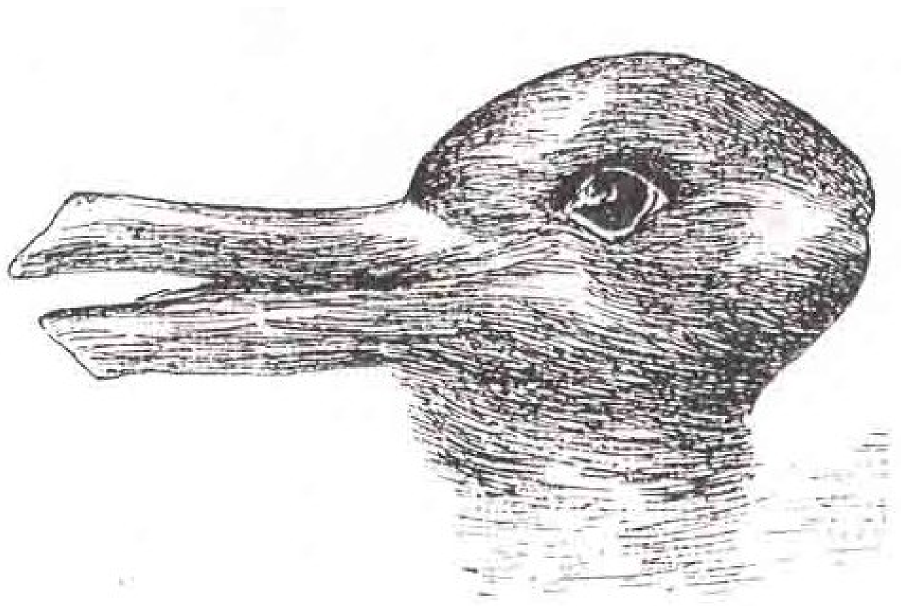
Verifying the Semantics of Disambiguation Rules

Using Parse Tree Repairing for Showing Safety and Completeness of Associativity and Priority Rules



Luka Miljak

Verifying the Semantics of Disambiguation Rules

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Luka Miljak
born in Gouda, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: The famous duck-rabbit ambiguous image. Jastrow, J. (1899). The mind's eye. Popular Science Monthly, 54, 299-312.

Verifying the Semantics of Disambiguation Rules

Author: Luka Miljak
Student id: 4368916

Abstract

Context-free grammars (CFGs) provide a well-known formalism for the specification of programming languages. They describe the structure of a program in terms of parse trees. One major issue of CFGs is ambiguity, where one sentence can sometimes have multiple different parse trees. Some formalisms like SDF3 or YACC allow annotating a grammar with disambiguation rules, such as priority or associativity. Disambiguation rules filter out certain parse trees, making a grammar less ambiguous. Giving a formal semantics for these disambiguation rules is still an ongoing research topic. In this thesis we *verify* an existing semantics for these rules by Souza Amorim and Visser (2019) for a subset of expression grammars. These grammars may contain infix, prefix, and postfix expressions. We verify the semantics by proving that it is both *safe* and *complete*. Safety states adding disambiguation rules will not change the underlying language of the grammar, meaning each sentence in the language will have at least one valid parse tree that does not get filtered out. Completeness guarantees that a grammar is unambiguous, meaning that each sentence in the language will have at most one valid parse tree that does not get filtered out. We have *mechanized* the proofs in the Coq Proof Assistant, increasing the confidence in their correctness. As part of the proofs, we also provide a verified implementation for disambiguation rules.

Supplement Material `https://zenodo.org/record/4680987#.YHRVeOgzZjE`

Thesis Committee:

Chair:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. R. J. Krebbers, Faculty of Science, Radboud University
Committee Member:	Dr. N. Yorke-Smith, Faculty EEMCS, TU Delft
Daily Supervisor:	Dr. R. J. Krebbers, Faculty of Science, Radboud University

Preface

This thesis would not have existed without my supervisors. Thank you Robbert Krebbers, for introducing me to the field of Software Verification. You mentioned that it sometimes feels like playing a video game, which I could not agree more with. Your continued assistance during this thesis has been invaluable. Also a thank you for Eelco Visser, who suggested the thesis subject to me. Without your open-mindedness towards my ideas, this thesis would still remain stuck in its first phase.

Also a special thank you to Bram and Taico, who used up some of their free time to review this thesis.

Luka Miljak
Bergambacht, the Netherlands
April 23, 2021

Contents

Chapter 1

Introduction

Context-free grammars (CFGs) provide a well-known formalism for the specification of programming languages. A CFG describes the structure of a program in terms of parse trees. A parser can be seen as an implementation for a specific CFG, which maps sentences to parse trees. These parsers are typically created by a parser generator, which compiles a CFG into a parser. Figure 1.1 shows a simple CFG for arithmetic expression using SDF3 (*syntax definition formalism*) (Vollebregt, Kats, and Visser 2012; Spoofax Development Team 2020; Souza Amorim and Visser 2020).

The issue with plain context-free grammars, is that they can potentially introduce *ambiguities*. Meaning that a sentence in the language sometimes corresponds to multiple parse trees. A classic example of this are arithmetic expressions. It is generally understood that the expression $15 - 3 \cdot 4$ should be read as $15 - (3 \cdot 4)$, because multiplication takes *priority* over subtraction. However, the context-free grammar from Figure 1.1 does not convey this specific information, and therefore it specifies that the expression can also be read as $(15 - 3) \cdot 4$. The expressions $15 - 3 - 4$ is also ambiguous. We know this should be read left-to-right as $(15 - 3) - 4$, because subtraction is left *associative*. Our CFG specifies that $15 - (3 - 4)$ is also a valid order.

Grammar Rewriting These ambiguities can be resolved by grammar rewriting. For instance, to resolve the ambiguity issue between subtraction and multiplication, we can look at the CFG from Figure 1.2. We have simplified it such that it only contains productions for subtraction and multiplication. In this rewritten grammar, a subtraction cannot appear inside of a multiplication. We accomplished this by creating an additional nonterminal `MulExp`, where a `MulExp` can be derived from `Exp` (line 3), but not the other way around. Because multiplication is a part of `MulExp` and subtraction belongs to `Exp`, we ensure that subtractions can not be derived before multiplications.

In Figure 1.3 we give the full rewritten grammar for our arithmetic language. The problem is that grammar rewriting defeats one of the desirable features of context-free grammars,

```
context-free syntax
    Exp.Lit     = NUM
    Exp.Add     = Exp "+" Exp
    Exp.Sub     = Exp "-" Exp
    Exp.Minus   = "-" Exp
    Exp.Mul     = Exp "*" Exp
    Exp.Div     = Exp "/" Exp
```

Figure 1.1: CFG for simple arithmetic expressions, using SDF3 syntax.

```
context-free syntax
    MulExp.Lit     = NUM
    MulExp.Mul     = MulExp "*" MulExp
    Exp.M          = MulExp
    Exp.Sub        = Exp "-" Exp
```

Figure 1.2: Unambiguous CFG for subtraction and multiplication.

```
context-free syntax
    LitExp.Lit          = NUM
    MinExp.LitExp       = LitExp
    MinExp.Min          = "-" MinExp
    MulDivExp.MinExp    = MinExp
    MulDivExp.Mul       = MulDivExp "*" MinExp
    MulDivExp.Div       = MulDivExp "/" MinExp
    Exp.MulDivExp       = MulDivExp
    Exp.Add             = Exp "+" MulDivExp
    Exp.Sub             = Exp "-" MulDivExp
```

Figure 1.3: Rewritten grammar of Figure 1.1

which is to provide a concise and readable description for the syntax of languages. If somebody asks for the syntax of a language, we would rather give them Figure 1.1 than Figure 1.3. Grammar rewriting introduces complexity and the result can be difficult to read.

Disambiguation Rules We will call disambiguation using priority or associativity *declarative disambiguation rules*. The term *precedence* is often used to describe priority. Declarative disambiguation rules are not uncommon. Take the Java Language Specification as an example. Despite the fact that the full syntax of Java inside the Java Language Specification is unambiguous due to a rewritten grammar, descriptions of the syntax using precedence and associativity are spread throughout the many tutorials (Gosling et al. 2020). The same holds for many other programming languages. Reason being that declarative disambiguation rules are easy to understand. The syntax definition formalisms YACC (Johnson et al. 1975) and SDF3 both include mechanisms for specifying priority and associativity rules. Their corresponding parser generators take these rules into account when creating a parser. Figure 1.4 shows the same arithmetic expression grammar as Figure 1.1 with added priority and associativity declarations. The `left` keyword specifies left-associativity, whereas > specifies priority. Apart from the keyword `right` also being usable (corresponding to right-associativity), SDF3 includes many other disambiguation rules (such as `non-assoc`) which will not be discussed within this thesis. The focus here will solely lie on priority and associativity.

Formal Semantics The many uses of declarative disambiguation rules begs the question: *What are the formal semantics of these rules?* A draft paper by Souza Amorim and Visser (2019) attempts to answer this question by introducing the first direct semantics for declarative disambiguation rules. They define the semantics as *filters* over parse trees. Each disambiguation rule added in the grammar, adds another layer to the filter. Trees that pass all layers of the filter are accepted and considered valid.

The paper makes two claims about the semantics. The first is that the semantics are *safe* under certain conditions. Safety means that disambiguation preserves the underlying language of a grammar. In other words, there should always be *at least one* valid parse tree

```
context-free syntax
    Exp.Lit     = NUM
    Exp.Add     = Exp "+" Exp {left}
    Exp.Sub     = Exp "-" Exp {left}
    Exp.Minus   = "-" Exp
    Exp.Mul     = Exp "*" Exp {left}
    Exp.Div     = Exp "/" Exp {left}
context-free priorities
    Exp.Minus > {left: Exp.Div Exp.Mul} > {left: Exp.Add Exp.Sub}
```

Figure 1.4: CFG for simple arithmetic expressions with priority and associativity declarations.

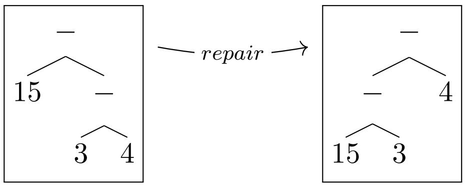


Figure 1.5: Repairing an incorrect parse tree into the correct version.

per sentence. Disambiguation is unsafe if there is a sentence for which all the parse trees are being filtered out. The second is that the semantics are *complete* under certain conditions. Meaning that the disambiguation rules make a grammar completely unambiguous, i.e., each sentence in the language should have *at most one parse* tree. In Chapter 2 we define these concepts formally. Souza Amorim and Visser (2019) have provided sketches of proofs to these claims.

Proof Mechanization The original goal of this thesis was to *mechanize* these proofs in the Coq Proof Assistant (Coq Development Team 2020). A proof assistant is a software tool that aids in giving formal proofs. It allows someone to write down logical definitions and theorems, and prove those theorems. The theorem proving aspect is done interactively, where the user can use tactics that apply some logical deduction to the proof. The advantage of proof assistants over regular handwritten paper proofs is that each step in the proof is checked for correctness. As such, it provides greater certainty that a proof is actually free from errors.

Parse Tree Repairing As said before, we tried to mechanize the safety and completeness proofs from Souza Amorim and Visser (2019) in Coq. Apart from resolving any errors, mechanizing the proofs has shown to be quite a hurdle due to difficult challenges such as showing termination of a rewrite system. Therefore this thesis contains its own new alternative proofs to safety and completeness. In Section 8.1 we go in more detail into the original proof, elaborate on the difficulties faced, and compare it to our alternative proof.

The key idea behind our proof is *parse tree repairing*. Recall that the expression $15 - 3 - 4$ is ambiguous because there are two ways of reading it: $15 - (3 - 4)$ and $(15 - 3) - 4$. The first we consider to be the incorrect, and the second the correct version. In Figure 1.5 we show the two parse trees corresponding to these expressions. The idea of parse tree repairing is that we have some function *repair*. This function takes a parse tree as input and, if incorrect, is able to transform the tree into the correct version. In Chapter 4 we show how this works.

We can use this function to prove safety and completeness. In Chapter 5 we show that safety corresponds to soundness of *repair*, meaning that the output $\mathit{repair}(t) = t'$ is indeed always a valid parse tree for all well-formed trees t, with t and t' corresponding to the same sentence. In Chapter 6 we show that completeness of the semantics corresponds to completeness of *repair*: If we have a valid parse tree t' that corresponds to the same sentence as a tree t, then $\mathit{repair}(t) = t'$. This would show that t' is the only unique valid parse tree.

The main reason this function was conceived, is to aid in proving safety and completeness. However, the function by itself can be considered interesting, as it can be seen an *implementation* of disambiguation rules. This could be a step towards creating a first verified parser generator for CFGs that may include disambiguation rules.

Contributions For this thesis we limited ourselves to a small subset of CFGs, namely *IPP Grammars*. These contain *infix*, *prefix*, and *postfix* expressions.

The contributions are as follows:

1. We define an algorithm that is able to *repair* parse trees, such that they follow the disambiguation rules.
2. We prove that the semantics from Souza Amorim and Visser (2019) is both *safe* and *complete* under certain restrictions. We do this by showing the *repair* function is both sound and complete respectively. Note that for grammars that contain all three types of expressions (infix, prefix, and postfix), we only managed to produce a *partial* proof for completeness. More details can be found in Section 7.3.
3. We prove that the restrictions given for safety and completeness are most general. Meaning that loosening the restriction will violate either safety or completeness.
4. We mechanized the semantics and proofs in the Coq Proof Assistant.[1]

Document Outline This document is structured as follows:

- In Chapter 2 we recap the semantics of disambiguation rules from Souza Amorim and Visser (2019). We limit ourselves here to *infix expression grammars*.
- Chapter 3 is an intermezzo, in which we give an overview of all the proofs that we will give in the three subsequent chapters.
- We show an implementation of these semantics in Chapter 4, by means of parse tree repairing.
- In Chapter 5 we discuss when a grammar is considered safe and provide a proof for the safety property.
- Chapter 6 does the same for the completeness property.
- In Chapter 7 we extend the semantics we give in Chapter 2 and implementation in Chapter 4 to also include *prefix* and *postfix* expressions.
- We discuss work related to this thesis in Chapter 8, which includes a comparison to the original proofs sketches by Souza Amorim and Visser (2019).
- A summary of the mathematical notations and symbols used within this document can be found in Appendix A.

[1]`https://zenodo.org/record/4680987#.YHRVeOgzZjE`

- The definitions, lemmas, and theorems (excluding proofs) written in Coq can be found in Appendix B.
- Appendix C contains the the full algorithm for repairing parse trees, both using mathematical notation as well as a Coq implementation.

Chapter 2

Grammars and Declarative Disambiguation Rules

The main question we will try to answer in this chapter is: *What is the semantics of declarative disambiguation rules*?. We will mostly summarize the semantics from the paper by Souza Amorim and Visser (2019). This semantics is of the most interest as it is the first *direct* semantics for declarative disambiguation rules.

In Section 2.1 we recap the textbook definitions of context-free grammars and languages. Section 2.2 shows the general idea of the semantics for disambiguation rules by means of a *filter* over parse trees. We also define what it means for a filter to be *safe* and *complete*. Finally, in Section 2.3 we give the specifics of the filter for *infix expression grammars*, which is a small subset of context-free grammars. In Chapter 7 we will extend this to also include *prefix* and *postfix* expressions.

2.1 General Context-free Grammars

Before formally defining disambiguation rules, in this section we will first define context-free grammars (CFGs) and languages in the classic way.

Definition 2.1 (Context-Free Grammar)**.** A context-free grammar G is a tuple (Σ, N, P), with Σ a set of terminal symbols, N a set of non-terminals symbols, and P a set of productions. Denote V_G as the set of symbols $\Sigma \cup N$. A production has the form $A.C = X_1...X_n$ with $A \in N$, $n \geqslant 0$, $X_i \in V_G$, and C a unique constructor name. C is used to identify individual productions in a parse tree.

The syntax used for productions corresponds to the syntax of the SDF3 formalism. Figure 1.1 is an example of a CFG for arithmetic expressions with a single nonterminal `Exp` and terminals $\{$`NUM`, `+`, `-`, `*`, `/`$\}$.

Definition 2.2 (Well-Formed Parse Trees)**.** For the tree syntax: a plain symbol represents a leaf of a tree. The notation $[A.C = t_1...t_n]$ represents a tree node with $A.C$ being the label of the node, and $t_1...t_n$ its subtrees. If the context allows it, we omit the label $A.C$ and just write $[t_1...t_n]$.

We define the family T_G^X of well-formed parse trees on a grammar $G = (\Sigma, N, P)$, indexed over $X \in V_G$, representing the root of the tree, inductively as follows:

$$\frac{a \in \Sigma}{a \in T_G^a} \qquad \frac{(A.C = X_1...X_n) \in P, \quad \forall\, 1 \leqslant i \leqslant n.\ t_i \in T_G^{X_i}}{[A.C = t_1...t_n] \in T_G^A}$$

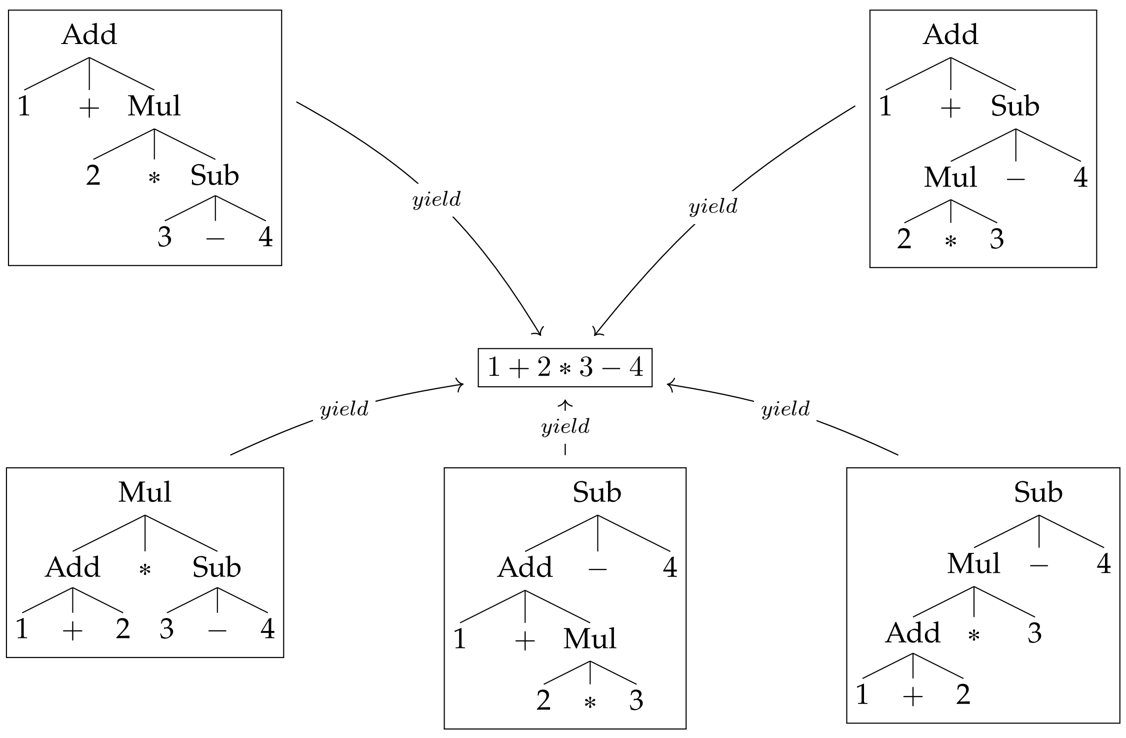


Figure 2.1: Parse trees whose yield is $1 + 2 * 3 - 4$ for a grammar $G = (\Sigma, N, P)$ with $\Sigma = \{\texttt{NUM}, +, -, *\}$, $N = \{\texttt{Exp}\}$, and the productions $P = \{\texttt{Lit}, \texttt{Add}, \texttt{Sub}, \texttt{Mul}\}$ defined in the usual way.

We say $t \in T_G$ if there exists an $X \in V_G$ for which $t \in T_G^X$. [1]

As an example, Figure 2.1 shows several tree diagrams. All of these parse trees are well-formed on the grammar from Figure 1.1.

Definition 2.3 (Yields)**.** The yield $w \in \Sigma^*$ of a parse tree t, written $w = \mathit{yield}(t)$, is the concatenation of its leaves.

The parse trees from Figure 2.1 all have the same yield. This immediately shows that the grammar is ambiguous, as an unambiguous grammar would only have at most one corresponding parse tree for a $w \in \Sigma^*$. If we included the disambiguation rules from Figure 1.4, then the tree in the bottom middle of the figure would be allowed: $[[1 + [2 * 3]] - 4]$.

Definition 2.4 (Language)**.** A sequence of terminals $w \in \Sigma^*$ is in the language of G, if there is a parse tree $t \in T_G$ such that $w = \mathit{yield}(t)$. Let L_G denote the language of G. The sequence w is called a *sentence* in L_G.

We could have also chosen to define a language using *derivations* rather than using parse trees. This would have been an equivalent definition (Aho et al. 2007). With a derivation you start with a symbol such as `Exp`, and keep rewriting using the productions until you get a sentence. For example:

$$\texttt{Exp} \rightarrow \texttt{Exp} + \texttt{Exp} \rightarrow \texttt{Exp} + \texttt{Exp} - \texttt{Exp} \rightarrow \texttt{Exp} + \texttt{Exp} * \texttt{Exp} - \texttt{Exp} \rightarrow \ldots \rightarrow 1 + 2 * 3 - 4.$$

However, as we shall see in Section 2.2, the semantics for disambiguation rules will be defined as *filters* over parse trees. This made yields a more fitting candidate than derivations for defining languages.

[1]Usually a context-free grammar also needs a start symbol $S \in V_G$. However, for the purpose of the disambiguation semantics, this is not a requirement. If it did have a start symbol S, then $t \in T_G$ only if $t \in T_G^S$.

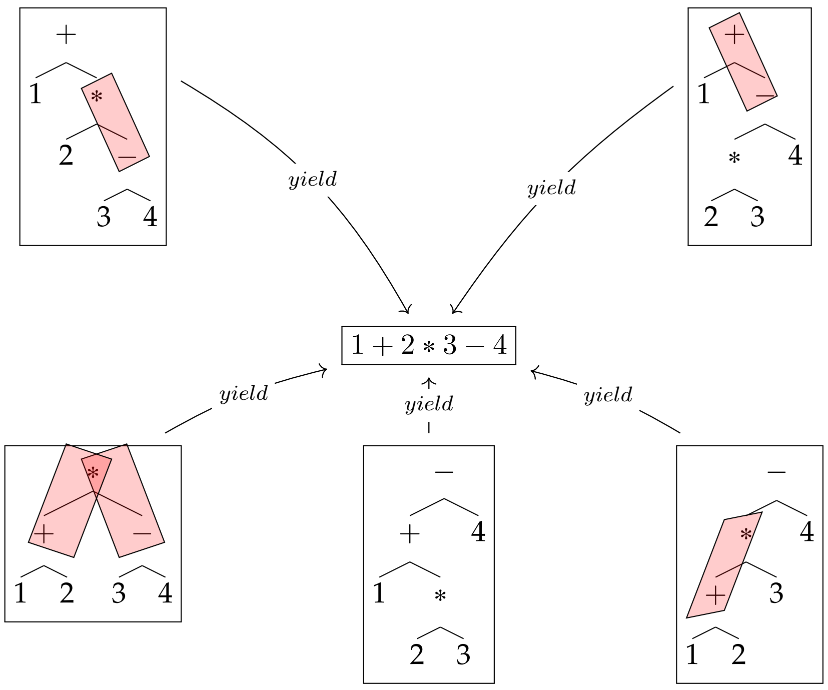


Figure 2.2: Parse trees with highlighted conflicts.

