p, t = c(t_1^{r_1}, \ldots, t_n^{r_n}), \\ & p \subseteq r_i \\ t^r, & \text{otherwise} \end{cases}$$

If the current term t^r equals u^p including the source region, we yield the replacement v. We recursively impose the empty source region on term v to mark it as being synthesized. We use the source region p of u to navigate to and recurse on subterm $t_i^{r_i}$ of t^r such that p is included in r_i. If we cannot find an appropriate subterm, we yield the current term t^r unchanged.

We are not only interested in defining functions like replace but also in the metatheoretical properties of these functions. In particular, we want to reason about the soundness of code completion, which means that code completion yields well-formed terms. Thus, before moving on to the other functions, we define precisely when an application of replace is sound.

Theorem 6.3.1 (Soundness of replace)**.** *Given $t^r \in T_\Sigma^{R,s}$ for some sort s, a replacement* $\mathsf{replace}(t^r, u^p, v) = w^q$ *is sound iff $w^q \in T_\Sigma^{R,s}$. If $u^p \in T_\Sigma^{R,s'}$ for some sort s' and $v \in T_\Sigma^{s'}$, then* $\mathsf{replace}(t^r, u^p, v)$ *is sound.*

That is, a replacement is sound if it yields well-formed terms of the same sort as input t^r. Specifically, a replacement of u by v in t is sound if u and v have the same sort. For the proof of this theorem it is important that we impose the empty region on v, such that the result of the replacement has well-formed regions.

We will use replace to inject proposed code fragments in place of placeholders $\$s$. Function proposals computes a list of proposed code fragments for a given sort s. Here, we only specify proposals abstractly, reasoning about its soundness.

Definition 7 (Proposals function)**.** *Given a signature $\Sigma = \langle S, C \rangle$, a proposal function* proposals $: S \to (T_{\Sigma^\$})^*$ *maps each sort $s \in S$ to a sequence of proposed terms. A proposal function* proposals *is sound iff for all $s \in S$, the terms proposed for s have sort s:* $\mathsf{proposals}(s) \in (T_{\Sigma^\$}^s)^*$.

Our proposal function permits context-free syntactic code-completion proposals based on the expected sort. Based on proposals and replace, we can model code completion by (i) navigating to the placeholder at the current cursor position $c \in \mathbb{N}$, (ii) getting proposals for that placeholder, (iii) replacing the placeholder by one of the proposed terms. Function propose performs steps (i) and (ii). That is, propose takes a term $t^r \in T_{\Sigma^\$}^R$ with placeholders as well as source regions and it takes a cursor position $c \in \mathbb{N}$. It finds and yields the term at the cursor position together with a possibly empty list of proposals for it.

Definition 8 (Function propose)**.**

$$\mathsf{propose}(t^r, cur) = \begin{cases} \langle \$s^r, \mathsf{proposals}(s) \rangle, & \text{if } t = \$s, cur \in r \\ \mathsf{propose}(t_i^{r_i}, cur), & \text{if } t = c(t_1^{r_1}, \ldots, t_n^{r_n}), cur \in r_i \\ \langle t^r, \varepsilon \rangle, & \text{otherwise} \end{cases}$$

Finally, function complete uses propose and replace to implement full code completion. To model the user's behavior, we use an oracle $\phi : (T_\Sigma)^+ \to T_\Sigma$ to select one of the proposed terms.

Definition 9 (Function complete)**.**

$\mathsf{complete}(t^r, cur, \phi) =$
  let $\langle u^p, ts \rangle = \mathsf{propose}(t^r, cur)$ in
    if $ts = \varepsilon$
      then t^r
      else $\mathsf{replace}(t^r, u^p, \phi(ts))$

Theorem 6.3.2 (Soundness of complete)**.** *Given $t^r \in T_\Sigma^{R,s}$ for some sort s and arbitrary cur and ϕ, a function* complete$(t^r, cur, \phi) = w^q$ *is sound iff $w^q \in T_\Sigma^{R,s}$. If function* proposals *is sound, then function* complete *is sound for all $t^r \in T_\Sigma^{R,s}$.*

That is, a completion is sound if the resulting term is well-formed and has the same sort as the input. Specifically, for any sound proposal function that only proposes terms of the required sort, code completion is indeed sound. This holds because replace is sound and for all proposal $\langle \$s^r, ts \rangle$, the sort of terms $t \in ts$ is s.

Code completion should also be *complete*. That is, starting from some placeholder $\$s$, all terms of sort s should be constructible through code completion (and by typing lexemes of sort LEX). Complete completion enables a purely projectional user interaction where no typing is necessary except for names of variables etc.

6.3.4 *Implementation in Spoofax*

As an instantiation of our formal model for syntactic code completion, we implemented a generic completion framework in the Spoofax Language Workbench. Spoofax provides the syntax definition formalism SDF3 for specification of syntax. The distinguishing feature of SDF3 is the introduction of explicit layout specified in a template as the body of a context-free production [133]. A template production defines the usual sequence of symbols of a production and an abstract syntax tree constructor. In addition, the whitespace between the symbols is interpreted as a specification-by-example for the purpose of producing a pretty-printer. Thus, a single syntax definition serves to define a grammar, a scannerless generalized parser for that grammar, an abstract syntax tree schema (in the form of an algebraic signature), *and* a pretty-printer mapping ASTs to text.

```
context-free syntax // regular productions

  Statement.Assign = [[VarRef] = [Exp];]
  Statement.If = [
    if([Exp]) [Statement]
    else [Statement]]
  Statement.While = [while([Exp]) [Statement]]
  Statement.Block = [
    {
      [{Statement "\n"}*]
    }]
  Statement.VarDecl = [[Type] [ID];]

context-free syntax // derived productions

  VarRef.VarRef-Plhdr       = [$VarRef]
  Exp.Exp-Plhdr             = [$Exp]
  Statement.Statement-Plhdr = [$Statement]
  Type.Type-Plhdr           = [$Type]
  ID.ID-Plhdr               = [$ID]
```

(a)

```
rules // derived rewrite rules

  rewrite-placeholder:
    Statement-Plhdr() -> Assign(VarRef-Plhdr(), Exp-Plhdr())

  rewrite-placeholder:
    Statement-Plhdr() -> If(Exp-Plhdr(), Statement-Plhdr(),
                                         Statement-Plhdr())

  rewrite-placeholder:
    Statement-Plhdr() -> While(Exp-Plhdr(), Statement-Plhdr())

  rewrite-placeholder:
    Statement-Plhdr() -> Block([])

  rewrite-placeholder:
    Statement-Plhdr() -> VarDecl(Type-Plhdr(), ID-Plhdr())
```

(b)

Figure 6.6 (a) Extending the grammar with placeholder productions and (b) automatically generating rewrite rules for placeholder expansion from the syntax definition.

We support explicit placeholders as part of a language by automatically extending the language's syntax definition with extra template productions. As specified in the formalization, the goal is to allow a placeholder to appear

whenever it is possible to parse a non-terminal at a certain position in the program. The second *context-free syntax* section of Figure 6.6a illustrates the generated template productions from the regular productions defined in the first section.

In our implementation, we instantiate the abstract function proposals as the function templates returning a list of proposals for a sort $s \in \Sigma$.

Definition 10 (Templates function)**.** *Given a signature* $\Sigma = \langle S, C \rangle$*, we define the set of concrete proposals returned by function* templates $: S \to (T_{\Sigma^\$})^*$ *such that:*

$$\frac{c : s_1 \times \ldots \times s_n \to s \in \Sigma}{c(\$s_1, \ldots, \$s_n) \in \mathsf{templates}(s)}$$

We can reason about the soundness of our function templates based on the definition of the abstract function proposals.

Theorem 6.3.3. *Our implementation of the function* proposals *provided by the function* templates *is sound.*

Proof. By the definition of templates, all the terms that we generate as an expansion for a placeholder of sort s have sort s. Thus, according to the soundness criterion of the abstract function proposals, templates is sound. □

Moreover, since the templates function generates all alternatives for a sort s, it is straightforward to establish that our automatically derived completions are complete.

As specified in the function templates, in Spoofax we not only automatically derive placeholders from the SDF3 but also derive their respective proposals. Each template production with a constructor in the syntax definition defines a possible placeholder expansion. The rules in Figure 6.6b shows an example of generated rewrite rules in the Stratego transformation language [129] that transform a placeholder of sort s into all its abstract expansions. As a design decision, placeholder expansions do not include placeholders for nullable symbols such as lists with zero or many elements or optional nodes, generating empty lists or optionals by default and expanding them by placeholder inference as we will present in Section 6.4.

Spoofax constructs source regions as attachments of terms when parsing a program and imploding the parse tree. We use this information to navigate to a placeholder in the program, as specified by the function replace. We produce concrete proposals by pretty-printing the abstract expansions collected from the rewrite rules using the generated pretty-printer from SDF3. Completing the program replaces the placeholder text by the pretty-printed text of its selected expansion. Thus, completing a program preserves its structure except for the placeholder being expanded.

6.4 CODE EXPANSION BY PLACEHOLDER INFERENCE

In this section, we investigate how to use placeholders to propose expansions of complete programs. A complete program contains no placeholders thus,

the method described in the previous section fails to generate any proposals. However, we want to use code completion to propose expansions of complete programs. In this chapter, we focus on adding elements to lists and adding previously missing optional elements. For example, class `Main` in Figure 6.7 does not define the optional extends clause. An invocation of code completion should propose defining the optional element. To this end, we introduce a method for expanding complete programs by inferring and inserting placeholders.

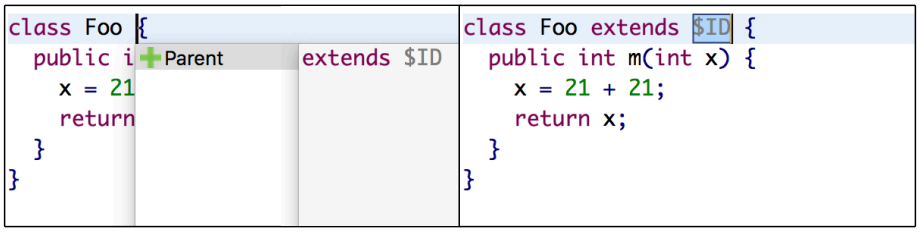


Figure 6.7 Inferring a placeholder inside an optional node.

6.4.1 *Placeholder Inference for Optionals*

We first investigate placeholder inference for optionals, which we represent according to the following definition.

