MASTER

# Parse forest disambiguation

van der Sanden, L.J.

*Award date:*
2014

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

# Parse Forest Disambiguation

*Master's Thesis*

Bram van der Sanden

Supervisor:
Mark van den Brand

Eindhoven, August 29, 2014

# Abstract

Context-free grammars are the most widely used formalism to express the concrete syntax of general-purpose and domain-specific languages. Generalized parsers are able to handle any context-free grammar, which allows grammar engineers to write grammars in a natural way. A consequence of supporting the whole class of context-free grammars is that also ambiguous grammars are supported. This means that parsing an input sentence may lead to multiple derivations rather than a unique derivation. In generalized parsing, the set of all parse trees of a given input sentence is embedded in a parse forest. By using disambiguation rules and disambiguation filters, undesired derivations can be removed.

In this thesis we will define a set of parser-technology independent parse forest filters, that allows the removal of undesired parse trees from a shared packed parse forest. These filters remove all parse trees from a parse forest that contain some construct, such as an invalid path. Given declarative disambiguation rules expressing the associativity, and precedence of operators, we can create filters that specify the constructs that should be removed. We will describe a new filter based on precede and follow restrictions, that allows the disambiguation of expression grammars containing mixfix operators with respect to associativities and precedences of operators. As a case study we will look into the disambiguation of the mCRL2 language, where also other disambiguation filters like keyword restriction and prefer filtering are used.

Often disambiguation filtering can be partially applied on parse time, avoiding the creation of undesired parse trees in the first place. We will show how our disambiguation filters for expression grammars can be incorporated into the GLL algorithm. This integration leads to a substantial decrease in the total time needed for parsing and disambiguation of input sentences containing many ambiguities.

# Contents

# Preface

This thesis is the result of my gradation project at Eindhoven University at Technology. Before starting this project, I have done a honors project and seminar on GLL parsing together with Bram Cappers and Josh Mengerink. Together, we have implemented an object-oriented variant of the GLL parsing algorithm extended with a plug-in architecture for supporting error handling. This project also sparked my interest to continue research in the field of software language engineering.

I would like to thank Mark van den Brand for being my supervisor in both projects. During my graduation project, he gave me a lot of freedom to explore the topic of parse forest disambiguation, and provided me with good advice. Furthermore, he suggested the topic of disambiguation which turned out to be very interesting.

During my studies I have met a lot of wonderful people at the university, which made the time I spend there very enjoyable. I would like to thank the friends I made over the past five years for making the courses and studying fun. In particular I would like to thank Bram Cappers and Josh Mengerink, with whom I shared many great moments during my master studies. Furthermore I would like to thank my family for supporting me during my entire studies.

I would like to thank Elizabeth Scott for providing me with thorough feedback and suggestions on ideas, and parts of this thesis. Also thanks to Sjoerd Cranen and Wieger Wesselink for providing me with the mCRL2 grammar, and describing the issues that arise while parsing mCRL2 files. Finally, I would like to thank Adrian Johnstone, Elizabeth Scott, and Tim Willemse for serving on my examination committee.

Bram van der Sanden
August 12, 2014
Eindhoven

# Chapter 1

# Introduction

Context-free grammars are prevalent in software engineering to define a wide range of languages. From a context-free grammar, parsers that derive the structure of a sentence in the language can be automatically generated. Classical deterministic parser generators in the line of YACC [30] are able to deal with almost all modern programming languages and are used for parsing languages like RUBY, OCAML, and PHP. The output of a deterministic parser is a parse tree that reflects the derivation of the sentence. Over the years it has been shown that these kind of parser generators are not able to solve modern challenges in software engineering [6, 9, 15]. These challenges include issues related to modularity of grammars, domain-specific languages, and reverse engineering.

For these and other reasons, there has been a lot of interest in generalized parsing techniques [35, 46, 51, 52] in the last decade. Parser generators that are based on generalized parsing algorithms have several advantages. The main advantage is that they can generate a working parser for *any* context-free grammar automatically. When using a subclass of all context-free languages, some user grammar may not be supported and rewriting may be necessary to fit it into the subclass. Supporting the full class of context-free grammars means that the grammar can be written in a natural way, which ensures better maintainability and comprehensibility. Furthermore the full class of context-free grammars is closed under composition. That is, when two context-free grammars are combined, the result is again a context-free grammar. This allows for modular grammars and reuse of grammars. One can think of examples like template languages where programming code is added into the template [5], or integrating database query languages into general purpose programming languages.

One of the consequences of supporting the whole class of context-free grammars is that also *ambiguous grammars* are supported. In an ambiguous grammar there are sentences in the language that can be derived in multiple ways. Each derivation could potentially reflect a different interpretation of the sentence, depending on the semantics of the language. Detecting whether a context-free grammar is ambiguous is in general undecidable [19]. There has, however, been a substantial amount of research in the area of ambiguity detection for specific types of ambiguities [8,11,18,42]. These techniques can detect various types of ambiguity by looking at the structure of the grammar and generating a set of input sentences to see whether they are ambiguous.

In most cases one is interested in a specific derivation rather than all possible derivations. By means of *disambiguation*, the intended derivation is selected from the set of of possible derivations. Using a set of declarative *disambiguation rules* one can specify properties like the priority and associativity of operators. Based on these rules, disambiguation methods can be designed that enforce these rules, removing undesired derivations.

Since generalized parsers generate all possible derivations given a grammar and a sentence, the output of the parse can be a set of trees rather than a single parse tree. In most generalized parsing algorithms, a *parse forest* is used to represent the set of parse trees. In this data structure sharing is often used to allow a more compact storage of all trees.

In this thesis we will look at the disambiguation of *shared packed parse forests* (SPPFs), which can be generated by generalized algorithms like Earley [45], generalized LR (GLR) variants like RIGLR [45] and RNGLR [43], and generalized LL (GLL) [46]. As the name already indicates, sharing is used to reduce the required space for storing all parse trees. So rather than storing each tree separately, subtrees that occur in multiple trees are shared.

## 1.1 Motivation

Since the start of research in the area of parsing technology, there has been a lot of interest in the topic of ambiguity and disambiguation. Before the introduction of generalized parsing methods, ambiguities where most of the time avoided by rewriting the grammar or handled by changing the actions in shift-reduce based parsers. While these techniques reached their goal – the outcome was a single desired parse tree – the side effects where complex grammars, and disambiguation semantics directly defined based on steps taken in the specific parser algorithm.

With the introduction of generalized parsing algorithms it became possible to generate all parse trees, and clearly define the desired and undesired trees based on the notion of disambiguation filters. In this thesis we will look at SPPF filters to remove parse trees from the SPPF containing some construct, such as paths between two nodes that each correspond to some production. In this way, we can remove all parse trees having this construct at once, rather than first expanding the SPPF into a set of all parse trees. These SPPF filters are used in the specification of disambiguation filters aimed at resolving specific types of ambiguities. We will look at disambiguation filters that resolve ambiguities in expression grammars, mostly related to the associativity and precedence of operators. The ambiguities are resolved by finding undesired constructs, and removing them using our SPPF filters.

This approach has a number of advantages. First of all, we can precisely define which parse trees we want to remove. Exactly those parse trees can be removed using the SPPF filters, while keeping valid parse trees in the SPPF. Because the filters are defined on trees, they are parser-independent. Based on the semantics of the filter, we can possibly implement them in the generated parsers, to avoid generating the undesired trees in the first place. In this situation, care has to be taken that valid trees will still be present in the output SPPF.

## 1.2 Research Questions

The objective of this thesis is to look at disambiguation in shared packed parse forests that are generated by generalized parsing algorithms such as GLL and GLR. We will investigate the problem of ambiguity in expression grammars with operators of arbitrary fixity (infix, prefix, postfix, and closed operators), and higher arity (e.g. unary, binary), and use parse forest disambiguation as a method for resolving these ambiguities. The main difficulty in filtering SPPFs is the fact that sharing is used. Because of this sharing, we must always be very cautious in removing certain nodes or edges to avoid removing valid parse trees.

**Research Question 1** How can we remove undesired parse trees from an SPPF while keeping desired parse trees?

Given algorithms to remove undesired parse trees from an SPPF, we would like to look at ambiguity resolution in expression grammars with unary and binary operators. These kinds of grammars are part of practically any modern programming language. What kind of ambiguities occur, and how can we specify and implement parse forest filters that resolve these ambiguities?

**Research Question 2** What kind of ambiguities occur in expression grammars and how to implement parse forest filters to resolve these ambiguities?

Given the parse forest filters for expression grammars, we would like to see whether these filters can be adapted to apply to the wider class of mixfix expressions having operators of arbitrary fixity.

**Research Question 3** Are the parse forest filters able to resolve all ambiguities occurring in mixfix grammars?

## 1.3 Scope

In this thesis we will look at ambiguity resolving in expression grammars and mixfix grammars. These kind of grammars are often part of programming languages, and are typically used as a case study to demonstrate disambiguation filters. Another class where ambiguity resolving plays a big role is the class of natural language grammars. In the research area of natural language processing (NLP), generalized parsing algorithms are used for parsing natural languages. Often statistics are used to determine the most likely derivation, for instance in the Stanford Parser [32]. We will not look into natural language grammars, but the algorithms given for removing parse trees in the SPPF are grammar independent and can hence also be applied in this research area.

## 1.4 Outline

The remainder of this thesis is structured in the following way. In Chapter 2 we formally introduce the notation that we will use throughout this thesis, and describe the structure of shared-packed parse forests. In Chapter 3, we focus on the topic of disambiguation and introduce the notion of disambiguation filters.

Chapter 4 tries to answer our first research question, and looks into removing parse trees from an SPPF. Throughout this chapter we will give examples of the usefulness and applicability of our removal filters.

Chapters 5 and 6 look at the disambiguation of expression grammars, and mixfix grammars respectively. The disambiguation is done by means of a set of disambiguation filters. As a case study of a mixfix grammar language, we will look at the disambiguation of mCRL2 [20] in Chapter 6.

In Chapter 7, we describe the experiments that have been run to test our filters on mCRL2 input files. We also show the performance gain, when filters are integrated into the parser. The implementation of our GLL parser generator and filters that have been used for running the experiments, are described in Chapter 8.

Finally, Chapter 9 concludes this paper by summing up the contributions of this thesis and providing suggestions for future research.

# Chapter 2

# Preliminaries

In this thesis the following definitions are used. A *context-free grammar* (CFG) $\Gamma$ is defined as a quadruple $\langle T, N, P, S\rangle$, where $T$ is a set of terminal symbols, $N$ a set of nonterminal symbols, $S$ is the start symbol, and $P \subseteq N \times (T \cup N)^*$ a set of *productions*. We write $A ::= \alpha$ for a production $p = \langle A, \alpha\rangle \in P$, where $A$ is called the *head*, and $\alpha$ the *body* of the production. Productions with the same left hand sides are often composed into a single rule using the alternation symbol, $A ::= \alpha_1 \mid \ldots \mid \alpha_p$. We refer to strings $\alpha_i$ as the *alternates of A*. Productions are also sometimes called *grammar rules*.

A *sentential form* is a finite string in $(T \cup N)^*$, that can be derived from the start symbol in zero or more steps. A *sentence* is a sentential form without nonterminal symbols. The symbol $\varepsilon$ denotes the empty sentential form. We use lowercase greek characters $\alpha, \beta, \gamma, \ldots$ for variables over sentential forms. A *derivation step* has the form $\gamma A\beta \Rightarrow \gamma\alpha\beta$ where $\gamma, \beta \in (T \cup N)^*$ and $A ::= \alpha$ is a production. A *derivation* of $\tau$ from $\sigma$ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \beta_n = \tau$, also written as $\sigma \overset{*}{\Rightarrow} \tau$, or, if $n > 0, \sigma \overset{+}{\Rightarrow} \tau$. A *full derivation* is a sequence of production rule applications that starts with the start symbol and ends with a sentence. A derivation is *left-most* if at each step the left-most nonterminal is replaced. The *language* $L(\Gamma)$ of a grammar $\Gamma$ is the set of all sentences derivable from $S$, i.e. the set $u \in T^*$ such that $S \overset{*}{\Rightarrow} u$. A grammar is *ambiguous* if there is a sentence *u* for which there is more than one left-most derivation $S \overset{*}{\Rightarrow} u$.