2.2 Disambiguation Filters

Intuitively, what disambiguation does is tell you which parse trees are valid and which are not. The semantics in this chapter are designed in exactly this way. Figure 2.2 is a flash-forward of what we are going to build towards. In this figure we have highlighted certain conflicts within the parse trees. A conflict can be considered two nodes in the tree that are positioned in an invalid manner. What is considered a conflict depends on the disambiguation rules. Conflict-free trees are accepted by the filter and considered valid. In this section we give definitions that make the idea of filters and conflict patterns concrete. In the next section we explain how this relates to specific disambiguation rules. Describing disambiguation by means of filters has first been introduced by Klint and Visser (1994).

Definition 2.5 (Disambiguation Filter)**.** For a grammar G, a disambiguation filter is a function $F : T_G \rightarrow \textit{bool}$. It tells whether if a parse tree t is "valid" or not.

Definition 2.6 (Disambiguated Language)**.** A sequence of terminals $w \in \Sigma^*$ is in the disambiguated language of G using F, denoted by L_G^F, if there is a parse tree $t \in T_G$, such that $F(t) = \textit{true}$, and $\textit{yield}(t) = w$.

Now that we have formalized the idea of filters, we start with formalizing the idea of conflicts. To achieve this, we say that each conflict is represented by some *pattern* the tree can match. Suppose we have a set of these supposed patterns. If a tree t matches *any* of these patterns, we want $F(t)$ to return *false*, or *true* otherwise. The following definitions make this more concrete.

Definition 2.7 (Tree Patterns)**.** We define the set TP_G of well-formed tree patterns on a grammar G inductively as follows:

$$\frac{X \in V_G}{\_ \in TP_G^X} \qquad \frac{(A.C = X_1 ... X_n) \in P, \quad \forall\, 1 \leqslant i \leqslant n.\ q_i \in TP_G^{X_i}}{[A.C = q_1 ... q_n] \in TP_G^A}$$

Where $q \in TP_G$ if there is an $X \in V_G$ and $q \in TP_G^X$.

Tree patterns can be viewed as ordinary parse trees that have 'holes' in them, represented by underscores _. A parse tree matches a tree pattern, if the holes in the pattern can be substituted such that they become equal.

Definition 2.8 (Matching)**.** For a grammar G, a tree $t \in T_G$ and pattern $q \in TP_G$, define $M(t, q)$ (t matches q) inductively as follows:

$$\frac{}{M(t, \_)} \qquad \frac{\forall\, 1 \leqslant i \leqslant n.\ M(t_i, q_i)}{M([A.C = t_1 ... t_n], [A.C = q_1 ... q_n])}$$

Definition 2.9 (Set Matching)**.** For a grammar G, a tree $t \in T_G$ and set of tree patterns $Q \subseteq TP_G$, we say $M(t, Q)$ (t matches Q) if there exists a $q \in Q$ such that $M(t, q)$.

Definition 2.10 (Subtree Exclusion and Conflict Patterns)**.** For a grammar G and $Q \subseteq TP_G$ a set of tree patterns, also called the *conflict patterns*. We define the set T_G^Q of well-formed parse trees under subtree exclusion inductively as follows:

$$\frac{a \in \Sigma}{a \in T_G^Q} \qquad \frac{t \in T_G, \quad \neg M(t, Q), \quad \forall\, 1 \leqslant i \leqslant n.\ t_i \in T_G^Q}{t = [A.C = t_1 ... t_n] \in T_G^Q}$$

This means that for t to be a well-formed parse tree under subtree exclusion, both t and all of its subtrees are not allowed to match *any* conflict pattern in Q. In this case, t is called a *conflict-free* parse tree.

Definition 2.11 (Subtree Exclusion Filter)**.** For a set of tree patterns Q, define the filter F_Q, called the subtree exclusion filter, as follows:

$$F_Q(t) = \begin{cases} \textit{true} & \text{if } t \in T_G^Q \\ \textit{false} & \text{if } t \notin T_G^Q \end{cases}$$

The question regarding the semantics now becomes: How do we determine the set Q of subtree exclusion patterns in an intuitive way? We will answer this question in Section 2.3. What we can still do here is define the core properties we want our semantics to adhere to. These are *safety* and *completeness*.

Definition 2.12 (Safe Filter)**.** Given a grammar G, a disambiguation filter F is a safe filter if it does not change the underlying language, so $L_G = L_G^F$. In other words, for all $w \in L_G$, there should be *at least* one $t \in T_G$ such that $F(t) = \textit{true}$ and $\textit{yield}(t) = w$.

The reason why we want a safe filter, is because we do not want to accidentally reject sentences from our language by introducing disambiguation rules. The goal of disambiguation is to reduce the number of valid parse trees per sentence, not all of the parse trees

Definition 2.13 (Complete Filter)**.** Given a grammar G, a disambiguation filter F is a complete filter if for all $w \in L_G$, there is *at most* one valid corresponding parse tree t. Formally: If $t_1, t_2 \in T_G$, $F(t_1) = F(t_2) = \textit{true}$, and $\textit{yield}(t_1) = \textit{yield}(t_2)$, then $t_1 = t_2$.

A complete filter *guarantees* us that our new disambiguated language is unambiguous.

Looking back at Figure 2.2: Safety and completeness together should mean that there is *exactly* one tree that is conflict-free. This should apply to all sentences.

```
1  context-free syntax
2      Exp.Lit = NUM
3      Exp.Add = Exp "+" Exp
4      Exp.Sub = Exp "-" Exp
5      Exp.Mul = Exp "*" Exp
```

Figure 2.3: Example of an Infix Expression Grammar

2.3 Disambiguation Semantics for Infix Expression Grammars

In this thesis we limit ourselves to *expression grammars*, as they are the most applicable for associativity and priority disambiguation rules. An expression grammar is a CFG that only has one nonterminal A. Souza Amorim and Visser (2019) specify how to obtain the set of conflict patterns Q from an expression grammar. They do this incrementally for expression grammars of increasing complexity, starting with the simplest form: *Infix expression grammars*. In this thesis we will follow this pattern by starting with these infix grammars. In Chapter 7 we will extend this to also include *prefix* and *postfix expressions*.

Definition 2.14 (Infix Expression Grammar)**.** An infix expression grammar (IG) is a tuple (L, O_{in}), where L and O_{in} are disjoint sets of lexical and infix operator symbols. It represents a context-free grammar $G = (L \cup O_{in}, \{A\}, P)$. Here, A is a single nonterminal symbol. The set P contains productions

$$\begin{aligned} &A.C = l && \text{(Atomic Productions), and} \\ &A.C = A\ o\ A && \text{(Infix Productions)} \end{aligned}$$

for all $l \in L$ and $o \in O_{in}$.

Figure 2.3 shows an example of an infix expression grammar. The production on line 2 is an atomic production, and the productions on lines 3-5 are infix productions. The parse trees from Figure 2.1 are also well-formed for this grammar.

Because each operator o and lexical symbol l is unique, for simplicity we can use those symbols to denote productions rather than using their corresponding constructor names C. For denoting parse trees, we often ignore the $A.C$ label of a node. Instead we can just write $[t_1\ o\ t_2]$ rather than $[A.C = t_1\ o\ t_2]$. For a visual tree diagram we can also use the terminals to label nodes (see Figure 2.2 as an example).

We will now move on to defining disambiguation rules.

Definition 2.15 (Disambiguation Rules for Infix Grammars)**.** A disambiguation rule set PR for an IG $G = (L, O_{in})$ is a tuple $(>, \texttt{left}, \texttt{right})$. This consists of a priority relation $>$ and associativity relations `left` and `right` between the infix productions of G. So $>, \texttt{left}, \texttt{right} \subseteq O_{in} \times O_{in}$. The disambiguation rules are *well-formed* if there is at most one rule for a pair of operators. For instance, having both $o_1\ \texttt{left}\ o_2$ and $o_1\ \texttt{right}\ o_2$ is not allowed.

The well-formedness property we enforce on the disambiguation rules PR differs from the well-formedness property by Souza Amorim and Visser (2019). We will justify our definition at the end of this section. In Sections 5.1 and 6.1 we give additional comparisons to the different restrictions.

In Figure 2.4 we define a set of disambiguation rules for the grammar of Figure 2.3. We defined it in what should be the intuitive way: Meaning that multiplication takes priority over addition and subtraction, and everything else should be left-associative.

We now want to create a logical set of conflict patterns Q, for usage in the subtree exclusion method from Definition 2.10. It should reflect the following wanted behavior of priority and associativity:

```
Exp.Add left Exp.Sub, Exp.Sub left Exp.Add,
Exp.Add left Exp.Add, Exp.Sub left Exp.Sub,
Exp.Mul > Exp.Add, Exp.Mul > Exp.Sub, Exp.Mul left Exp.Mul
```

Figure 2.4: A set of disambiguation rules for the grammar of Figure 2.3

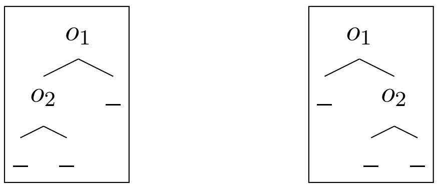


Figure 2.5: The common infix conflict patterns *CL* (left) and *CR* (right)

1. If $o_1 > o_2$, it means that we do not want to allow parse trees where a node o_1 has an o_2 node as a direct subtree.
2. If o_1 `left` o_2, we do not want an o_2 node to appear as a direct right subtree of an o_1 node.
3. If o_1 `right` o_2, we do not want an o_2 node to appear as a direct left subtree of an o_1 node.

Definition 2.16 (Common Infix Conflict Patterns)**.** There are two types of tree patterns that will be commonly used. We define them here as functions:

$$CL(o_1, o_2) = [[\_\ o_2\ \_]\ o_1\ \_]$$
$$CR(o_1, o_2) = [\_\ o_1\ [\_\ o_2\ \_]]$$

CL is also called a *left-conflict* and *CR* a *right-conflict*.

Informally, this means that if we have $CL(o_1, o_2)$ or $CR(o_1, o_2)$ in the set of conflict-patterns Q, then an o_2 node is not allowed to be a direct left-subtree or a direct right-subtree of an o_1 node, respectively. Figure 2.5 shows these two conflict patterns visually.

Definition 2.17 (Infix Conflict Patterns)**.** Let G be an infix expression grammar and $PR = (>, \texttt{left}, \texttt{right})$ a disambiguation rule set for G. We define the set $Q(PR)$ of conflict patterns as follows:

$$\frac{o_1 > o_2}{CR(o_1, o_2) \in Q(PR)} \qquad \frac{o_1 > o_2}{CL(o_1, o_2) \in Q(PR)}$$
$$\frac{o_1\ \texttt{left}\ o_2}{CR(o_1, o_2) \in Q(PR)} \qquad \frac{o_1\ \texttt{right}\ o_2}{CL(o_1, o_2) \in Q(PR)}$$

Figure 2.6 shows all the conflict patterns generated by the disambiguation rules from Figure 2.4. In Figure 2.7 we have redrawn the parse trees for the expression $1 + 2 * 3 - 4$ with each conflict highlighted. It shows that the tree we considered to be correct (the bottom-center one), is indeed the only conflict-free parse tree for this yield.

Restrictions on Disambiguation Rules In Definition 2.15 we imposed a well-formedness restriction on disambiguation rules *PR*. That is, a pair of infix operators (o_1, o_2) can belong in at most one of the tree relations $>$, `left`, or `right`. Now that we have given the semantics in Definition 2.17 we can justify these restrictions. Suppose we have o_1 `left` o_2 and o_1 `right` o_2. This generates the conflict patterns $CR(o_1, o_2)$ and $CL(o_1, o_2)$. These same conflict patterns

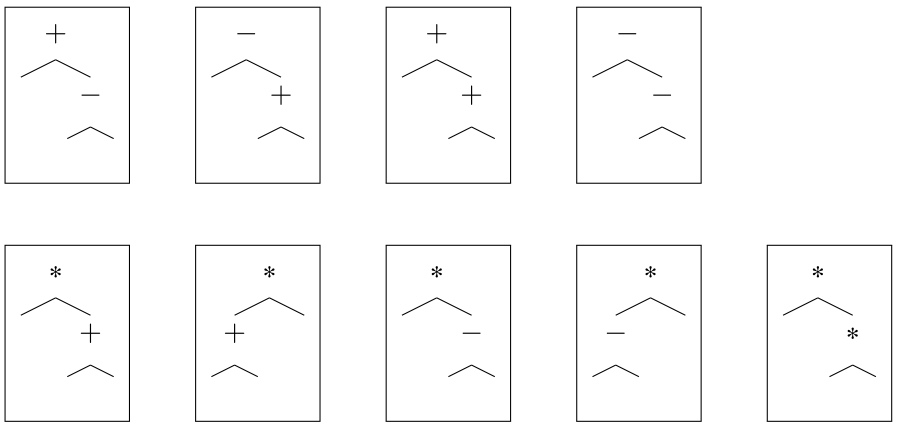


Figure 2.6: Conflict patterns for the disambiguation relations from Figure 2.4. The underscores have been hidden so as not to cause confusion with the minus symbol.

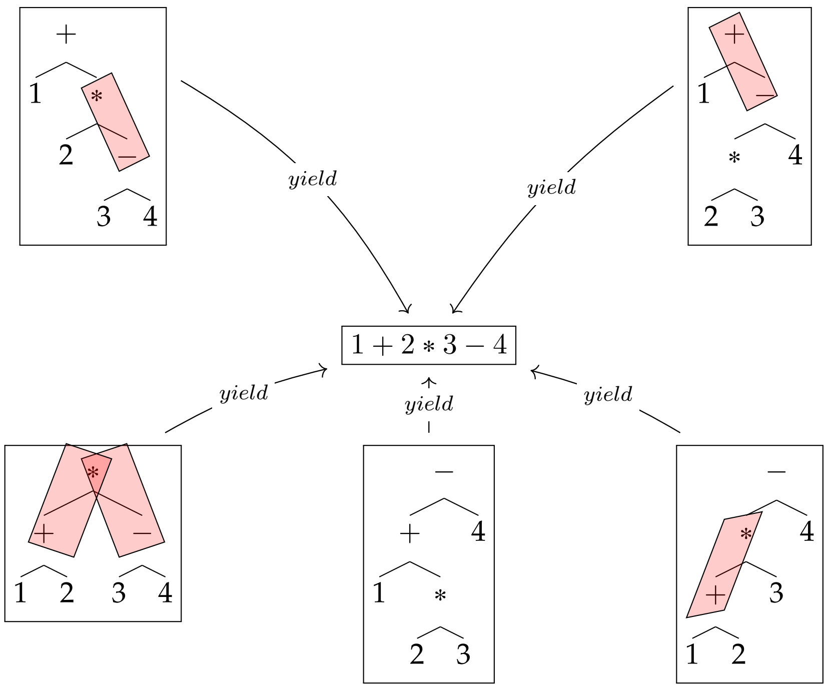


Figure 2.7: Parse trees with highlighted conflicts. This figure is a copy of Figure 2.2, as it has been a few pages back.

can be obtained by having rule $o_1 > o_2$. Therefore it is impractical to make two operators be both left- as well as right-associative. Similarly, if we have $o_1 > o_2$, then adding the rules o_1 `left` o_2 or o_1 `right` o_2 will have no effect.

Later in this thesis, we will impose additional restrictions on the disambiguation rules PR. This would include properties such as priority $>$ being transitive and irreflexive. Associativity should be symmetric. These restrictions will together ensure safety and completeness of the filter. However, we defer defining these restrictions to the relevant parts about safety (Chapter 5) and completeness (Chapter 6).

Chapter 3

Intermezzo — Overview of Proofs

This short intermezzo exists as an overview for Chapters 4 to 6. In these we will prove that the semantics we gave in Chapter 2 is both safe and complete for infix expression grammars. Figure 3.1 gives an overview of every definition relevant to to safety and completeness, including every relation between them which we have proven.

Safety and Completeness in General Back in Section 2.3 we mentioned that we will impose restrictions on the disambiguation rules PR. These restrictions are needed as a premise for safety and completeness. In total, we will have three different definitions of safety, as well as three for completeness:

1. Safety and completeness of disambiguation rules PR (Definitions 5.3 and 6.3).
2. Safety and completeness of the conflict patterns $Q(PR)$ (Definitions 5.1 and 6.1).
3. Safety and completeness of the subtree exclusion filter $F_{Q(PR)}$ (Definitions 2.12 and 2.13).

Safety and completeness of PR and $Q(PR)$ are the restrictions we will impose to ensure safety and completeness of the filter. When we speak of safety and completeness in general terms, we talk about the relation between these three. Safety in general means

$$\text{safe } PR \Rightarrow \text{safe } Q(PR) \wedge \text{safe } Q(PR) \Rightarrow \text{safe } F_{Q(PR)}.$$

Similarly, completeness in general means

$$\text{complete } PR \Rightarrow \text{complete } Q(PR) \wedge \text{complete } Q(PR) \Rightarrow \text{complete } F_{Q(PR)}.$$

Most General Restrictions While proving the implications above is the core goal of those chapters, we will also show some extra properties hold. For safety, we also show that the restrictions we impose on PR or $Q(PR)$ are the most general restrictions we can have for safety of $F_{Q(PR)}$. In other words, we show the following:

$$\text{safe } PR \Leftarrow \text{safe } Q(PR) \wedge \text{safe } Q(PR) \Leftarrow \text{safe } F_{Q(PR)}.$$

Unfortunately, we did not manage to find a most general set of restrictions for completeness. However, we do show the completeness restrictions are most general for safe disambiguation rules. In other words, assuming we have a safe set of disambiguation rules PR, the following holds:

$$\text{complete } PR \Leftarrow \text{complete } Q(PR) \wedge \text{complete } Q(PR) \Leftarrow \text{complete } F_{Q(PR)}.$$

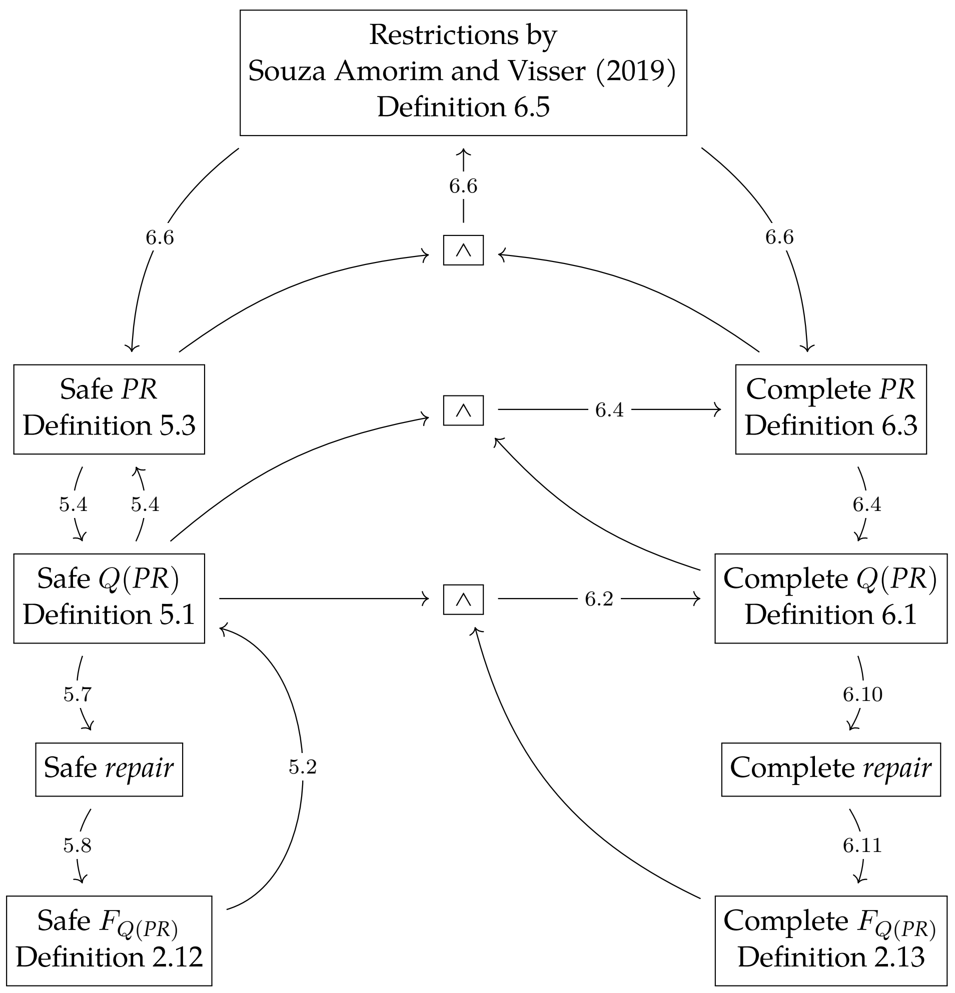


Figure 3.1: Overview of relevant definitions and the proven relations between them. Each node represents a property, for which we give the corresponding definition. Each edge represents an implication which we have proven in this thesis, for which we also give a reference to where we have proven it.

Restrictions by Souza Amorim and Visser (2019) The paper by Souza Amorim and Visser (2019) also impose restrictions on the disambiguation rules *PR*. However, they define it slightly differently. Instead of just safety and completeness restrictions, the paper also adds a well-formedness property to the disambiguation rules that is tighter than our well-formedness property from Definition 2.15. However, the essence of the restrictions is exactly the same. This gives rise to the following property:

$$\text{well-formed } PR \wedge \text{safe } PR \wedge \text{complete } PR$$
$$\iff$$
$$\text{restrictions by Souza Amorim and Visser (2019) on } PR.$$

The Repair Function In Chapter 4 we define the *repair* function. This function has three main purposes:

1. Being an implementation for disambiguation rules.
2. Assisting in proving safety.

3. Assisting in proving completeness.

As we shall see in Chapter 5, proving safety requires the ability to transform *any* well-formed parse tree into a conflict-free parse tree that has the same yield. We call this repairing the tree, as we are fixing a parse tree by removing all its conflicts. In other words, safety of *repair* implies safety of $F_{Q(PR)}$. Instead of safety of repair, we also sometimes use the term *soundness*.

Similarly, we prove completeness by also showing *repair* is complete. Repair is complete if for any two yield-equivalent trees t and t', if t' is conflict-free, then $\mathit{repair}(t) = t'$.

Chapter 4

Implementation for Disambiguation Semantics

In this chapter we will propose an implementation for the disambiguation semantics for infix expression grammars (IGs) discussed in Chapter 2. Before doing that, we first have to answer the question: What can even be considered an implementation for disambiguation semantics? The obvious answer is to literally copy the semantics, i.e., a *filter* that takes a set of a parse trees as input and removes all parse trees that have conflicts. This is however terribly inefficient. A sentence that contains just 15 infix operators will have nearly 10 million different well-formed parse trees. The bottleneck here lies in the parser. We expect it to give us all well-formed parse trees as input for our filter. This is an unreasonable thing to ask. Existing implementations, such as YACC and SDF3, handle this by compiling disambiguation rules directly into the parser itself (Johnson et al. 1975; Souza Amorim and Visser 2020). The problem with this method is that it makes it a more difficult candidate for verification. Ideally we want to split the parsing process from the disambiguation process.