Definition 11 (Optional terms)**.** *Given a sort s, $Opt(s)$ is the sort of optional terms. For each s, constructor $Some_s : s \rightarrow Opt(s)$ indicates the presence of a term of sort s whereas constructor $None_s : Opt(s)$ indicates its absence.*

Placeholder inference for optionals is similar to completion for explicit placeholders, where term $\texttt{None}_s\texttt{()}$ plays the role of placeholder $\$s$. Thus, in a first approximation we could extend function propose to generate proposals for $\texttt{None}_s\texttt{()}$ terms as well as placeholders. However, it is not as straightforward as that. Since a $\texttt{None}_s\texttt{()}$ term corresponds to the empty string, it does not have a source region. Hence, in contrast to a placeholder, we cannot select a $\texttt{None}_s\texttt{()}$ term using the cursor. Furthermore, there may be multiple $\texttt{None}_s\texttt{()}$ terms that are candidates for expansion in the area around the cursor. For example, consider the term $\texttt{VarDecl(Ident("x", None}_{type}\texttt{()), None}_{exp}\texttt{())}$ that represents a variable declaration for an identifier with optional type and optional initializer. When the cursor is placed after the identifier and code completion is triggered, we would like to see proposals for expanding the type as well as the initializer.

To formalize this we need the notion of adjacency of terms to the cursor. A subterm t_i of a term t is adjacent to the cursor if none of the other subterms of t capture the cursor in their source region. Since terms like $\texttt{None}_s\texttt{()}$ have empty source regions, multiple subterms can be adjacent to the cursor simultaneously.

Definition 12 (Function adjacent)**.**

$$\mathsf{adjacent}(t, cur) = \begin{cases} \{t_i^{r_i} \mid r_1 \cup \ldots \cup r_{i-1} < \{cur\} < r_{i+1} \cup \ldots \cup r_n\}, \\ \quad \textit{if } t = C(t_1^{r_1}, \ldots, t_n^{r_n}) \\ \emptyset, \textit{otherwise} \end{cases}$$

Function $\mathsf{infer_o}$ infers completion proposals for *all* optionals that are adjacent to the cursor.

Definition 13 (Function $\mathsf{infer_o}$)**.**

$$\mathsf{infer_o}(t^r, cur) = \begin{cases} \{\ \langle t^r, \mathsf{proposals_o}(s)\rangle\ \}, \\ \quad \textit{if } t = None_s() \\ \bigcup\{\mathsf{infer_o}(t_i^{r_i}, cur) \mid t_i^{r_i} \in \mathsf{adjacent}(t, cur)\}, \\ \quad \textit{if } t = C(t_1^{r_1}, \ldots, t_n^{r_n}) \end{cases}$$

$$\mathsf{proposals_o}(s) = \{Some_s(t) \mid t \in \mathsf{proposals}(s)\}$$

In the first case, if the current term is `None`$_s$`()`, we generate replacement proposals for sort s. This corresponds to the case for placeholders $\$s$ of $\mathsf{propose}$ except that here we generate proposals of optional terms using $\mathsf{proposals_o}$. The second case applies inference recursively to those subterms $t_i^{r_i}$ that are adjacent to the cursor.

6.4.2 *Placeholder Inference for Lists*

We now consider placeholder inference for list terms, which we represent according to the following definition.

Definition 14 (List terms)**.** *Given a sort s, $List(s)$ is the sort of list terms. For each s, constructor $Cons_s : s \times List(s) \to List(s)$ indicates a non-empty list with head and tail and $Nil_s : List(s)$ indicates an empty list.*

Placeholder inference for lists generates proposals for inserting elements into a list. To that end, inference selects the sublist that directly *follows* the cursor (modulo layout) and generates proposals for the syntactic sort of the elements in the list. When the user selects a proposal, we insert the selected element at the cursor position. For example, in Figure 6.8, the cursor is positioned between two statements in a list of statements. Code completion proposes the insertion of a new statement at the cursor position.

Function infer_* generates completion proposals for list elements.

Definition 15 (Function infer_*)**.**

$$\mathsf{infer}_*(t^r, cur) = \begin{cases} \{\ \langle t^r, \mathsf{proposals}_*(s,t)\rangle\ \}, \\ \qquad \textit{if } t = Nil_s() \\ \{\ \langle t^r, \mathsf{proposals}_*(s,t)\rangle\ \} \cup\ \mathsf{infer}_*(hd^p, cur) \\ \qquad \textit{if } t = Cons_s(hd^p, tl^q), \{cur\} < p \\ \mathsf{infer}_*(hd^p, cur) \\ \qquad \textit{if } t = Cons_s(hd^p, tl^q), cur \in p \\ \mathsf{infer}_*(tl^q, cur) \\ \qquad \textit{if } t = Cons_s(hd^p, tl^q), \{cur\} > p \\ \bigcup\{\mathsf{infer}_*(t_i^{r_i}, cur) \mid t_i^{r_i} \in \mathsf{adjacent}(t, cur)\}, \\ \qquad \textit{if } t = C(t_1^{r_1}, \ldots, t_n^{r_n}), C \neq Cons \end{cases}$$

$$\mathsf{proposals}_*(s, tl) = \{Cons_s(hd, tl) \mid hd \in \mathsf{proposals}(s)\}$$

In the first case, if the current term is the empty list $\texttt{Nil}_s\texttt{()}$, we generate proposals for element sort s. This corresponds to the $\texttt{None}_s\texttt{()}$ case of infer_0 and to the case for placeholders $\$s$ of $\mathsf{propose}$ except that here we generate proposals for list terms using $\mathsf{proposals}_*$. Specifically, we propose to replace the empty list with a singleton list where the head element is a proposal for sort s. In the second case of infer_*, if the current term is a `Cons` term and the cursor to the left of the head element, we propose to prepend another element. We also recursively infer completions in the head element to support proposals for nested lists. In the third case, if the cursor is within the source region of the head element, we recursively infer proposals there. Otherwise, if the cursor is to the right of the head element, we recursively infer proposals for the tail of the list. Finally, for terms that are not lists we recursively infer proposals for all subterms that are adjacent to the cursor.

We illustrate a concrete example in Figure 6.8. Note that the cursor is at position 58, that is, we want to add a statement in between the two existing statements for variable declaration and assignment. We start computing proposals by applying infer_* to node $\texttt{Method}^{[16,92]}$. Since that node is not a list, the function reaches the fourth case, returning the union of a recursive application of infer_* on all adjacent children. However, node $\texttt{Cons}^{[39,74]}$ is the only adjacent subterm.

Since that node is a non-empty list, we check whether the cursor is before or after the head of the list. Since the head element $\texttt{VarDecl}^{[39,44]}$ precedes the cursor at 58, the third case of infer_* applies and recurses into the tail of the list $\texttt{Cons}^{[63,74]}$. This time, the cursor position precedes the head element $\texttt{Assign}^{[63,74]}$ and the second case of infer_* applies. Thus, we propose completions that prepend a statement to $\texttt{Cons}^{[63,74]}$. The recursive call in the second case of infer_* does not yield any additional completions because the head element does not contain a nested list adjacent to the cursor.

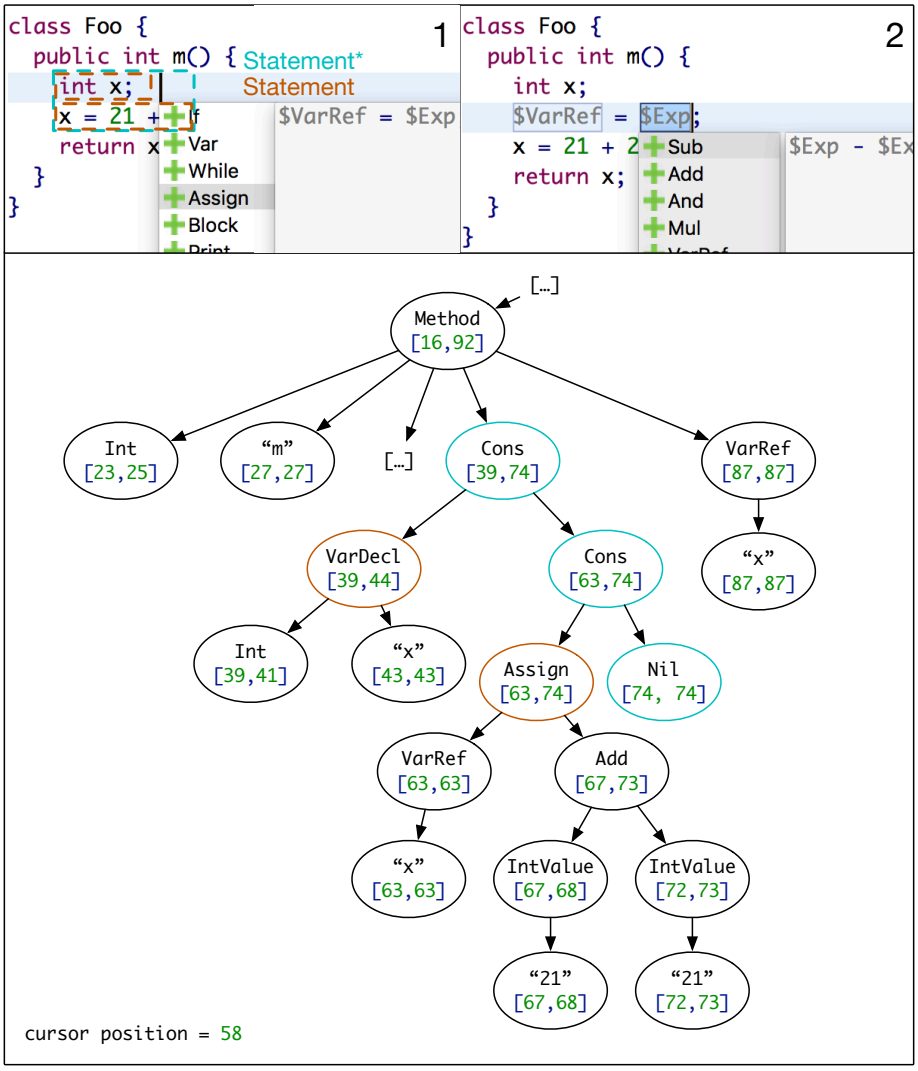


Figure 6.8 Placeholder inference inside a list. At the bottom, excerpt of the AST with source regions before expansion.

6.4.3 *Code Expansion by Placeholder Inference*

Similar to code completion, we can combine the two inference functions infer_o and infer_* together with replace. Since we do not rewrite incomplete program fragments but insert code into complete program fragments, we call this *code expansion* rather than code completion.

The following function expand defines code expansion formally. To model the user's behavior, here we use two oracle functions ϕ_1 and ϕ_2. Through the

first oracle ϕ_1, the user selects which one of the subterms adjacent to the cursor to expand. Through the second oracle ϕ_2, the user selects the expansion for the selected subterm.