A nonterminal $A$ is *nullable* if $A \overset{*}{\Rightarrow} \varepsilon$. If there is a $\gamma \in (T \cup N)^*$ such that $A \overset{+}{\Rightarrow} A\gamma$ then $A$ is said to be *left recursive*, and if $A \overset{*}{\Rightarrow} \beta A\gamma$ where $\beta \overset{+}{\Rightarrow} \varepsilon$ we say $A$ has *hidden* left recursion.

A *grammar slot* defines the position immediately before or after any symbol in any alternate. A *grammar pointer* points to a grammar slot in the grammar. Grammar slots are written in the same fashion as LR(0) items, so $S ::= \alpha \cdot \beta$ is the position in the grammar immediately after the last symbol in $\alpha$.

Define $\textsc{first}_T(A) = \{t \in T \mid \exists\alpha(A \overset{*}{\Rightarrow} t\alpha)\}$, and $\textsc{follow}_T(A) = \{t \in$

$T \mid \exists\alpha,\beta(S \overset{*}{\Rightarrow} \alpha A t \beta)\}$. If $A$ is nullable, define $\textsc{first}(A) = \textsc{first}_T(A) \cup \{\varepsilon\}$, otherwise $\textsc{first}(A) = \textsc{first}_T(A)$. If $S \overset{*}{\Rightarrow} \alpha A$ define $\textsc{follow}(A) = \textsc{follow}_T(A) \cup \{\$\}$, otherwise $\textsc{follow}(A) = \textsc{follow}_T(A)$.

In a *bracketed derivation* [24], each application of a rule is recorded by a pair of brackets. For example $S \Rightarrow (\alpha E \beta) \Rightarrow (\alpha(E + E)\beta) \Rightarrow (\alpha(E + (E * E))\beta)$. Brackets are (implicitly) indexed with their corresponding rule. In our example $S$ derives $E$, which in turn derives $E + E$ and so forth. Bracketed derivations will only be used for epsilon-free grammars.

A *parse tree* is an ordered finite tree representation of a full derivation of a specific sentence. The root node is labeled with the start symbol, the interior nodes are labeled with nonterminals and the leaf nodes are labeled with elements of $T \cup \{\varepsilon\}$. The children of an interior node $n$ labeled $A$ are ordered and labeled with the symbols (in order) of some alternate $\alpha_i$ of $A$. We define $head(n) = A$, and $prod(n) = \langle A, \alpha_i \rangle$. Define $\mathcal{T}(t)$ to be the subtree rooted at $t$. When we say $n \in \mathcal{T}(t)$, we mean that $n$ is contained in the set of nodes of the tree with root $t$.

The *extent* of a node with symbol $X$ denotes the substring on which the symbol has been matched. For every $n \in t$ we define $extent(n) = \langle i_n, j_n \rangle$ with $1 \leq i_n \leq j_n \leq m$, where the input string has length $|m|$. Variable $i_n$ refers to the *left* extent of the node, and defines the starting position in the input string. Variable $j_n$ refers to the *right* extent of the node, and defines that the symbol is matched up to (so not including) position $j_n$.

A *parse forest* is a set of parse trees, extended with some construct to denote ambiguities. A parse forest represents full derivations of a *single* sentence. In this thesis we will use the term *parse forest* to denote a shared-packed parse forest, that is used for instance in the generalized LL parsing algorithm [46]. The shared-packed parse forest is described in Section 2.2. The *yield* of a tree $t$ is the string containing all leaves from the left to right. The function can be lifted to a set of parse trees by $yield(\Phi) = \{yield(t) \mid t \in \Phi\}$. If there are two or more parse trees with the same yield in grammar $\Gamma$, the grammar is called ambiguous.

A *recognizer* for $\Gamma$ is a terminating function that takes any sentence $\alpha$ as input and returns true if and only if $S \overset{*}{\Rightarrow} \alpha$. A *parser* for $\Gamma$ is a terminating function that takes any sentence $\alpha$ as input and returns a *parse forest* for $\alpha$. *Scannerless parsing* is a term used to indicate parsing without a separate lexical analysis phase. In scannerless parsing, a syntax definition is a context-free grammar with characters as terminals.

## 2.1 Generalized LL Parsing

There are two main flavors of parsing algorithms: bottom-up parsers, and top-down parsers. Bottom-up parsers start with the target string and try to derive the start symbol. This type of parsers use a parse table and shift/reduce actions. The parse table is typically constructed automatically and makes the bottom-up parsers generated hard to read. Top-down parsers begin at the start symbol of the grammar and try to apply production rules to arrive at the input string. This kind of parsers follow the structure of the grammar, and is therefore easier to debug and understand. One of the most recent top-down algorithms is the generalized LL (GLL) [44,46] parsing algorithm. This algorithm generates parsers that can deal with ambiguity and left-recursion, and runs in worst case cubic time. Because it is a top-down parser, the generated parser code is easy to read, and debug and can be easily generated by means of a code template. It is even feasible to do the implementation of the parser for a given grammar by hand.

The GLL algorithm is based on the idea of traversing the grammar Γ, using an input string $u$. For the traversal we use a pointer into the grammar (a grammar slot), and another pointer into the input string. Multiple traversals are handled using descriptors. A descriptor $(L, s, i, w)$ is processed by restarting a traversal with the grammar pointer at grammar slot $L$, $s$ as the stack top and the input pointer at position $i$. Pointer $w$ refers to a SPPF node in the SPPF that is being generated. We need a reference to the stack top to know where we have to continue the derivation after deriving the current symbol. There are potentially infinitely many descriptors for a given input string, because the number of stacks may be unbounded. The solution to this problem, introduced by Tomita [50], is to use a graph-structured stack (GSS). Lower portions of stacks that are identical are merged, and stack tops are recombined when possible. Cycles are added to the GSS in order to deal with left-recursion. Because of this sharing, a descriptor can record several partial traversals when they restart at the same grammar and at the same input position. When the set of pending descriptors is empty, all possible traversals have been explored, and all valid derivations have been determined.

## 2.2 Shared Packed Parse Forest (SPPF)

A GLL parser uses a *shared packed parse forest* (SPPF) to represent the complete set of derivation trees for a given input string. An SPPF is a bipartite graph where sharing is used to reduce the total space required to represent all derivation trees. Nodes which have the same subtree are shared, and nodes are combined which correspond to different derivations of the same substring from the same nonterminal.

There are three types of nodes in an SPPF associated with a GLL parser: *symbol nodes*, *packed nodes*, and *intermediate nodes*. In the visualizations symbol nodes are shown as rectangles with rounded corners, packed nodes are shown as circles, or ovals when the label is visualized, and intermediate nodes are shown as rectangles.

Symbol nodes have labels of the form $(x, j, i)$ where $x$ is a terminal, nonterminal, or $\varepsilon$ (i.e. $x \in T \cup N \cup \{\varepsilon\}$), and $0 \leq j \leq i \leq m$ with $m$ being the length of the input sentence. The tuple $(j, i)$ is called the *extent*, and denotes that the symbol $x$ has been matched on the substring from position $j$ up to position $i$. Here $j$ is called the *left extent*, and $i$ is called the *right extent*.

Packed nodes have labels of the form $(t, k)$, where $0 \leq k \leq m$. Here $k$ is called the *pivot*, and $t$ is of the form $X ::= \alpha \cdot \beta$. The value of $k$ represents that the first symbol of $\beta$ starts at position $k$ of the input string. Packed nodes are used to represent multiple derivation trees. When multiple derivations are possible with the same extent, starting from the same nonterminal symbol node, a separate packed node is added to the symbol node for each derivation. We will sometimes refer to these packed nodes as *alternatives*.

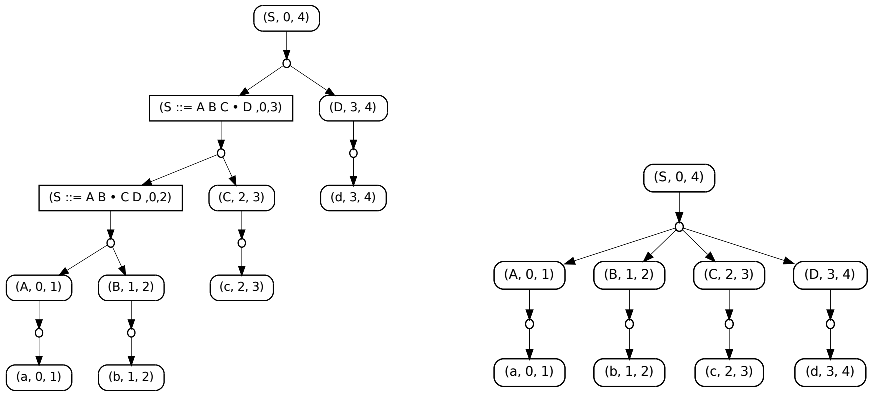
Intermediate nodes are used to binarize the SPPF. They are introduced from the left, and group the children of packed nodes in pairs from the left. The binarization ensures that the size of the SPPF is worst-case cubic in the size of the input sentence. The fact that the SPPF is binarized does not mean that each node in the SPPF has at most two children. A symbol node or intermediate node can still have as many packed node children as there are ambiguities starting from it. Intermediate nodes have labels of the form $(t, j, i)$ where $t$ is a grammar slot, and $(j, i)$ is the extent. There are no intermediate nodes of the shape $(A ::= \alpha \cdot, j, i)$, where the grammar pointer of the grammar slot is at the end of the alternate. These grammar slots are present in the form of symbol nodes.

Consider Grammar $\Gamma_{2.1}$. The SPPF resulting from this grammar and sentence $abcd$ is shown in Figure 2.1a. Suppose that the intermediate nodes had not been added to the SPPF. Then the nonterminal symbol nodes for $A$, $B$, $C$, and $D$ would have been attached to the nonterminal symbol node $S$ as is shown in Figure 2.1b. This example shows how intermediate nodes ensure that the tree is binarized.

$$S ::= ABCD \quad A ::= a \quad B ::= b \quad C ::= c \quad D ::= d \tag{$\Gamma_{2.1}$}$$

### Cycles

Grammars that contain cycles can generate sentences which have infinitely many derivation trees. A context-free grammar is cyclic if there exists a nonterminal $A \in N$ and a derivation $A \overset{+}{\Rightarrow} A$. Note that a cyclic context-free grammar implies that the context-free grammar is left-recursive, but the converse does not hold. The derivation trees for a cyclic grammar are represented in the finite SPPF by introducing cycles in the graph.



(a) SPPF with intermediate nodes (b) SPPF without intermediate nodes

Figure 2.1: SPPF for grammar $S ::= ABCD \quad A ::= a \quad B ::= b \quad C ::= c \quad D ::= d$ and sentence $abcd$.

Consider as an example Grammar $\Gamma_{2.2}$. All derivation trees of sentence $a$ generated by this grammar can be represented by the SPPF shown in Figure 2.2.

$$S ::= SS \mid a \mid \varepsilon. \qquad (\Gamma_{2.2})$$

A particular derivation tree can be retrieved from the SPPF by unwinding the cycle the required number of times, and selecting one packed node below each symbol and intermediate node.