Repairing Parse Trees The solution to this problem actually originally comes from our attempt to prove *safety* in Chapter 5. Proving safety requires the ability to transform *any* well-formed parse tree into a conflict-free parse tree that has the same yield. We therefore construct a *repair* function, which has also shown to be useful in proving completeness (Chapter 6).

So how can this algorithm be used as an effective implementation of disambiguation rules? Firstly, because instead of requiring the parser to give us *all* well-formed parse trees, we reduce the problem to just finding *one*. We can apply *repair* on this parse tree to obtain a parse tree that follows the disambiguation semantics. The only restriction is that the disambiguation rules *PR* (Definition 2.15) should be *safe*, otherwise $\mathit{repair}(t)$ will not be conflict-free. We will elaborate on this in Chapter 5.

Since $\mathit{repair}(t)$ will always produce exactly one result, it can be problematic if the disambiguation rules might still allow ambiguities. This means that this is not really a full implementation of the semantics. Therefore *PR* should also be *complete* in order to make $\mathit{repair}(t)$ the unique result we want it to be. This will be elaborated on in Chapter 6.

4.1 The Repair Function

As mentioned above, we will craft a function $\mathit{repair} : T_G \to T_G$. So the goal is to transform a parse tree t into a tree $\mathit{repair}(t) = t'$ such that $\mathit{yield}(t) = \mathit{yield}(t')$ and t' has no conflicts. We will construct t' in a bottom-up manner: First we start with an 'empty' t'. Then we will iterate through all symbols in t in a right-to-left order, inserting each symbol to the left side

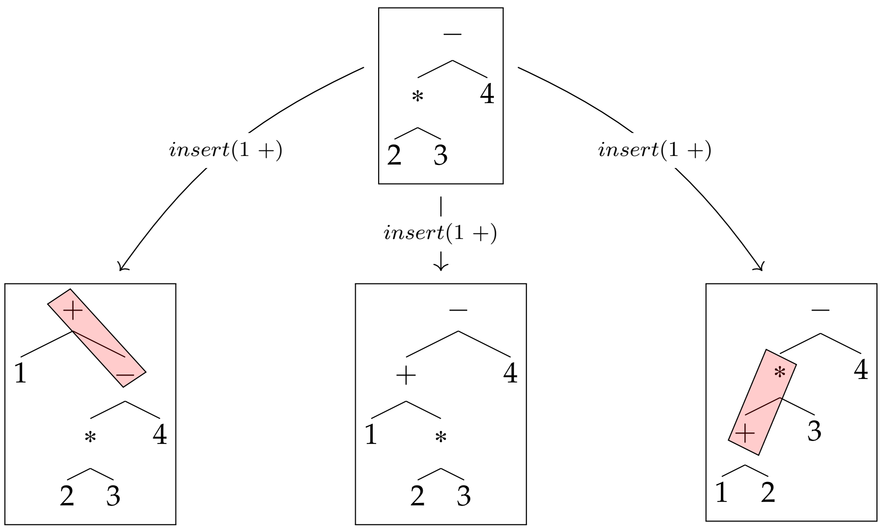


Figure 4.1: Multiple possibilities for left-insertion.

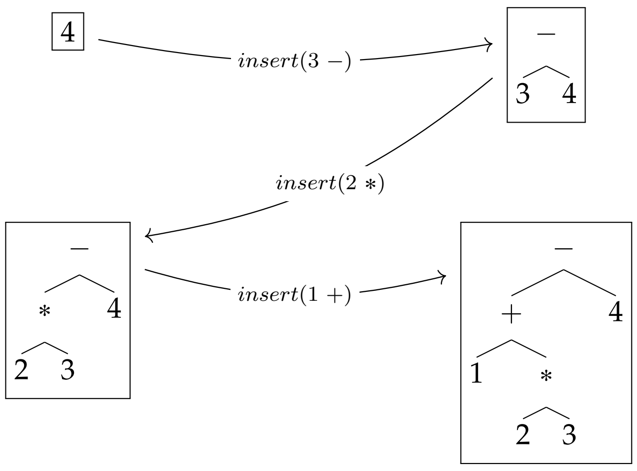


Figure 4.2: Repairing a tree with yield $1 + 2 * 3 - 4$ using left-insertion. Note that *insert* corresponds to Definition 4.1.

of t'. Each insertion should be safe, i.e., not introduce any conflicts. The result is a tree t' that has the same yield as t and is also conflict-free. Figure 4.2 illustrates this concept.

Insertion How do we approach insertion into a tree? Suppose we have some lexical symbol l and infix operator o, there are several ways to insert this to the leftmost side of a tree. Take a look at Figure 4.1. There we start with the tree $[[2 * 3] - 4]$. The goal is to insert the number "1", and the "+" operator to the left side of the tree. There are exactly three places we can do this: Above the minus sign, in between the minus and multiplication, and below the multiplication. In all cases the yield becomes $1 + 2 * 3 - 4$. Which of these three should we pick? Our goal is to not introduce any conflicts. Using the disambiguation rules from Figure 2.4 we have that $CR(+, -)$ and $CL(*, +)$ are in Q. We have highlighted the parts in the trees that match these conflict patterns. This means that the only valid insertion from Figure 4.1 is the middle tree, as it does not match any of the conflict patterns.

Now that the basic concept has been explained, we will formalize this idea of repairing trees using three separate functions:

1. *repair* : $T_G \to T_G$, the top level repair function.

2. $\mathit{repair}_{in} : (T_G \times O_{in} \times T_G) \to T_G$, for repairing infix nodes. The term $\mathit{repair}_{in}(t_1, o, t_2)$ is used to repair a tree $[t_1\ o\ t_2]$.

3. $\mathit{insert}_{in} : (L \times O_{in} \times T_G) \to T_G$, also for repairing infix nodes, but where the left subtree is (or has been reduced to) a lexical symbol in L. The function has the insertion behavior discussed before. If we call $\mathit{insert}_{in}(l_1, o, t_2)$, it means we try to repair a tree $[l_1\ o\ t_2]$ by left-inserting the symbols l_1 and o safely in t_2.

We used this naming convention in anticipation of prefix and postfix grammars (Chapter 7), where we will also have a insert_{pre} and a insert_{post} function.

Remark: Before we give the full definitions of these three functions, first we give a remark to help avoid confusion. We just introduced insert_{in} by giving it the header $(L \times O_{in} \times T_G) \to T_G$. In our definition we will generalize this so that the first argument in L can also be an entire tree in T_G. Meaning that the header is actually $(T_G \times O_{in} \times T_G) \to T_G$. Doing this allows inserting entire trees into another tree, rather than just a lexical symbol. When we introduce prefix and postfix expression in Chapter 7, this will have practical use cases. Furthermore, this generalized function will be useful when proving completeness in Chapter 6.

Definition 4.1 (IG insert_{in})**.** For an infix grammar $G = (L, O_{in})$ with a set of conflict patterns Q, we define the function $\mathit{insert}_{in} : (T_G \times O_{in} \times T_G) \to T_G$ as follows:

$$\mathit{insert}_{in}(t_1, o, t_2) = \begin{cases} [t_1\ o\ l_2], & \text{if } t_2 = l_2 \in L \\ [t_1\ o\ [t_{21}\ o_2\ t_{22}]], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}] \text{ and } CR(o, o_2) \notin Q \\ [\mathit{insert}_{in}(t_1, o, t_{21})\ o_2\ t_{22}], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}] \text{ and } CR(o, o_2) \in Q \end{cases}$$

The behavior of this function is that it tries to insert t_1 and o as high as possible in the tree t_2. If it would introduce a *right-conflict* (*CR*-conflict), we recursively try to insert it in the left branch of t_2. Note how our function only explicitly avoids right-conflicts and not left-conflicts. In Chapter 5, which is about safety, we will prove that it also avoids left-conflicts under certain conditions.

Definition 4.2 (IG repair_{in})**.** For an infix grammar $G = (L, O_{in})$ with a set of conflict patterns Q, we define the function $\mathit{repair}_{in} : (T_G \times O_{in} \times T_G) \to T_G$ as follows:

$$\mathit{repair}_{in}(t_1, o, t_2) = \begin{cases} \mathit{insert}_{in}(l_1, o, t_2), & \text{if } t_1 = l_1 \in L \\ \mathit{repair}_{in}(t_{11}, o_1, \mathit{repair}_{in}(t_{12}, o, t_2)) & \text{if } t_1 = [t_{11}\ o_1\ t_{12}] \end{cases}$$

This function reduces the left subtree in order to be able to use insertion. This has the right-to-left iteration behavior as we mentioned at the start of this section. We iterate on the input tree t_1 from right-to-left, inserting each symbol to the left of the tree t_2.

Now we can finally define the *repair* function.

Definition 4.3 (IG *repair*)**.** For an infix grammar $G = (L, O_{in})$ with a set of conflict patterns Q, we define the function $\mathit{repair} : T_G \to T_G$ as follows:

$$\mathit{repair}(t) = \begin{cases} l, & \text{if } t = l \in L \\ \mathit{repair}_{in}(t_1, o, \mathit{repair}(t_2)), & \text{if } t = [t_1\ o\ t_2] \end{cases}$$

What *repair* essentially boils down to is a composite application of *insert*$_{in}$ for all symbols of the tree. Let us take a random example tree from before, that has conflicts. We are going to fix the tree $[[1+2]*[3-4]]$, which corresponds to the bottom-left tree from Figure 2.7.

$$\begin{aligned} repair([[1+2]*[3-4]]) &= repair_{in}([1+2], *, repair([3-4])) \\ &= repair_{in}([1+2], *, repair_{in}(3, -, repair(4))) \\ &= repair_{in}([1+2], *, repair_{in}(3, -, 4)) \\ &= repair_{in}(1, +, repair_{in}(2, *, repair_{in}(3, -, 4))) \\ &= insert_{in}(1, +, insert_{in}(2, *, insert_{in}(3, -, 4))) \\ &= \text{See Figure 4.2} \end{aligned}$$

4.2 The Repair Function and Yields

While the two most interesting properties of *repair*, safety[1] and completeness, will be discussed in Chapter 5 and Chapter 6, we can already state two useful and easy to prove properties of *repair*. The first property states that *repair* preserves the yield of a tree. Meaning that the resulting tree has the same yield as the input tree t. The second property states that *repair* is 'fully dependent' on the yield of the tree. Meaning that the function will produce the same result for any trees that have the same yields. Another way to state this is that yield equivalence is the same as repair equivalence.

Lemma 4.4 (Repair Preserves Yields). *For all trees $t \in T_G$ and conflict patterns Q, it follows that $yield(t) = yield(repair(t))$.*

Proof. By the fact that insertion iterates over trees in right-to-left order, and each symbol gets inserted on the left-most side of the tree. We can formally prove this by first proving the auxiliary properties $yield(insert_{in}(t_1, o, t_2)) = yield([t_1\ o\ t_2])$ and $yield(repair_{in}(t_1, o, t_2)) = yield([t_1\ o\ t_2])$. The former can be shown by induction over the structure of tree t_2, and the latter by induction over t_1. □

Lemma 4.5 (Repair is Fully Yield-Dependent). *For any two trees $t_1, t_2 \in T_G$ and conflict patterns Q, if $yield(t_1) = yield(t_2)$, then it follows that $repair(t_1) = repair(t_2)$.*

Proof. Suppose we have a tree t. Expanding $repair(t)$ results in a nested application of *insert*$_{in}$ on the yield of t. We can also express this using the equations

$$\begin{aligned} repair(t) &= insert_{in}(l_1\ o_1, ...insert_{in}(l_{n-1}\ o_{n-1}, l_n)...) \\ yield(t) &= l_1\ o_1\ ...\ l_{n-1}\ o_{n-1}\ l_n. \end{aligned}$$

To help explain this, we can take the example from the previous section:

$$repair([[1+2]*[3-4]]) = ... = insert_{in}(1, +, insert_{in}(2, *, insert_{in}(3, -, 4)))$$

Concatenating each terminal symbol from the final expression here gives $1+2*3-4$, which is the yield of $[[1+2]*[3-4]]$.

We formalize this idea with a *parse* function that takes as input a sentence and outputs a tree.

$$parse(w) = \begin{cases} l, & \text{if } w = l \in L \\ insert_{in}(l, o, parse(w_t)), & \text{if } w = l\ o\ w_t \\ error, & \text{otherwise} \end{cases}$$

[1]Safety here can also be interpreted as soundness.

Note that this parser returns an *error* in case the input w is not a valid sentence. It is not hard to show that $\mathit{parse}(\mathit{yield}(t)) = \mathit{repair}(t)$. Therefore, if we have $\mathit{yield}(t_1) = \mathit{yield}(t_2)$, we get the equality $\mathit{parse}(\mathit{yield}(t_1)) = \mathit{parse}(\mathit{yield}(t_2))$, which in turn can be rewritten to $\mathit{repair}(t_1) = \mathit{repair}(t_2)$. $\square$

For the proof of Lemma 4.5 we built a parser for infix expression grammars. We could turn this into a *verified* parser by also giving a soundness proof (i.e., if $\mathit{parse}(w) = t$, then $\mathit{yield}(t) = w$). While not that difficult, this lies outside of the scope of this thesis. Building a parser for (infix) expression grammars is easy. Our interest lies more in disambiguating an existing tree using *repair*.

Chapter 5

Safe Disambiguation

In Definition 2.12 we defined a safe filter as one that does not change the underlying language. Meaning that if we have a sentence in the language, that sentence should also be in the disambiguated language. If we expand this definition for the subtree exclusion filter, it says the following: For all sentences w, if there exists a well-formed parse tree t with $\mathit{yield}(t) = w$, then there should also exists a well-formed conflict-free parse tree t' with $\mathit{yield}(t') = w$. To obtain the tree t', in Chapter 4 we have created a function *repair* $: T_G \to T_G$ from trees to conflict-free trees. This function preserves the yield of the input tree t (Lemma 4.4). The only thing we have not proven is that the output t' is indeed conflict-free.

In this chapter we will provide a proof that *repair* indeed only produces conflict-free trees, and therefore that subtree exclusion is also safe for infix expression grammars (IGs). However, we shall see that safety does not hold for all disambiguation rules *PR*. In Section 5.1 we impose restrictions on the rules. Souza Amorim and Visser (2019) also impose restrictions, but are more restrictive that the ones we will propose in this section. In Section 5.2 we prove these indeed guarantee safety.

```
context-free syntax
    Exp.Lit = NUM
    Exp.Add = Exp "+" Exp
    Exp.Mul = Exp "*" Exp
```

```
Exp.Mul > Exp.Add, Exp.Add > Exp.Mul
```

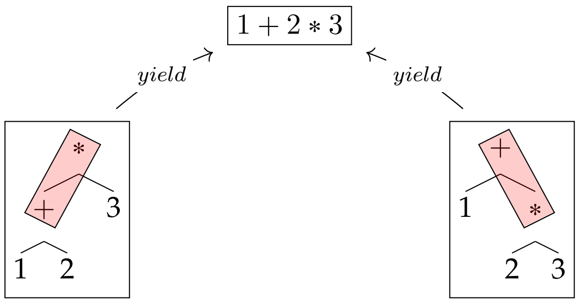


Figure 5.1: Example of unsafe disambiguation rules.

5.1 Safe Disambiguation Rules

As mentioned, safety is not guaranteed for all sets of disambiguation rules *PR*. A simple example would be if two productions both have priority over each other. In Figure 5.1 we made the unnatural decision of having addition have priority over multiplication and multiplication have priority over addition. This demonstrates that a simple expression with just those two operators is already unsafe, because all corresponding parse trees have conflicts. To resolve this issue, we impose the following restrictions on the disambiguation rules:

Definition 5.1 (Safe Infix Conflict Patterns). For a disambiguation rule set *PR*, its conflict patterns $Q(PR)$ is *safe* if for each pair of infix operators (o_1, o_2), the conflict patterns $CR(o_1, o_2)$ and $CL(o_2, o_1)$ are not both in $Q(PR)$.

In Section 5.2 we will show that these restrictions indeed imply safety. In this section we show that these restrictions are also a *lower bound* for safety. Meaning that if we have safety, then $Q(PR)$ is also safe.

Lemma 5.2 (Lower Bound for Safety). *Assuming we have at least one atomic production, if a subtree exclusion filter for PR is safe, then $Q(PR)$ is safe.*

Proof. By contraposition, suppose $Q(PR)$ is unsafe. This means that there exist two infix operators o_1 and o_2 for which we have $CR(o_1, o_2), CL(o_2, o_1) \in Q(PR)$. Let l be a lexical symbol for which we have an atomic production. The sentence $w = l\ o_1\ l\ o_2\ l$ has exactly two well-formed parse trees:

1. $t = [l\ o_1\ [l\ o_2\ l]]$, which conflicts with $CR(o_1, o_2)$.
2. $t = [[l\ o_1\ l]\ o_2\ l]$, which conflicts with $CL(o_2, o_1)$.

Because there are no conflict-free parse trees whose yield outputs w, we have that w is not in the disambiguated language. This means that the subtree exclusion filter for *PR* is not safe. Figure 5.1 is a visual example that demonstrates this concept. □

Relation to Disambiguation Rules In Definition 5.1 we define safety of the disambiguation rules in terms of the conflict patterns it produces. We can also define it directly using the disambiguation rules.

Definition 5.3 (Safe Infix Disambiguation Rules). A disambiguation rules set *PR* is *safe* if and only if for each pair of infix operators (o_1, o_2), a maximum of one of the following two properties holds:

1. $o_1 > o_2$ or o_1 `left` o_2
2. $o_2 > o_1$ or o_2 `right` o_1

Lemma 5.4. *A disambiguation rule set PR is safe if and only if its conflict patterns $Q(PR)$ is also safe.*

Proof. Following the rules from 2.17, Definition 5.3 is logically equivalent to 5.1. □

Restrictions by Souza Amorim and Visser (2019) The restrictions in Definition 5.3 are less restrictive than the original restrictions by Souza Amorim and Visser (2019). This is partially due to the fact that they also impose rather strict *well-formedness* properties on *PR*, on top of safety properties. We give the full Definition in Section 6.1.

While their restrictions make sense intuitively, they prevents us from examining the least restrictive set of rules for safety. Meaning that we would not be able to state Lemma 5.2.

```
1  context-free syntax
2      Exp.Lit     = NUM
3      Exp.Add     = Exp "+" Exp {left}
4      Exp.Mul     = Exp "*" Exp {left}
5  context-free priorities
6      {right: Exp.Mul} > Exp.Add
```

Figure 5.2: An unsafe grammar in SDF3.

Restrictions by SDF3 Ideally, parser generators should check whether the safety (and also completeness) restrictions hold. They should give the user either an error or a warning if they do. We investigated the syntax definition formalism SDF3 [1] and compared it to our restrictions.

SDF3 does not statically check any of the safety properties. It is possible in SDF3 to write rules such as `Exp.Add > Exp.Add`. This causes an expression such as $1 + 2 + 3$ to give a parse error. It is also possible to make some production both left- as well as right-associative with itself. In Figure 5.2 multiplication is left-associative, as well as right-associative, causing an expression such as $1 * 2 * 3$ to give a parse error.

To avoid unsafe disambiguation rules, SDF3 would have to change two things. Firstly, to avoid a case such as in Figure 5.2, SDF3 can remove the feature where a language developer can write an associativity rule inline for a production. For example, the `left` keyword should not be allowed on line 3 and line 4. Instead these should be written in the context-free priorities section of the grammar. Secondly, SDF3 should statically disallow cases where a production appears more than once in the context-free priorities section. This prevents cases where a production has priority over itself.

5.2 Infix Disambiguation is Safe

Our goal is to show that if we have a sentence in the language, that it is also in the disambiguated language. Meaning that we have to be able to transform a well-formed parse tree t into a conflict-free parse tree t' with equivalent yields. For this we will apply the *repair* function from Chapter 4 on t. Lemma 4.4 already shows that the yields are equivalent. So what is left is to prove that $repair(t)$ is conflict-free. For this we will use the restrictions from Definition 5.1.

The most crucial part of proving this is to first show that the function $insert_{in}$ is safe. In other words, if t_2 is conflict-free, then $insert_{in}(l_1, o, t_2)$ should be conflict-free. Because *repair* is just a repeated application of $insert_{in}$, *repair* will also be safe.

Before we go into the formal proof, we give a brief intuition as to *why* it should work. Back in Section 4.1 we noted how the function explicitly avoids right-conflicts (conflicts of the form $CR(o_1, o_2)$). The idea is that our safety restrictions on PR also cause the avoidance of left-conflicts. Figure 5.3 shows this. Here we try to left-insert the operator o_2 to the left of an o_1 node. The $insert_{in}$ function first tries to place o_2 above o_1, causing a potential right-conflict of $CR(o_2, o_1)$, marked using red color in the figure. If this indeed conflicts, $insert_{in}$ will instead try to insert it one level lower. This would potentially cause a left-conflict $CL(o_1, o_2)$, which we marked blue in the figure. However, due to the safety restrictions we imposed, these can not both be conflict patterns (Definition 5.1). This means that $insert_{in}$ both avoids left- as well as right-conflicts.

We will prove this more formally here:

[1]Specifically the version of SDF3 included in Spoofax version 2.5.14 (Kats and Visser 2010).

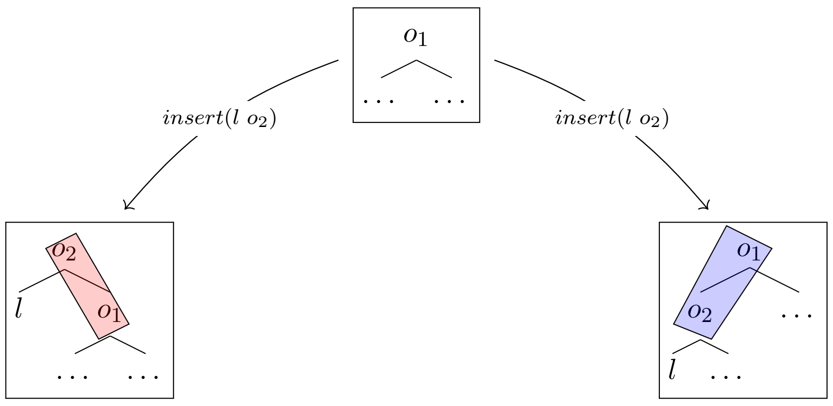


Figure 5.3: The function *insert*$_{in}$ avoids both right- and left-conflicts under safety restrictions.

Lemma 5.5 (IG *insert*$_{in}$ is safe)**.** *Suppose PR is safe and t_2 is a conflict-free tree, then $t' = insert_{in}(l_1, o, t_2)$ is also conflict-free.*

Proof. By induction over the structure of t_2.

1. Suppose $t_2 = l_2 \in L$, then we have
$$\begin{aligned} t' &= insert_{in}(l_1, o, l_2) \\ &= [l_1\ o\ l_2]. \end{aligned}$$
The tree $[l_1\ o\ l_2]$ is conflict-free.

2. Suppose $t_2 = [t_{21}\ o_2\ t_{22}]$ with $CR(o, o_2) \notin Q$, then we have
$$\begin{aligned} t' &= insert_{in}(l_1, o, [t_{21}\ o_2\ t_{22}]) \\ &= [l_1\ o\ [t_{21}\ o_2\ t_{22}]]. \end{aligned}$$
Because $t_2 = [t_{21}\ o_2\ t_{22}]$ is conflict-free, and o does not right-conflict with o_2, we have that t' is conflict-free.