Definition 16 (Function expand).

$$
\begin{array}{l}
\mathsf{expand}(t^r, cur, \phi_1, \phi_2) = \\
\quad \mathsf{let}\ props = \mathsf{infer}_*(t^r, cur) \cup \mathsf{infer}_\mathsf{o}(t^r, cur)\ \mathsf{in} \\
\quad\quad \mathsf{if}\ props = \emptyset \\
\quad\quad\quad \mathsf{then}\ t^r \\
\quad\quad\quad \mathsf{else\ let}\ \langle u^p, ts \rangle = \phi_1(props)\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{if}\ ts = \varepsilon \\
\quad\quad\quad\quad\quad \mathsf{then}\ t^r \\
\quad\quad\quad\quad\quad \mathsf{else}\ \mathsf{replace}(t^r, u^p, \phi_2(ts))
\end{array}
$$

Theorem 6.4.1 (Soundness of expand). *Given* $t^r \in T_\Sigma^{R,s}$ *for some sort* s *and arbitrary* cur, ϕ_1, *and* ϕ_2, *an expansion* $\mathsf{expand}(t^r, cur, \phi_1, \phi_2) = w^q$ *is sound iff* $w^q \in T_\Sigma^{R,s}$. *If function* proposals *is sound, then function* expand *is sound for all* $t^r \in T_\Sigma^{R,s}$.

That is, an expansion is sound if the resulting term is well-formed and has the same sort as the input. Specifically, for any sound proposal function that only proposes terms of the required sort, code expansion is indeed sound. This holds because replace is sound and we have setup $\mathsf{proposals}_\mathsf{o}$ and $\mathsf{proposals}_*$ such that for all proposal $\langle u, ts \rangle$, the sort of terms $t \in ts$ is identical to the sort of u.

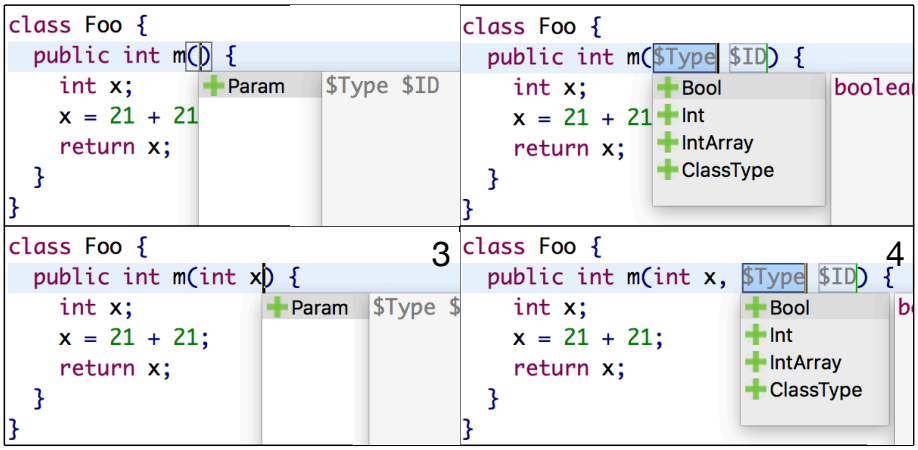


Figure 6.9 Inserting an element into a list: Spoofax preserves the surrounding layout and inserts list separators as needed.

A pragmatic concern when inserting elements into a list is the formatting of the source code. Our formal model abstracts from this issue by considering

ASTs only. As illustrated in Figure 6.9, our implementation in Spoofax preserves the layout of all existing code and only formats the inserted element, also inserting list separators as needed.

6.5 CODE COMPLETION FOR INCORRECT PROGRAMS

In this section, we consider syntactic code completion for syntactically incorrect programs, i.e. for which parsing fails. Such syntactic errors occur frequently during editing. For example, when the developer writes an assignment statement, the program text remains incorrect until the developer terminates the statement with a semicolon. We want to provide code completion for incorrect programs to assist developers in completing code fragments as they write them. Specifically, we address the following scenario:

- We only consider syntax errors at the cursor position; we ignore errors elsewhere in the program text.
- We only consider the insertion of symbols into the program text; we ignore other forms of manipulation such as symbol removal.
- Soundness applies as before: The proposed recoveries must yield correct programs at the cursor position.
- We relax the requirement on completeness: Not all programs are necessarily constructible from the proposed recoveries.

Figure 6.10 illustrates the expected behaviour of the completion framework for an incorrect program using the grammar of Figure 6.6. Here, `x` is the first symbol of a statement and the framework should propose all statements that can start with symbol `x`. As shown in the top-right and bottom-right boxes, upon selection of a proposal, the framework inserts the missing symbols to make the program *syntactically* correct. We insert placeholders for subterms, thus allowing the user to subsequently complete the program as described in Section 6.3. In the remainder of this section, we present our solution for computing proposals based on the insertion of missing symbols. Placeholders play a crucial role for our solution as we will discuss in Section 6.5.2.

6.5.1 *Constructing Proposals by Inserting Symbols*

To construct the list of proposals, we compute all possible ways to recover a correct program by inserting symbols at the cursor position. To perform symbol insertions, we use an error-recovering technique based on permissive grammars and insertion productions [65].

Permissive grammars are grammars that can parse a more relaxed version of the input by either skipping individual symbols or simulating the insertion of missing symbols. Here, we only consider the insertion of missing symbols as an alternative to fix the error. In addition to regular productions, a permissive grammar for completion contains *insertion productions* that denote which symbols may be inserted. For example, Figure 6.11 shows the insertion productions for our imperative language from Figure 6.6. An insertion production

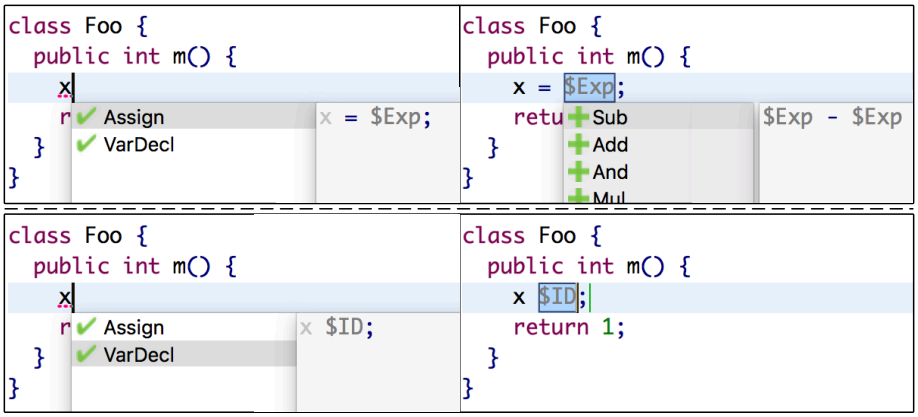


Figure 6.10 Fixing syntax errors by code completion.

```
// derived insertion rules for placeholders
context-free syntax // derived productions

  VarRef.VarRef-Plhdr       = {symbol-insertion}
  Exp.Exp-Plhdr             = {symbol-insertion}
  Statement.Statement-Plhdr = {symbol-insertion}
  Type.Type-Plhdr           = {symbol-insertion}
  ID.ID-Plhdr               = {symbol-insertion}

// derived insertion rules for literals
lexical syntax

  "="     = {symbol-insertion}
  "if"    = {symbol-insertion}
  "else"  = {symbol-insertion}
  "while" = {symbol-insertion}
  "("     = {symbol-insertion}
  ")"     = {symbol-insertion}
  "{"     = {symbol-insertion}
  "}"     = {symbol-insertion}
  ";"     = {symbol-insertion}
```

Figure 6.11 Extending the grammar with insertion rules.

recognizes the *empty string* — the right-hand side of the production is empty. Thus, if a regular production expects some symbol, which is not present in the text, the insertion production can parse the empty string to pretend it is there anyway. We automatically generate such insertion productions for each lexeme and placeholder of the grammar.

To compute the list of proposals, we use generalized parsing [125, 130, 131]

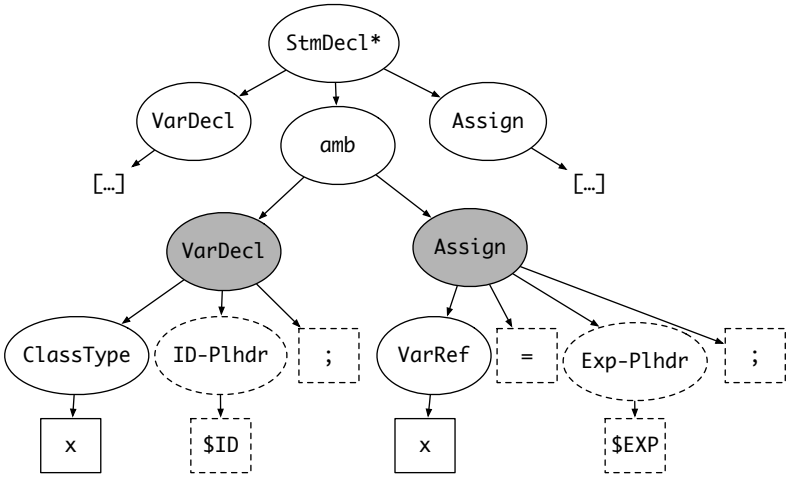


Figure 6.12 Recovered AST with inserted nodes (dashed line) and proposals (shaded fill).

on the permissive grammar. Generalized parsing supports ambiguous inputs and constructs a parse forest with one AST for each possible parse result. Thus, if alternative insertions lead to a correct parse result, we retrieve all alternatives from the generalized parser. Generalized parsers typically compact the parse forest, using ambiguity nodes `amb` to denote alternative subtrees.

Figure 6.12 shows the parse forest we retrieve for parsing the program from Figure 6.10 using the permissive grammar. The parser found two alternatives for completing the program. First, we can interpret the lexeme `x` as a class type, insert an `ID` placeholder, and a semicolon lexeme. Or, we can interpret `x` as a variable reference, insert an equality lexeme, an `Exp` placeholder, and a semicolon lexeme. In Figure 6.12, we mark the inserted symbols using dashed shapes and we use shaded nodes to mark nodes that become proposals. To avoid an excessive search for possible recoveries, we limit the search space using placeholders.

6.5.2 *Limiting the Search Space for Recoveries*

Insertion productions indicate to the parser to insert missing symbols. However, arbitrarily applying insertion productions would lead to non-termination. In our example, we could keep inserting symbols to add more statements to the list or even construct additional classes. This happens because insertion productions produce the empty string. As a result, the parser does not consume any input when applying insertion productions, leading to an infinite number of possible parses. Hence, we need to restrict the application of insertion productions to guarantee termination.