### Ambiguities

A parse forest is *ambiguous* if and only if it contains at least one ambiguity. An ambiguity arises when a symbol node or intermediate node has at least two packed nodes as its children. We will call such a node *ambiguous*. Consider for instance Grammar $\Gamma_{2.3}$ and input sentence $1 + 1 + 1$. This gives the SPPF as shown in Figure 2.3.

$$E ::= E + E \mid 1 \qquad (\Gamma_{2.3})$$

In this SPPF, symbol node $(E, 0, 5)$ has two packed nodes as children. This means that there are at least two different parse trees starting at this node, the parse trees representing derivations $(E+(E+E))$ and $((E+E)+E)$ respectively.

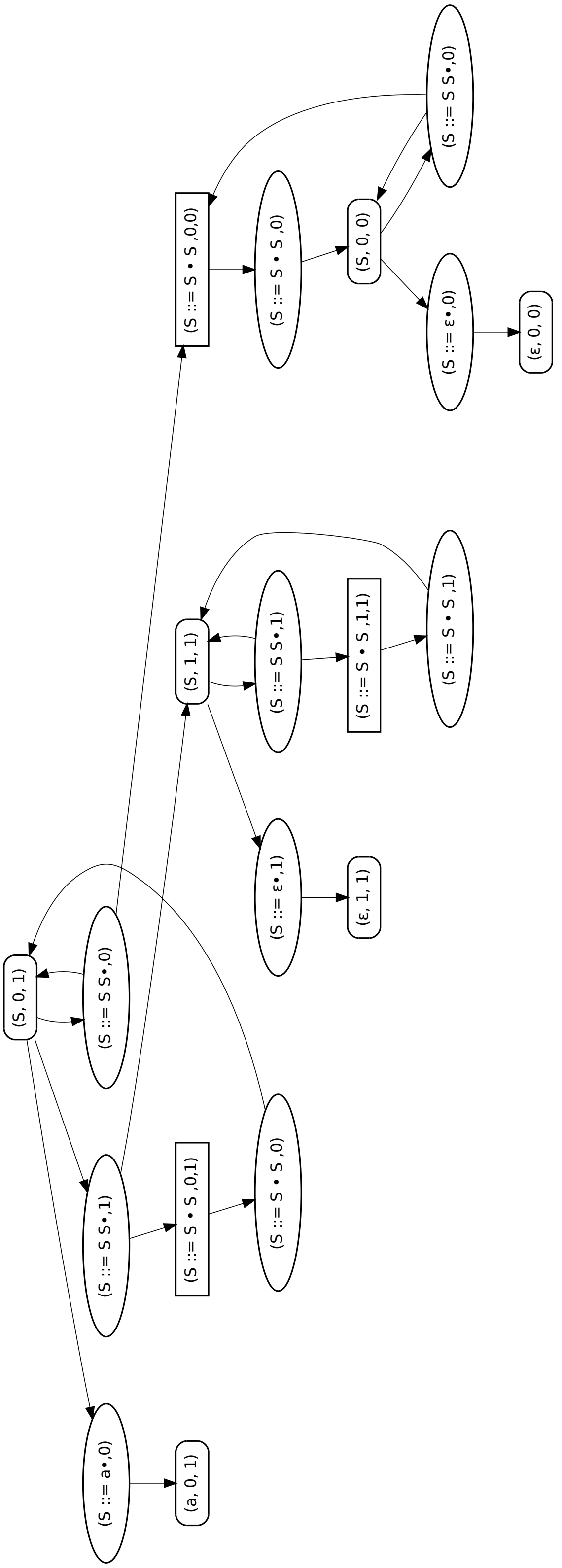


Figure 2.2: SPPF for grammar $S ::= SS \mid a \mid \varepsilon$ and sentence $a$.

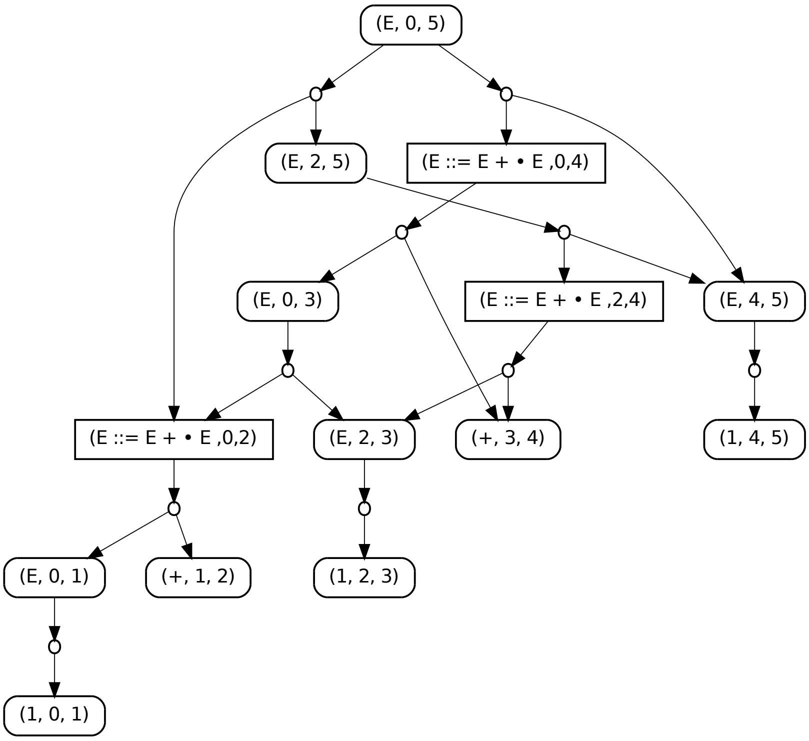


Figure 2.3: SPPF for grammar $E ::= E + E \mid 1$ and sentence $1 + 1 + 1$.

All parse trees can be retrieved from the SPPF by unfolding it. When a symbol node has multiple packed nodes as child, one of these has to be selected. The intermediate nodes and packed nodes can be removed in the second step. When removing a node, its children are attached to the parent of the node. It is important that the order of the children stays the same.

### Structural properties of an SPPF

There are various structural properties that are useful when reasoning about SPPFs in general. At first note that each symbol node $(E, j, i)$ with $E \in T \cup N \cup \{\varepsilon\}$ is unique, so an SPPF does not contain two symbol nodes $(A, k, l)$ and $(B, m, n)$ with $A = B, k = m$, and $l = n$.

Terminal symbol nodes have no children. These nodes represent the leaves of the parse forest. Nonterminal symbol nodes $(A, j, i)$ have packed node children of the form $(A ::= \gamma\cdot, k)$ with $j \leq k \leq i$, and the number of children is not limited to two.

Intermediate nodes $(t, j, i)$ have packed node children with labels of the form $(t, k)$, where $j \leq k \leq i$.

Packed nodes $(t, k)$ have one or two children. The right child is a symbol node $(x, k, i)$ and the left child (if it exists) is a symbol or intermediate node with label $(s, j, k)$, where $j \leq k \leq i$. Packed nodes have always exactly one parent which is a symbol node or intermediate node.

It is useful to observe that the SPPF is a bipartite graph, with on the one hand the set of intermediate and symbol nodes and on the other hand the set of packed nodes. Therefore edges always go from a node of the first type to a node of the second type, or the other way round. As a consequence, cyles in the SPPF are always of even length.

# Chapter 3

# Disambiguation

When a given input sentence can be parsed in various ways, the resulting parse forest will contain multiple parse trees. Each of these parse trees has a different structure, corresponding to a different derivation. In order to choose the intended parse tree when multiple parse trees are available, a set of disambiguation rules is given next to the context-free grammar. These disambiguation rules can be used to deal with ambiguities before, after, and during parsing. Figure 3.1 shows this architecture, which has been first introduced by Klint and Visser [33]. In this chapter we will look at some of the most used disambiguation rules. Section 3.2 describes how they are used for dealing with ambiguities during various moments of parsing. Section 3.3 introduces the notion of filters to specify which parse trees are undesired.

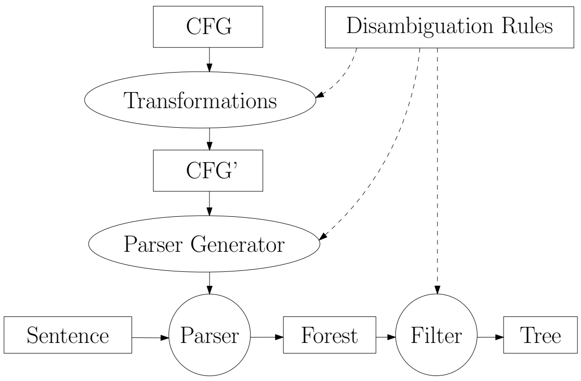


Figure 3.1: Use of disambiguation rules at different phases.

## 3.1 Disambiguation Rules

Already in 1975, Aho and Johnston [4] noted that various syntactic constructs in programming languages can be specified more naturally and concisely using an ambiguous grammar rather than an equivalent unambiguous grammar. In their work, the ambiguous grammar is accompanied by a set of *disambiguation rules* that are used to resolve *parsing action conflicts*. Using these disambiguation rules for instance operator precedence, and associativity can be handled, as well as the well-known dangling else problem. This approach, of using a set of rules aimed at resolving parsing action conflicts, has been implemented for the first time in the still widely used YACC parser generator [30].

The problem with this approach is that the disambiguation rules are guided by parser algorithms [10]. In order to understand the semantics of this kind of disambiguation rule, one must understand the implementation of the parsing algorithm.

Ideally one would have a set of *declarative disambiguation rules* that have a natural form that is easy to understand by language engineers. These kinds of rules can be implemented as a generic filtering mechanism that remove undesired parse trees from a set of parse trees [31,33]. Using explicit disambiguation rules makes it possible to apply them selectively, possibly using context-information that is not always available during parsing.

We will now list a number of often encountered disambiguation rules. Some of these rules are specific to scannerless parsers. The rules listed below have been implemented in the parser framework of SDF2 [53].

*Priority* rules specify the relative priorities between productions. For instance the production for addition may not be a direct child of the production for multiplication.

*Associativity* rules are used to express the associativity of an operator. For instance, since addition is left-associative, the production of the addition operator may not be a direct right child of itself. Besides left-associative, an operator can be right-associative or non-associative. Non-associativity means that no nesting is allowed at all.

*Follow restriction* rules are a simplification of adjacency restriction rules introduced by Salomon et al. [40, 41], and they are used to achieve longest match disambiguation. Follow restriction rules restrict the symbols that may follow a certain symbol. For instance, using a follow restriction one can specify that an identifier may not be followed by any character in the regular expression `[A-Za-z0-9]`.

*Reject production* rules are used to implement a reserved keywords mechanism. Using reject production rules one can for instance specify that a production may never derive certain keywords that are reserved in the language.

*Preference/Avoid* rules are used for selecting a preferred derivation or avoiding a derivation respectively, when multiple derivations are present. They can for example be used to disambiguate the dangling else construction. This is done by preferring the production with the if-then over the if-then-else production.

## 3.2 Moment of disambiguation

Disambiguation can be carried out before parsing by rewriting the grammar, during parser generation by modifying the parsers being generated, and as post-parse filtering on the parse forest. In general, deferring disambiguation is expensive but also makes it more generic.