3. Suppose $t_2 = [t_{21}\ o_2\ t_{22}]$ with $CR(o, o_2) \in Q$, then we have
$$\begin{aligned} t' &= insert_{in}(l_1, o, [t_{21}\ o_2\ t_{22}]) \\ &= [insert_{in}(l_1, o, t_{21})\ o_2\ t_{22}] \\ &= [t'_1\ o_2\ t_{22}] \end{aligned}$$
with $t'_1 = insert_{in}(l_1, o, t_{21})$. Because t_2 is conflict-free, we have that its subtrees t_{21} and t_{22} are also conflict free. By the induction hypothesis and the fact that t_{21} is conflict-free, it means that t'_1 is also conflict free. We know that o_2 does not right-conflict with t_{22}, because t_2 is conflict-free. To finish the proof that t' is conflict-free, we still have to show that o_2 also does not left-conflict with t'_1.

   The uppermost operator of t'_1 is either o, or it is the top operator from t_{21} (assuming it has one). This follows directly from the definition of *insert*$_{in}$. The latter case is the easiest, due to the fact that t_2 is conflict-free, we already have that o_2 does not left-conflict with the uppermost operator of t_{21}. What remains is to show that o_2 does not left-conflict with o. Remember that we assumed that $CR(o, o_2) \in Q$. By safety of PR this means that $CL(o_2, o) \notin Q$. In other words, o_2 does indeed not left-conflict with o.

   To conclude, we have that t'_1 and t_{22} are conflict-free, o_2 does not left-conflict with t'_1, and it does not right-conflict with t_{22}. This means that $t' = [t'_1\ o_2\ t_{22}]$ is also conflict-free.

□

Generalizing the lemma Note the *insert*$_{in}$ function allows for the first argument to be an entire tree t_1 rather than being limited to a lexical symbol l_1. We could generalize this lemma by supporting an arbitrary tree input t_1. To do this we have to impose restrictions on t_1: Firstly, it should be conflict-free. Secondly, the input o should not left-conflict with t_1. Because this generalization will not be important until the inclusion of prefix and postfix grammars in Chapter 7, we did not do this for this chapter.

Lemma 5.6 (IG *repair*$_{in}$ is safe)**.** *Suppose PR is safe and t_2 is a conflict-free tree, then $t' = repair_{in}(t_1, o, t_2)$ is also conflict-free.*

Proof. Follows directly by safety of *insert*$_{in}$ (Lemma 5.5) using induction over the structure of t_1. □

Lemma 5.7 (IG *repair* is safe)**.** *Suppose PR is safe, then for all trees t we have $t' = repair(t)$ is conflict-free.*

Proof. By induction over the structure of t.

1. Suppose $t = l \in L$, then $t' = l$ is conflict-free.
2. Suppose $t = [t_1\ o\ t_2]$, then $t' = repair_{in}(t_1, o, repair(t_2))$. By the induction hypothesis we have that $repair(t_2)$ is conflict-free. Using Lemma 5.6 we get that t' is conflict-free.

□

We conclude this chapter with one final theorem that summarizes the core idea.

Theorem 5.8 (IG safety)**.** *The subtree exclusion filter for infix expression grammars is safe if and only if the disambiguation rules PR is safe, assuming the grammar has at least one atomic production.*

Proof. By Lemmas 5.2 and 5.4 we have that if the subtree exclusion filter is safe, then *PR* is safe. Conversely, suppose *PR* is safe. To show the filter is safe, we have to prove that for a sentence w in the language of the grammar, w is also in the disambiguated language. If $w \in L_G$, we have that there is a well-formed tree $t \in T_G$ with $yield(t) = w$. Because the tree $repair(t)$ is both conflict-free (Lemma 5.7) and has yield w (Lemma 4.4), we have that w is in the disambiguated language. □

Chapter 6

Complete Disambiguation

In Definition 2.13 we defined a complete filter as one that causes the disambiguated language to be unambiguous. Meaning that if we have a sentence in the language, it should have at most one corresponding well-formed and conflict-free parse tree. If we expand this definition for subtree exclusion, it says the following: If there exist two well-formed and conflict-free parse trees t_1, t_2 with $\mathit{yield}(t_1) = \mathit{yield}(t_2)$, then $t_1 = t_2$. Similarly to how we have proven safety in Chapter 5, we are going to use the *repair* function from Chapter 4 to prove completeness. By Lemma 4.5 we already get that $\mathit{repair}(t_1) = \mathit{repair}(t_2)$ if $\mathit{yield}(t_1) = \mathit{yield}(t_2)$. If we can also show that for all conflict-free trees t, we have $\mathit{repair}(t) = t$, we are done.

Similarly to how safety does not hold for all disambiguation rules *PR*, completeness also does not hold. Therefore we impose restrictions on *PR* in Section 6.1. We also give a comparison to our restrictions and the restrictions by Souza Amorim and Visser (2019). In Section 6.2 we prove that our restrictions indeed guarantee completeness.

6.1 Complete Disambiguation Rules

As mentioned, completeness is not guaranteed for all sets of disambiguation rules *PR*. In Figure 6.1 we made subtraction left-associative with addition. However, we deliberately de-

```
context-free syntax
    Exp.Lit = NUM
    Exp.Add = Exp "+" Exp
    Exp.Sub = Exp "-" Exp
```

```
Exp.Add left Exp.Add, Exp.Sub left Exp.Sub,
Exp.Sub left Exp.Add
```

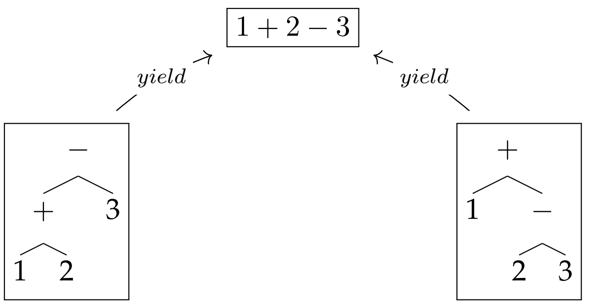


Figure 6.1: Example of incomplete disambiguation rules.

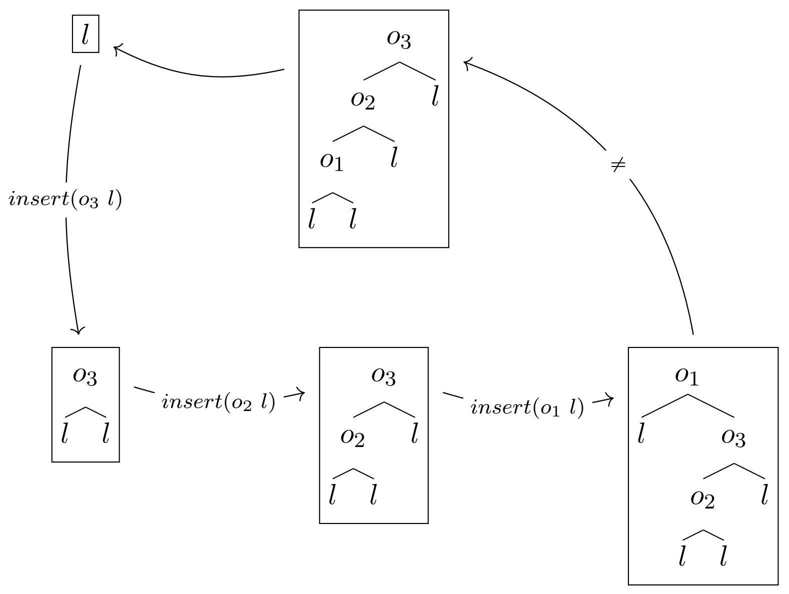


Figure 6.2: Repairing a conflict-free tree, giving another tree. This shows ambiguity. Here $CR(o_1, o_2)$, $CR(o_2, o_3)$ are conflict patterns, but $CR(o_1, o_3)$ is not.

cided that addition is not left-associative with subtraction. From this we manage to construct two conflict-free parse trees that have equivalent yields. To resolve this issue, we impose the following restrictions on the disambiguation rules:

Definition 6.1 (Complete Infix Conflict Patterns)**.** For a disambiguation rule set PR, its conflict patterns $Q(PR)$ is complete if the following properties hold:

1. One of $CR(o_1, o_2)$ and $CL(o_2, o_1)$ should be in $Q(PR)$.
2. If $CR(o_1, o_2), CR(o_2, o_3) \in Q(PR)$, then $CR(o_1, o_3) \in Q(PR)$
3. If $CL(o_1, o_2), CL(o_2, o_3) \in Q(PR)$, then $CL(o_1, o_3) \in Q(PR)$

The first property of this Definition would fix the ambiguity in Figure 6.1. The usefulness of the second and third properties are slightly harder to see. While we could give examples of two conflict-free and yield-equivalent trees to show ambiguity, it would still not be clear as to *why* this happens. Instead we will give an example that uses the *repair* function to show why $repair(t) \neq t$ for some conflict-free t if one of the properties does not hold.

We are going to try and repair the already conflict-free tree $[[l\ o_1\ l]\ o_2\ ]\ l\ o_3\ l]$ (see Figure 6.2), but our set of conflict patterns Q will violate the second property: Suppose we have designed the disambiguation rules such that $CR(o_1, o_2)$, $CR(o_2, o_3) \in Q$, but $CR(o_1, o_3) \notin Q$. With these conflict patterns, inserting o_2 goes well. The issue arises when inserting o_1: It is being inserted above o_3 because there is no right-conflict, even though we want it to get inserted below o_2. This problem arises when something should be inserted deep in the tree, but lacks the required conflict relation with the operators that are higher in the tree. A similar example can be constructed from disambiguation rules that violate the third property.

For safety, we managed to come up with a most general set of restrictions, as shown by Lemma 5.2. Unfortunately, we did not manage to the same for completeness. However, the restrictions in Definition 6.1 are most general *under the assumption* that they are also safe.

Lemma 6.2 (Lower Bound for Completeness)**.** *Suppose we have at least one atomic production and a safe set of disambiguation rules PR (Definition 5.1). If the subtree exclusion filter for PR is complete, then $Q(PR)$ is complete.*

Proof. By contraposition, suppose $Q(PR)$ is incomplete. This gives three cases, for which we will construct counter-examples to show that the subtree exclusion filter is also not complete.

1. Suppose both $CR(o_1, o_2)$ and $CL(o_2, o_1)$ are not in $Q(PR)$. This gives an ambiguous sentence $w = l\ o_1\ l\ o_2\ l$ with two different conflict-free trees (Figure 6.1).
2. Suppose $CR(o_1, o_2)$ and $CR(o_2, o_3)$ are in $Q(PR)$, but $CR(o_1, o_3)$ is not in $Q(PR)$. By safety of $Q(PR)$ we get that $CL(o_2, o_1)$ and $CL(o_3, o_2)$ are not conflict patterns in $Q(PR)$. The sentence $w = l\ o_1\ l\ o_2\ l\ o_3$ has two conflict-free trees (Figure 6.2).
3. Suppose $CL(o_1, o_2)$ and $CL(o_2, o_3)$ are in $Q(PR)$, but $CL(o_1, o_3)$ is not in $Q(PR)$. Analogous to the previous case, the sentence $w = l\ o_3\ l\ o_2\ l\ o_1$ has two conflict-free trees.

□

Relation to Disambiguation Rules In Definition 6.1 we defined completeness of the disambiguation rules in terms of the conflict patterns it produces. We can also define it directly using the disambiguation rules.

Definition 6.3 (Complete Disambiguation Rules)**.** A disambiguation rule set PR is *complete* if the following conditions hold for all operators:

1. $o_1 > o_2 \vee o_1\ \texttt{left}\ o_2 \vee o_2 > o_1 \vee o_2\ \texttt{right}\ o_1$
2. If $o_1 > o_2$, and $o_2 >|\texttt{left}|\texttt{right}\ o_3$, then $o_1 > o_3$
3. If $o_1 >|\texttt{left}|\texttt{right}\ o_2$, and $o_2 > o_3$, then $o_1 > o_3$
4. If $o_1\ \texttt{left}\ o_2$, and $o_2\ \texttt{left}\ o_3$, then $o_1\ \texttt{left}\ o_3$
5. If $o_1\ \texttt{right}\ o_2$, and $o_2\ \texttt{right}\ o_3$, then $o_1\ \texttt{right}\ o_3$
6. If $o_1\ \texttt{left}\ o_2$, then $o_2\ \texttt{right}\ o_3$ should not hold for all o_3
7. If $o_1\ \texttt{right}\ o_2$, then $o_2\ \texttt{left}\ o_3$ should not hold for all o_3.

Note that we use the notation $>|\texttt{left}|\texttt{right}$ to denote the union of the three relations.

Lemma 6.4. *For a well-formed and safe disambiguation rule set PR, it is complete if and only if its conflict patterns $Q(PR)$ is also complete.* [1]

Definition 6.5 (Restrictions by Souza Amorim and Visser (2019))**.** A set of disambiguation rules is well-formed according to Souza Amorim and Visser (2019) if the following properties hold:

1. The priority relation $>$ is transitive and irreflexive.
2. Associativity relations `left` and `right` are transitive and symmetric.
3. A production is in at most one associativity relation, i.e.,
$$domain(\texttt{left}) \cap domain(\texttt{right}) = \varnothing$$
4. Associativity groups have the same priority, i.e., if $p_1 > p_2$ and $p_2\ \texttt{left}|\texttt{right}\ p_3$, then $p_1 > p_3$, and if $p_1\ \texttt{left}|\texttt{right}\ p_2$ and $p_2 > p_3$, then $p_1 > p_3$.

[1]Due to the numerous cases required, we have not shown the proof here. However, each case is relatively simple, and we have proven them in the Coq Proof Assistant.

```
context-free syntax
    Exp.Lit     = NUM
    Exp.Add     = Exp "+" Exp
    Exp.Sub     = Exp "-" Exp
    Exp.Mul     = Exp "*" Exp
    Exp.Div     = Exp "/" Exp
    Exp.Pow     = Exp "^" Exp
context-free priorities
    {left: Exp.Pow} > {Exp.Mul Exp.Div} > {left: Exp.Add Exp.Sub}
```

Figure 6.3: An incomplete grammar in SDF3.

PR is safe if each pair of productions belong to at most one relation ($>$, `left`, `right`). *PR* is complete if each pair of productions belong to at most one relation ($>$, `left`, `right`).

Lemma 6.6. *The combination of the well-formedness restriction (Definition 2.15), safety restriction (Definition 5.3), and completeness restrictions (Definition 6.3), is logically equivalent to the combination of well-formedness, safety, and completeness as defined by Souza Amorim and Visser (2019).*

This Lemma gives a formal justification for the well-formedness properties Souza Amorim and Visser (2019) impose on the disambiguation rules. It shows that without these restrictions, either safety or completeness will not hold.

Restrictions in SDF3 Back in Section 5.1 we stated that SDF3 does enforce the safety properties. The same holds for the majority of the completeness properties. There are two rules that are enforced:

1. Transitivity of the priority relation, i.e., if $o_1 > o_2$ and $o_2 > o_3$, then $o_1 > o_3$.
2. Symmetry of the associativity relations, e.g., if o_1 `left` o_2, then o_2 `left` o_1.

These rules do not take the form of restrictions, but rather as properties that are semantically guaranteed by SDF3. For instance, writing `Exp.Pow` $>$ `Exp.Mul` $>$ `Exp.Add` also means that `Exp.Pow` $>$ `Exp.Add` is a disambiguation rule. Similarly, writing `{left: Exp.Add Exp.Sub}` means both `Exp.Add left Exp.Sub` as well as `Exp.Sub left Exp.Add`.

Note that while symmetry of the associativity relations is not included in our restrictions from Definition 6.1, it is a property that follows if we have *both* safety as well as completeness.

In Section 5.1 we proposed some static semantics to SDF3 to guarantee safety. This included each production having *at most* one occurrence in the context-free priorities section of the grammar. For completeness we want each production appearing *at least* once in this section. Furthermore, each priority group should be annotated with an associativity rule. For example, the grammar in Figure 6.3 should give either a warning or an error, because the priority group `{Exp.Mul Exp.Div}` is missing an associativity rule. Lastly, if a production occurs in some priority group with the `left` or `right` rule, this production should then also be left- or right-associativity with itself. In Figure 6.3 this would mean that `Exp.Add` and `Exp.Sub` should be left-associative with themselves. The current implementation of SDF3 does not do this. Strangely enough, if a priority group only contains one production, such as `{left: Exp.Pow}`, then that production will actually be left-associative with itself.

6.2 Infix Disambiguation is Complete

In the previous section we already gave examples that give an intuition as to why the completeness restrictions are necessary. In this section we will give a formal proof that these restrictions are sufficient. The goal is the prove that $\mathit{repair}(t) = t$ if t is conflict-free. By expansion, this requires proving that $\mathit{repair}_{in}(t_1, o, t_2) = [t_1\, o\, t_2]$ given that $[t_1\, o\, t_2]$ is conflict-free. As it turns out, this is quite difficult to show directly. Instead we will show that $\mathit{repair}_{in}(t_1, o, t_2)$ rewrites to $\mathit{insert}_{in}(t_1, o, t_2)$. The idea being that solving the problem for insert_{in} is easier than for repair_{in}. This is one of the reasons we allowed the first input argument of insert_{in} to be an entire tree t_1 rather than just a lexical symbol l_1.

Lemma 6.7. *Suppose PR is complete. We have that* $\mathit{repair}_{in}(t_1, o, t_2) = \mathit{insert}_{in}(t_1, o, t_2)$ *if* t_1 *is conflict-free and* o *does not left-conflict with* t_1*, i.e., if* $t_1 = [t_{11}\ o_1\ t_{12}]$*, then* $CL(o, o_1) \notin Q$.

Proof. By induction over the structure of t_1.
Base case: if $t_1 = l_1$ is a lexical symbol, then $\mathit{repair}_{in}(l_1, o, t_2) = \mathit{insert}_{in}(l_1, o, t_2)$ by definition.
Inductive case: $t_1 = [t_{11}\ o_1\ t_{12}]$.

$$\begin{aligned}
\mathit{repair}_{in}(t_1, o, t_2) &= \mathit{repair}_{in}([t_{11}\ o_1\ t_{12}], o, t_2) \\
&= \mathit{repair}_{in}(t_{11}, o_1, \mathit{repair}_{in}(t_{12}, o, t_2)) \\
&= \mathit{insert}_{in}(t_{11}, o_1, \mathit{repair}_{in}(t_{12}, o, t_2)) && \text{(IH, 1)} \\
&= \mathit{insert}_{in}(t_{11}, o_1, \mathit{insert}_{in}(t_{12}, o, t_2)) && \text{(IH, 2)} \\
&= \mathit{insert}_{in}([t_{11}\ o_1\ t_{12}], o, t_2) && \text{(Lemma 6.8)} \\
&= \mathit{insert}_{in}(t_1, o, t_2)
\end{aligned}$$

Here we justify the usages of the induction hypothesis:

1. Because t_1 is conflict-free, we have that o_1 does not left-conflict with t_{11}, and t_{11} is conflict-free. This permits usage of the induction hypothesis.

2. Because t_1 is conflict-free, we have that o_1 does not left-conflict with t_{11}, and t_{12} is conflict-free. In order to be able to use the induction hypothesis, we still need to show that o does not left-conflict with the top operator of t_{12}, call it o_{12}. We will show this by contradiction: Suppose o conflicts with o_{12}, i.e. $CL(o, o_{12})$ is a conflict pattern. We have that $CR(o_1, o_{12})$ is not a conflict pattern, therefore by completeness it must follow that $CL(o_{12}, o_1)$ is a conflict pattern. Because $CL(o, o_{12})$ and $CL(o_{12}, o_1)$ are both conflict patterns, then by Lemma 6.4 we get that $CL(o, o_1)$ is a conflict pattern. This is a contradiction because we assumed that o does not left-conflict with t_1.

□

Lemma 6.8. *Suppose PR is complete. We have that*

$$\mathit{insert}_{in}([t_{11}\ o_1\ t_{12}], o, t_2) = \mathit{insert}_{in}(t_{11}, o_1, \mathit{insert}_{in}(t_{12}, o, t_2))$$

if $[t_{11}\ o_1\ t_{12}]$ *is conflict-free and* $CL(o, o_1) \notin Q(PR)$.

Proof. By induction over the structure t_2. Note that by completeness of PR we get $CR(o_1, o) \in Q(PR)$.

1. Base case: $t_2 = l_2$ is a lexical symbol. Then we get

$$\begin{aligned}
\mathit{insert}_{in}([t_{11}\ o_1\ t_{12}], o, t_2) &= \mathit{insert}_{in}([t_{11}\ o_1\ t_{12}], o, l_2) \\
&= [[t_{11}\ o_1\ t_{12}]\ o\ l_2] \\
&= [\mathit{insert}_{in}(t_{11}, o_1, t_{12})\ o\ l_2] && \text{(Lemma 6.9)} \\
&= \mathit{insert}_{in}(t_{11}, o_1, [t_{12}\ o\ l_2]) \\
&= \mathit{insert}_{in}(t_{11}, o_1, \mathit{insert}_{in}(t_{12}, o, l_2)) \\
&= \mathit{insert}_{in}(t_{11}, o_1, \mathit{insert}_{in}(t_{12}, o, t_2))
\end{aligned}$$

2. Suppose $t_2 = [t_{21}\ o_2\ t_{22}]$ and $CR(o, o_2) \notin Q(PR)$. This case is analogous to the previous case.

3. Suppose $t_2 = [t_{21}\ o_2\ t_{22}]$ and $CR(o, o_2) \in Q(PR)$. Note that by Lemma 6.4 we also get $CR(o_1, o_2) \in Q(PR)$.

$$\begin{aligned}
\mathit{insert}_{in}([t_{11}\ o_1\ t_{12}], o, t_2) &= \mathit{insert}_{in}([t_{11}\ o_1\ t_{12}], o, [t_{21}\ o_2\ t_{22}]) \\
&= [\mathit{insert}_{in}([t_{11}\ o_1\ t_{12}], o, t_{21})\ o_2\ t_{22}] \\
&= [\mathit{insert}_{in}(t_{11}, o_1, \mathit{insert}_{in}(t_{12}, o, t_{21}))\ o_2\ t_{22}] && \text{(IH)} \\
&= \mathit{insert}_{in}(t_{11}, o_1, [\mathit{insert}_{in}(t_{12}, o, t_{21})\ o_2\ t_{22}]) \\
&= \mathit{insert}_{in}(t_{11}, o_1, \mathit{insert}_{in}(t_{12}, o, [t_{21}\ o_2\ t_{22}]) \\
&= \mathit{insert}_{in}(t_{11}, o_1, \mathit{insert}_{in}(t_{12}, o, t_2))
\end{aligned}$$

□

Lemma 6.9 (IG *insert*$_{in}$ is complete)**.** *If* $[t_1\ o\ t_2]$ *is conflict-free, then*

$$\mathit{insert}_{in}(t_1, o, t_2) = [t_1\ o\ t_2]$$

Proof. If $[t_1\ o\ t_2]$ has no conflicts, then o does not right-conflict with t_2. Thus by definition it follows directly that $\mathit{insert}_{in}(t_1, o, t_2) = [t_1\ o\ t_2]$. □

With this relation between repair_{in} and insert_{in}, our proof that $\mathit{repair}(t) = t$ for a conflict-free t just became a lot easier.

Lemma 6.10 (IG *repair* is complete)**.** *If PR is complete and t is a conflict-free tree, then* $\mathit{repair}(t) = t$.

Proof. By induction over the structure of t.