First, recall that we are merely interested in code completion, rather than error-recovery parsing in general. Thus, we can restrict insertions to the cursor

position modulo layout. Conversely, we prohibit the parser from applying insertion productions elsewhere in the program.

Second, we assume that part of a proposal is already in the input. Therefore, fixing the error preserves the existing fragments of a proposal and only adds the missing symbols necessary to finish a structure. From this restriction, we disallow the application of regular productions on only recovered nodes. Note that error recovery can recover either placeholders or literal strings from the program. Recovering placeholders is essential to limit the search space as we do not need to recursively recover complex subterms that may be constructed by placeholder expansion later.

Third, we define our recovery approach as greedy, assuming that proposals contain as many symbols as possible from the input. A proposal can also include multiple nodes implying that the AST of the program contains an erroneous branch. Thus, we construct a single proposal as the smallest subtree containing all proposal nodes and we construct multiple proposals by flattening ambiguities containing multiple proposal nodes. By doing that, we guarantee that completing the program chooses only one alternative of the ambiguity and proposals only fix a single node (or branch). Most importantly, we guarantee that our strategy is sound as selecting a proposal does not introduce errors, but introduces a fix instead. The list of proposals is partially complete, i.e., we produce all fixes that include the elements that are already part of the input.

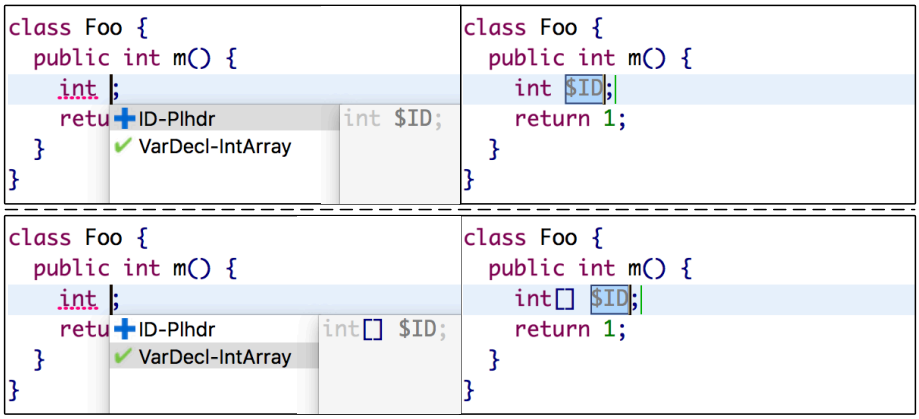


Figure 6.13 At the top the proposal inserts the infix of an incomplete variable declaration (in this case either a single placeholder `$ID`). At the bottom, an example of a nested proposal, where it is necessary to fix multiple nodes in the AST to recover from the error (turn the Int type into IntArray and insert `$ID`).

Figure 6.13 shows examples of the third restriction. In the top program, we do not create two proposals, for example, using *int* as the prefix of a variable declaration, and semicolon as a suffix for an assignment. Instead, we use both symbols as part of only one proposal for a variable declaration. Moreover, at the bottom, we show an example of a proposal that *fixes* more than a single

node of the program. In this case, we change the inner node for the type of the variable declaration to array of integers, and add the missing placeholder for the identifier to complete the variable declaration itself.

6.5.3 *Implementation in Spoofax*

To implement the syntactic code completion for incorrect programs in Spoofax, we use the scannerless generalized-LR parsing algorithm (SGLR). The complete algorithm for SGLR is described in [130, 131] but as we are only interested in restricting the application of grammar rules to construct proposals, we only modified the reducer method of SGLR, applying the restrictions on the search space of possible recoveries we described before.

SGLR is a generalized shift-reduce parser that handles multiple stacks in parallel. Each conflict action in the parse table generates a new stack so that parsing can continue with that action. Given a parse table for some grammar and a string, the parser returns a parse forest containing all possible alternatives to parse the string according to the grammar described in the table. This makes SGLR a perfect match for collecting all possible recoveries as ambiguities in the resulting parse tree.

To produce the list of proposals, we do a traversal on the resulting parse tree, collecting all proposals as described before and pretty-printing them to present to the user. A selected proposal only adds the missing fragments necessary to fix the program. SGLR deals with other errors in the program using its regular error recovering strategy based on permissive grammars [65].

With the approach described in this section, the completion framework can handle incorrect programs since the parser is able to construct a list of proposals by inserting missing symbols, fixing an error at the cursor position. From there, the framework provides syntactic code completion following the approach we described before, either by expanding explicit placeholders or by placeholder inference as illustrated in Figure 6.2.

Our solution also preserves the generic aspect of the completion framework, as we derive insertion productions from the syntax definition. If the error at the cursor position does not follow the assumptions we made previously, error recovery does not produce any proposal. Moreover, syntax errors that occur in other locations and do not influence code completion are just preserved since insertion productions to create proposals are only applied at the cursor position.

6.6 EVALUATION

We have applied our approach to generate syntactic code completion for Pascal, a subset of Java, and IceDust, a domain-specific language for data modelling [57]. We automatically generate placeholder transformations and construct the proposals with the pretty-printer generated from the syntax definition. Recovered proposals are constructed by our adapted version of SGLR.

We observed that the way production rules are organized in the syntax definition directly affects the number of placeholders and proposals for each placeholder. Moreover, when considering placeholder inference, inferring a placeholder when multiple optionals and empty lists are adjacent to the cursor makes the list of proposals even larger. Ideally, it should not be necessary to massage the grammar to produce better proposals. However, in the current implementation the grammar structure can affect the generated proposals.

In general, the completion framework produced acceptable proposals for all languages we evaluated. Deriving syntactic code completion from the syntax definition allowed us to implement the completion service for each of these languages without additional effort.

6.7 RELATED WORK

We have implemented a generic content completion framework that is able to derive sound and complete syntactic code completion from language definitions. We adopt placeholders to represent incomplete structures for a program in a textual editor, similar to structural editors. For programs that still contain syntax errors due to incomplete structures we construct the list of proposals by error recovery. We compare our approach to projectional editors in the literature, textual language workbenches and discuss syntactic error recovery.

Syntactic Completion in Textual Language Workbenches Textual language workbenches such as Spoofax [68] and Xtext [43] derive syntactic completions from the syntax definition. However, these language workbenches currently do not have a representation of incomplete programs. In the case of Xtext, proposals involve only the following token that can appear in the input. Since non-terminal symbols can reference each other in the syntax definition, proposals may also involve predefined names or types. To extend the automatically generated completions, the language engineer can customize code templates in automatically generated Java methods.

As for the old implementation of syntactic code completion in Spoofax, a descriptor language for editor services contains the specification of completion templates, defining expansions given the context of a non-terminal symbol from the grammar. Placeholders inside proposals contain default strings, and the possibility to directly navigate to them is lost when leaving the completion mode. Furthermore, completing the program might lead to syntax errors, as the framework calculates proposals based on whether it is possible to parse a non-terminal symbol at the cursor position, possibly inserting incomplete structures.

Projectional Editing with Placeholders Placeholders allow for directly manipulating the AST of a program, a characteristic of projectional/structural editors. The Generic Syntax-directed Editor (GSE) [78] is part of the ASF+SDF Meta-environment [71] and generates an interactive editor that is hybrid, i.e., both textual and projectional from the language specification extended with placeholders.

In GSE, the editor uses the cursor position to determine the smallest node in the AST being edited. Only the content inside the focus is actually parsed, with the guarantee that the remainder of the input is syntactically correct. Whenever the focus is in a placeholder, the editor can expand the node following the grammar rules for the placeholder. However, one of the consequences for supporting hybrid editing is that GSE stores both the textual and abstract representation of the program in memory, creating a two-way mapping between them. Our approach is only based on textual editors, and we rely on source positions mapped as attachments to nodes in the AST, constructing them whenever parsing the program.

Another issue is that GSE does not support error recovery, so a focus is either syntactically correct or it is not. Thus, to properly provide code completion the user needs to first manually fix syntax errors. In addition, we only provide a single editor operation (control + space) to invoke the completion framework, whereas GSE uses the focus to determine the completions for a placeholder. Furthermore, our approach supports free textual editing, without any need for substring parsing to keep track of focuses.

Language workbenches such as CENTAUR [21], MPS [134] and mbeddr [136] generate projectional editors from language specifications. In such editors, the user edits the program by manipulating the AST directly instead of editing pure text. Proposals are automatically derived from the projections defined by the language engineer, making the completion service sound by definition. Code completion also alleviates the problems when writing programs in projectional editors, since the normal editing behaviour does not resemble classical text editing [137].

The Synthesizer Generator [98] has a representation for unexpanded terms as *completing operators*. Completing operators act as placeholders and can be structurally edited by specifying rules as commands that insert code templates. In our implementation, code templates are defined by the grammar, whereas the definition of code templates is disjoint from parsing rules. Moreover, since the language definition is based on attribute grammars [75], template proposals can also use semantic information by evaluating attributes derived from syntactic sorts of completion operators. Proxima [108] also uses placeholders as *holes* that can appear inside textual or structural presentation elements. As our solution is implemented in a textual editor, we only support placeholders as part of textual elements of the program.

Error Recovery To handle incorrect programs, our approach recovers missing symbols to construct a valid AST from which the framework creates an expansion proposal. There are different approaches to support error recovery from syntax errors [41]. The current approach implemented in the Spoofax language workbench is based on island grammars [89, 90] and recovery rules providing error recovery for a generalized parsing algorithm [65].

In the generalized scenario of SGLR, it is necessary to investigate multiple branches, and the detection of syntax errors occurs at the point where the last branch failed. This point might not even be local to the actual root cause of the error, making error reporting more difficult. Scannerless parsing also

contributes to make the recovery strategy more complex. Common strategies based on token insertion or deletion to fix the error are ineffective when considering single characters.

Our approach benefits from the assumptions that we know the error location and that only missing elements contribute to the error. Therefore, it is not necessary to skip parts of the input nor backtrack to find the actual error location. Furthermore, we benefit from the fact that SGLR constructs a parse forest as result. Thus we return all possible fixes, reporting them to the user as proposals.

6.8 FUTURE WORK

Character-based Completions Our current recovery strategy does not recover from incomplete words, producing only insertion symbols for literals and placeholders. The completion framework could handle partial keywords by manipulating the input to reconstruct keywords and use them as a starting point for recovering a proposal.