**Grammar rewriting** Grammar transformations are often used to eliminate left recursion, or nullable productions (productions that derive $\varepsilon$ in one or more steps). The transformations are always language preserving, but the shape of the derivations will look different. Grammar rewriting is often used to avoid ambiguities arising from associativity and precedence by encoding these rules in the grammar directly. Consider as an example grammar $\Gamma_{3.1}$.

$$E ::= E + E \mid E * E \mid 1 \tag{$\Gamma_{3.1}$}$$

When parsing a sentence like $1 + 1 * 1$, the resulting parse forest will contain two derivations: $(E + (E * E))$ and $((E + E) * E)$. To add the higher precedence of the $*$ over the precedence of $+$ to the grammar, we layer the grammar. First we create a nonterminal for each level of precedence, then we isolate the corresponding part of the grammar, and finally we force the parser to recognize the higher precedence sub-expressions first. When performing these steps we get the modified grammar $\Gamma_{3.2}$.

$$\begin{aligned} S &::= E \\ E &::= E + T \mid T \\ T &::= T * F \mid F \\ F &::= 1 \end{aligned} \tag{$\Gamma_{3.2}$}$$

The rewritten grammar is no longer ambiguous, and correctly adheres to the higher precedence of $*$ over $+$. One can see that the rewritten grammar becomes very large and incomprehensible when many precedence levels are integrated into the grammar. There are also more rewrite steps needed to reach some of the terminal symbols. For instance, to obtain a number from $S$ we have the derivation $S \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow number$. Another problem arises when new operators need to be added to the modified grammar. If these operators have a precedence level that is not yet present in the grammar, we need to modify several rules in the grammar. In the original grammar we would just add alternates for the operators and leave the rest of the grammar untouched.

It is not always obvious how to eliminate ambiguities by rewriting the grammar. Furthermore there are ambiguities that cannot be removed by rewriting [29]. A grammar for which no equivalent unambiguous grammar exists is called an *inherently ambiguous grammar*.

**Parser modification** A technique often deployed in deterministic parser algorithms is modifying the parsers being generated. By disallowing the creation of unwanted derivations, ambiguities can be avoided. For instance in LR parsing, various shift/reduce conflict can be avoided by using precedence declarations. Based on the topmost terminal symbol, the lookahead, and the precedence declarations either a shift or a reduce is performed. This mechanism is used in tools like YACC [30].

The ambiguities that can be solved using these techniques are mostly ambiguities like associativity and priority ambiguities. Ambiguities that are more complex are often not supported by these standard mechanisms. Such ambiguities can only be resolved by creating the parser by hand, which is far from trivial and error-prone.

**Parse forest disambiguation** As mentioned before, generalized parsers produce a parse forest containing all the possible derivations of the input sentence according to the grammar. The ambiguities encountered are encoded in the parse forest itself. Using disambiguation, the ambiguities in the parse forest can be resolved by filtering undesired derivations. This technique can be used for any parsing algorithm that generates a parse forest. Filtering parse forests is for instance supported by the parser framework of SDF2 [27,53], based on the scannerless GLR parsing algorithm [52].

Parse forest filtering can be implemented in various ways. For instance, the paradigm of term rewriting has been used as a mechanism to filter ambiguities in parse forests. Afroozeh et al. [3] use a form of tree rewriting in order to resolve ambiguities in SPPFs. Disambiguation is performed bottom-up, starting at the leaves of the parse forest. Because sharing is used in an SPPF to reduce the required storage space, care has to be taken that no valid parse trees are removed by applying a filter.

Brand et al. [13] use term rewriting to filter ambiguities based on semantical information. They use a preprocessing step that transforms the parse forest into a single tree that represents the parse forest. This tree has a special *ambiguity constructor* to make the ambiguities in a parse forest visible to a term rewriting system. Since term rewriting is done on a tree rather than a forest, it is easier to prove that no valid parse trees are removed unintentionally. However, the consequence of using a single tree rather than a forest is that the amount of storage space required might become exponentially larger. By using data structures that enforce maximal sharing, such as *Annotated Terms* (ATerms) [12], this can be prevented.

In general, we can observe that filtering a parse forest consists of two steps. First, we need to find the invalid parse trees in the SPPF. Given these invalid parse trees, we need algorithms that remove all invalid parse trees from the SPPF, while preserving all valid parse trees. Of course, these two steps can be alternated to remove invalid parse trees as soon as they are found. In the remainder of this chapter we will describe some parse filters that have been introduced in the literature, and describe how they can be implemented. The next chapter will focus on removing invalid parse trees from an SPPF.

## 3.3 Disambiguation Filters

Given a set of disambiguation rules, we need a mechanism to specify which parse trees are desired and which ones are not desired. Klint and Visser [33] have proposed a framework of *filters* to describe and compare many disambiguation methods in a parser-independent way. Using a filter one can select the intended trees from the set of all parse trees being generated. In many cases a filter is defined in negative terms by specifying which trees are undesired, and should be removed. The application of filters it not limited to post-parse filtering. In some cases filters can be applied earlier, during parsing or sometimes even during parser generation.

Let $\Phi$ denote a set of parse trees, and consider a grammar $\Gamma$. A *disambiguation filter* $\mathcal{F}$ is a function from a set of parse trees to a set of parse trees, where $\mathcal{F}(\Phi) \subseteq \Phi$. This condition ensures that a filter does not invent new parse trees. The set of parse trees in $\Phi$ can contain parse trees for different input strings. Given a set of derivation trees, define $\mathcal{F}(\Phi, w) = \{t \in \Phi \mid \mathit{yield}(t) = w\}$. When for all $w \in L(\Gamma)$, $|\mathcal{F}(\Phi, w)| \leq 1$, filter $\mathcal{F}$ is *completely disambiguating* [33]. A filter is $\mathcal{F}$ *complete* when for all $w \in L(\Gamma)$, $|\mathcal{F}(\Phi, w)| = 1$ [48].

In the remainder of this section we look at various disambiguation methods that can be described using the filters. These filters typically use some additional data $Q$. They formally describe which parse trees are the intended ones, but leave open the implementation. For each of the disambiguation methods expressed as filter, possible implementations are given.

### 3.3.1 Subtree Exclusion

Thorup [49] introduces a disambiguation method that consists of specifying a finite set of forbidden sub-parse trees that may not occur in any of the parse trees produced.

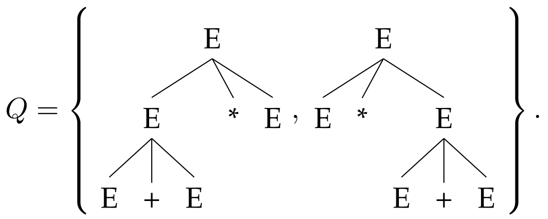
Given a finite set $Q$ of tree patterns, called the set of *forbidden sub-tree patterns*, the subtree exclusion filter $\mathcal{F}_{SE}$ is defined by

$$\mathcal{F}_{SE}(\Phi) = \{t \in \Phi \mid \neg\exists s \in sub(t) : Q \text{ matches } s\}.$$

Function $sub(t)$ returns the set of subtrees starting from node $t$. The tree patterns in $Q$ are finite, and can for instance be used to enforce priority rules among binary expressions. Consider Grammar $\Gamma_{3.3}$, and assume the standard semantics of $+$ as plus and $*$ as multiplication, where $*$ binds stronger than $+$.

$$E ::= E + E \mid E * E \mid Id \qquad (\Gamma_{3.3})$$

We define $Q$ to forbid subtrees where $E + E$ occurs in the subtree of $E * E$, since this would mean that $+$ would bind stronger than $*$. The set $Q$ looks as follows



This disambiguation method can implemented as a grammar transformation [49]. The transformation avoids the ambiguities related to precedence of operators from occurring in the modified grammar $\Gamma'$, but the size of $\Gamma'$ may be exponential with respect to the original grammar $\Gamma$.

It is also possible to implement this disambiguation method as a post-parse filter. There are various ways in which this can be done. Afroozeh [2] uses term rewriting as a mechanism to remove subtrees in the parse forest that match one of the elements in $Q$.

### 3.3.2 Reject rules

Reject rules [14] is a disambiguation method that is used to filter restricted keywords from identifier nonterminals. For instance, in programming languages a variable name may never be equal to restricted keywords such as *if*, *then* or *while*.

The data that is being used is a set $Q$ of pairs $\langle A, k\rangle$, where $A$ is a nonterminal and $k$ a terminal string representing the keyword. Define $\mathcal{T}(t)$ to be the subtree rooted at $t$. Then the filter for reject rules, $\mathcal{F}_R$ is defined by

$$\mathcal{F}_R = \{t \mid t \in \Phi \wedge \neg\exists(A, y) \in Q, w \in \mathcal{T}(t) : A = w \wedge yield(w) = y\}.$$

Reject rules can be implemented as a post-parse filter, for instance using term rewriting. One has to look at parse trees that have nodes with label $A$ with a child $k$ such that $\langle A, k\rangle \in Q$, and remove these trees.

### 3.3.3 Precede-follow restrictions

Follow restrictions are often used as a disambiguation method to achieve longest match disambiguation. They are a simplification of the adjacency restriction rules [40]. The dual of follow restrictions are precede restrictions [16]. These are not as widely used as the former but are also quite powerful.

Consider a parse tree $t \in \Phi$, not containing $\varepsilon$ symbols. For each $v \in t$, we will define the *first*, *last*, *follow*, and *precede* values of that node as follows. Let $w = yield(t)$, with $|w| = m$. If $w$ is not equal to the empty string then $w = w_0 \dots w_{m-1}$. Define $w_{-1} = w_m = \$$, where \$ is a dummy symbol. Now for a node $v$ with $extent(v) = \langle i, j\rangle$ we define

$$\textsc{first}(v) = \begin{cases} \varepsilon & \text{if } i = j \\ w_i & \text{otherwise} \end{cases} \qquad \textsc{last}(v) = \begin{cases} \varepsilon & \text{if } i = j \\ w_{j-1} & \text{otherwise} \end{cases}$$

and $\textsc{precede}(v) = w_{i-1}$ and $\textsc{follow}(v) = w_j$. Figure 3.2 shows an example parse tree with the values for an internal node.

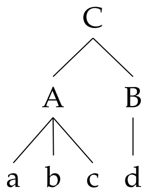


Figure 3.2: Parse tree where $extent(A) = \langle 0, 3\rangle$, $\textsc{precede}(a) = \$, \textsc{follow}(A) = d, \textsc{first}(A) = a, \textsc{last}(A) = c$.

Let $Q$ be a pair $\langle \mathcal{P}, \mathcal{F}\rangle$, where $\mathcal{P}$ and $\mathcal{F}$ are sets of pairs $\langle p, term\rangle$ with production $p \in P$ and terminal $term \in T$, representing the precede and follow restrictions respectively. Define $\mathcal{T}(u)$ to be the set of its descendants of $u$, included $u$ itself, and let $prod(u)$ return the corresponding production of a node $u$. Then the filter for precede-follow restrictions $\mathcal{F}_{PF}$ is defined by

$$\begin{aligned}\mathcal{F}_{PF} = \{t \in \Phi \mid \neg\exists\langle p, a\rangle \in \mathcal{P}, \langle q, b\rangle \in \mathcal{F}, w \in \mathcal{T}(t) :\\ (p = prod(w) \wedge \textsc{precede}(w) = a)\vee\\ (q = prod(w) \wedge \textsc{follow}(w) = b)\}\end{aligned}$$

This filter removes all trees where a node corresponding to production $p$ has $t$ as its precede value for pairs $\langle p, t\rangle \in \mathcal{P}$, and nodes corresponding to a production $q$ with $u$ as its follow value for pairs $\langle q, u\rangle \in \mathcal{F}$.

Precede restrictions can be implemented in parser algorithms that *predict* new occurrences of production rules while parsing [16], like Earley [21] or GLL [46]. In these algorithms the predict is not performed if the preceding input symbol is forbidden. Follow restrictions can also be incorporated in the parsers being generated by these algorithm. In the Earley algorithm, no *reduce* action is performed if the next terminal is forbidden. In the GLL algorithm, the *pop* is not executed if the next terminal is forbidden.