1. Base case: $t = l$ gives $\mathit{repair}(t) = l$

2. Inductive case: $t = [t_1\ o\ t_2]$, which gives

$$\begin{aligned}
\mathit{repair}(t) &= \mathit{repair}([t_1\ o\ t_2]) \\
&= \mathit{repair}_{in}(t_1, o, \mathit{repair}(t_2)) \\
&= \mathit{repair}_{in}(t_1, o, t_2) && \text{(IH)} \\
&= \mathit{insert}_{in}(t_1, o, t_2) && \text{(Lemma 6.7)} \\
&= [t_1\ o\ t_2] && \text{(Lemma 6.9)} \\
&= t
\end{aligned}$$

□

We conclude this chapter with the final completeness theorem:

Theorem 6.11. *The subtree exclusion filter for infix grammars is complete if the disambiguation rules PR is complete. The converse holds if PR is safe as well.*

Proof. To show the subtree exclusion filter is safe, we have to prove that if $\mathit{yield}(t_1) = \mathit{yield}(t_2)$ holds holds for two conflict-free parse trees t_1 and t_2, then $t_1 = t_2$. By Lemma 4.5 we get that $\mathit{repair}(t_1) = \mathit{repair}(t_2)$. Lemma 6.10 tells us that $\mathit{repair}(t_1) = t_1$ and $\mathit{repair}(t_2) = t_2$. Thus we have $t_1 = t_2$.

The converse holds from Lemma 6.2. □

Chapter 7

Extending with Prefix and Postfix Expressions

So far we have only considered expression grammars that contain infix and atomic expressions. In this chapter we extend the semantics and corresponding proofs with *prefix* and *postfix expressions*.

Definition 7.1 (IPP Expression Grammar). An IPP expression grammar (IPPG) is a tuple $(L, O_{in}, O_{pre}, O_{post})$, where L, O_{in}, O_{pre}, and O_{post} are disjoint sets of lexical, infix operator, prefix operator, and postfix operator symbols respectively. It represents a context-free grammar $G = (L \cup O_{in} \cup O_{pre} \cup O_{post}, \{A\}, P)$, with A a single nonterminal symbol and where there are productions

$$\begin{aligned}
&A.C = l && \text{(Atomic Productions)}\\
&A.C = A\ o_{in}\ A && \text{(Infix Productions)}\\
&A.C = o_{pre}\ A && \text{(Prefix Productions)}\\
&A.C = A\ o_{post} && \text{(Postfix Productions)}
\end{aligned}$$

for all $l \in L$, $o_{in} \in O_{in}$, $o_{pre} \in O_{pre}$, and $o_{post} \in O_{post}$.

Figure 7.1 shows an example of an IPP grammar. The `Minus` and `Lambda` [1] productions are both prefix expressions, whereas `Incr` is a postfix production.

In this chapter we will extend the semantics from Chapter 2 to also support IPP grammars (Section 7.1). The new semantics will introduce more complexity. It has new types of

[1]The `Lambda` production can be considered a prefix production, because the terminals `lambda`, `ID`, and the dot `.` can be conjoined into one single terminal.

```
context-free syntax
    Exp.Lit    = NUM
    Exp.Var    = ID
    Exp.Add    = Exp "+" Exp {left}
    Exp.Minus  = "-" Exp
    Exp.Lambda = "lambda" ID "." Exp
    Exp.Incr   = Exp "++"
context-free priorities
    {left: Exp.Minus Exp.Incr} > Exp.Add > Exp.Lambda
```

Figure 7.1: Example of an IPP Grammar

conflict patterns, that include prefix and postfix nodes. Furthermore, we define new types of matching, where a node can conflict with another node deeper in the tree (deep matching). This allows us to keep ensuring completeness. In Sections 7.2 and 7.3 we extend our *repair* function from Chapter 4 to support prefix and postfix expressions respectively. The extended repair function will take these new concepts into account.

7.1 Semantics for IPP Grammars

Back in Chapter 2 we used the disambiguation rules to generate conflict patterns. We expand the semantics from Definition 2.17, which only work for infix productions, to also include prefix and postfix productions.

This section is also directly based on the semantics given by Souza Amorim and Visser (2019). The differences between their semantics and the semantics we provide in this section are mostly notational. We write almost every possible combination disambiguation rules and conflicts between the different types operators. While our notation gives an explicit overview of every possible case, making it an easier candidate to mechanize in the Coq Proof Assistant, the notation used by Souza Amorim and Visser (2019) is more compact and pleasing to the eye.

First we update our common conflict patterns CR and CL to also include tree patterns that have prefix and postfix nodes.

Definition 7.2 (Common IPP Conflict Patterns)**.**

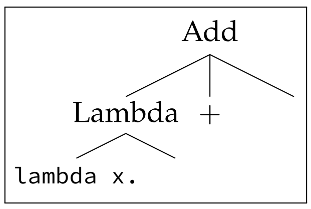
$$\begin{aligned}
CL_{in,in}(o_1, o_2) &= [[\_\ o_2\ \_]\ o_1\ \_] & CR_{in,in}(o_1, o_2) &= [\_\ o_1\ [\_\ o_2\ \_]] \\
CL_{in,pre}(o_1, o_2) &= [[o_2\ \_]\ o_1\ \_] & CR_{pre,in}(o_1, o_2) &= [o_1\ [\_\ o_2\ \_]] \\
CR_{in,post}(o_1, o_2) &= [\_\ o_1\ [\_\ o_2]] & CL_{post,in}(o_1, o_2) &= [[\_\ o_1\ \_]\ o_1] \\
CR_{pre,post}(o_1, o_2) &= [o_1\ [\_\ o_2]] & CL_{post,pre}(o_1, o_2) &= [[\_\ o_2]\ o_1]
\end{aligned}$$

These new conflict patterns will be used in Definition 7.6. Note that this list at first seems to be incomplete. For instance, the pattern $CR_{in,pre}(o_1, o_2) = [\_\ o_1\ [o_2\ \_]]$ is not included. This pattern causes unsafe semantics as we can construct example sentences that get completely filtered out. For instance, using the grammar from Figure 7.1, suppose $CR_{in,pre}(+, \texttt{lambda ID.}) = [\_ + [\texttt{lambda ID.}\ \_]]$ is a conflict pattern. The sentence $5+\texttt{lambda x. x}$ only has one well-formed tree: $[5 + [\texttt{lambda x. x}]]$. This tree unfortunately matches the conflict pattern. The same problem would hold for a conflict pattern $CL_{in,post}(o_1, o_2) = [[\_\ o_2]\ o_1\ \_]$. Therefore these two types conflicts patterns are not allowed to be constructed and we excluded them from the definition. Similarly, we have also not defined the conflict patterns $CR_{pre,pre}$ and $CL_{post,post}$.

Incomplete Semantics It now seems intuitive to include prefix and postfix expressions to our semantics of priority and associativity rules in the same manner as we did for infix expressions in Definition 2.17. However, *completeness* becomes a new issue. First we will discuss an example that works as we would expect, then we will show one that gives rise to ambiguity in the grammar.

Take the grammar from Figure 7.1. In particular, take the priority rule `Minus` > `Add`. Following the same logic in Definition 2.17, this should give rise to the conflict pattern $CR_{pre,in}(-, +) = [-\ [\_ + \_]]$. Suppose we have the expression $-5 + 3$, which has two parse trees $[-[5 + 3]]$ and $[[-5] + 3]$. The first correctly gets filtered out, whereas the second is conflict-free. Safety and completeness are consistently guaranteed for priority rules where we have a prefix or postfix production that has priority over an infix production. Meaning that the conflict pattern types $CR_{pre,in}$ and $CL_{post,in}$ work as we would expect.

A problem arises when we have an infix production, that has priority over a prefix or postfix production. As an example, take the priority rule `Add` > `Lambda`. The idea behind



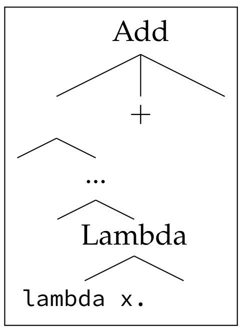


Figure 7.2: A shallow priority conflict (left) compared to a deep priority conflict (right) based on the grammar from Figure 7.1.

`Lambda` having low priority, is that body of a lambda expression should extend to the right as far as possible. For instance, the body of the lambda expression in the sentence `lambda x.` $-$ `x` $+ 5 + 3$, should encompass the entire expression $-$`x` $+ 5 + 3$. With the conflict pattern $CL_{in,pre}(+,$ `lambda ID.`$) = [[$`lambda ID.` $\_] + \_]$, this particular sentence will only have one conflict-free tree and is therefore correctly unambiguous. Unfortunately, we can also construct sentences that are ambiguous. Take the expression $5 +$ `lambda x.` $6 + 7$. This has two conflict-free parse trees

$$[5 + [\texttt{lambda x.}\ [6 + 7]]$$
$$[[5 + [\texttt{lambda x.}\ 6]] + 7].$$

It should be clear that we want the second parse tree to be filtered, as the lambda expression does not extend all the way to the right. The problem is that it does not match the conflict pattern $[[$`lambda ID.` $\_] + \_]$. The semantics is *incomplete*.

Deep Priority Conflicts This problem occurs because the lambda expression is nested deeper inside the tree, while the conflict pattern is only able to handle *shallow* conflicts. Figure 7.2 gives a visual example of the occurring problem. The left tree gets properly filtered as it matches the conflict pattern. The problem occurs when the lambda production occurs further down in the tree, as a rightmost descendant. We will call these *deep priority conflicts*. Deep priority conflicts can also occur if we have an infix production that has priority over a postfix production. Similarly, any disambiguation rule between a prefix and postfix production can also create deep priority conflicts. These types of conflicts can not be filtered out using our shallow conflict patterns. To solve this problem, Souza Amorim and Visser (2019) introduce deep conflict patterns.

It is not enough to just define regular shallow matching as in Definition 2.8. We define the concept of *right-* and *left-most matching*.

Definition 7.3 (Right-Most and Left-Most Matching)**.** For a grammar G, a tree $t \in T_G$ and pattern $q \in TP_G$, define $M^{rm}(t, q)$ and $M^{lm}(t, q)$ inductively as follows:

$$\frac{M(t,q)}{M^{rm}(t,q)} \quad \frac{M^{rm}(t_n,q)}{M^{rm}([...\ t_n],q)}$$
$$\frac{M(t,q)}{M^{lm}(t,q)} \quad \frac{M^{rm}(t_1,q)}{M^{lm}([t_1\ ...],q)}$$

The idea is that t right-most matches q if t either shallowly matches q, or any of its right-most descendants do. Left-most matching works analogously.

How can we apply this new matching scheme for our 'problematic' conflict patterns? Take the previous example where $CL_{in,pre}(+, \texttt{lambda ID.}) = [[\texttt{lambda ID.}\ \_]\ +\ \_]$ is a conflict pattern. Suppose we have a tree $[t_1 + t_2]$. This tree should match this pattern, if t_1 right-most matches the pattern $[\texttt{lambda ID.}\ \_]$. This would filter any trees that have a shape of the form as shown in Figure 7.2.

Definition 7.4 (Deep Matching)**.** For a grammar G, a tree $t \in T_G$ and pattern $q \in TP_G$, define $D^{rm}(t, q)$ and $D^{lm}(t, q)$ inductively as follows:

$$\frac{\forall\, 1 \leqslant i \leqslant n.\ M^{rm}(t_i, q_i)}{D^{rm}([A.C = t_1 ... t_n], [A.C = q_1 ... q_n])} \qquad \frac{\forall\, 1 \leqslant i \leqslant n.\ M^{lm}(t_i, q_i)}{D^{lm}([A.C = t_1 ... t_n], [A.C = q_1 ... q_n])}$$

For a set of tree patterns $Q \subseteq TP_G$, we say $D^{rm}(t, Q)$ if there is a $q \in Q$ such that $D^{rm}(t, q)$. We define $D^{lm}(t, Q)$ analogously.

Let us take a look back at the example tree $t = [[5 + [\texttt{lambda x.}\ 6]] + 7]$. The conflict pattern $q = [[\texttt{lambda ID.}\ \_] + \_]$ does not shallowly match the tree, i.e., $M(t, q)$ does not hold. However, it does right-most deeply match the tree. Below we give a derivation for this claim:

$$\cfrac{\cfrac{\cfrac{M([\texttt{lambda x.}\ 6], [\texttt{lambda ID.}\ \_])}{M^{rm}([\texttt{lambda x.}\ 6], [\texttt{lambda ID.}\ \_])}}{M^{rm}([5 + [\texttt{lambda x.}\ 6]], [\texttt{lambda ID.}\ \_])}}{D^{rm}([[5 + [\texttt{lambda x.}\ 6]] + 7], [[\texttt{lambda ID.}\ \_] + \_])}$$

Before we had only one set Q of shallow conflict patterns. Now we have a tuple $Q = (Q^i, Q^{rm}, Q^{lm})$ with three sets of conflict patterns. Q^i, handles shallow conflicts, whereas Q^{rm}, and Q^{lm}, handle left-most and right-most deep priority conflicts. Below we redefine what it means for a tree to be well-formed under subtree exclusion.

Definition 7.5 (Subtree Exclusion)**.** For a grammar $G = (\Sigma, N, P)$ and $Q = (Q^i, Q^{rm}, Q^{lm})$ with $Q^i, Q^{rm}, Q^{lm} \subseteq TP_G$ sets of tree patterns, we define the set T_G^Q of well-formed parse trees under subtree exclusion inductively as follows:

$$\frac{a \in \Sigma}{a \in T_G^Q} \qquad \frac{t \in T_G, \quad \neg M(t, Q^i),\ \neg D^{rm}(t, Q^{rm}),\ \neg D^{lm}(t, Q^{lm}), \quad \forall\, 1 \leqslant i \leqslant n.\ t_i \in T_G^Q}{t = [A.C = t_1 ... t_n] \in T_G^Q}$$

Now that we have redefined subtree exclusion to also support prefix and postfix expressions, we can finally show how to obtain the conflict patterns from the disambiguation rules *PR*.

Definition 7.6 (IPP Conflict Patterns)**.** Let $G = (L, O_{in}, O_{pre}, O_{post})$ be an IPP expression grammar and $PR = (>, \texttt{left}, \texttt{right})$ a disambiguation rule set for G. We define the tuple

$Q(PR) = (Q^i(PR), Q^{rm}(PR), Q^{lm}(PR))$ as follows:

$$\frac{o_1, o_2 \in O_{in}, \quad o_1 > o_2}{CR_{in,in}(o_1, o_2) \in Q^i(PR)} \qquad \frac{o_1, o_2 \in O_{in}, \quad o_1 \ \texttt{left}\ o_2}{CR_{in,in}(o_1, o_2) \in Q^i(PR)}$$

$$\frac{o_1, o_2 \in O_{in}, \quad o_1 > o_2}{CL_{in,in}(o_1, o_2) \in Q^i(PR)} \qquad \frac{o_1, o_2 \in O_{in}, \quad o_1 \ \texttt{right}\ o_2}{CL_{in,in}(o_1, o_2) \in Q^i(PR)}$$

$$\frac{o_1 \in O_{pre}, o_2 \in O_{in}, \quad o_1 > o_2}{CR_{pre,in}(o_1, o_2) \in Q^i(PR)} \qquad \frac{o_1 \in O_{pre}, o_2 \in O_{in}, \quad o_1 \ \texttt{left}\ o_2}{CR_{pre,in}(o_1, o_2) \in Q^i(PR)}$$

$$\frac{o_1 \in O_{in}, o_2 \in O_{pre}, \quad o_1 > o_2}{CL_{in,pre}(o_1, o_2) \in Q^{rm}(PR)} \qquad \frac{o_1 \in O_{in}, o_2 \in O_{pre}, \quad o_1 \ \texttt{right}\ o_2}{CL_{in,pre}(o_1, o_2) \in Q^{rm}(PR)}$$

$$\frac{o_1 \in O_{post}, o_2 \in O_{in}, \quad o_1 > o_2}{CL_{post,in}(o_1, o_2) \in Q^i(PR)} \qquad \frac{o_1 \in O_{post}, o_2 \in O_{in}, \quad o_1 \ \texttt{right}\ o_2}{CL_{post,in}(o_1, o_2) \in Q^i(PR)}$$

$$\frac{o_1 \in O_{in}, o_2 \in O_{post}, \quad o_1 > o_2}{CR_{in,post}(o_1, o_2) \in Q^{lm}(PR)} \qquad \frac{o_1 \in O_{in}, o_2 \in O_{post}, \quad o_1 \ \texttt{left}\ o_2}{CR_{in,post}(o_1, o_2) \in Q^{lm}(PR)}$$

$$\frac{o_1 \in O_{pre}, o_2 \in O_{post}, \quad o_1 > o_2}{CR_{pre,post}(o_1, o_2) \in Q^{lm}(PR)} \qquad \frac{o_1 \in O_{pre}, o_2 \in O_{post}, \quad o_1 \ \texttt{left}\ o_2}{CR_{pre,post}(o_1, o_2) \in Q^{lm}(PR)}$$

$$\frac{o_1 \in O_{post}, o_2 \in O_{pre}, \quad o_1 > o_2}{CL_{post,pre}(o_1, o_2) \in Q^{rm}(PR)} \qquad \frac{o_1 \in O_{post}, o_2 \in O_{pre}, \quad o_1 \ \texttt{right}\ o_2}{CL_{post,pre}(o_1, o_2) \in Q^{rm}(PR)}$$

Take Figure 7.3 as an example, where we give the conflict patterns for a specific IPP grammar. In Figure 7.4 we give an example of a sentence for that grammar, with its corresponding parse trees. We highlighted the conflicts occurring in those parse trees.

For our safety and completeness proofs, we impose restrictions on the disambiguation rules *PR* analogous to how we defined them in Sections 5.1 and 6.1. We elevated the Definitions 5.3 and 6.3 to also include prefix and postfix operators in the exact same manner.[2]

7.2 Repairing for Prefix and Infix Expressions

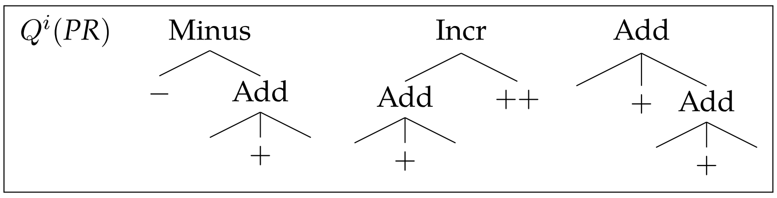
In this section we will update our *repair* function from Chapter 4 in order to support parse trees with both infix and prefix expressions (IP Grammars). For the purpose of this section, let an IP grammar be defined analogously to IPP grammars, with the postfix operators O_{post} and productions excluded.

Due to our new deep conflict patterns, it may seem like the function will suddenly take on a much higher complexity. This is not the case. Recall that we designed *repair* in such a way that we only avoid creating right-conflicts, i.e., conflicts of the *CR* type. The idea is that safety of the disambiguation rules ensures that left-conflicts also be avoided. Luckily, with prefix expressions, the only deep conflict pattern is $CL_{in,pre}$. This is a left-conflict, which we ignore. The right-conflict pattern $CR_{pre,in}$ is a shallow conflict. We start by extending our $insert_{in}$ function to support insertion *into* prefix nodes.

Definition 7.7 (IP $insert_{in}$)**.** For an IP grammar $G = (L, O_{in}, O_{pre})$ with conflict patterns $Q = (Q^i, Q^{rm})$, we define the function $insert_{in} : (T_G \times O_{in} \times T_G) \to T_G$ as follows:

[2]Unlike with Infix Expression Grammars, we have not proven that if the subtree exclusion filter is safe, then the conflict patterns $Q(PR)$ is safe. Although we do conjecture it holds for IPP Grammars.

```
context-free syntax
    Exp.Lit    = NUM
    Exp.Var    = ID
    Exp.Add    = Exp "+" Exp {left}
    Exp.Minus  = "-" Exp
    Exp.Lambda = "lambda" ID "." Exp
    Exp.Incr   = Exp "++"
context-free priorities
    {left: Exp.Minus Exp.Incr} > Exp.Add > Exp.Lambda
```



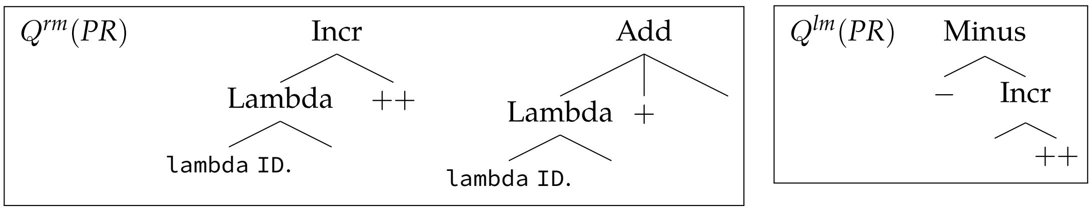


Figure 7.3: A disambiguated IPP Grammar, together with corresponding conflict pattern sets Q^i, Q^{rm}, and Q^{lm}.

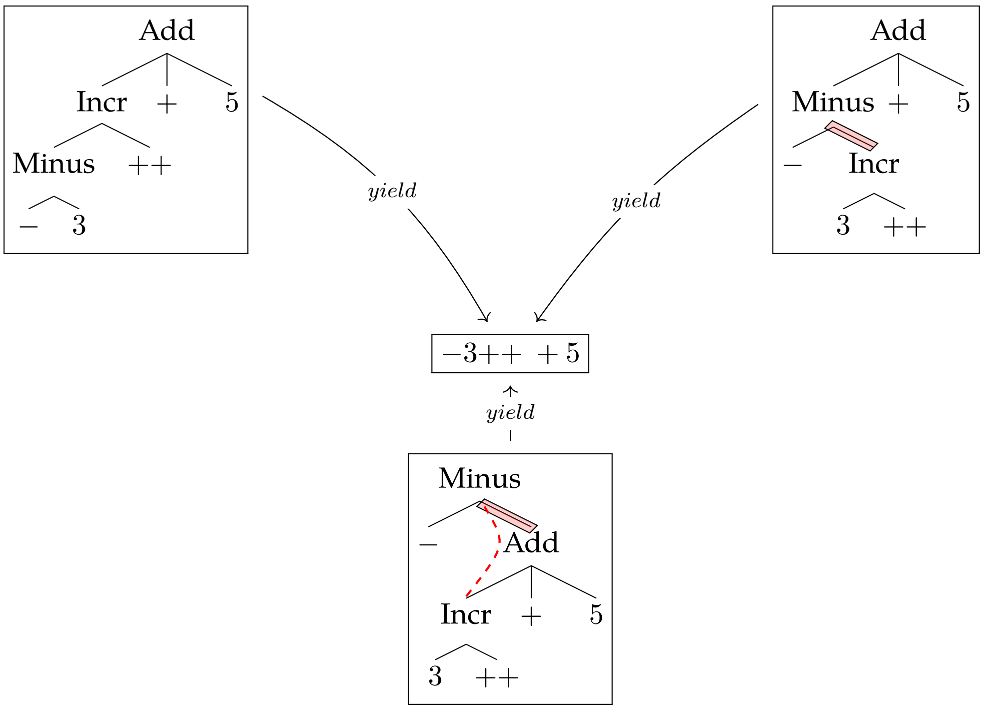


Figure 7.4: Parse trees with highlighted conflicts. Note that the bottom parse tree contains two conflicts. The dashed line represents a deep conflict.