Inlining and Ordering Proposals The current approach might generate too many proposals depending on the productions in the grammar. For this issue, ordering suggestions might improve the final user experience [99]. Inlining proposals can also improve the framework for cases when it is necessary many placeholder replacements to create a final code template.

Semantic Completions The completions in this chapter are restricted to *syntactic* completions. Mainstream IDEs typically have spent more effort in the support for *semantic* completions, i.e. proposing names (e.g. of variables or methods) that are valid to use in the cursor context. In future work, we plan to explore providing *generic* support for such semantic completions based on our work on name [76, 91] and type resolution [13]. Using the results of name and type resolution, we can propose completions for lexical placeholders to insert declared names. Moreover, semantic information can also be used to filter the list of syntactic proposals such that sound content completion guarantees the absence of syntactic *and* semantic errors.

6.9 CONCLUSION

Code completion avoids misspellings and enables language exploration. However, the support for syntactic completion is not fully implemented by most IDEs. The completion implementation is ad-hoc, unsound and incomplete.

The separation of programs into different states allowed us to provide code completion with a "divide and conquer" strategy. For correct programs, we implement code completion by expanding placeholders that can appear implicitly or explicitly in programs. For incorrect programs, we used the nature of errors in the completion scenario to propose an adapted error recovery strategy to construct the list of proposals.

Finally, our formalization allowed us to reason about soundness and completeness of code completion. We implemented the framework by modifying the scannerless GLR parsing algorithm and by generating placeholders and its expansions from syntax definitions in the Spoofax Language Workbench. Our framework addresses the requirements derived from the analysis of state-of-the-art implementations of syntactic code completion (Section 6.2). This work opens up a path to rich editing services based on the (context-sensitive) structure of a program in purely textual IDEs.

Conclusion

7

In this dissertation we presented techniques that enable declarative syntax specifications to be used by modern language workbenches. In the first part of this dissertation, we investigated how to provide support for *efficient declarative disambiguation*, since grammars used to define programming languages are often ambiguous. In the second part we investigated declarative syntax definitions in language workbenches from two different perspectives: the challenges to support languages with unique features, such as *layout sensitivity* and how to support complex syntactic editor services, such as *syntactic code completion*. We extensively evaluated each of our techniques using the syntax definition formalism SDF3 and the Spoofax Language Workbench.

7.1 THE THESIS REVISITED

In the introduction of this dissertation we listed open problems related to using declarative syntax definitions in language workbenches. Below, we show how we addressed those problems.

The key principle underlying the design of the family of syntax definition formalisms SDF is to enable *declarative* syntax definition, so that users do not need to adapt their grammars to a particular parsing algorithm. SDF3, as the latest generation of SDF and an evolution of SDF2, has been developed in this dissertation under the same principle, improving various issues of its predecessor, and serving the needs of modern language workbenches.

The new semantics for disambiguating expression grammars shown in Chapter 2 guarantees that declarative disambiguation of expression grammars in SDF3 is *safe* and *complete*. We also show the need for efficient disambiguation, developing a study of priority conflicts in real programs in Chapter 3. Furthermore, we provide an efficient implementation of this semantics in Chapter 4, such that the generated parser can perform disambiguation without performance penalties.

To support parsing and pretty-printing of layout-sensitive languages, we have also equipped SDF3 with *layout declarations*, as shown in Chapter 5. Layout declarations enable language engineers to automatically derive a layout-sensitive parser and pretty-printer for the syntax definition, increasing maintainability and supporting rapid development and prototyping of such languages.

Finally, we proposed a principled approach for syntactic code completion in Chapter 6. We extended the syntax definition with the notion of placeholders, using them to address code completion for incomplete, complete, and incorrect programs. We showed that our approach is sound and complete, supporting rich and efficient editor services in tools such as language workbenches.

We believe that our thesis has been supported by the work presented in this dissertation, and that SDF3 can be used to effectively define the syntax of programming languages, and generate an efficient generalized parser, a (layout-sensitive) pretty-printer, and sound and complete code completion.

7.1.1 *Summary of Contributions*

Below, we summarize our core contributions. For a more detailed account of our contributions, we refer to the individual chapters of this dissertation.

Design of SDF3
: We present the design of the Syntax Definition Formalism SDF3, which provides many improvements over its predecessor. As shown by our case studies, SDF3 has been used to successfully define several programming languages, including domain-specific languages such as IceDust, Tiger, and Jasmin, and general purpose languages such as Java, OCaml, and Haskell.

Safe and Complete Semantics for Disambiguation of Expression Grammars
: We propose a semantics for declarative disambiguation of operator precedence conflicts that is safe and complete, addressing the issues in the semantics from SDF2. Our semantics can also handle conflicts that require unbounded depth analysis of the tree called deep priority conflicts, including lower precedence prefix, dangling prefix and suffix, longest match, and ambiguities due to indirect recursion, which occur in various types of grammars that define expressions.

Empirical Study on Deep Priority Conflicts
: We present a study of deep priority conflicts using real-world programs. Our study has indicated that such conflicts occur relatively often in practice and that grammar transformation techniques might not always be efficient when dealing with deep priority conflicts.

Efficient Disambiguation of Deep Priority Conflicts
: We introduced a more efficient technique for disambiguating deep priority conflicts. By using a data-dependent approach, we can address deep priority conflict disambiguation at parse time, reducing the cost for disambiguation and its overhead when parsing programs without conflicts.

Layout Declarations
: We propose declarative specifications of indentation rules in programming languages as layout declarations. These declarations can be used to derive a more efficient layout-sensitive parser and correct pretty-printers for layout-sensitive languages.

Principled Syntactic Code Completion
We introduce a novel approach for principled sound and complete syntactic code completion. Syntactic code completion as an IDE feature in a language workbench increases language discoverability and helps programmers avoid misspellings.

7.2 SUGGESTIONS FOR FUTURE WORK

In this section, we list some suggestions for future work based on the work presented in this dissertation.

Semantic Code Completion In this dissertation we focused on deriving sound and complete syntactic code completion from a syntax definition. However, mainstream IDEs spend more effort in the support of semantic completions, i.e. proposing names (e.g. of variables or methods) that are valid to use in the cursor context. Thus, we propose as future work, exploring the integration of both approaches, that is, using syntactic code completion in combination with semantic completion to produce better proposals. Semantic code completion can be used to empower textual editors with structural editor features, guaranteeing semantic well-formedness for incomplete programs [93]. In the context of Spoofax, syntactic and semantic code completion can be implemented by using the results of name and type analysis [76, 13, 14]. Furthermore, semantic analysis can be used to filter the list of syntactic proposals to guarantee that the resulting program does not contain syntactic nor semantic errors.

Comment Preserving Layout-Sensitive Pretty-printing While we proposed techniques to automatically derive a pretty-printer for a layout-sensitive language from the language specification, our technique does not consider comments when reconstructing the pretty-printed program. Recent work has been done on layout preservation in refactoring transformations [66]. Therefore, exploring an integration of both techniques may allow deriving a pretty-printer that also considers comments when pretty-printing a program of a layout-sensitive language. One of the challenges consists of reconstructing comments such that they do not interfere with the meaning of the original program, but are still correctly preserved, that is, comments still relate to the structure they refer to.

Declarative Disambiguation and Layout Sensitive Languages One challenge that still remains when using SDF3 to specify a layout-sensitive language consists of efficient disambiguation of operator precedence conflicts. Despite our efforts on providing a safe and complete semantics for disambiguating such conflicts, we believe that our implementation needs to be adapted to handle layout-sensitive languages. The problem is that, for such languages, layout can be used as brackets, to perform explicit disambiguation. Thus, in some cases, our solution may be unsafe for layout-sensitive languages, filtering trees that do not belong to an ambiguity, because the tree that obeys the priority rules in the language

specification has been invalidated due to layout, that is, disambiguation has already been performed. Because both layout-sensitive disambiguation, and our approach for data-dependent disambiguation of deep priority conflicts happen at parse-time, we believe that they can be combined to guarantee safe disambiguation even for programs of layout-sensitive languages.

Bibliography

[1] Annika Aasa. "Precedences in Specifications and Implementations of Programming Languages". In: *Theoretical Computer Science* 142.1 (1995), pp. 3–26. DOI: http://dx.doi.org/10.1016/0304-3975(95)90680-J.

[2] Paul W. Abrahams. "A final solution to the Dangling else of ALGOL 60 and related languages". In: *Communications of the ACM* 9.9 (1966), pp. 679–682. DOI: http://doi.acm.org/10.1145/365813.365821.

[3] Michael D. Adams. "Principled parsing for indentation-sensitive languages: revisiting landin's offside rule". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 511–522. ISBN: 978-1-4503-1832-7. DOI: http://doi.acm.org/10.1145/2429069.2429129.

[4] Michael D. Adams and Matthew Might. "Restricting Grammars with Tree Automata". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 82:1–82:25. ISSN: 2475-1421. DOI: 10.1145/3133906. URL: http://doi.acm.org/10.1145/3133906.

[5] Ali Afroozeh, Mark G. J. van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen J. Vinju. "Safe Specification of Operator Precedence Rules". In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer, 2013, pp. 137–156. ISBN: 978-3-319-02653-4. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_8.

[6] Ali Afroozeh and Anastasia Izmaylova. "Faster, Practical GLL Parsing". In: *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Björn Franke. Vol. 9031. Lecture Notes in Computer Science. Springer, 2015, pp. 89–108. ISBN: 978-3-662-46662-9. DOI: http://dx.doi.org/10.1007/978-3-662-46663-6_5.

[7] Ali Afroozeh and Anastasia Izmaylova. "One parser to rule them all". In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Gail C. Murphy and Guy L. Steele Jr. ACM, 2015, pp. 151–170. ISBN: 978-1-4503-3688-8. DOI: http://doi.acm.org/10.1145/2814228.2814242.

[8] Ali Afroozeh and Anastasia Izmaylova. "Operator precedence for data-dependent grammars". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, 2016, pp. 13–24. ISBN: 978-1-4503-4097-7. DOI: http://doi.acm.org/10.1145/2847538.2847540.

[9] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. "Deterministic Parsing of Ambiguous Grammars". In: *Communications of the ACM* 18.8 (1975), pp. 441–452. DOI: http://doi.acm.org/10.1145/360933.360969.

[10] Alfred V. Aho, Steven C. Johnson, and Jeffrey D. Ullman. "Deterministic Parsing of Ambiguous Grammars". In: *POPL*. 1973, pp. 1–21.

[11] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[12] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. "A Study of Visual Studio Usage in Practice". In: *SANER*. 2016. (Acceptance Ratio: 52/140 = 37%).