# Chapter 4

# Filtering in the SPPF

In this chapter we will look into filtering of shared packed parse forests (SPPF). An SPPF is a single data structure that encodes all parse trees, and uses compaction mechanisms to reduce the required space to represent multiple derivation trees. There are two methods to reduce the space required; *subtree sharing* and *local ambiguity packing* [39]. By means of subtree sharing, nodes having the same tree below them are shared. Using local ambiguity packing, nodes which correspond to different derivations of the same subsentence from the same nonterminal are combined by creating a packed node for each alternative.

If there are multiple valid derivations for a given input string, the resulting SPPF will contain ambiguities. Disambiguation is needed to remove undesired derivations from the parse forest, while retaining those that are valid. Removing an undesired derivation boils down to removing the corresponding parse tree that is embedded in the SPPF. In many cases it is not possible to simply remove the nodes of such a parse tree from the SPPF, since these nodes are shared with other parse trees embedded in the SPPF.

As stated in the previous chapter, disambiguation filters can be used to define trees that are not desired. Because we are dealing with an SPPF rather than a set of parse trees, we need to specify the filters in terms of parse trees embedded in an SPPF. For this purpose we introduce the concept of SPPF filters. Many filters are defined in negative terms, stating that parse trees containing certain patterns should be removed. Examples include subtree exclusion, reject rules, and precede-follow restrictions that have been described in the previous chapter.

In this chapter we will describe how parse trees are embedded in an SPPF, and look at removing specific parse trees from the SPPF. We will introduce SPPF filters that are able to remove all parse trees from an SPPF containing some node, edge, or path. In Chapter 5 we will look at disambiguating expression grammars, were we will use these filters for removing undesired parse tree embeddings. Filters for removing parse trees containing some subtree (that is not a path) are left for future work.

## 4.1 Parse Trees in an SPPF

An SPPF $\mathcal{S}$ contains a set of SPPF parse trees according to some grammar and an input string. An SPPF parse tree is the same as a parse tree, except that it has intermediate nodes and packed nodes that need to be removed. The set of SPPF parse trees in an SPPF is defined in the following way.

**Definition 4.1.1** (Set of all SPPF parse trees in an SPPF)**.** Let $\mathcal{S}$ be an SPPF, and $SPT_{\mathcal{S}}$ be the set of SPPF parse trees embedded in $\mathcal{S}$. A SPPF parse tree $t \in SPT_{\mathcal{S}}$ can be obtained in the following way.

Start at the root node of $\mathcal{S}$, and walk the tree by choosing one packed node below each visited node, and choosing all the children of a visited packed node in a recursive manner.

Any SPPF parse tree can be obtained by making particular choices, and any tree constructed this way is an SPPF parse tree.

To illustrate this set, and the relationship between SPPF parse trees and normal parse trees consider the grammar shown in Grammar $\Gamma_{4.1}$. In this grammar there is a single operator, plus, that can be used as prefix, binary, and postfix operator.

$$S ::= +S \mid S + S \mid S+ \mid 1 \qquad (\Gamma_{4.1})$$

Now suppose we have an input sentence $I$ = "1+++1". Given the grammar, this sentence is ambiguous and results in three parse trees. The SPPF $\mathcal{S}$ is shown in Figure 4.1.

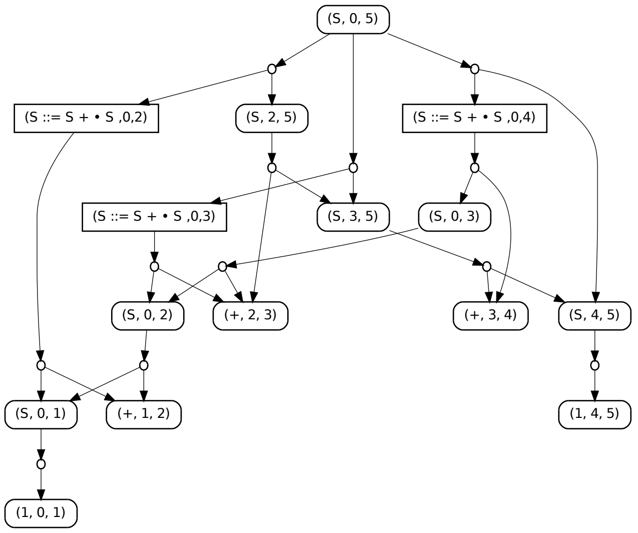


Figure 4.1: SPPF for Grammar $\Gamma_{4.1}$ and input $I$ = "1+++1"

The SPPF parse trees embedded in $\mathcal{S}$, which are elements of the set $SPT_{\mathcal{S}}$, are shown in Figure 4.2. Figure 4.3 shows the parse trees that result upon removal of the intermediate and packed nodes.

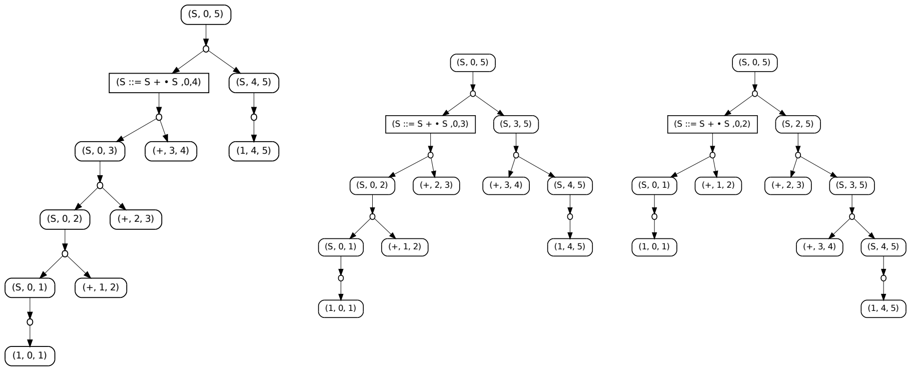


Figure 4.2: SPPF parse trees in $\mathcal{S}$.

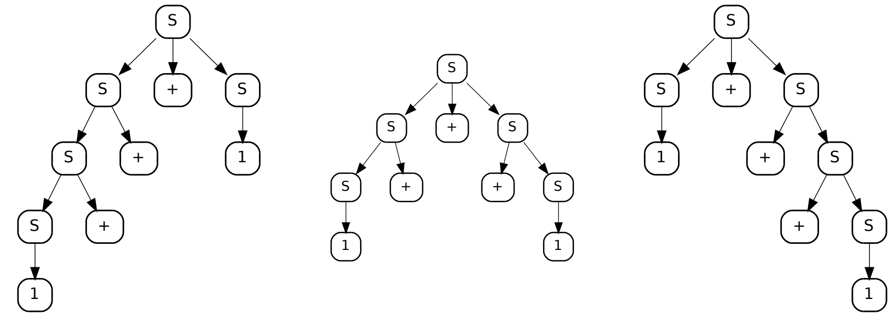


Figure 4.3: Parse trees in $\mathcal{S}$.

Intermediate nodes and packed nodes in an SPPF parse tree can be removed by removing the node itself, and attaching its children to its parent node. It is important to note that the order of the children needs to be preserved.

## 4.2 SPPF Filters

In order to remove certain parse trees from the SPPF we introduce the notion of an SPPF filter.

**Definition 4.2.1** (SPPF filter)**.** An SPPF filter $\mathcal{F}$ is a function $\mathcal{F} : \mathcal{S}_{in} \rightarrow S_{out}$, where $S_{in}$ and SPPF $S_{out}$ are sets of SPPFs, satisfying the requirement that the set of SPPF parse tree embeddings in the elements of $S_{out}$ is a subset of the set of SPPF parse tree embeddings in the elements of $S_{in}$. Stated formally, we require that

$$\bigcup_{\mathcal{S}' \in S_{out}} SPT_{\mathcal{S}'} \subseteq SPT_{\mathcal{S}_{in}}.$$

This restriction ensures that an SPPF filter can never introduce new parse trees. Note that the output of a filter is a set instead of a single SPPF. This is due to the fact that the SPPF needs to be split if we want to remove all SPPF parse trees from the SPPF containing some path or subtree. Note that there are infinitely many SPPF parse trees if the SPPF contains a cycle.

Given a grammar $\Gamma$ and an input sentence $I$, the corresponding SPPF is unique. When this SPPF is altered by means of a filter, some parse trees in the SPPF are removed, and the SPPF can be split into multiple smaller SPPFs. A *complete SPPF* is an SPPF which given a grammar and an input sentence contains all parse trees. After applying a filter, the output set might contain incomplete SPPFs, where undesired parse trees embedded in the SPPF have been removed.

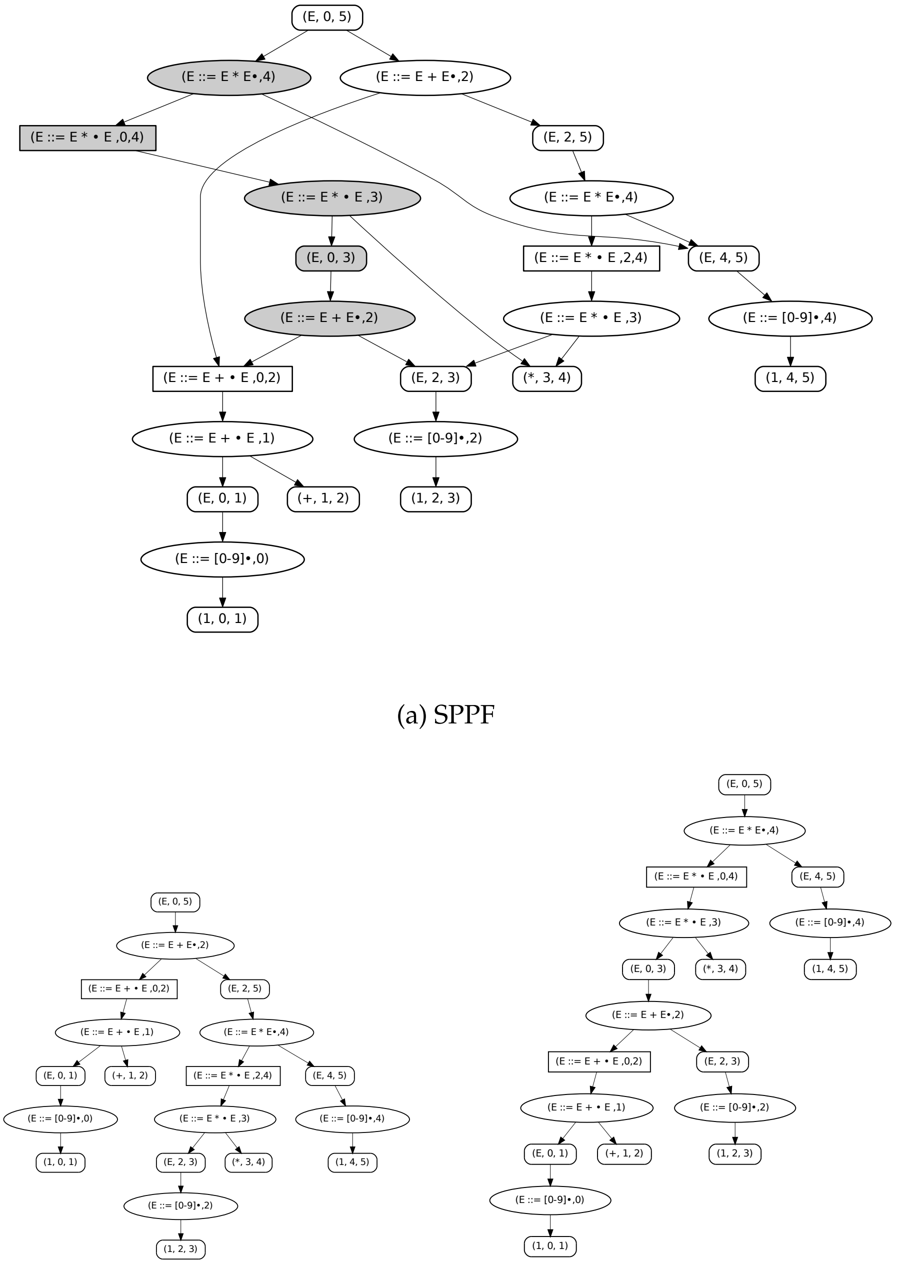
## 4.3 Removing parse trees from the SPPF

Ambiguity reduction in an SPPF can be done by removing parse trees from the SPPF that are undesired. Removing parse trees from an SPPF means that at a certain point the structure of the SPPF needs to be changed by removing edges and nodes.