$insert_{in}(t_1, o, t_2) =$

$$\begin{cases} [t_1\ o\ l_2], & \text{if } t_2 = l_2 \in L \\ [t_1\ o\ [t_{21}\ o_2\ t_{22}]], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } CR_{in,in}(o, o_2) \notin Q^i \\ [insert_{in}(t_1, o, t_{21})\ o_2\ t_{22}], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } CR_{in,in}(o, o_2) \in Q^i \\ [t_1\ o\ [o_2\ t_{21}]] & \text{if } t_2 = [o_2\ t_{21}], o_2 \in O_{pre} \end{cases}$$

The bottom case is the newly added case where we insert into prefix nodes. Because $CR_{in,pre}$ is not a type of conflict pattern, we do not have to account for this and guarantee safety from at least right-conflicts.

The next step is to add another function $insert_{pre} : (O_{pre} \times T_G) \to T_G$ to allow insertion of a prefix operator into an arbitrary tree.

Definition 7.8 (IP $insert_{pre}$)**.** For an IP grammar $G = (L, O_{in}, O_{pre})$ with conflict patterns $Q = (Q^i, Q^{rm})$, we define the function $insert_{pre} : (O_{pre} \times T_G) \to T_G$ as follows:

$insert_{pre}(o, t_2) =$

$$\begin{cases} [o\ l_2], & \text{if } t_2 = l_2 \in L \\ [o\ [t_{21}\ o_2\ t_{22}]], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } CR_{pre,in}(o, o_2) \notin Q^i \\ [insert_{pre}(o, t_{21})\ o_2\ t_{22}], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } CR_{pre,in}(o, o_2) \in Q^i \\ [o\ [o_2\ t_{21}]] & \text{if } t_2 = [o_2\ t_{21}], o_2 \in O_{pre} \end{cases}$$

This function works similarly to $insert_{in}$ in that we explicitly avoid all right-conflicts. In this case we recursively insert the input operator o deeper in the tree if it would conflict with an operator o_2 with $CR_{pre,in}(o, o_2) \in Q^i$.

Definition 7.9 (IP $repair_{in}$)**.** For an IP grammar $G = (L, O_{in}, O_{pre})$ with conflict patterns $Q = (Q^i, Q^{rm})$, we define the function $repair_{in} : (T_G \times O_{in} \times T_G) \to T_G$ as follows:

$repair_{in}(t_1, o, t_2) =$

$$\begin{cases} insert_{in}(l_1, o, t_2), & \text{if } t_1 = l_1 \in L \\ repair_{in}(t_{11}, o_1, repair_{in}(t_{12}, o, t_2)), & \text{if } t_1 = [t_{11}\ o_1\ t_{12}], o_1 \in O_{in} \\ insert_{pre}(o_1, repair_{in}(t_{11}, o, t_2)), & \text{if } t_1 = [o_1\ t_{11}], o_1 \in O_{pre} \end{cases}$$

Definition 7.10 (IP $repair$)**.** For an IP grammar $G = (L, O_{in}, O_{pre})$ with a set of conflict patterns $Q = (Q^i, Q^{rm})$, we define the function $repair : T_G \to T_G$ as follows:

$$repair(t) = \begin{cases} l, & \text{if } t = l \in L \\ repair_{in}(t_1, o, repair(t_2)), & \text{if } t = [t_1\ o\ t_2], o \in O_{in} \\ insert_{pre}(o, repair(t_1)), & \text{if } t = [o\ t_1], o \in O_{pre} \end{cases}$$

Both Definition 7.9 and 7.10 should be self-explanatory. These extensions with prefix expression follow the same logic as they did with just infix expressions.

Safety and Completeness Proof We have proven this new repair function is indeed both safe and complete using the Coq Proof Assistant. The proofs work similarly to the proofs in Chapter 5 and Chapter 6 for infix expression grammars. Giving the full proof here would be rather tedious, as the proof contains many cases. So instead of a full proof, we will work out one very specific case, which shows that $insert_{pre}$ is free from any deep conflicts. This should give an idea as to how we worked with these deep priority conflicts in the proofs.

Lemma 7.11. *Let $G = (L, O_{in}, O_{pre})$ be an IP Grammar and PR a safe set of disambiguation rules for G. For operators $o \in O_{pre}$ and a conflict-free infix node $t_2 = [t_{21}\ o_2\ t_{22}]$ with $CR_{pre,in}(o, o_2) \in Q^i(PR)$, we have that $t' = insert_{pre}(o, t_2)$ does not match any deep conflicts in $Q^{rm}(PR)$.*

Proof. By definition, we get $t' = insert_{pre}(o, t_2) = [insert_{pre}(o, t_{21})\ o_2\ t_{22}]$. We have to show that the tree does not deeply match any conflict pattern in $Q^{rm}(PR)$. Note that the all patterns in Q^{rm} have the type $CL_{in,pre}$.

We will give a proof by contradiction: Suppose t' deeply matches some conflict pattern $CL_{in,pre}(x_1, x_2) = [[x_2\ \_]\ x_1\ \_]$. This means that $x_1 = o_2$ and that the subtree $insert_{pre}(o, t_{21})$ right-most matches the pattern $[x_2\ \_]$. By performing case analysis on $insert_{pre}(o, t_{21})$, it can be shown that either $o = x_2$, or t_{21} right-most matches the pattern $[x_2\ \_]$.

1. Suppose $o = x_2$. Recall that we also had that $o_2 = x_1$. This means that both $CR_{pre,in}(o, o_2)$ and $CL_{in,pre}(o_2, o)$ are conflict patterns. By safety of PR this can not occur.

2. Suppose t_{21} right-most matches the pattern $[x_2\ \_]$. Recall that we assumed that the tree $t_2 = [t_{21}\ o_2\ t_{22}]$ is conflict-free. This means it should not match our conflict pattern $CL_{in,pre}(x_1, x_2)$. Because we had $x_1 = o_2$, this can only occur if t_{21} does not right-most match the pattern $[x_2\ \_]$.

□

7.3 Repairing for Postfix, Prefix, and Infix Expressions

In the previous section we have extended our *repair* function to support prefix expressions. In this section we will also include postfix expressions. Appendix C contains the full uninterrupted raw definitions for the function.

We will start by updating $insert_{in}$ again. Recall that we only care about explicitly avoiding right-conflicts. With prefix expression we got lucky, because all possible right-conflicts are shallow conflicts. With the addition of postfix expressions, we get two types of deep right-most conflicts: $CR_{in,post}$ and $CR_{pre,post}$. Our updated $insert_{in}$ function has to take these deep conflicts into account. To help simplify our design for insertion, we give another definition that formalizes that idea of right-conflicts and left-conflicts:

Definition 7.12 (Right- and Left-Conflicts)**.** Suppose we have an IPP grammar $G = (L, O_{in}, O_{pre}, O_{post})$ with conflict patterns $Q = (Q^i, Q^{rm}, Q^{lm})$. For an operator $o \in O_{in} \cup O_{pre}$ and $t_2 \in T_G$, we define $R_Q(o, t_2)$ (o right-conflicts with t_2) as follows:

$$\frac{o \in O_{in}, o_2 \in O_{in}, q = CR_{in,in}(o, o_2) \in Q^i, M([t_{21}\ o_2\ t_{22}], q)}{R_Q(o, [t_{21}\ o_2\ t_{22}])}$$

$$\frac{o \in O_{pre}, o_2 \in O_{in}, q = CR_{pre,in}(o, o_2) \in Q^i, M([t_{21}\ o_2\ t_{22}], q)}{R_Q(o, [t_{21}\ o_2\ t_{22}])}$$

$$\frac{o \in O_{in}, o_2 \in O_{post}, q = CR_{in,post}(o, o_2) \in Q^{lm}, D^{lm}([t_{21}\ o_2], q)}{R_Q(o, [t_{21}\ o_2])}$$

$$\frac{o \in O_{pre}, o_2 \in O_{post}, q = CR_{pre,post}(o, o_2) \in Q^{rm}, D^{rm}([t_{21}\ o_2], q)}{R_Q(o, [t_{21}\ o_2])}$$

Define $L_Q(o, t_1)$ (o left-conflicts with t_1) with $o \in O_{in} \cup O_{post}$ and $t_1 \in T_G$ analogously.

Using this new mechanism will help us more easily extend our insertion functions.

Definition 7.13 (IPP *insert$_{in}$*)**.** For an IPP grammar $G = (L, O_{in}, O_{pre}, O_{post})$ with conflict patterns Q, we define the function *insert$_{in}$* : $(T_G \times O_{in} \times T_G) \to T_G$ as follows:

$$insert_{in}(t_1, o, t_2) =$$

$$\begin{cases} [t_1\ o\ l_2], & \text{if } t_2 = l_2 \in L \\ [t_1\ o\ [t_{21}\ o_2\ t_{22}]], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{in}(t_1, o, t_{21})\ o_2\ t_{22}], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } R_Q(o, t_2) \\ [t_1\ o\ [o_2\ t_{21}]] & \text{if } t_2 = [o_2\ t_{21}], o_2 \in O_{pre} \\ [t_1\ o\ [t_{21}\ o_2]], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{in}(t_1, o, t_{21})\ o_2], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } R_Q(o, t_2) \end{cases}$$

As before, the first four cases handle insertion into an atomic, infix, or prefix nodes. We have slightly changed how we defined insertion into infix nodes to make use of our newly defined relation R_Q, but the principle remains the same: We wish to explicitly avoid right-conflicts. The main difference from before is that we now also check for deep conflicts.

The last two cases involve insertion into postfix nodes. Because insertion to the top can introduce right-conflicts, these are handled analogously to infix nodes.

We will now extend *insert$_{pre}$* in a similar way.

Definition 7.14 (IPP *insert$_{pre}$*)**.** For an IPP grammar $G = (L, O_{in}, O_{pre}, O_{post})$ with conflict patterns Q, we define the function *insert$_{pre}$* : $(O_{pre} \times T_G) \to T_G$ as follows:

$$insert_{pre}(o, t_2) =$$

$$\begin{cases} [o\ l_2], & \text{if } t_2 = l_2 \in L \\ [o\ [t_{21}\ o_2\ t_{22}]], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{pre}(o, t_{21})\ o_2\ t_{22}], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } R_Q(o, t_2) \\ [o\ [o_2\ t_{21}]], & \text{if } t_2 = [o_2\ t_{21}], o_2 \in O_{pre} \\ [o\ [t_{21}\ o_2]], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{pre}(o, t_{21})\ o_2], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } R_Q(o, t_2) \end{cases}$$

For repairing postfix expressions, we introduce a new function *insert$_{post}$* : $(T_G \times O_{post}) \to T_G$. Calling *insert$_{post}$*$(t_1, o)$ repairs the tree $[t_1\ o]$ by right-inserting o into t_1.

Definition 7.15 (IPP *insert$_{post}$*)**.** For an IPP grammar $G = (L, O_{in}, O_{pre}, O_{post})$ with conflict patterns Q, we define the function *insert$_{post}$* : $(T_G \times O_{post}) \to T_G$ as follows:

$$insert_{post}(t_1, o) =$$

$$\begin{cases} [l_1\ o], & \text{if } t_1 = l_1 \in L \\ [[t_{11}\ o_1\ t_{12}]\ o], & \text{if } t_1 = [t_{11}\ o_1\ t_{12}], o_1 \in O_{in}, \text{ and } \neg L_Q(o, t_1) \\ [t_{11}\ o_1\ insert_{post}(t_{12}, o)], & \text{if } t_1 = [t_{11}\ o_1\ t_{12}], o_1 \in O_{in}, \text{ and } L_Q(o, t_1) \\ [[o_1\ t_{12}]\ o], & \text{if } t_1 = [o_1\ t_{12}], o_1 \in O_{pre}, \text{ and } \neg L_Q(o, t_1) \\ [o_1\ insert_{post}(t_{12}, o)], & \text{if } t_1 = [o_1\ t_{12}], o_1 \in O_{pre}, \text{ and } L_Q(o, t_1) \\ [[t_{11}\ o_1]\ o], & \text{if } t_1 = [t_{11}\ o_1], o_1 \in O_{post} \end{cases}$$

This function is defined analogously to *insert$_{pre}$*. Note how, unlike in all previous cases, we explicitly avoid left-conflicts instead of right-conflicts. This is because we insert in the opposite direction. This makes more sense for postfix expressions, where the operator is found on the opposite side.

Now we will also extend $repair_{in}$ to handle postfix nodes.

Definition 7.16 (IPP $repair_{in}$)**.** For an IPP grammar $G = (L, O_{in}, O_{pre}, O_{post})$ with conflict patterns Q, we define the function $repair_{in} : (T_G \times O_{in} \times T_G) \to T_G$ as follows:

$$repair_{in}(t_1, o, t_2) =$$

$$\begin{cases} insert_{in}(l_1, o, t_2), & \text{if } t_1 = l_1 \in L \\ repair_{in}(t_{11}, o_1, repair_{in}(t_{12}, o, t_2)), & \text{if } t_1 = [t_{11}\ o_1\ t_{12}], o_1 \in O_{in} \\ insert_{pre}(o_1, repair_{in}(t_{11}, o, t_2)), & \text{if } t_1 = [o_1\ t_{11}], o_1 \in O_{pre} \\ insert_{in}([t_{11}\ o_1], o, t_2), & \text{if } t_1 = [t_{11}\ o_1], o_1 \in O_{post} \end{cases}$$

This new case $repair_{in}([t_{11}\ o_1], o, t_2) = insert_{in}([t_{11}\ o_1], o, t_2)$ seems counter-intuitive. However, back in Chapter 4 we hinted that there would be cases where we use $insert_{in}$ to insert entire trees, rather than just a lexical symbol l_1. To explain why we are allowed to insert the entire tree $[t_{11}\ o_1]$, we first explain what $insert_{in}$ actually requires: The tree $t' = insert_{in}(t_1, o, t_2)$ is conflict-free if

1. t_2 is conflict-free, and
2. t_1 is conflict-free, and
3. o does not left-conflict with t_1.

These requirements are there because $insert_{in}$ only fixes right-conflicts between o and t_2. So what about if $t_1 = [t_{11}\ o_1]$ like we use it in $repair_{in}$? The second requirement states that this should be conflict-free. This is the case if we let the top-level *repair* function recursively repair the subtree t_1 first, like it already does with t_2. The third requirement states that o should not left-conflict with $[t_{11}\ o_1]$. This requirement holds because there exists no conflict pattern $CL_{in,post}$, nor does a deep conflict pattern exist where o can conflict deeper in t_{11}.

Definition 7.17 (IPP *repair*)**.** For an IPP grammar $G = (L, O_{in}, O_{pre}, O_{post})$ with conflict patterns Q, we define the function $repair : T_G \to T_G$ as follows:

$$repair(t) = \begin{cases} l, & \text{if } t = l \in L \\ repair_{in}(repair(t_1), o, repair(t_2)), & \text{if } t = [t_1\ o\ t_2], o \in O_{in} \\ insert_{pre}(o, repair(t_1)), & \text{if } t = [o\ t_1], o \in O_{pre} \\ insert_{post}(repair(t_1), o), & \text{if } t = [t_1\ o], o \in O_{post} \end{cases}$$

Note that this new definition of *repair* also recursively repairs the left-subtree t_1 of infix nodes. This is to account for the case in $repair_{in}$ where we required t_1 to be conflict-free in case it is a postfix node.

Safety and Partial Completeness Proof Using this new repair function, we have proven that the semantics is indeed safe and *partially* proven it is complete using the Coq Proof Assistant. The missing part of the completeness proof is the lemma that states that *repair* is fully dependant on the yield of the tree (Lemma 4.5). This was one of the properties we used for proving completeness for infix and prefix grammars.

Conjecture 7.18 (IPPG Repair is fully yield-dependent)**.** *For any two trees $t_1, t_2 \in T_G$ and sets of conflict patterns $Q = (Q^i, Q^{rm}, Q^{lm})$, if $yield(t_1) = yield(t_2)$, then it follows that $repair(t_1) = repair(t_2)$.*

The reason why we were able to prove this for infix and prefix grammars, but not when we also add postfix, is because the *repair* function now has *inconsistent behavior* for yield-equivalent trees. For infix expression grammars, repairing a tree t resulted in a nested application of $insert_{in}$ on the yield of t. We gave the following example to illustrate this concept:

$$repair([[1 + 2] * [3 - 4]]) = ... = insert_{in}(1, +, insert_{in}(2, *, insert_{in}(3, -, 4)))$$

No matter how you reorder the tree $[[1 + 2] * [3 - 4]]$, it will eventually reduce to this same nested application of $insert_{in}$. The same idea holds when we add prefix expressions:

$$repair(-[1 + [-2]]) = ... = insert_{pre}(-, insert_{in}(1, +, insert_{pre}(-, 2)))$$

The reason this works is because we consistently use left-insertion and we only insert one symbol at a time. As soon as you introduce both prefix *and* postfix, this same consistency is difficult to achieve. Postfix nodes are easier to repair with right-insertion and we also have cases where we insert entire trees with $insert_{in}$. For example, take the expression $1 * 2{+}{+}$, which has two parse trees $[1 * [2{+}{+}]]$ and $[[1 * 2]{+}{+}]$. Repairing these gives the following behaviors:

$$\begin{aligned} repair([1 * [2{+}{+}]]) &= insert_{in}(1, *, repair([2{+}{+}])) = insert_{in}(1, *, [2{+}{+}]) = ... \\ repair([[1 * 2]{+}{+}]) &= insert_{post}(repair([1 * 2]), {+}{+}) = insert_{post}([1 * 2], {+}{+}) = ... \end{aligned}$$

The first parse tree uses $insert_{in}$, whereas the second parse tree uses $insert_{post}$ to get the result. This especially becomes complicated when you get to even larger trees with a mixture of all types of expressions. This made proving the property much more difficult compared with we had just infix and prefix expressions.

Chapter 8

Related Work

In this chapter we compare this thesis to some of the other work in literature. We begin very close to the subject of this thesis, and compare our proof the original proof idea by Souza Amorim and Visser (2019) in Section 8.1. In Section 8.2 we compare the semantics of disambiguation rules from Souza Amorim and Visser (2019) to some other ideas for semantics in literature. Lastly, in Section 8.3 we look at some other proof mechanizations related to context-free grammars. In particular, we will look at some verified parser generators, as quite a handful of them can be found throughout literature.

8.1 Alternate Safety and Completeness Proof

Our safety and completeness proofs from Chapters 5 and 6 were made using tree insertion. We constructed conflict-free and unique trees in a bottom-up fashion. This entire thesis is built on the draft paper by Souza Amorim and Visser (2019), which first introduced the direct semantics for disambiguation rules. There they used a different approach to proving safety and completeness using *tree reorderings*. The original idea of this thesis was to complete the proofs using that approach, but instead we deviated and used the tree insertion method. In this section we will show what tree reorderings are and how they could potentially be used to prove safety and completeness. For simplicity, we will only consider the context of infix expression grammars in this section.

Definition 8.1 (Infix Reordering)**.** We define the relation $\overset{RI}{\rightarrow}$ over parse trees inductively as follows:

$$\frac{}{[t_1\ o\ [t_{21}\ o_2\ t_{22}]] \overset{RI}{\rightarrow} [[t_1\ o\ t_{21}]\ o\ t_{22}]} \qquad \frac{t_1 \overset{RI}{\rightarrow} t_1'}{[t_1\ o\ t_2] \overset{RI}{\rightarrow} [t_1'\ o\ t_2]} \qquad \frac{t_2 \overset{RI}{\rightarrow} t_2'}{[t_1\ o\ t_2] \overset{RI}{\rightarrow} [t_1\ o_2\ t_2']}$$

Let $\overset{RI*}{\longleftrightarrow}$ be the reflexive, symmetric, transitive closure of $\overset{RI}{\rightarrow}$. We say a tree t_2 is a *reordering* of t_1 if $t_1 \overset{RI*}{\longleftrightarrow} t_2$.

In Figure 8.1 we show all parse trees for the expression $1 + 2 * 3 - 4$ (with the traditional arithmetic grammar). The figure shows that all these parse trees are reorderings of each other. In fact, we can generalize this relation between yields and reorderings in the following lemma:

Lemma 8.2 (Infix Yields and Reorderings)**.** *For two parse trees t_1 and t_2, it follows that $yield(t_1) = yield(t_2)$ iff $t_1 \overset{RI*}{\longleftrightarrow} t_2$.*

Proof. As shown by Souza Amorim and Visser (2019). □

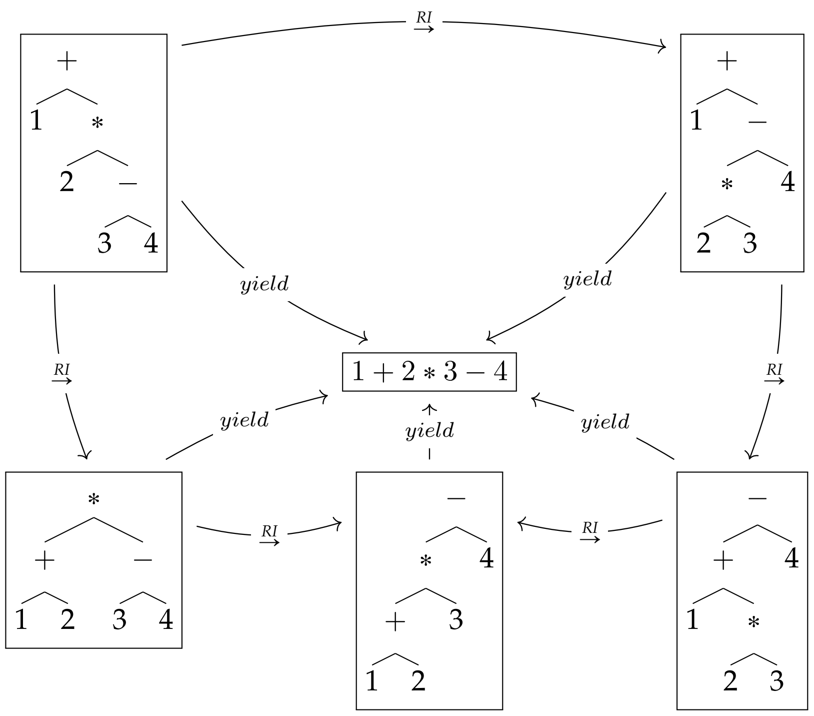


Figure 8.1: Parse tree reorderings.

So far, the reordering relation does not take ambiguity into account. For that we define a subset of reorderings that attempt to repair conflicts.

Definition 8.3 (Infix Disambiguation Reordering)**.** We define the relation $\xrightarrow{DI}$ over parse trees inductively as follows:

$$\frac{CR(o, o_2) \in Q}{[t_1\ o\ [t_{21}\ o_2\ t_{22}]] \xrightarrow{DI} [[t_1\ o\ t_{21}]\ o\ t_{22}]} \qquad \frac{t_1 \xrightarrow{DI} t_1'}{[t_1\ o\ t_2] \xrightarrow{DI} [t_1'\ o\ t_2]}$$

$$\frac{CL(o, o_1) \in Q}{[[t_{11}\ o_1\ t_{12}]\ o\ t_2] \xrightarrow{DI} [t_{11}\ o_1\ [t_{12}\ o\ t_2]]} \qquad \frac{t_2 \xrightarrow{DI} t_2'}{[t_1\ o\ t_2] \xrightarrow{DI} [t_1\ o_2\ t_2']}$$

Let $\xrightarrow{DI*}$ be the reflexive and transitive closure of $\xrightarrow{DI}$.