[13] Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A constraint language for static semantic analysis based on scope graphs". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Martin Erwig and Tiark Rompf. ACM, 2016, pp. 49–60. ISBN: 978-1-4503-4097-7. DOI: http://doi.acm.org/10.1145/2847538.2847543.

[14] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). DOI: https://doi.org/10.1145/3276484.

[15] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998. ISBN: 0-521-58388-8.

[16] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. "The FORTRAN Automatic Coding System". In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*. IRE-AIEE-ACM '57 (Western). New York, NY, USA: ACM, 1957, pp. 188–198. DOI: 10.1145/1455567.1455599.

[17] John Warner Backus. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". In: *IFIP Congress*. 1959, pp. 125–131.

[18] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. "The Coq proof assistant reference manual: Version 6.1". PhD thesis. Inria, 1997.

[19] Bas Basten. "Ambiguity Detection for Programming Language Grammars". PhD thesis. Universiteit van Amsterdam, Dec. 2011.

[20] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language, Version 1.2*. Available on: `http://yaml.org/spec/1.2/spec.html`. 2009.

[21] Patrick Borras, Dominique Clément, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. "CENTAUR: The System". In: *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. New York, USA: ACM, 1988, pp. 14–24.

[22] R. Boulton. *Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing*. 390. University of Cambridge, Computer Laboratory, 1996.

[23] Eric Bouwers, Martin Bravenboer, and Eelco Visser. "Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking". In: *Electronic Notes in Theoretical Computer Science* 203.2 (2008), pp. 85–101. DOI: `http://dx.doi.org/10.1016/j.entcs.2008.03.046`.

[24] Claus Brabrand, Robert Giegerich, and Anders Møller. "Analyzing ambiguity of context-free grammars". In: *Science of Computer Programming* 75.3 (2010), pp. 176–191. DOI: `http://dx.doi.org/10.1016/j.scico.2009.11.002`.

[25] M.G.J. van den Brand. *Generation of Language Independent Modular Prettyprinters*. Tech. rep. P9315. University of Amsterdam, July, 1993.

[26] M.G.J. van den Brand. *Prettyprinting Without Losing Comments*. Tech. rep. P9327. University of Amsterdam, October, 1993.

[27] Mark G. J. van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. "Efficient annotated terms". In: *Software: Practice and Experience* 30.3 (2000), pp. 259–291.

[28] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. "Disambiguation Filters for Scannerless Generalized LR Parsers". In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 143–158. ISBN: 3-540-43369-4. DOI: `http://link.springer.de/link/service/series/0558/bibs/2304/23040143.htm`.

[29] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: `http://dx.doi.org/10.1016/j.scico.2007.11.003`.

[30] Martin Bravenboer, Éric Tanter, and Eelco Visser. "Declarative, formal, and extensible syntax definition for AspectJ". In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*. Ed. by Peri L. Tarr and William R. Cook. ACM, 2006, pp. 209–228. ISBN: 1-59593-348-4. DOI: http://doi.acm.org/10.1145/1167473.1167491.

[31] Martin Bravenboer and Eelco Visser. "Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*. Ed. by John M. Vlissides and Douglas C. Schmidt. Vancouver, BC, Canada: ACM, 2004, pp. 365–383. ISBN: 1-58113-831-8. DOI: http://doi.acm.org/10.1145/1028976.1029007.

[32] Leonhard Brunauer and Bernhard Mühlbacher. "Indentation Sensitive Languages". Unpublished Manuscript. July, 2006. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.2933&rep=rep1&type=pdf (visited on 05/16/2013).

[33] Raymond P.L. Buse and Westley R. Weimer. "A Metric for Software Readability". In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA '08. Seattle, WA, USA: ACM, 2008, pp. 121–130. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390647. URL: http://doi.acm.org/10.1145/1390630.1390647.

[34] David G. Cantor. "On The Ambiguity Problem of Backus Systems". In: *Journal of the ACM* 9.4 (1962), pp. 477–479. DOI: http://doi.acm.org/10.1145/321138.321145.

[35] Minder Chen and Jay F. Nunamaker. "MetaPlex: an integrated environment for organization and information system development". In: *Proceedings of the International Conference on Information Systems, ICIS 1989, 1989, Boston, Massachusetts, USA*. Association for Information Systems, 1989, pp. 141–151. DOI: http://doi.acm.org/10.1145/75034.75047.

[36] N. Chomsky and M.P. Schützenberger. "The Algebraic Theory of Context-Free Languages*". In: *Computer Programming and Formal Systems*. Ed. by P. Braffort and D. Hirschberg. Vol. 35. Studies in Logic and the Foundations of Mathematics. Elsevier, 1963, pp. 118–161. DOI: https://doi.org/10.1016/S0049-237X(08)72023-8. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08720238.

[37] Noam Chomsky. *Syntactic Structures*. The Hague, The Netherlands: Mouton, Feb. 1957.

[38] K. Clarke. *The Top-down Parsing of Expressions*. University of London. Queen Mary College. Department of Computer Science and Statistics, 1986.

[39] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available on: http://www.grappa.univ-lille3.fr/tata. release October, 12th 2007. 2007.

[40] Nils Anders Danielsson. "Correct-by-construction pretty-printing". In: *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*. Ed. by Stephanie Weirich. ACM, 2013, pp. 1–12. ISBN: 978-1-4503-2384-0. DOI: http://doi.acm.org/10.1145/2502409.2502410.

[41] Pierpaolo Degano and Corrado Priami. "Comparison of Syntactic Error Handling in LR Parsers". In: *Software: Practice and Experience* 25.6 (1995), pp. 657–679.

[42] Jay Earley. "Ambiguity and Precedence in Syntax Description". In: *Acta Informatica* 4 (1974), pp. 183–192.

[43] Sven Efftinge and Markus Völter. "oAW xText: A framework for textual DSLs". In: *Workshop on Modeling Symposium at Eclipse Summit*. 2006.

[44] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, 2007, pp. 884–885. ISBN: 978-1-59593-865-7. DOI: http://doi.acm.org/10.1145/1297846.1297938.

[45] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. "Layout-Sensitive Generalized Parsing". In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, 2012, pp. 244–263. ISBN: 978-3-642-36089-3. DOI: http://dx.doi.org/10.1007/978-3-642-36089-3_14.

[46] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge". In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer, 2013, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_11.

[47] Moritz Eysholdt and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way". In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 307–309. ISBN: 978-1-4503-0240-1. DOI: http://doi.acm.org/10.1145/1869542.1869625.

[48] Robert W. Floyd. "On ambiguity in phrase structure languages". In: *Communications of the ACM* 5.10 (1962), p. 526. DOI: http://doi.acm.org/10.1145/368959.368993.

[49] Bryan Ford. "Packrat parsing: simple, powerful, lazy, linear time, functional pearl". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming (ICFP 2002)*. 2002, pp. 36–47. DOI: http://doi.acm.org/10.1145/581478.581483.

[50] Bryan Ford. "Parsing expression grammars: a recognition-based syntactic foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 111–122. ISBN: 1-58113-729-X. DOI: http://doi.acm.org/10.1145/964001.964011.

[51] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. DOI: http://www.martinfowler.com/articles/languageWorkbench.html.

[52] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *OOPSLA '07: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2007. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297033. URL: http://doi.acm.org/10.1145/1297027.1297033.

[53] Seymour Ginsburg and Joseph S. Ullian. "Ambiguity in context free languages". In: *Journal of the ACM* 13.1 (1966), pp. 62–89. DOI: http://doi.acm.org/10.1145/321312.321318.

[54] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. 1st. Addison-Wesley Professional, 2013. ISBN: 0133260224, 9780133260229.

[55] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification. Java SE 8 Edition*. Mar. 2014.

[56] John Gruber. *Markdown: Syntax*. Available on: https://daringfireball.net/projects/markdown/syntax. 2004.

[57] Daco Harkes, Danny M. Groenewegen, and Eelco Visser. "IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. ISBN: 978-3-95977-014-9. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.11.

[58] Jan Heering, P. R. H. Hendriks, Paul Klint, and Jan Rekers. "The syntax definition formalism SDF - reference manual". In: *SIGPLAN Notices* 24.11 (1989), pp. 43–75. DOI: http://doi.acm.org/10.1145/71605.71607.

[59] Jan Heering, Paul Klint, and Jan Rekers. "Incremental Generation of Parsers". In: *PLDI*. 1989, pp. 179–191.

[60] Mark Hills, Paul Klint, and Jurgen Vinju. "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: ACM, 2013, pp. 325–335. ISBN: 978-1-4503-2159-4. DOI: 10.1145/2483760.2483786. URL: http://doi.acm.org/10.1145/2483760.2483786.

[61] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Boston, MA, USA: Addison-Wesley, 2006.

[62] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-61622-X.

[63] Trevor Jim, Yitzhak Mandelbaum, and David Walker. "Semantics and algorithms for data-dependent grammars". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 417–430. ISBN: 978-1-60558-479-9. DOI: http://doi.acm.org/10.1145/1706299.1706347.

[64] S. C. Johnson. *YACC—yet another compiler-compiler*. Tech. rep. CS-32. Murray Hill, N.J.: AT & T Bell Laboratories, 1975.

[65] Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. "Natural and Flexible Error Recovery for Generated Modular Language Environments". In: *ACM Transactions on Programming Languages and Systems* 34.4 (2012), p. 15. DOI: http://doi.acm.org/10.1145/2400676.2400678.

[66] Maartje de Jonge and Eelco Visser. "An Algorithm for Layout Preservation in Refactoring Transformations". In: *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*. Ed. by Anthony M. Sloane and Uwe Aßmann.

Vol. 6940. Lecture Notes in Computer Science. Springer, 2011, pp. 40–59. ISBN: 978-3-642-28829-6. DOI: http://dx.doi.org/10.1007/978-3-642-28830-2_3.

[67] Tomas Kalibera and Richard Jones. "Rigorous Benchmarking in Reasonable Time". In: *ISMM '13: Proceedings of the International Symposium on Memory Management*. ACM, 2013. DOI: 10.1145/2464157.2464160.

[68] Lennart C. L. Kats and Eelco Visser. "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: http://doi.acm.org/10.1145/1869459.1869497.

[69] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. "Pure and declarative syntax definition: paradise lost and regained". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, 2010, pp. 918–932. ISBN: 978-1-4503-0203-6. DOI: http://doi.acm.org/10.1145/1869459.1869535.

[70] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978. ISBN: 0-13-110163-3.