As an introduction into parse tree removal, consider Grammar $\Gamma_{4.2}$ with two binary operators, addition and multiplication, that can be applied on single digit numbers. Given input sentence $1 + 1 * 1$ there are two derivations: $(1 + (1 * 1))$ and $((1 + 1) * 1)$.

$$E ::= E + E \mid E * E \mid [0\text{-}9] \qquad (\Gamma_{4.2})$$

Since multiplication has precedence over addition, we want to remove trees where "+" productions are direct children of "*" productions. In order to remove such trees, we look for paths from a packed node with label $(E ::= E * E\cdot, k)$ to a packed node with label $(E ::= E + E\cdot, j)$ for some $k, j$. If there are other packed nodes in between these nodes on the path, then they must be related to the parent packed node. In our example we have packed node $(E ::= E * \cdot E, k)$ that satisfies this criterium. All embedded parse trees in the SPPF $\mathcal{S}$ that contain this path need to be removed from $\mathcal{S}$. Figure 4.4 shows $\mathcal{S}$ and the two SPPF parse trees embedded in $\mathcal{S}$. In $\mathcal{S}$ we have highlighted all paths, in our case just one, where a "+" production is a direct child of a "*" production. Removing this path $p$ will remove the parse tree corresponding to $((1 + 1) * 1)$ from $\mathcal{S}$.



(a) SPPF

(b) SPPF tree corresponding to derivation $(1 + (1 * 1))$

(c) SPPF tree corresponding to derivation $((1 + 1) * 1)$

Figure 4.4: SPPF and SPPF tree embeddings for Grammar $\Gamma_{4.2}$ and input sentence $1 + 1 * 1$.

After removing nodes and edges from an SPPF we have to check that the resulting SPPF is still a valid SPPF. For instance, nodes may become unreachable if all parents of the node are removed. Another issue is that as a side effect valid SPPF trees are possibly removed from the SPPF. For example, suppose that node $(E, 0, 3)$ in $\mathcal{S}$ has another incoming edge from a packed node $u$ with a "+" production. Then removing path $p$ will also remove any tree that contains path $\langle u, (E, 0, 3), (E ::= E + E\cdot, 2)\rangle$, which should remain in $\mathcal{S}$. To avoid this problem we split the SPPF into multiple copies and then remove some edges in each of the copies to eliminate all parse trees containing the specific path.

To deal with these kind of issues, we need algorithms that remove all parse trees from $\mathcal{S}$ that contain some pattern, without removing valid parse trees. A pattern can be a node, edge, path, or even a certain subtree. Removing all parse trees containing some node results in a single smaller SPPF. Node removal can for instance be used for prefer-avoid filtering, where we remove packed nodes below a symbol node.

It turns out that edge removal can be expressed in terms of node removal. When a path needs to be removed, we need to create a number of copies of the SPPF, and in each copy remove some particular edge. In the following sections we will describe the algorithms to perform these operations. At the end of this chapter we will also briefly look at the usefulness of removing parse trees that contain some subtree.

### 4.3.1 Removing all parse trees containing some node

Suppose we have an SPPF $\mathcal{S}$, and a node $x$. Now we want to remove all parse trees from $\mathcal{S}$ that contain node $x$. The first step in the process is to remove node $x$ from $\mathcal{S}$, and all its outgoing and incoming edges. After removing all outgoing edges, nodes that become unreachable in $\mathcal{S}$ are no longer part of any SPPF parse tree in $\mathcal{S}$. These are the descendant nodes of $x$ that are not part of any other SPPF parse tree not containing $x$.

After removing $x$ from $\mathcal{S}$, it might be the case that certain ancestors also need to be removed. Otherwise we might end up with new trees that were not embedded in the original $\mathcal{S}$. The following lemmas directly follow from Definition 4.1.1, and are used to reason about ancestor removal. We will use $V(\mathcal{S})$ and $E(\mathcal{S})$ to define the set of nodes and edges in $\mathcal{S}$ respectively.

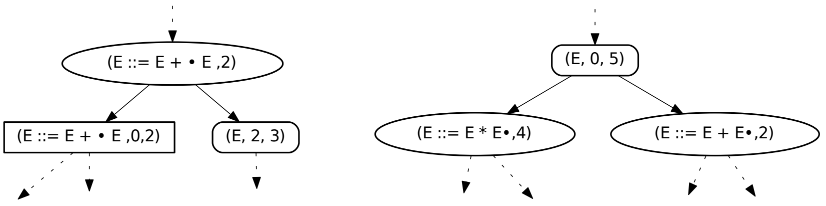
**Lemma 4.3.1.** *Let $t$ be an SPPF parse tree embedded in an SPPF $\mathcal{S}$, and let $u \in V(\mathcal{S})$ be a packed node. If $t$ contains $u$, then $t$ contains all children of $u$.*

When we remove a child of a packed node, the packed node becomes invalid and must be removed. Otherwise we would end up with a corrupt SPPF parse tree embedded in $\mathcal{S}$ that misses part of the derivation.

**Lemma 4.3.2.** *Let $t$ be an SPPF parse tree embedded in an SPPF $\mathcal{S}$, and let $u \in V(\mathcal{S})$ be an intermediate or a nonterminal symbol node. If $t$ contains $u$, it must include at least one of the children of $u$.*

If we remove a packed node, we must check whether the parent has multiple children, and remove the parent if this is not the case. This ensures that we avoid the situation where a nonterminal symbol node or intermediate node becomes a leaf.

To illustrate, consider Figure 4.5a. In Figure 4.5a, selecting packed node $(E ::= E + \cdot E, 2)$ means selecting its two children. This corresponds to Lemma 4.3.1. In Figure 4.5b, given node $(E, 0, 5)$ we can select either packed node $(E ::= E * E\cdot, 4)$ or $(E ::= E + E\cdot, 2)$, corresponding to Lemma 4.3.2.



(a) (b)

Figure 4.5

Now we have all the ingredients necessary to construct an algorithm that removes all embedded SPPF parse trees in an SPPF containing some node $x$.

**Algorithm** *RemoveParseTrees*($\mathcal{S}, x$)

```
$\mathcal{S}' \leftarrow \mathcal{S}$
$Q = [x]$
while $Q \neq [\,]$
    do $node$ ← Dequeue(Q) (* remove element from Q *)
       $parents \leftarrow node.parents$
       remove $node$ from $\mathcal{S}'$, and remove all its incoming and outgoing edges
       if $node.type = symbol \lor node.type = intermediate$
         then for $parent \in parents$
                do Enqueue($Q, parent$)
         else (* $node.type = packed$ *)
              for $parent \in parents$
                do if $|parent.children| = 0$
                     then Enqueue($Q, parent$)
return $\mathcal{S}'$
```

**Theorem 4.3.3.** *Given an SPPF $\mathcal{S}$, and node $x \in V(\mathcal{S})$, Algorithm RemoveParseTrees returns SPPF $\mathcal{S}'$, where the trees embedded in $\mathcal{S}'$ are precisely those SPPF trees from $\mathcal{S}$ not containing $x$.*

*Proof.* Recall Definition 4.1.1 which defines $SPT_{\mathcal{S}}$; the set of trees encoded in an SPPF. Each $t \in SPT_{\mathcal{S}}$ can be obtained by starting at the root node of $\mathcal{S}$, and walking the tree selecting one packed node below each visited node, and all the children of a visited packed node recursively.

Let $X$ be the set of all SPPF parse trees in $\mathcal{S}$ containing $x$, and $\overline{X}$ the set of all SPPF parse trees in $\mathcal{S}$ *not* containing $x$.

We need to prove that:

1. there are no invalid SPPF parse trees in $\mathcal{S}'$ that are not in $\mathcal{S}$;

2. all SPPF parse trees in $\mathcal{S}$ not containing $x$ are in $\mathcal{S}'$.

First note that we only remove nodes from $\mathcal{S}$, so we can not create new SPPF parse trees by adding nodes. The only way in which we can create an invalid embedded tree in $\mathcal{S}'$ is after removing some node. An invalid embedded tree can be created in two situations:

(i) if a non-leaf node in $\mathcal{S}$ becomes a leaf node in $\mathcal{S}'$;

(ii) if a packed node in $\mathcal{S}$ is present in $\mathcal{S}'$ and has less child nodes.

Whenever we remove a node $u$, we must ensure that this node is present in all SPPF parse trees in $X$ and not present in any SPPF parse tree in $\overline{X}$. When we remove $u$, the descendants of $u$ may become unreachable. In this case these nodes are not part of any parse tree not containing $u$. If they stay reachable, they are present in some SPPF parse tree embedded in $\mathcal{S}$ not containing $u$. If they stay reachable, they are present in some SPPF parse tree embedded in $\mathcal{S}$ not containing $u$. Upon removal of $u$, we need to show that situations (i) and (ii) are not applicable for the parents of $u$.

- Consider the case where $u$ is a symbol node or an intermediate node. Then its parents are packed nodes. By Lemma 4.3.1 we must remove these parents, because otherwise we would end up with a corrupt SPPF tree embedded in $\mathcal{S}$. After removing the parents, we need to check the grandparents for these situations.

- In the case where $u$ is a packed node, a parent of $u$ may have multiple packed nodes as children. If $u$ has no siblings given a parent $v$, $v$ becomes a leaf. This would mean that we introduce a new tree in $SPT_{\mathcal{S}}$, which is corrupt since $v$ cannot be a terminal symbol node. Therefore node $v$ must be removed. If $v$ has multiple children, then $v$ is not removed and situations (i) and (ii) are not applicable. If $v$ is removed, we need to check the parents for situations (i) and (ii) and see if these parents must be removed or not.

On termination of the algorithm, all SPPF parse trees containing node $x$ are removed from $\mathcal{S}$, resulting in the new SPPF $\mathcal{S}'$. Note that $\mathcal{S}'$ contains unreachable nodes that are no longer part of any SPPF. Using a final step we can walk $\mathcal{S}'$ and output just the nodes reachable from the root. □

**Use cases**

To understand the practical use of removing all parse trees containing some node, recall the precede-follow restriction mechanism. As described in Chapter 2, each symbol node in the SPPF contains a label $(x, i, j)$, where $x$ is a terminal, nonterminal, or $\varepsilon$, and $\langle i, j\rangle$ is the extent. Using this information we can look at the symbols at input positions $i - 1$ and $j + 1$ and see whether a precede or follow restriction applies. In that case we need to delete all parse trees containing this node. Chapter 5 will describe precede-follow filters to disambiguate simple expression grammars that use this node removal.

Preference/Avoid rules can also be implemented using node removal. Whenever there is a choice for several subderivations, the nonterminal symbol node has multiple packed nodes as children, each representing a different alternative. All alternative derivations that have the *avoid* attribute, each represented by a packed node, are removed by removing the packed node, but only if other alternative derivations are there that do not have this attribute. If there are derivations with the *prefer* attribute, represented by a packed node, then all derivations not having this attribute are removed. These derivations can be removed by removing all parse trees containing the associated packed node. Care has to be taken that the order of applying the rules could possibly influence the results. For instance if two prefer rules are applicable, choosing either one might result in a different outcome.