This new relation repairs a conflict with each step. If there are no longer any conflicts in a tree t, it is in *normal form*, i.e., there exists no t' such that $t \xrightarrow{DI} t'$. We can use this to try and prove both safety and completeness.

Proving Safety Recall that the goal of proving safety is to find a yield-equivalent and conflict-free tree for any tree t. We can use $\xrightarrow{DI*}$ to find such a tree. If we have $t \xrightarrow{DI*} t'$, with t' in normal form, it means we are done. In other words, we must show that $\xrightarrow{DI}$ is *terminating* (or *strongly normalizing*). Note that termination requires proving that each path in $\xrightarrow{DI}$ is finite. For safety, we only need *one* finite path for each tree to a normal form. So an alternative is proving the relation is *weakly normalizing*: $\exists t', t \xrightarrow{DI*} t'$ with t' in normal form for each t.

Proving Completeness To show completeness, we should have that each conflict-free tree is unique within its yield equivalence class. In terms of reordering using $\xrightarrow{DI}$, we must show that two yield-equivalent trees will have the same unique normal form. Assuming we have already proven *termination* from before, Souza Amorim and Visser (2019) shows that all we still need to do is prove $\overset{DI}{\to}$ is *locally confluent*: If $t \overset{DI}{\to} t_1$ and $t \overset{DI}{\to} t_2$, then there exists a t' with $t_1 \xrightarrow{DI*} t'$ and $t_2 \xrightarrow{DI*} t'$.

Comparison with Insertion Proving *termination* and *local confluence* can be quite difficult. This caused the deviation in this thesis into a different approach, using the bottom-up insertion method from Chapter 4. This does not mean there are no large similarities between the two approaches. The first thing that may come to mind is Lemma 8.2. It is effectively almost the same as Lemmas 4.4 and 4.5, which together state that yield equivalence is the same as repair equivalence. In a way, this means that $repair(t) = t'$ can be seen as a very specific reordering between t and t'. This specific reordering can then be used to prove the weak normalization property, required for proving safety.

8.2 Alternate Semantics

While the formalism and semantics for context-free grammars are already well-established, and first developed in the mid-1950s (Hopcroft and Ullman 1979), the semantics for disambiguation rules are relatively new and still an active research topic.

Souza Amorim and Visser (2019) distinguishes between two different types of semantics for disambiguation rules: *indirect* and *direct*. Indirect approaches use a translation into another formalism, whereas direct approaches do not use a translation and directly describe the behavior of associativity and priority rules. The semantics in Souza Amorim and Visser (2019), which we also used in this thesis, define the semantics as disambiguation filters over parse trees. This is an example of a direct semantics.

Indirect Semantics Indirect approaches are generally undesirable, as those require understanding of another formalism. Johnson et al. (1975) define the semantics of these disambiguation rules by means how a parser would solve shift/reduce conflicts. This requires understanding of the workings and generation of an LR parse table. Adams and Might (2017) use tree automata (Comon 1997) to describe disambiguation, which is also an indirect approach.

Direct Semantics Direct semantics throughout literature are mostly done with the idea of negatively using a set of 'illegal' sub-parse trees, and was first introduced by Thorup (1994) and Klint and Visser (1994) by means of disambiguation filters. Thorup (1996) shows this method guarantees that the language remains unchanged, i.e., *safety*. The semantics of SDF2 (predecessor of SDF3) (Visser et al. 1997; Visser et al. 1995) are defined as tree patterns that filter over subtrees directly using disambiguation filters. These semantics can be seen as the predecessor to Souza Amorim and Visser (2019), on which this thesis is based. The SDF2 semantics have been shown to be *unsafe* for some grammars by Afroozeh et al. (2013). This lead to the creation of this new semantics, which also introduced deep conflict patterns.

Unlike the semantics discussed before, Afroozeh et al. (2013) uses a filter over *derivations* rather than parse trees. We briefly mentioned derivations in Section 2.1 as the alternative and more classical way of describing the semantics of context-free grammars. Similarly to how we defined tree patterns, they defined patterns for derivations. This is also a direct semantics. As an implementation for the semantics, they used grammar rewriting to transform a context-free grammar into another context-free grammar. This can be seen as an alternative to our method of parse tree repairing.

8.3 Verified Parsing

This thesis is not the first to include a context-free grammar related mechanization in a proof assistant. *Verified parser generators* in particular have taken their own spot literature.

Definition 8.4 (Parsers)**.** A parser for a context-free grammar $G = (\Sigma, N, P)$ is defined as a function $\mathit{parse} : \Sigma^* \rightarrow \mathcal{P}(T_G)$.

A parser is *sound* if for all sentences $w \in \Sigma^*$ and $t \in \mathit{parse}(w)$, we have $\mathit{yield}(t) = w$. Conversely, a parser is *complete* if for all sentences $w \in \Sigma^*$ and trees $t \in T_G$ with $\mathit{yield}(t) = w$, we have $t \in \mathit{parse}(w)$.

A parser is considered verified if there is a proof (in a proof assistant) for both soundness and completeness. We found five verified parser generators which we will list in this section.

Barthwal and Norrish (2009) made the first verified parser generator, which supports SLR grammars (DeRemer 1969). This was done in the HOL theorem prover (HOL Development Team 2019).

Jourdan, Pottier, and Leroy (2012) support LR(1) grammars, which is a class of context-free grammars higher than SLR grammars. However, the main difference with Barthwal and Norrish (2009) is that this is not a parser generator, but a verified parser *validator*. A validator takes a parser as input and checks whether it is sound and complete. Note that this validator can give false negatives. Meaning that it can reject sound and complete parsers. The validator itself is implemented in the Coq Proof Assistant.

Lasser et al. (2019) made a verified parser generator for LL(1) grammars. This is a class of grammars that supports significantly fewer grammars than SLR and LR(1) grammars. However, unlike the previous cases, they also have a proof that guarantees that the parser also terminates.

There are two verified parser interpreters by Koprowski and Binsztok (2010) and Blaudeau and Shankar (2020). The former implemented in the Coq Proof Assistant and the latter in the PVS Specification and Verification System (PVS Development Team 2014). The grammars they support are *Parsing Expression Grammars* (PEGs) (Ford 2004), which is an alternative to context-free grammars. One difference is that PEGs are closer to an implementation of a parser than CFGs. As such, PEGs are not compiled down to a parse table first, but are used directly with a *parser interpreter*.

Chapter 9

Future Work

This chapter lists some of the direct future work that may arise from this thesis.

Full Completeness Proof We managed to proof both safety and completeness for grammars that contain both infix and prefix expressions. In Section 7.3 we mentioned that we were unable to fully proof completeness for expression grammars that also contain postfix expressions. This is probably the biggest disappointment of this thesis. In that section we mention the specific problem with our attempt to prove it. We hope that future work will include a solution.

More CFG Support The largest set of context-free grammars we supported in this thesis are IPP grammars, which contain infix, prefix, and postfix expressions. As future work, it makes sense to extend this to support more types of CFGs. This means extending the semantics of associativity and priority to support them, as well as verifying safety and completeness for the new semantics. In particular, there are some extensions already supported by Souza Amorim and Visser (2019). We will specifically mention some of these below. For some we may include an intuition as to how a safety and completeness proof could look like for the grammars, by proposing an extension to the *repair* function.

Closed Expressions In this thesis we have not included *closed expressions*. These are productions of the form $A.C = o_1\ A\ o_2$, with o_1 and o_2 being terminals. These expressions are easy to include because they do not introduce any ambiguity. A simple example of a practical closed production is the bracket production: `Exp = ( Exp )`. In fact, bracket expressions in particular are often used for *explicit disambiguation*. With explicit disambiguation a user can use these expressions to explicitly encode some order in a sentence. The sentence $1 + 2 - 3$ should be read as $(1+2)-3$ because of left-associativity. However, with a bracket expression you can choose to write down $1 + (2 - 3)$, explicitly giving priority to the minus operator.

Although these expressions do not introduce any ambiguity, for a safety and completeness proof we still need to update our *repair* function from Chapter 4. This can be done by first repairing each closed node $[o_1\ t\ o_2]$ in the parse tree, by recursively repairing the inner tree t. Then you can treat each of these closed nodes as if they were a lexical symbol.

Mixfix Expressions A mixfix expression grammar permits productions where you can have an arbitrary number of terminals interleaved with the nonterminal A. A mixfix expression grammar simply covers all expression grammars: Which is a CFG with only a single nonterminal A. We exclude the trivial production $A.C = A$.

Figure 9.1 shows an example of a Mixfix Expression Grammar. In particular, the `Cond`, `IfElse`, `Idx`, and `While` production we were not able to express using IPP grammars.

```
context-free syntax
    Exp.Lit    = NUM
    Exp.Var    = ID
    Exp.Add    = Exp "+" Exp
    Exp.Minus  = "-" Exp
    Exp.Incr   = Exp "++"
    Exp.Cond   = Exp "?" Exp ":" Exp
    Exp.IfElse = "if" Exp "then" Exp "else" Exp
    Exp.Idx    = Exp "[" Exp "]"
    Exp.While  = "while" Exp "do" Exp "end"
```

Figure 9.1: Example of a Mixfix Expression Grammar

The trick to repairing parse trees containing mixfix expression nodes, is to pretend they are a mixture of closed expressions, and one of infix, prefix, or postfix. In particular, we distinguish between *infix mixfix*, *prefix mixfix*, *postfix mixfix*, and *closed mixfix*. For instance, in Figure 9.1, the `Cond` production is an example of infix mixfix. This is because we have two nonterminals on two far sides of the production and all terminals are on the inside. Similarly, `IfElse` is prefix mixfix, `Idx` is postfix mixfix, and `While` is closed mixfix.

First take the `While` production as an example. This may have two nonterminals on the inside, but has the exact same behavior as an ordinary closed expression. We can repair a parse tree containing `While` nodes, by recursively repairing the two inner nodes, then treating it as if it were a plain terminal (atomic node).

Now take the `IfElse` production. We can split this production into two parts: The closed part and the prefix part. The closed part is the beginning `"if" Exp "then" Exp "else"`. When repairing a parse tree containing an `IfElse` node, we can first recursively repair the two inner trees of its closed part. Recall that we can now treat this as if it were an ordinary terminal, which we will call c. This new expression has the form c `Exp`. By pretending c is just a terminal symbol, we can treat this entire production as it if were a prefix production, and repair it the same way we would an ordinary prefix node.

For infix and postfix mixfix productions such as `Cond` and `Idx`, we can similarly split them up into a closed part and infix or postfix part. After repairing the closed part, we can treat them as if they were infix and postfix productions.

Operator Overlap In our definition for IPP grammars $G = (L, O_{in}, O_{pre}, O_{post})$, we assumed that the terminal sets L, O_{in}, O_{pre}, and O_{post} have no overlap. Operator overlap can introduce ambiguities that associativity and priority rules cannot solve. Souza Amorim and Visser (2019) distinguish between two types of ambiguities: *Reordering ambiguities* and *non-reordering ambiguities*.

Reordering ambiguities are the type which we have supported throughout this thesis. Back in Section 8.1 we explained how yield-equivalent parse trees can be considered reorderings of each other. These parse trees are made up out of the same productions. The ambiguity arises when these productions appear in different positions, but can be solved through reordering.

Non-reordering ambiguities do not have the property where two ambiguous parse trees can be reordered into each other. Take the grammar with operator overlap from Figure 9.2. Suppose we have the expression $1 - -2$. This sentence is ambiguous because it has two corresponding parse trees: $[\texttt{Exp.Sub} = 1 - [\texttt{Exp.Min} = -2]]$ and $[\texttt{Exp.Sub} = [\texttt{Exp.Dec} = 1-] - 2]$. This is a non-reordering ambiguity, because these two parse trees are made up out of different productions, and as such can not be reordered into each other.

```
context-free syntax
    Exp.Lit    = NUM
    Exp.Sub    = Exp "-" Exp
    Exp.Min    = "-" Exp
    Exp.Dec    = Exp "-"
```

Figure 9.2: Example of an Expression Grammar with Operator Overlap

Non-reordering ambiguities will have no effect on the *safety* of the semantics, but will influence *completeness*. Looking at our proof for completeness, this non-reordering ambiguity will give problems to the property stated in Lemma 4.5. It states that if two parse trees have equivalent yield $yield(t_1) = yield(t_2)$, then they repair to the same tree $repair(t_1) = repair(t_2)$. This *only* works if t_1 and t_2 are reorderings of each other. This is because *repair* can be seen as a specific reordering. It does not change the productions the tree is made of.

Souza Amorim and Visser (2019) consider different types of operator overlap, and propose a classification between *harmless* and *harmful overlap*. Harmful overlap, such as the one in Figure 9.2 should be explicitly forbidden, because they will cause ambiguities. Future work could include verifying safety and completeness for grammars that allow non-harmful overlap. But also verify that harmful overlap indeed cause ambiguities.

Remark: In our Coq proof for safety and completeness, we permitted overlap between prefix and infix operators. This is because it is a non-harmful overlap and therefore did not stand in the way of proving completeness.

Generalising Completeness Constraints In Chapter 5, we defined some constraints on the disambiguation rules that ensure safety. We showed that safety holds if *and only if* the disambiguation rules are safe. Meaning that if the disambiguation rules are not safe, then we can also guarantee that the grammar will be unsafe.

The same does not hold for the completeness constraints we have defined in Chapter 6. Lemma 6.2 does state this, but only under the condition that the disambiguation rules are also safe. It is desirable to find a set of constraints, such that if these constraints do not hold, that a grammar is guaranteed to be ambiguous. In other words, we wish to *generalize* the completeness constraints.

Proof for General Constraints of IPP Grammars We have not extended Lemmas 5.2 and 6.2, which are about most general safety and completeness constraints, for IPP Grammars. Although we do conjecture that this holds, this has not been formally proven yet.

Verified Implementation Probably the most far-reaching out of all the future work suggestions in this chapter, a semantics means there could be a potential verified implementation of that semantics. In Section 8.3 we listed some verified parsers, but none of them support ambiguous grammars. New research territory would be creating a new parser generator that also supports grammars with disambiguation rules. The idea of repairing parse trees could serve a potential big role in such an implementation.

IDE Support Suppose a language developer is developing their new programming language using the Spoofax Language Workbench (Kats and Visser 2010), which is a platform for developing (domain-specific) programming languages. It includes SDF3 for specifying the syntax of the language. As of writing this thesis, Spoofax does not warn a user for potential ambiguities in the SDF3 code. Using the safety and completeness proof from this thesis as justification, the platform could give the language developer a warning if their disambiguation rules do not fit the completeness constraints from Chapter 6. The warning message could suggest potential disambiguation rules between productions. We made some suggestions for this in Sections 5.1 and 6.1.

Chapter 10

Conclusion

In this thesis we aimed at solving the problem of *safety* and *completeness* for the semantics of disambiguation rules from Souza Amorim and Visser (2019). That is, give formal and mechanized proofs for these two properties. We succeeded in solving this problem for expression grammars containing infix and prefix expressions (IP grammars). When also adding postfix expressions (IPP grammars), we also managed a safety proof, but only a *partial* completeness proof. The proofs are implemented in the Coq proof assistant [1], which gives an additional layer of certainty that the proof is correct.

We did this by means of *parse tree repairing*, which by itself is interesting even without considering safety and completeness. Parse tree repairing can be considered a new implementation for disambiguation rules, and as such is open for further research.

For safety of infix expression grammars, we managed to find a most general possible constraints on the disambiguation rules. It is most general in the sense that we can guarantee a grammar is not safe if the disambiguation rules do not follow these constraints. We managed to do the same for completeness, but under the assumption that the disambiguation rules are also safe. Our safety and completeness restrictions are logically equivalent to the restrictions imposed by Souza Amorim and Visser (2019), which shows that those restrictions are also most general for safety and completeness.

Lastly, we made some suggestions for syntax definition formalisms, such as SDF3, to enforce safe and complete disambiguation rules.

[1] `https://zenodo.org/record/4680987#.YHRVeOgzZjE`

Bibliography

Adams, Michael D. and Matthew Might (Oct. 2017). “Restricting Grammars with Tree Automata”. In: *Proc. ACM Program. Lang.* 1.OOPSLA. DOI: `10.1145/3133906`. URL: `https://doi.org/10.1145/3133906`.

Afroozeh, Ali et al. (2013). “Safe Specification of Operator Precedence Rules”. In: *Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, pp. 137–156. ISBN: 978-3-319-02654-1.

Aho, Alfred V. et al. (2007). *Compilers: principles, techniques, and tools*. 2nd ed. Pearson/Addison Wesley, pp. 191–302.

Barthwal, Aditi and Michael Norrish (2009). “Verified, executable parsing”. In: *European Symposium on Programming*. Springer, pp. 160–174.

Blaudeau, Clement and Natarajan Shankar (2020). “A Verified Packrat Parser Interpreter for Parsing Expression Grammars”. In: *arXiv preprint arXiv:2001.04457*.

Comon, Hubert (1997). “Tree automata techniques and applications”. In: *http://www. grappa. univ-lille3. fr/tata*.

Coq Development Team (Jan. 2020). *The Coq Proof Assistant*. URL: `https://coq.inria.fr/`.

DeRemer, Franklin Lewis (1969). “Practical translators for LR (k) languages.” PhD thesis. Massachusetts Institute of Technology.

Ford, Bryan (2004). “Parsing expression grammars: a recognition-based syntactic foundation”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 111–122.

Gosling, James et al. (Feb. 2020). *The Java Language Specification*. URL: `https://docs.oracle.com/javase/specs/jls/se14/html/index.html`.

HOL Development Team (Aug. 2019). *HOL Interactive Theorem Prover*. URL: `https://hol-theorem-prover.org/`.

Hopcroft, John E. and Jeff D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company.

Johnson, Stephen C et al. (1975). *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

Jourdan, Jacques-Henri, François Pottier, and Xavier Leroy (2012). “Validating LR (1) parsers”. In: *European Symposium on Programming*. Springer, pp. 397–416.

Kats, Lennart CL and Eelco Visser (2010). “The spoofax language workbench: rules for declarative specification of languages and IDEs”. In: *ACM sigplan notices* 45.10, pp. 444–463.

Klint, Paul and Eelco Visser (1994). “Using filters for the disambiguation of context-free grammars”. In: *Proc. ASMICS Workshop on Parsing Theory*. Milan, Italy, pp. 1–20.

Koprowski, Adam and Henri Binsztok (2010). “TRX: A formally verified parser interpreter”. In: *European Symposium on Programming*. Springer, pp. 345–365.

Lasser, Sam et al. (2019). “A Verified LL (1) Parser Generator”. In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

PVS Development Team (Feb. 2014). *PVS Specification and Verification System*. URL: `https://pvs.csl.sri.com/index.shtml`.

Souza Amorim, Luís Eduardo de and Eelco Visser (2019). “A Direct Semantics for Declarative Disambiguation of Expression Grammars”. In:

— (2020). “Multi-purpose Syntax Definition with SDF3”. In: *Software Engineering and Formal Methods*. Ed. by Frank de Boer and Antonio Cerone. Cham: Springer International Publishing, pp. 1–23. ISBN: 978-3-030-58768-0.

Spoofax Development Team (Dec. 2020). *Syntax Definition with SDF3*. URL: `http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/`.

Thorup, Mikkel (May 1994). “Controlled Grammatic Ambiguity”. In: *ACM Trans. Program. Lang. Syst.* 16.3, pp. 1024–1050. ISSN: 0164-0925. DOI: `10.1145/177492.177759`. URL: `https://doi.org/10.1145/177492.177759`.

— (1996). “Disambiguating grammars by exclusion of sub-parse trees”. In: *Acta Informatica* 33.6, pp. 511–522.

Visser, Eelco et al. (1995). “A case study in optimizing parsing schemata by disambiguation filters”. In: *Report P9507, Dept. of Computer Science, University of Amsterdam, the Netherlands*.

— (1997). *A family of syntax definition formalisms*. Universiteit van Amsterdam. Programming Research Group.

Vollebregt, Tobi, Lennart CL Kats, and Eelco Visser (2012). “Declarative specification of template-based textual editors”. In: *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*. ACM, p. 8.

Appendix A

Mathematical Symbols and Notation

This appendix list all the mathematical symbols used throughout this thesis.

$$
\begin{aligned}
G &:= \text{a context-free grammar (CFG)} \\
a \in \Sigma &:= \text{terminals} \\
w \in \Sigma^* &:= \textit{words}\text{, sequences of terminal symbols} \\
A \in N &:= \text{nonterminals} \\
X \in V = \Sigma \cup N &:= \text{symbols} \\
(A.C = X_1...X_n) \in P &:= \text{productions}
\end{aligned}
$$

$$
\begin{aligned}
l \in L \subseteq \Sigma &:= \text{lexical symbols} \\
o \in O \subseteq \Sigma &:= \text{operator symbols}
\end{aligned}
$$

Numbers, mathematical operators such as $+$ and $-$, and the terms `NUM` and `ID` are also often used as an examples for terminals in Σ. The term `Exp` is often used as a specific example for a nonterminal in N.

$$
\begin{aligned}
t \in T_G &:= \text{parse trees} \\
q \in Q \subseteq TP_G &:= \text{tree patterns / conflict patterns}
\end{aligned}
$$

Symbols in Σ are used to represent leaves of trees. The notation $[A.C = t_1...t_n]$ represents a node in a tree, or just $[t_1...t_n]$ if $A.C$ is clear from the context. A node $[t_1\ o\ t_2]$ represents an infix node, $[o\ t]$ a prefix node, and $[t\ o]$ a postfix node. An underscore _ represents the empty tree pattern.

Appendix B

Coq Definitions and Theorems

This Appendix contains all definitions and theorems (safety and completeness) for IPP grammars. Not included are:

- Definitions and Theorems for Infix Expression grammars and IP grammars. We have proven these separately, but since IPP grammars covers all of these grammars, showing those would be unnecessary.
- The proofs, as these would be unreadable without the use of an actual interactive Coq IDE.
- The definition of the repair function (see Appendix C).
- Each individual (sub) Lemma.

These items can still be found in the full source code.