[71] P. Klint. "A Meta-environment for Generating Programming Environments". In: *ACM Trans. Softw. Eng. Methodol.* 2.2 (Apr. 1993), pp. 176–201. ISSN: 1049-331X. DOI: 10.1145/151257.151260. URL: http://doi.acm.org/10.1145/151257.151260.

[72] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. "EASY Meta-programming with Rascal". In: *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by Joao M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 6491. Lecture Notes in Computer Science. Springer, 2009, pp. 222–289. ISBN: 978-3-642-18022-4. DOI: http://dx.doi.org/10.1007/978-3-642-18023-1_6.

[73] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 2009, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.28.

[74] Paul Klint and Eelco Visser. "Using Filters for the Disambiguation of Context-free Grammars". In: *Proceedings of the ASMICS Workshop on Parsing Theory*. Milano, Italy: Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, Oct. 1994.

[75] Donald E. Knuth. "Semantics of Context-Free Languages". In: *In Mathematical Systems Theory*. 1968, pp. 127–145.

[76] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. "Declarative Name Binding and Scope Rules". In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, 2012, pp. 311–331. ISBN: 978-3-642-36089-3. DOI: http://dx.doi.org/10.1007/978-3-642-36089-3_18.

[77] Gabriël Konat, Luís Eduardo de Souza Amorim, Sebastian Erdweg, and Eelco Visser. *Bootstrapping, Default Formatting, and Skeleton Editing in the Spoofax Language Workbench*. Language Workbench Challenge (LWC@SLE). 2016.

[78] J.W.C. Koorn. "GSE: a generic text and structure editor". In: *University of Amsterdam*. 1992, pp. 168–177.

[79] Ralf Lämmel. "Grammar Testing". In: *Fundamental Approaches to Software Engineering, FASE 2001*. Ed. by Heinrich Hußmann. Vol. 2029. Lecture Notes in Computer Science. Springer, 2001, pp. 201–216. ISBN: 3-540-41863-6. DOI: http://link.springer.de/link/service/series/0558/bibs/2029/20290201.htm.

[80] Peter J. Landin. "The next 700 programming languages". In: *Communications of the ACM* 9.3 (1966), pp. 157–166. DOI: http://doi.acm.org/10.1145/365230.365257.

[81] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions". In: *Journal of Software: Evolution and Process* 28.7 (2016). JSME-15-0028.R1, pp. 589–618. ISSN: 2047-7481. DOI: 10.1002/smr.1760. URL: http://dx.doi.org/10.1002/smr.1760.

[82] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. "Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 507–518. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.53. URL: https://doi.org/10.1109/ICSE.2017.53.

[83] Nicolas Laurent and Kim Mens. "Parsing expression grammars made practical". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*. Ed. by Richard F. Paige, Davide Di Ruscio, and Markus Völter. ACM, 2015, pp. 167–172. ISBN: 978-1-4503-3686-4. DOI: http://doi.acm.org/10.1145/2814251.2814265.

[84] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.07: Documentation and user's manual*. Intern report. Inria, July 2018, pp. 1–752. URL: https://hal.inria.fr/hal-00930213.

[85] Huiqing Li, Simon Thompson, and Claus Reinke. "The Haskell Refactorer, HaRe, and its API". In: *Electronic Notes in Theoretical Computer Science* 141.4 (2005), pp. 29–34. DOI: http://dx.doi.org/10.1016/j.entcs.2005.02.053.

[86] Simon Marlow. *Haskell 2010 Language Report*. Available on: https://www.haskell.org/onlinereport/haskell2010. 2010.

[87] Jim Melton. "SQL Language Summary". In: *ACM Computing Surveys* 28.1 (1996), pp. 141–143. DOI: db/journals/csur/Melton96.html.

[88] Cleve B. Moler. "MATLAB: A Mathematical Visualization Laboratory". In: *COMPCON 88, Digest of Papers, Thirty-Third IEEE Computer Society International Conference, San Francisco, California, USA, February 29 - March 4, 1988*. IEEE Computer Society, 1988, pp. 480–481.

[89] Leon Moonen. "Generating Robust Parsers Using Island Grammars". In: *WCRE*. 2001, p. 13. DOI: http://computer.org/proceedings/wcre/1303/13030013abs.htm.

[90] Leon Moonen. "Lightweight Impact Analysis using Island Grammars". In: *10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*. IEEE Computer Society, 2002, pp. 219–228. ISBN: 0-7695-1495-2. DOI: http://computer.org/proceedings/iwpc/1495/14950219abs.htm.

[91] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A Theory of Name Resolution". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 205–231. ISBN: 978-3-662-46668-1. DOI: http://dx.doi.org/10.1007/978-3-662-46669-8_9.

[92] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. *An overview of the Scala programming language*. Tech. rep. 2004.

[93] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. "Hazelnut: a bidirectionally typed structure editor calculus". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 86–99. ISBN: 978-1-4503-4660-3. DOI: http://dl.acm.org/citation.cfm?id=3009900.

[94] Hubert Österle, Jörg Becker, Ulrich Frank, Thomas Hess, Dimitris Karagiannis, Helmut Krcmar, Peter Loos, Peter Mertens, Andreas Oberweis, and Elmar J. Sinz. "Memorandum on design-oriented information systems research". In: *EJIS* 20.1 (2011), pp. 7–10. DOI: http://dx.doi.org/10.1057/ejis.2010.55.

[95] Terence John Parr. *The De finitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers, May 2007.

[96] Terence John Parr and Kathleen Fisher. "LL(*): the foundation of the ANTLR parser generator". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 425–436. ISBN: 978-1-4503-0663-8. DOI: http://doi.acm.org/10.1145/1993498.1993548.

[97] Terence John Parr, Sam Harwell, and Kathleen Fisher. "Adaptive LL(*) parsing: the power of dynamic analysis". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by Andrew P. Black and Todd D. Millstein. ACM, 2014, pp. 579–598. ISBN: 978-1-4503-2585-1. DOI: http://doi.acm.org/10.1145/2660193.2660202.

[98] Thomas Reps and Tim Teitelbaum. "The Synthesizer Generator". In: *SIGSOFT Softw. Eng. Notes* 9.3 (Apr. 1984), pp. 42–48. ISSN: 0163-5948. DOI: 10.1145/390010.808247. URL: http://doi.acm.org/10.1145/390010.808247.

[99] Romain Robbes and Michele Lanza. "How Program History Can Improve Code Completion". In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L Aquila, Italy*. IEEE, 2008, pp. 317–326. DOI: http://dx.doi.org/10.1109/ASE.2008.42.

[100] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. ISBN: 1906966141, 9781906966140.

[101] Lisa F. Rubin. "Syntax-Directed Pretty Printing - A First Step Towards a Syntax-Directed Editor". In: *IEEE Trans. Software Eng.* 9.2 (1983), pp. 119–127.

[102] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manual*. Addison-Wesley-Longman, 1999. ISBN: 978-0-201-30998-0.

[103] D. J. Salomon and G. V. Cormack. "Scannerless NSLR(1) parsing of programming languages". In: *SIGPLAN Not.* 24.7 (1989). DOI: http://doi.acm.org/10.1145/74818.74833.

[104] D.J. Salomon and G.V. Cormack. *The disambiguation and scannerless parsing of complete character-level grammars for programming languages*. Tech. rep. 95/06. Winnipeg, Canada: Department of Computer Science, University of Manitoba, 1995.

[105] Leonardo Vieira dos Santos Reis, Roberto da Silva Bigonha, Vladimir Oliveira Di Iorio, and Luis Eduardo de Souza Amorim. "Adaptable Parsing Expression Grammars". In: *Programming Languages - 16th Brazilian Symposium, SBLP 2012, Natal, Brazil, September 23-28, 2012. Proceedings*. 2012, pp. 72–86. DOI: 10.1007/978-3-642-33182-4_7.

[106] Leonardo Vieira dos Santos Reis, Roberto da Silva Bigonha, Vladimir Oliveira Di Iorio, and Luis Eduardo de Souza Amorim. "The formalization and implementation of Adaptable Parsing Expression Grammars". In: *Sci. Comput. Program.* 96 (2014), pp. 191–210. DOI: 10.1016/j.scico.2014.02.020.

[107] Sylvain Schmitz. "Conservative Ambiguity Detection in Context-Free Grammars". In: *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*. Ed. by Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki. Vol. 4596. Lecture Notes in Computer Science. Springer, 2007, pp. 692–703. ISBN: 978-3-540-73419-2. DOI: http://dx.doi.org/10.1007/978-3-540-73420-8_60.

[108] Martijn M. Schrage. "Proxima – a presentation-oriented editor for structured documents". PhD thesis. Utrecht University, The Netherlands, Oct. 2004. ISBN: 90-393-3803-5. URL: http://www.oblomov.com/Documents/Thesis.pdf.

[109] Elizabeth Scott and Adrian Johnstone. "GLL Parsing". In: *Workshop on Language Descriptions, Tools and Applications (LDTA'09)*. 2009.

[110] T. Sedano. "Code Readability Testing, an Empirical Study". In: *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. Apr. 2016, pp. 111–117. DOI: 10.1109/CSEET.2016.36.

[111] Klaas Sikkel. *Parsing schemata - a framework for specification and analysis of parsing algorithms*. Springer, 1997. ISBN: 978-3-540-61650-4.

[112] Charles Simonyi, Magnus Christerson, and Shane Clifford. "Intentional software". In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*. Ed. by Peri L. Tarr and William R. Cook. ACM, 2006, pp. 451–464. ISBN: 1-59593-348-4. DOI: http://doi.acm.org/10.1145/1167473.1167511.

[113] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. "MetaEdit - A Flexible Graphical Environment for Methodology Modelling". In: *CAiSE*. 1991, pp. 168–193.

[114] Paul G. Sorenson, J. Paul Tremblay, and Andrew J. McAllister. "The Metaview System for Many Specification Environments". In: *IEEE Software* 5.2 (1988), pp. 30–38. DOI: http://doi.ieeecomputersociety.org/10.1109/52.2008.

[115] Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. "Principled syntactic code completion using placeholders". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, 2016, pp. 163–175. ISBN: 978-1-4503-4447-0. DOI: http://dl.acm.org/citation.cfm?id=2997374.

[116] Luis Eduardo de Souza Amorim, Timothée Haudebourg, and Eelco Visser. *Declarative Disambiguation of Deep Priority Conflicts*. Tech. rep. TUD-SERG-2017-014. Delft University of Technology, 2017.

[117] Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. "Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. Ed. by David Pearce 0005, Tanja Mayerhofer, and Friedrich Steimann. ACM, 2018, pp. 3–15. ISBN: 978-1-4503-6029-6. DOI: https://doi.org/10.1145/3276604.3276607.