## 4.3.2 Removing all parse trees containing some edge

Suppose that we want to remove all parse trees containing an edge $(u, v)$, from an SPPF $\mathcal{S}$. Then there are four different edge configurations possible as shown in Figure 4.6. An edge always connects a packed node with an intermediate node or symbol node, or the other way round (see also Section 2.2).

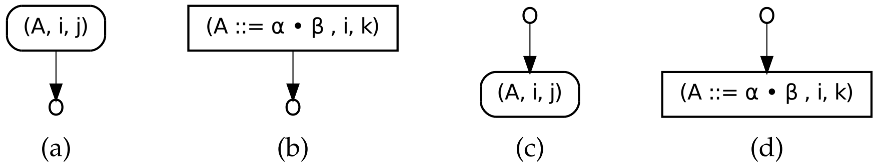


Figure 4.6: Edge configurations in an SPPF

Removing all parse trees containing edge $(u, v)$ from an SPPF is equivalent to removing the packed node, which is either $u$ or $v$.

**Lemma 4.3.4.** *Removing all SPPF parse trees containing an edge $(u, v)$ in SPPF $\mathcal{S}$ is equal to removing all SPPF parse trees containing $u$ if $u$ is a packed node, or $v$ if $v$ is a packed node.*

*Proof.* If we remove a node, all its incoming and outgoing edges are also removed. What we need to prove is that we do not remove SPPF parse trees from $\mathcal{S}$ not containing edge $(u, v)$.

- Suppose $u$ is a packed node. Then if an SPPF parse tree embedded in $\mathcal{S}$ contains $u$, it will also include each child $v$ of $u$ by Definition 4.1.1 and edge $(u, v)$ for $v \in V(\mathcal{S})$ and $(u, v) \in E(\mathcal{S})$.

- Suppose $v$ is a packed node. Then if an SPPF parse tree embedded in $\mathcal{S}$ contains $v$, it must contain the (unique) parent of $v$, which is node $u$ and edge $(u, v)$.

Removing all parse trees containing the packed node will therefore remove exactly those SPPF parse trees embedded in $\mathcal{S}$ that contain the edge $(u, v)$. □

The algorithm to remove all SPPF parse trees containing an edge in an SPPF now follows direct from Lemma 4.3.4 and Theorem 4.3.3.

```
Algorithm RemoveParseTreesE($\mathcal{S}$, $(u, v)$)
Input: SPPF $\mathcal{S}$ with $(u, v) \in E(\mathcal{S})$
    if $u.type = packed$
        then return RemoveParseTrees($\mathcal{S}$, $u$)
        else  return RemoveParseTrees($\mathcal{S}$, $v$)
```

**Theorem 4.3.5.** *Algorithm RemoveParseTreesE correctly removes all parse trees from $\mathcal{S}$ containing some invalid edge $(u, v)$.*

*Proof.* Follows directly by Theorem 4.3.3, and Lemma 4.3.4. □

**Corollary 4.3.6.** *Removing all SPPF parse trees from an SPPF containing an edge is equal to removing the packed node attached to the edge.*

### Use cases

The removal of parse trees containing some edges is not especially useful by itself. It is however used in the filter of removing all parse trees containing some path, which will be described next.

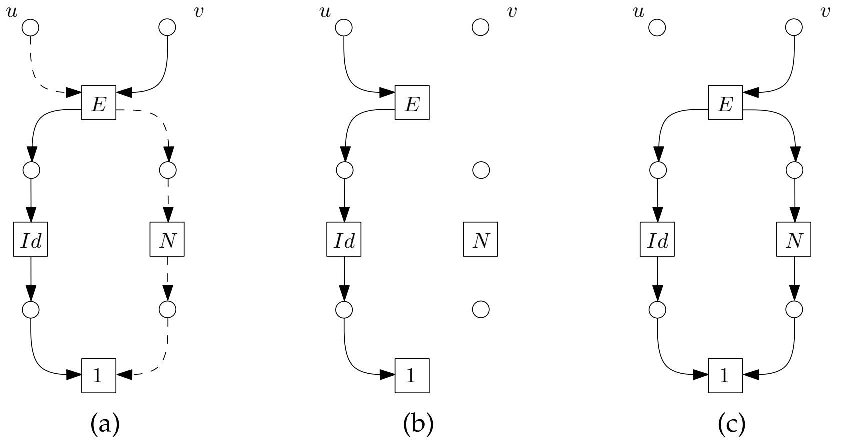


Figure 4.7: The dashed path needs to be removed. The paths shown in (b) and (c) must still be present in the new SPPF.

### 4.3.3 Removing all parse trees containing some path

Suppose we want to remove all parse trees containing a path $p = \langle u_1, \ldots, u_k \rangle$ from an SPPF $\mathcal{S}$. Then simply removing some nodes and edges in $\mathcal{S}$ will no longer suffice. Consider the example shown in Figure 4.7, where we want to remove the dashed path, shown in (a). We cannot simply remove the entire path since a subpath of it may be used by other paths. Removing the first edge is also not valid, since then the path shown in (b) is removed. Removing the last edge will remove the right path $\langle v, E, \_, N, \_, 1 \rangle$ present in (c).

In order to correctly remove path $p$ from $\mathcal{S}$, we need to split $\mathcal{S}$ into several smaller SPPFs. We require that the grammar corresponding to the SPPF does not generate a cycle in the SPPF. Otherwise, we can not always split the SPPF into several smaller SPPFs to remove all parse trees containing some path. At the end of this section we will illustrate the problem with cycle-generating grammars.

Given a path of length $k$, we need to split $\mathcal{S}$ into at most $k - 1$ smaller SPPFs as is proven in the next theorem. After proving this theorem we will improve this bound.

**Theorem 4.3.7.** *Removing path $p = \langle u_1, \ldots, u_k \rangle$ from $\mathcal{S} = (V, E)$ results in set $\mathcal{C}$ containing at most $k - 1$ copies with $1 \leq i < k$. Path $p$ must be present in $\mathcal{S}$, and $\mathcal{S}$ may not contain any cycle.*

*Proof.* Let $C_i \in \mathcal{C}$ be a copy such that $SPT_{C_i} = SPT_{\mathcal{S}} \setminus SPT_{\mathcal{S}}(u_i, u_{i+1})$, where $SPT_{\mathcal{S}}(u_i, u_{i+1})$ is the set of SPPF parse trees in $\mathcal{S}$ where each tree in this set contains edge $(u_i, u_{i+1})$.

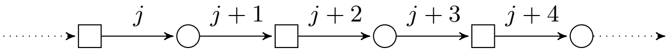
We need to show that all parse trees *not* containing $p$ are still embedded in some $\mathcal{C}_i$. For every SPPF tree in $\mathcal{S}$ that does not contain $p$ we have that it does not contain some edge of $p$. Let this edge be $(u_i, u_{i+1})$ with $1 \leq i < k$. Then by definition of $C_i$, this SPPF tree is embedded in copy $C_i$.

By correctness of the edge removal algorithm, we do not introduce any new invalid SPPF parse trees in any created copy.

□

**Theorem 4.3.8.** *Removing path $p = \langle u_1, \ldots, u_k \rangle$ from $\mathcal{S}$, corresponding to a grammar not generating cycles in $\mathcal{S}$, results in at most $\lceil \frac{k}{2} \rceil$ copies $\mathcal{C}_i$ with $1 \leq i < \lceil \frac{k}{2} \rceil$.*

*Proof.* Given Theorem 4.3.7, we know that we can create $k - 1$ copies of $\mathcal{S}$ that contain all SPPF parse trees not containing path $p$. Given the property that $\mathcal{S}$ is a bipartite graph, any path is of the following shape:



.

Each edge connects an intermediate node or symbol node (shown as a rectangle) with a packed node (shown as a circle). By Corollary 4.3.6 we know that removing an edge removes the same parse trees from $\mathcal{S}$ as removing the packed node attached to the edge. In our example, removing edge $j$ or edge $j + 1$ removes the same set of parse trees.

Suppose that path $p$ has length $k$. Then either $p$ starts with an intermediate or symbol node, or with a packed node.

Now the number of copies, $|\mathcal{C}|$, is given by the following equation:

$$|\mathcal{C}| = \begin{cases} k \;\mathsf{div}\; 2 + k \;\mathsf{mod}\; 2 & \text{if } u_1 \text{ is a symbol or intermediate node} \\ k \;\mathsf{div}\; 2 + 1 & \text{otherwise} \end{cases}$$

Since $k \;\mathsf{div}\; 2 + k \;\mathsf{mod}\; 2 \leq \lceil \frac{k}{2} \rceil$ we have proven the theorem. □

**Theorem 4.3.9.** *Given Theorem 4.3.7, we know that we can create $k - 1$ copies that contain all parse trees which do not contain path $p$. Removing path $p = \langle u_1, \ldots, u_k \rangle$ from $\mathcal{S}$, corresponding to a grammar not generating cycles in $\mathcal{S}$, results in at most $1 + |W|$ copies $\mathcal{C}_i$, where*

$$\begin{aligned} W = \{u_x \mid \;& (u_x.type = symbol \vee u_x.type = intermediate) \\ & \wedge degree(u_x) > 2 \wedge 1 < x < k\}. \end{aligned}$$

*The degree of a node $u_x$ is given by the number of incoming edges plus the number of outgoing edges.*

*Proof.* We will proof the theorem by case distinction on the size of $W$.

- If $|W| = 0$, all nodes on path $p$ have a degree of one or two. Consider an intermediate or symbol node $u$ on path $p$. By Lemma 4.3.2, each SPPF parse tree containing $u$ must include at least one child of $u$.

Since the degree of $u$ is not bigger than 2, there is only one child. For any packed node $v$ on path $p$ we have that all its children are present in the same SPPF parse trees by Lemma 4.3.1. Combining these observations yields that any two nodes of path $p$ are part of the same set of SPPF parse trees in $\mathcal{S}$. No matter which edge in subpath $p$ we remove, the resulting copy will be exactly the same.

- If $|W| = 1$, there is a symbol or intermediate node on path $p$ having a degree larger than two. This can mean that the node has multiple incoming edges, or multiple outgoing edges. In the first case, the parent node not on the path is not necessarily present in all SPPF parse trees in $\mathcal{S}$. Consider Figure 4.8a. Here nodes $u_i, u_{i+1}$ and $u_{i+2}$ are present on path $p$, and $v$ is not on the path. Edge $(u_{i+1}, u_{i+2})$ is present in all SPPF parse trees having edge $(u_i, u_{i+1})$, but because of node $v$ this does not hold the other way round. Therefore removing edge $(u_i, u_{i+1})$ or $(u_{i+1}, u_{i+2})$ will result in different copies. A similar argument can be made for a symbol or intermediate node having multiple outgoing edges (cf. Figure 4.8b).

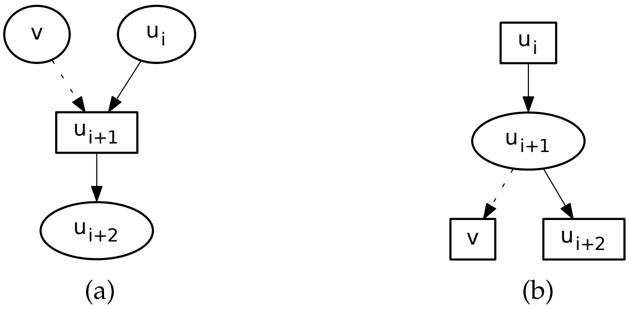
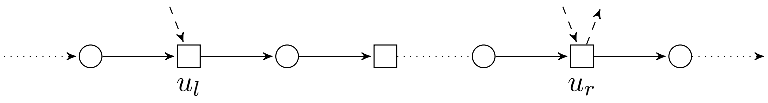


Figure 4.8

- Now consider the case where $|W| > 1$. Given the property that $\mathcal{S}$ is a bipartite graph, consider subpath $p' = \langle u_l, \ldots, u_r \rangle$ of $p$, and assume, $degree(u_l) > 2$, $degree(u_r) > 2$, $degree(u_j) = 2$ for $l < j < r$:



.