IPP Grammars

```
Inductive prod OP :=
  | InfixProd : OP → prod OP
  | PrefixProd : OP → prod OP
  | PostfixProd : OP → prod OP.

Record ippg := mkIppgrammar {
  LEX : Type;
  OP : Type;
  prods: prod OP → Prop
}.

Definition word g := list (LEX g + OP g).
```

Well-formed Parse Trees

```
Inductive parse_tree (g : ipg) :=
  | AtomicNode : LEX g → parse_tree g
  | InfixNode : parse_tree g → OP g → parse_tree g → parse_tree g
  | PrefixNode : OP g → parse_tree g → parse_tree g
  | PostfixNode : parse_tree g → OP g → parse_tree g.

Inductive wf_parse_tree g : parse_tree g → Prop :=
  | Atomic_wf l :
     wf_parse_tree g (AtomicNode l)
  | Infix_wf t1 o t2 :
     g.(prods) (InfixProd o) →
     wf_parse_tree g t1 →
     wf_parse_tree g t2 →
     wf_parse_tree g (InfixNode t1 o t2)
  | Prefix_wf o t :
     g.(prods) (PrefixProd o) →
     wf_parse_tree g t →
     wf_parse_tree g (PrefixNode o t)
  | Postfix_wf o t :
     g.(prods) (PostfixProd o) →
     wf_parse_tree g t →
     wf_parse_tree g (PostfixNode t o).
```

Yields

```
Fixpoint yield {g} t : word g :=
 match t with
  | AtomicNode l ⇒ [inl l]
  | InfixNode t1 o t2 ⇒ yield t1 ++ inr o :: yield t2
  | PrefixNode o t ⇒ inr o :: yield t
  | PostfixNode t o ⇒ yield t ++ [inr o]
 end.
```

Tree Patterns

```
Inductive tree_pattern g :=
  | HPatt : tree_pattern g (* Empty Tree Pattern *)
  | InfixPatt : tree_pattern g → OP g → tree_pattern g → tree_pattern g
  | PrefixPatt : OP g → tree_pattern g → tree_pattern g
  | PostfixPatt : tree_pattern g → OP g → tree_pattern g.
```

Shallow Matching

```
Inductive matches {g} : parse_tree g → tree_pattern g → Prop :=
  | HMatch t :
     matches t HPatt
  | InfixMatch t1 t2 q1 q2 o :
     matches t1 q1 →
     matches t2 q2 →
     matches (InfixNode t1 o t2) (InfixPatt q1 o q2)
  | PrefixMatch t q o :
     matches t q →
     matches (PrefixNode o t) (PrefixPatt o q)
  | PostfixMatch t q o :
     matches t q →
     matches (PostfixNode t o) (PostfixPatt q o).

Definition matches_set {g} t (Q : tree_pattern g → Prop) : Prop :=
 exists q, Q q ∧ matches t q.
```

Deep Rightmost Matching

```
Inductive matches_rm {g} : parse_tree g → tree_pattern g → Prop :=
  | Match_rm t q :
     matches t q →
```

```
   matches_rm t q
 | InfixMatch_rm t1 o t2 q :
   matches_rm t2 q →
   matches_rm (InfixNode t1 o t2) q
 | PrefixMatch_rm o t q :
   matches_rm t q →
   matches_rm (PrefixNode o t) q.

Inductive matches_drm {g} : parse_tree g → tree_pattern g → Prop :=
 | InfixMatch_drm t1 t2 q1 q2 o :
   matches_rm t1 q1 →
   matches_rm t2 q2 →
   matches_drm (InfixNode t1 o t2) (InfixPatt q1 o q2)
 | PrefixMatch_drm t q o :
   matches_rm t q →
   matches_drm (PrefixNode o t) (PrefixPatt o q)
 | PostfixMatch_drm t o q :
   matches_rm t q →
   matches_drm (PostfixNode t o) (PostfixPatt q o).

Definition matches_drm_set {g} t (Q : tree_pattern g → Prop) : Prop :=
 exists q, Q q ∧ matches_drm t q.
```

Deep Leftmost Matching

```
Inductive matches_lm {g} : parse_tree g → tree_pattern g → Prop :=
 | Match_lm t q :
   matches t q →
   matches_lm t q
 | InfixMatch_lm t1 o t2 q :
   matches_lm t1 q →
   matches_lm (InfixNode t1 o t2) q
 | PostfixMatch_lm o t q :
   matches_lm t q →
   matches_lm (PostfixNode t o) q.

Inductive matches_dlm {g} : parse_tree g → tree_pattern g → Prop :=
 | InfixMatch_dlm t1 t2 q1 q2 o :
   matches_lm t1 q1 →
   matches_lm t2 q2 →
   matches_dlm (InfixNode t1 o t2) (InfixPatt q1 o q2)
 | PrefixMatch_dlm t q o :
   matches_lm t q →
   matches_dlm (PrefixNode o t) (PrefixPatt o q)
 | PostfixMatch_dlm t o q :
   matches_lm t q →
   matches_dlm (PostfixNode t o) (PostfixPatt q o).

Definition matches_dlm_set {g} t (Q : tree_pattern g → Prop) : Prop :=
 exists q, Q q ∧ matches_dlm t q.
```

Subtree Exclusion

```
(* Free from shallow conflicts *)
```

```
Inductive i_conflict_free {g} (Q : tree_pattern g → Prop)
      : parse_tree g → Prop :=
  | Atomic_cf l :
     i_conflict_free Q (AtomicNode l)
  | Infix_cf t1 o t2 :
     ¬ matches_set (InfixNode t1 o t2) Q →
     i_conflict_free Q t1 →
     i_conflict_free Q t2 →
     i_conflict_free Q (InfixNode t1 o t2)
  | Prefix_cf t o :
     ¬ matches_set (PrefixNode o t) Q →
     i_conflict_free Q t →
     i_conflict_free Q (PrefixNode o t)
  | Postfix_cf t o :
     ¬ matches_set (PostfixNode t o) Q →
     i_conflict_free Q t →
     i_conflict_free Q (PostfixNode t o).

(* Free from deep rightmost conflicts *)
Inductive drm_conflict_free {g} (Q : tree_pattern g → Prop)
      : parse_tree g → Prop :=
  | Atomic_drmcf l :
     drm_conflict_free Q (AtomicNode l)
  | Infix_drmcf t1 o t2 :
     ¬ matches_drm_set (InfixNode t1 o t2) Q →
     drm_conflict_free Q t1 →
     drm_conflict_free Q t2 →
     drm_conflict_free Q (InfixNode t1 o t2)
  | Prefix_drmcf t o :
     ¬ matches_drm_set (PrefixNode o t) Q →
     drm_conflict_free Q t →
     drm_conflict_free Q (PrefixNode o t)
  | Postfix_drmcf t o :
     ¬ matches_drm_set (PostfixNode t o) Q →
     drm_conflict_free Q t →
     drm_conflict_free Q (PostfixNode t o).

(* Free from deep leftmost conflicts *)
Inductive dlm_conflict_free {g} (Q : tree_pattern g → Prop)
      : parse_tree g → Prop :=
  | Atomic_dlmcf l :
     dlm_conflict_free Q (AtomicNode l)
  | Infix_dlmcf t1 o t2 :
     ¬ matches_dlm_set (InfixNode t1 o t2) Q →
     dlm_conflict_free Q t1 →
     dlm_conflict_free Q t2 →
     dlm_conflict_free Q (InfixNode t1 o t2)
  | Prefix_dlmcf t o :
     ¬ matches_dlm_set (PrefixNode o t) Q →
     dlm_conflict_free Q t →
     dlm_conflict_free Q (PrefixNode o t)
  | Postfix_dlmcf t o :
    ¬ matches_dlm_set (PostfixNode t o) Q →
    dlm_conflict_free Q t →
    dlm_conflict_free Q (PostfixNode t o).

(* Free from all conflicts *)
Definition conflict_free {g} (Qi Qrm Qlm : tree_pattern g → Prop) t :=
 i_conflict_free Qi t ∧ drm_conflict_free Qrm t ∧ dlm_conflict_free Qlm t.
```

Disambiguation Rules

```
Record drules g := mkDrules {
 prio : prod g.(OP) → prod g.(OP) → Prop;
 left_a : prod g.(OP) → prod g.(OP) → Prop;
 right_a : prod g.(OP) → prod g.(OP) → Prop;
}.
```

Common Conflict Patterns

```
Definition CL_infix_infix {g} o1 o2 : tree_pattern g :=
 InfixPatt (InfixPatt HPatt o2 HPatt) o1 HPatt.
Definition CR_infix_infix {g} o1 o2 : tree_pattern g :=
 InfixPatt HPatt o1 (InfixPatt HPatt o2 HPatt).
Definition CL_infix_prefix {g} o1 o2 : tree_pattern g :=
 InfixPatt (PrefixPatt o2 HPatt) o1 HPatt.
Definition CR_infix_postfix {g} o1 o2 : tree_pattern g :=
 InfixPatt HPatt o1 (PostfixPatt HPatt o2).
Definition CR_prefix_infix {g} o1 o2 : tree_pattern g :=
 PrefixPatt o1 (InfixPatt HPatt o2 HPatt).
Definition CL_postfix_infix {g} o1 o2 : tree_pattern g :=
 PostfixPatt (InfixPatt HPatt o2 HPatt) o1.
Definition CR_prefix_postfix {g} o1 o2 : tree_pattern g :=
 PrefixPatt o1 (PostfixPatt HPatt o2).
Definition CL_postfix_prefix {g} o1 o2 : tree_pattern g :=
 PostfixPatt (PrefixPatt o2 HPatt) o1.
```

IPP Conflict Patterns

```
(* shallow conflict patterns (Qi) *)
Inductive i_conflict_pattern {g} (pr : drules g) : tree_pattern g → Prop :=
 | CLeft o1 o2 :
    pr.(left_a) (InfixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CR_infix_infix o1 o2)
 | CRight o1 o2 :
    pr.(right_a) (InfixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CL_infix_infix o1 o2)
 | CPrio_infix_infix_1 o1 o2 :
    pr.(prio) (InfixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CL_infix_infix o1 o2)
 | CPrio_infix_infix_2 o1 o2 :
    pr.(prio) (InfixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CR_infix_infix o1 o2)
 | CPrio_prefix_infix o1 o2 :
    pr.(prio) (PrefixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CR_prefix_infix o1 o2)
 | CLeft_prefix_infix o1 o2 :
    pr.(left_a) (PrefixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CR_prefix_infix o1 o2)
  | CPrio_postfix_infix o1 o2 :
    pr.(prio) (PostfixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CL_postfix_infix o1 o2)
  | CRight_postfix_infix o1 o2 :
    pr.(right_a) (PostfixProd o1) (InfixProd o2) →
    i_conflict_pattern pr (CL_postfix_infix o1 o2).

(* deep rightmost conflict patterns (Qrm) *)
Inductive rm_conflict_pattern {g} (pr : drules g) : tree_pattern g → Prop :=
  | CPrio_infix_prefix o1 o2 :
    pr.(prio) (InfixProd o1) (PrefixProd o2) →
    rm_conflict_pattern pr (CL_infix_prefix o1 o2)
  | CRight_infix_prefix o1 o2 :
    pr.(right_a) (InfixProd o1) (PrefixProd o2) →
    rm_conflict_pattern pr (CL_infix_prefix o1 o2)
  | CPrio_postfix_prefix o1 o2 :
    pr.(prio) (PostfixProd o1) (PrefixProd o2) →
    rm_conflict_pattern pr (CL_postfix_prefix o1 o2)
  | CRight_postfix_prefix o1 o2 :
    pr.(prio) (PostfixProd o1) (PrefixProd o2) →
    rm_conflict_pattern pr (CL_postfix_prefix o1 o2).

(* deep leftmost conflict patterns (Qlm) *)
Inductive lm_conflict_pattern {g} (pr : drules g) : tree_pattern g → Prop :=
  | CPrio_infix_postfix o1 o2 :
    pr.(prio) (InfixProd o1) (PostfixProd o2) →
    lm_conflict_pattern pr (CR_infix_postfix o1 o2)
  | CLeft_infix_postfix o1 o2 :
    pr.(left_a) (InfixProd o1) (PostfixProd o2) →
    lm_conflict_pattern pr (CR_infix_postfix o1 o2)
  | CPrio_prefix_postfix o1 o2 :
    pr.(prio) (PrefixProd o1) (PostfixProd o2) →
    lm_conflict_pattern pr (CR_prefix_postfix o1 o2)
  | CLeft_prefix_postfix o1 o2 :
    pr.(left_a) (PrefixProd o1) (PostfixProd o2) →
    lm_conflict_pattern pr (CR_prefix_postfix o1 o2).
```

Conflict-free w.r.t. Disambiguation Rules

```
Definition cfree {g} (pr : drules g) t : Prop :=
 conflict_free
   (i_conflict_pattern pr)
   (rm_conflict_pattern pr)
   (lm_conflict_pattern pr) t.
```

Disambiguated Language

```
Definition dlanguage {g} (pr : drules g) w : Prop :=
 exists t, wf_parse_tree g t ∧ yield t = w ∧
   cfree pr t.
```

Safety

```
Definition safe {g} (pr : drules g) : Prop :=
  forall w, language w → dlanguage pr w.
```

Completeness

```
Definition complete {g} (pr : drules g) : Prop :=
  forall t1 t2,
    yield t1 = yield t2 →
    cfree pr t1 →
    cfree pr t2 →
    t1 = t2.
```

Safe Disambiguation Rules

```
Definition safe_pr {g} (pr : drules g) : Prop :=
  forall p1 p2,
    (pr.(prio) p1 p2 ∨ (pr.(left_a)) p1 p2) →
    (pr.(prio) p2 p1 ∨ (pr.(right_a)) p2 p1) →
    False.
```

Complete Disambiguation Rules

```
Record complete_pr {g} (pr : drules g) := mkComplete_pr {
  complete_1 : forall o1 o2,
    pr.(prio) o1 o2 ∨ pr.(left_a) o1 o2 ∨
    pr.(prio) o2 o1 ∨ pr.(right_a) o2 o1;

  complete_2 : forall o1 o2 o3,
    pr.(prio) o1 o2 → pr.(prio) o2 o3 → pr.(prio) o1 o3;

  complete_3 : forall o1 o2 o3,
    pr.(prio) o1 o2 → pr.(prio) o2 o3 → pr.(prio) o1 o3;
  complete_4 : forall o1 o2 o3,
    pr.(prio) o1 o2 → pr.(left_a) o2 o3 → pr.(prio) o1 o3;
  complete_5 : forall o1 o2 o3,
    pr.(prio) o1 o2 → pr.(right_a) o2 o3 → pr.(prio) o1 o3;
  complete_6 : forall o1 o2 o3,
    pr.(left_a) o1 o2 → pr.(prio) o2 o3 → pr.(prio) o1 o3;
  complete_7 : forall o1 o2 o3,
    pr.(right_a) o1 o2 → pr.(prio) o2 o3 → pr.(prio) o1 o3;

  complete_8 : forall o1 o2 o3,
    pr.(left_a) o1 o2 → pr.(left_a) o2 o3 → pr.(left_a) o1 o3;
  complete_9 : forall o1 o2 o3,
    pr.(right_a) o1 o2 → pr.(right_a) o2 o3 → pr.(right_a) o1 o3;

  complete_10 : forall o1 o2 o3,
    pr.(left_a) o1 o2 → pr.(right_a) o2 o3 → False;
  complete_11 : forall o1 o2 o3,
    pr.(right_a) o1 o2 → pr.(left_a) o2 o3 → False;
}.
```

Safety Theorem

```
Theorem safety {g} (pr : drules g) :
  safe_pr pr →
  safe pr.
Proof.
  (* proof is here *)
Qed.
```

Completeness Theorem

```
Theorem completeness {g} (pr : drules g) :
  complete_pr pr →
  complete pr.
Proof.
  (* proof is here *)
Qed.
```

Appendix C

Repair Function in Full

Examining the full *repair* function from Chapter 7 may be annoying as it is interleaved with explanations and justifications. This appendix shows it in full for IPP grammars, together with the corresponding Coq implementation.

C.1 As Mathematical Notation

$repair : T_G \to T_G$
$repair(t) =$

$$\begin{cases} l, & \text{if } t = l \in L \\ repair_{in}(repair(t_1), o, repair(t_2)), & \text{if } t = [t_1\ o\ t_2], o \in O_{in} \\ insert_{pre}(o, repair(t_1)), & \text{if } t = [o\ t_1], o \in O_{pre} \\ insert_{post}(repair(t_1), o), & \text{if } t = [t_1\ o], o \in O_{post} \end{cases}$$

$repair_{in} : (T_G \times O_{in} \times T_G) \to T_G$
$repair_{in}(t_1, o, t_2) =$

$$\begin{cases} insert_{in}(l_1, o, t_2), & \text{if } t_1 = l_1 \in L \\ repair_{in}(t_{11}, o_1, repair_{in}(t_{12}, o, t_2)), & \text{if } t_1 = [t_{11}\ o_1\ t_{12}], o_1 \in O_{in} \\ insert_{pre}(o_1, repair_{in}(t_{11}, o, t_2)), & \text{if } t_1 = [o_1\ t_{11}], o_1 \in O_{pre} \\ insert_{in}([t_{11}\ o_1], o, t_2), & \text{if } t_1 = [t_{11}\ o_1], o_1 \in O_{post} \end{cases}$$

$insert_{in} : (T_G \times O_{in} \times T_G) \to T_G$
$insert_{in}(t_1, o, t_2) =$

$$\begin{cases} [t_1\ o\ l_2], & \text{if } t_2 = l_2 \in L \\ [t_1\ o\ [t_{21}\ o_2\ t_{22}]], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{in}(t_1, o, t_{21})\ o_2\ t_{22}], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } R_Q(o, t_2) \\ [t_1\ o\ [o_2\ t_{21}]] & \text{if } t_2 = [o_2\ t_{21}], o_2 \in O_{pre} \\ [t_1\ o\ [t_{21}\ o_2]], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{in}(t_1, o, t_{21})\ o_2], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } R_Q(o, t_2) \end{cases}$$

$insert_{pre} : (O_{pre} \times T_G) \rightarrow T_G$
$insert_{pre}(o, t_2) =$

$$\begin{cases} [o\ l_2], & \text{if } t_2 = l_2 \in L \\ [o\ [t_{21}\ o_2\ t_{22}]], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{pre}(o, t_{21})\ o_2\ t_{22}], & \text{if } t_2 = [t_{21}\ o_2\ t_{22}], o_2 \in O_{in}, \text{ and } R_Q(o, t_2) \\ [o\ [o_2\ t_{21}]], & \text{if } t_2 = [o_2\ t_{21}], o_2 \in O_{pre} \\ [o\ [t_{21}\ o_2]], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } \neg R_Q(o, t_2) \\ [insert_{pre}(o, t_{21})\ o_2], & \text{if } t_2 = [t_{21}\ o_2], o_2 \in O_{post}, \text{ and } R_Q(o, t_2) \end{cases}$$

$insert_{post} : (T_G \times O_{post}) \rightarrow T_G$
$insert_{post}(t_1, o) =$

$$\begin{cases} [l_1\ o], & \text{if } t_1 = l_1 \in L \\ [[t_{11}\ o_1\ t_{12}]\ o], & \text{if } t_1 = [t_{11}\ o_1\ t_{12}], o_1 \in O_{in}, \text{ and } \neg L_Q(o, t_1) \\ [t_{11}\ o_1\ insert_{post}(t_{12}, o)], & \text{if } t_1 = [t_{11}\ o_1\ t_{12}], o_1 \in O_{in}, \text{ and } L_Q(o, t_1) \\ [[o_1\ t_{12}]\ o], & \text{if } t_1 = [o_1\ t_{12}], o_1 \in O_{pre}, \text{ and } \neg L_Q(o, t_1) \\ [o_1\ insert_{post}(t_{12}, o)], & \text{if } t_1 = [o_1\ t_{12}], o_1 \in O_{pre}, \text{ and } L_Q(o, t_1) \\ [[t_{11}\ o_1]\ o], & \text{if } t_1 = [t_{11}\ o_1], o_1 \in O_{post} \end{cases}$$

$$\frac{o \in O_{in}, o_2 \in O_{in}, q = CR_{in,in}(o, o_2) \in Q^i, M([t_{21}\ o_2\ t_{22}], q)}{R_Q(o, [t_{21}\ o_2\ t_{22}])}$$

$$\frac{o \in O_{pre}, o_2 \in O_{in}, q = CR_{pre,in}(o, o_2) \in Q^i, M([t_{21}\ o_2\ t_{22}], q)}{R_Q(o, [t_{21}\ o_2\ t_{22}])}$$

$$\frac{o \in O_{in}, o_2 \in O_{post}, q = CR_{in,post}(o, o_2) \in Q^{lm}, D^{lm}([t_{21}\ o_2], q)}{R_Q(o, [t_{21}\ o_2])}$$

$$\frac{o \in O_{pre}, o_2 \in O_{post}, q = CR_{pre,post}(o, o_2) \in Q^{rm}, D^{rm}([t_{21}\ o_2], q)}{R_Q(o, [t_{21}\ o_2])}$$

$$\frac{o \in O_{in}, o_1 \in O_{in}, q = CL_{in,in}(o, o_2) \in Q^i, M([t_{11}\ o_1\ t_{12}], q)}{L_Q(o, [t_{11}\ o_1\ t_{12}])}$$

$$\frac{o \in O_{post}, o_1 \in O_{in}, q = CL_{post,in}(o, o_1) \in Q^i, M([t_{11}\ o_1\ t_{12}], q)}{L_Q(o, [t_{11}\ o_1\ t_{12}])}$$

$$\frac{o \in O_{in}, o_1 \in O_{pre}, q = CL_{in,pre}(o, o_1) \in Q^{rm}, D^{rm}([o_1\ t_{12}], q)}{L_Q(o, [o_1\ t_{12}])}$$

$$\frac{o \in O_{post}, o_2 \in O_{pre}, q = CL_{post,pre}(o, o_1) \in Q^{lm}, D^{lm}([o_1\ t_{12}], q)}{L_Q(o, [o_1\ t_{12}])}$$

C.2 As Coq Implementation

```
Fixpoint insert_in {g} (pr : drules g) t1 o t2 : parse_tree g :=
 match t2 with
 | InfixNode t21 o2 t22 ⇒
    if is_i_conflict_pattern pr (CR_infix_infix o o2)
    then InfixNode (insert_in pr t1 o t21) o2 t22
    else if has_infix_lm_conflicts pr o t2
    then InfixNode (insert_in pr t1 o t21) o2 t22
    else InfixNode t1 o t2
 | PostfixNode t21 o2 ⇒
    if has_infix_lm_conflicts pr o t2
    then PostfixNode (insert_in pr t1 o t21) o2
    else InfixNode t1 o t2
 | _ ⇒ InfixNode t1 o t2
 end.

Fixpoint insert_pre {g} (pr : drules g) o t2 : parse_tree g :=
 match t2 with
 | InfixNode t21 o2 t22 ⇒
    if is_i_conflict_pattern pr (CR_prefix_infix o o2)
    then InfixNode (insert_pre pr o t21) o2 t22
    else if has_prefix_lm_conflicts pr o t2
    then InfixNode (insert_pre pr o t21) o2 t22
    else PrefixNode o t2
 | PostfixNode t21 o2 ⇒
    if has_prefix_lm_conflicts pr o t2
    then PostfixNode (insert_pre pr o t21) o2
    else PrefixNode o t2
 | _ ⇒ PrefixNode o t2
 end.

Fixpoint insert_post {g} (pr : drules g) t1 o : parse_tree g :=
 match t1 with
 | InfixNode t11 o1 t12 ⇒
    if is_i_conflict_pattern pr (CL_postfix_infix o o1)
    then InfixNode t11 o1 (insert_post pr t12 o)
    else if has_postfix_rm_conflicts pr t1 o
    then InfixNode t11 o1 (insert_post pr t12 o)
    else PostfixNode t1 o
 | PrefixNode o1 t12 ⇒
    if has_postfix_rm_conflicts pr t1 o
    then PrefixNode o1 (insert_post pr t12 o)
    else PostfixNode t1 o
 | _ ⇒ PostfixNode t1 o
 end.

Fixpoint repair_in {g} (pr : drules g) t1 o t2 : parse_tree g :=
 match t1 with
 | InfixNode t11 o1 t12 ⇒ repair_in pr t11 o1 (repair_in pr t12 o t2)
 | PrefixNode o1 t12 ⇒ insert_pre pr o1 (repair_in pr t12 o t2)
 | _ ⇒ insert_in pr t1 o t2
 end.

Fixpoint repair {g} (pr : drules g) t : parse_tree g :=
 match t with
 | AtomicNode l ⇒ AtomicNode l
 | InfixNode t1 o t2 ⇒ repair_in pr (repair pr t1) o (repair pr t2)
 | PrefixNode o t2 ⇒ insert_pre pr o (repair pr t2)
 | PostfixNode t1 o ⇒ insert_post pr (repair pr t1) o
 end.
```