[118] Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. "Deep priority conflicts in the wild: a pilot study". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, 2017, pp. 55–66. ISBN: 978-1-4503-5525-4. DOI: http://doi.acm.org/10.1145/3136014.3136020.

[119] Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. "Towards Zero-Overhead Disambiguation of Deep Priority Conflicts". In: *Programming Journal* 2.3 (2018), p. 13. DOI: https://doi.org/10.22152/programming-journal.org/2018/2/13.

[120] Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. "Towards Zero-Overhead Disambiguation of Deep Priority Conflicts". In: *Programming Journal* 2 (2018), p. 13.

[121] Andreas Stefik and Susanna Siebert. "An Empirical Investigation into Programming Language Syntax". In: *Trans. Comput. Educ.* 13.4 (Nov. 2013), 19:1–19:40. ISSN: 1946-6226. DOI: 10.1145/2534973. URL: http://doi.acm.org/10.1145/2534973.

[122] Daniel Teichroew, P. Macasovic, E. A. Hershey, and Y. Yamamoto. "Application of the Entity-Relationship Approach to Information Processing Systems Modelling". In: *Entity-Relationship Approach to Systems Analysis and Design. Proc. 1st International Conference on the Entity-Relationship Approach*. Ed. by Peter P. Chen. North-Holland, 1979, pp. 15–38. ISBN: 0-444-85487-8. DOI: db/conf/er/TeichroewMHY79.html.

[123] Mikkel Thorup. "Disambiguating Grammars by Exclusion of Sub-Parse Trees". In: *Acta Informatica* 33.6 (1996), pp. 511–522.

[124] Masaru Tomita. "An Efficient Context-Free Parsing Algorithm for Natural Languages". In: *IJCAI*. 1985, pp. 756–764.

[125] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1985. ISBN: 0898382025.

[126] Eelco Visser. "A Case Study in Optimizing Parsing Schemata by Disambiguation Filters". In: *International Workshop on Parsing Technology (IWPT 1997)*. Massachusetts Institute of Technology. Boston, USA, Sept. 1997, pp. 210–224.

[127] Eelco Visser. "A Family of Syntax Definition Formalisms". In: *ASF+SDF 1995. A Workshop on Generating Tools from Algebraic Specifications*. Ed. by Mark G. J. van den Brand et al. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

[128] Eelco Visser. *A Family of Syntax Definition Formalisms*. Tech. rep. P9706. Programming Research Group, University of Amsterdam, Aug. 1997.

[129] Eelco Visser. "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9". In: *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*. Ed. by Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky. Vol. 3016. Lecture Notes in Computer Science. Springer, 2003, pp. 216–238. ISBN: 3-540-22119-0. DOI: http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3016&spage=216.

[130] Eelco Visser. *Scannerless Generalized-LR Parsing*. Tech. rep. P9707. Programming Research Group, University of Amsterdam, July 1997.

[131] Eelco Visser. "Syntax Definition for Language Prototyping". PhD thesis. University of Amsterdam, Sept. 1997.

[132] Tobi Vollebregt. "Declarative Specification of Template-Based Textual Editors". MA thesis. Delft, The Netherlands: Delft University of Technology, Apr. 2012. DOI: http://resolver.tudelft.nl/uuid:8907468c-b102-4a35-aa84-d49bb2110541.

[133] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. "Declarative specification of template-based textual editors". In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Ed. by Anthony Sloane and Suzana Andova. ACM, 2012, p. 8. ISBN: 978-1-4503-1536-4. DOI: http://doi.acm.org/10.1145/2427048.2427056.

[134] Markus Völter. "Language and IDE Modularization and Composition with MPS". In: *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 7680. Lecture Notes in Computer Science. Springer, 2011, pp. 383–430. ISBN: 978-3-642-35992-7. DOI: http://dx.doi.org/10.1007/978-3-642-35992-7_11.

[135] Markus Völter and Vaclav Pech. "Language modularity with the MPS language workbench". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE, 2012, pp. 1449–1450. ISBN: 978-1-4673-1067-3. DOI: http://dx.doi.org/10.1109/ICSE.2012.6227070.

[136] Markus Völter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. "mbeddr: Instantiating a Language Workbench in the Embedded Software Domain". In: *Journal of Automated Software Engineering* (2013). DOI: http://link.springer.com/article/10.1007%2Fs10515-013-0120-4.

[137] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. "Towards User-Friendly Projectional Editors". In: *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings*. Ed. by Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju. Vol. 8706. Lecture Notes in Computer Science. Springer, 2014, pp. 41–61. ISBN: 978-3-319-11244-2. DOI: http://dx.doi.org/10.1007/978-3-319-11245-9_3.

[138] Philip Wadler. "A Prettier Printer". In: *Journal of Functional Programming*. Palgrave Macmillan, 1998, pp. 223–244.

[139] Niklaus Wirth. "The Programming Language Pascal". In: *Acta Informatica* 1 (1971), pp. 35–63.

Acknowledgments

"Teaching is only demonstrating that it is possible. Learning is making it possible for yourself."

Paulo Coelho, The Pilgrimage

I would like to thank all the *teachers* who helped me during this journey. I would like to thank my previous supervisor Vladimir Di Iorio, who was essential in the first steps towards pursing a PhD. I want to thank Guido Wachsmuth for his great advice and for being so supportive whenever a need arose. I would like to thank my co-promotor, Sebastian Erdweg, for all the interesting conversations, the feedback, and for his work, which provided the foundations for part of the work in this dissertation. I would like to thank my promotor, Eelco Visser. Thank you so much for your help, for believing in me, and for the incredible opportunities throughout these years. I also want to thank all the members of the doctoral committee and the anonymous reviewers of the papers in this dissertation for the time they dedicated to my work.

Some people teach without even realizing they are doing it. I would like to thank all my colleagues at TU Delft for being so friendly and supportive, even when busy with their own work. In particular, I'd like to thank Augusto Passalaqua for the short time we spent working together, but also for your friendship, which I will appreciate for the rest of my life. I would also like to thank Daco Harkes for our many interesting conversations, and for the time we spent together at different conferences. I would like to thank Danny Groenewegen for being such a good friend, helping me many times, and making me feel welcome in the Netherlands. I would like to thank Michael Steindorfer for stepping outside his comfort zone to be part of much of the work in this dissertation. I want to thank Gabriël Konat for helping me more times that I can even count. You passion for computer science and programming is a big inspiration to me. I would also like to thank Peter Mosses for reviewing my work on many occasions, and providing great feedback. Finally, I would like to thank Timothée Haudebourg and Jasper Denkers for giving me the opportunity to be a teacher myself. I have learned so much from you, and you made me very proud. Last but not least, I would like to thank Elmer van Chastelet, Daniel Pelsmaeker, Robbert Krebbers, Martijn Dwars, Casper Poulsen, Tamás Szabó, Sven Keidel, Hendrik van Antwerpen, Jeff Smiths, Vlad Vergu, and Roniet Sharabi. It was a pleasure working with you, and a I wish you all the success in the world.

I was lucky to make many friends during my PhD studies. To all these amazing people, who are too numerous to mention individually, thank you very much! I would never go anywhere without the unconditional support of my family, which I felt even from far away. Uncles and aunts, cousins, I would like to thank you all. I would also like to thank my best friends who are like

brothers to me: Maykom Souza and Jairo da Costa. Maykom, you have always been someone I looked up to, and who was always there for me. Jairo, I have learned so many things from you, and your courage and determination are a great inspiration. I would like to thank my wife, Kate Pitcher, for her love, her advice, and for bringing me so much joy and happiness. I am very grateful to have you in my life. Finally, I would like to thank my mother, Elizalde, whose significant sacrifices have helped me to succeed. Thank you for your love and patience. Without you, none of the above would ever have happened.

The work presented in this thesis was partially funded by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil).

Curriculum Vitae

Luis Eduardo de Souza Amorim

7 June 1989
Born in Unai, Minas Gerais, Brazil

2008-2011
B.Sc. in Computer Science
Universidade Federal de Viçosa
Departamento de Informática
Medalha Arthur Bernardes Mod. Prata (with honor)

2011-2013
M.Sc. in Computer Science
Universidade Federal de Viçosa
Departamento de Informática

2014-2019
Ph.D. in Computer Science
Delft University of Technology
Department of Software Technology

List of Publications

- Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. "Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. Ed. by David Pearce 0005, Tanja Mayerhofer, and Friedrich Steimann. ACM, 2018, pp. 3–15. ISBN: 978-1-4503-6029-6. DOI: https://doi.org/10.1145/3276604.3276607

- Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. "Towards Zero-Overhead Disambiguation of Deep Priority Conflicts". In: *Programming Journal* 2.3 (2018), p. 13. DOI: https://doi.org/10.22152/programming-journal.org/2018/2/13

- Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. "Deep priority conflicts in the wild: a pilot study". In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Ed. by Benoît Combemale, Marjan Mernik, and Bernhard Rumpe. ACM, 2017, pp. 55–66. ISBN: 978-1-4503-5525-4. DOI: http://doi.acm.org/10.1145/3136014.3136020

- Luis Eduardo de Souza Amorim, Timothée Haudebourg, and Eelco Visser. *Declarative Disambiguation of Deep Priority Conflicts*. Tech. rep. TUD-SERG-2017-014. Delft University of Technology, 2017

- Gabriël Konat, Luís Eduardo de Souza Amorim, Sebastian Erdweg, and Eelco Visser. *Bootstrapping, Default Formatting, and Skeleton Editing in the Spoofax Language Workbench*. Language Workbench Challenge (LWC@SLE). 2016

- Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. "Principled syntactic code completion using placeholders". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. Ed. by Tijs van der Storm, Emilie Balland, and Dániel Varró. ACM, 2016, pp. 163–175. ISBN: 978-1-4503-4447-0. DOI: http://dl.acm.org/citation.cfm?id=2997374

- Leonardo Vieira dos Santos Reis, Roberto da Silva Bigonha, Vladimir Oliveira Di Iorio, and Luis Eduardo de Souza Amorim. "The formalization and implementation of Adaptable Parsing Expression Grammars". In: *Sci. Comput. Program.* 96 (2014), pp. 191–210. DOI: 10.1016/j.scico.2014.02.020

- Leonardo Vieira dos Santos Reis, Roberto da Silva Bigonha, Vladimir Oliveira Di Iorio, and Luis Eduardo de Souza Amorim. "Adaptable Parsing Expression Grammars". In: *Programming Languages - 16th Brazilian Symposium, SBLP 2012, Natal, Brazil, September 23-28, 2012. Proceedings*. 2012, pp. 72–86. DOI: 10.1007/978-3-642-33182-4_7