For any two nodes $u_m, u_n$ with $i < m < n < k$ we have that they are present in exactly the same set of SPPF parse trees in $\mathcal{S}$, following the same reasoning as in the case where $|W| = 0$. Now consider any edge $(u_j, u_{j+1})$ with $l < j < r$. No matter which edge in subpath $p'$ we remove, the resulting copy will be exactly the same.

Starting from index 1, each time we encounter a node $w \in W$, we have that removing edge $(u_{w-1}, u_w)$ is not equal to removing edge $(u_w, u_{w+1})$. This follows from the fact that edge $(u_{w-1}, u_w)$ need not be present in all SPPF parse trees containing $(u_w, u_{w+1})$ and vice versa. Therefore, in this case we need to add a new copy. Since we have $|W|$ such nodes, the maximal number of copies will be $1 + |W|$.

□

Algorithm *RemoveParseTreesP* creates the copies resulting of removing a path $\langle u_1, \ldots, u_k \rangle$ from an SPPF.

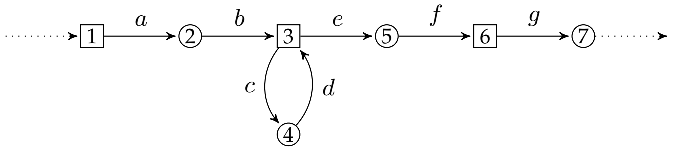
**Algorithm** *RemoveParseTreesP*($\mathcal{S}, \langle u_1, \ldots, u_k \rangle$)
**Input:** an SPPF $\mathcal{S}$ with path $p = \langle u_1, \ldots, u_k \rangle$
**Output:** set of SPPFs not containing $p$, but together containing all other paths in $\mathcal{S}$

```
$L = [(u_1, u_2)]$ (* list of edges *)
for $j = 2$ to $k - 1$
    do if $degree(u_j) > 2 \wedge (u_i.type = symbol \vee u_i.type = intermediate)$
        then $L \leftarrow L +\!\!+ [(u_j, u_{j+1})]$
$\mathcal{C} \leftarrow \emptyset$
for $i = 1$ to $|L|$
    do $\mathcal{C}_i \leftarrow$ RemoveParseTreeE($\mathcal{S}, L[i]$)
        if $\mathcal{C}_i \notin \mathcal{C}$ (* set of copies *)
            then $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathcal{C}_i\}$
return $\mathcal{C}$
```

The following theorem explains why the method of creating copies and removing edges does not work when there are cycles in the SPPF.

**Theorem 4.3.10.** *Removing path $p = \langle u_1, \ldots, u_k \rangle$ from $\mathcal{S} = (V, E)$, corresponding to a cycle generating grammar, cannot be done using Theorem 4.3.7.*

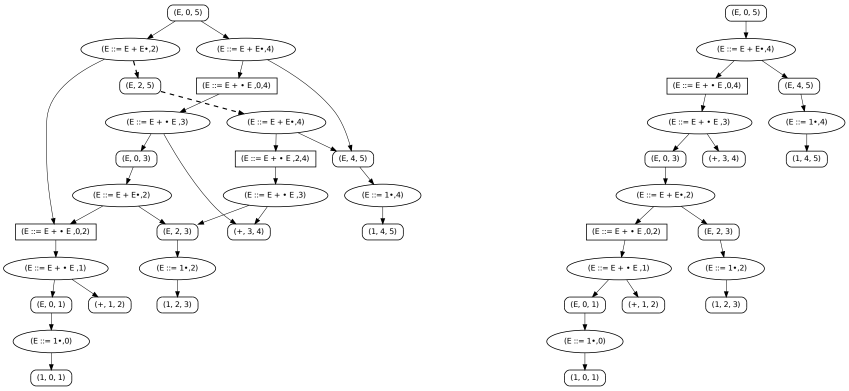
*Proof.* Consider the following subgraph of $\mathcal{S}$, and suppose that we want to remove the path $\langle 1, 2, 3, 5, 6, 7 \rangle$:



.

When removing one of the edges of $p$ in $\mathcal{S}$, no $\mathcal{C}_i \in \mathcal{C}$ will contain the path $\langle 1, 2, (3, 4)^+, 5, 6, 7 \rangle$. Intuitively we could add another copy $\mathcal{C}_j$, where we change the SPPF by expanding the cycle one time. But then we get two nodes with the same label, resulting in an SPPF that violates the structural SPPF property of unique labels for symbol nodes and intermediate nodes.

□



(a) SPPF for input string $1 + 1 + 1$, given grammar $E ::= E + E \mid 1$.

(b) SPPF after removing the dashed path. The result is a single SPPF parse tree.

Figure 4.9: Path removal example for removing ambiguities introduced by associativity.

### Use cases

Path removal is very generic, and can therefore be used as basis for implementing various kinds of disambiguation methods. For instance ambiguities caused by associativity can be disambiguated in a straightforward manner. The relationship between disambiguation methods and path removal is described in more detail in the next chapter.

As an illustration of path removal, consider the SPPF as shown in Figure 4.9a. Given are grammar $E ::= E + E \mid 1$, input string $1 + 1 + 1$, and the fact that operator $+$ is left-associative. Derivations of the shape $(E) \Rightarrow (E + E) \Rightarrow (E + (E + E))$, where $+$ is right-associative, need to be removed. This can be done by removing the dashed path shown in Figure 4.9a. In this path a packed node with label $(E ::= E + E\cdot, 2)$ has a packed node $(E ::= E + E\cdot, 4)$ occurring in the right subtree. After removing all parse trees containing this path, which is just a single parse tree in this case, we obtain the SPPF shown in Figure 4.9b.

**SDF priority and associativity** To further illustrate the applicability of path removal, we show how the priority and associativity constructs of SDF can be translated to path removal filters.

The following definitions are used in SDF for filtering expression grammars.

*Priority* Let priority relation $>$ be a partial order between recursive rules of an expression grammar. If $A ::= \alpha_1 A \alpha_2 > A ::= \beta_1 A \beta_2$, then all derivations $\gamma A\delta \Rightarrow \gamma(\alpha_1 A\alpha_2)\delta \Rightarrow \gamma(\alpha_1(\beta_1 A\beta_2)\alpha_2)\delta)$ are illegal.

*Associativity* If a recursive rule $A ::= A\alpha A$ is defined left associative, then any derivation $\gamma A\delta \Rightarrow \gamma(A\alpha A)\delta \Rightarrow \gamma(A\alpha(A\alpha A))\delta$ is illegal. If it is defined right associative, then any derivation $\gamma A\delta \Rightarrow \gamma(A\alpha A)\delta \Rightarrow \gamma((A\alpha A)\alpha A)\delta$ is illegal.

Note that the priority restriction is too restrictive. Consider the following expression grammar, where $E ::= E + E > E ::= E\ ?\ E\ !\ E$:

$$E ::= E\ ?\ E\ !\ E \mid E + E \mid a. \qquad (\Gamma_{4.3})$$

Then derivation $(a?(a+a)!a)$ is the only valid derivation for input sentence $a?a+a!a$, but will be removed by the filter. Another example is the grammar with both a binary and a unary minus, where $E ::= -E > E ::= E - E$:

$$E ::= -E \mid E - E \mid 1. \qquad (\Gamma_{4.4})$$

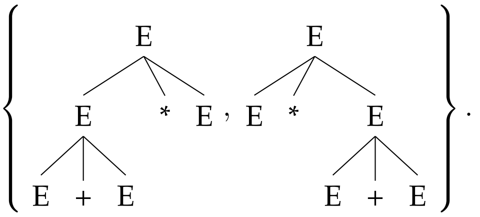
Here, input sentence $1 - -1$ has only one derivation; $(1 - (-1))$, but the filter removes this derivation.

In order to solve these problems, SDF introduces a special construct $\langle i \rangle$, to filter only below the $i$-th nonterminal in the production rule.

The translation of the priority and associativity construct to a path filter can be done in the following way.

1. Create a set of forbidden patterns. These patterns can be derived given a set of concrete production rules with their priorities. For instance, $E ::= E * E > E ::= E + E$ yields a set $Q$ containing two tree patterns:

   $Q =$

   Note that priorities are applied transitively by default, so $E ::= E^E > E ::= E * E > E ::= E + E$ yields 6 patterns. Each tree pattern is a two-level tree pattern. This means that we have a production rule where one of the nonterminals in this rule is expanded into another rule.

2. Transform each forbidden pattern to an SPPF tree pattern. This means introducing a packed node below each nonterminal symbol node, and binarizing the tree by introducing intermediate nodes. The extent or pivot values in the SPPF nodes are left undefined in order to

match any value. For our example, we obtain the two SPPF tree patterns shown in Figure 4.10.

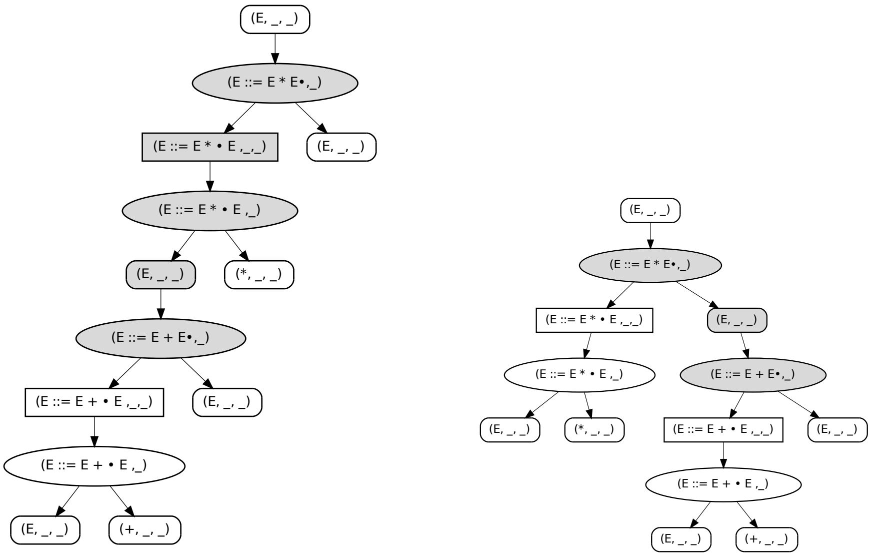


Figure 4.10: Tree patterns to remove rule $E ::= E * E$ directly below $E ::= E + E$ in a derivation.

3. Extract the path for each tree pattern. Find the packed nodes for the two production rules, and find the path between them. Removing all parse trees containing this path is the same as removing all parse trees having the corresponding tree pattern.

4. Remove all parse trees from the SPPF that contain any of these paths.

**Precede and follow restrictions** In Chapter 5 we will see that all Java expressions can be disambiguated, using SPPF filters expressed in terms of path removal. These filters are inspired by precede and follow restrictions, but also use the production rule that corresponds to the precede and follow symbols.

### 4.3.4 Removing all parse trees containing some subtree

Most filters that are used for enforcing associativities and priorities of productions can be expressed by removing all parse trees containing certain paths. However, when priorities can no longer be expressed on productions, we need to remove parse trees containing some *subtree*.

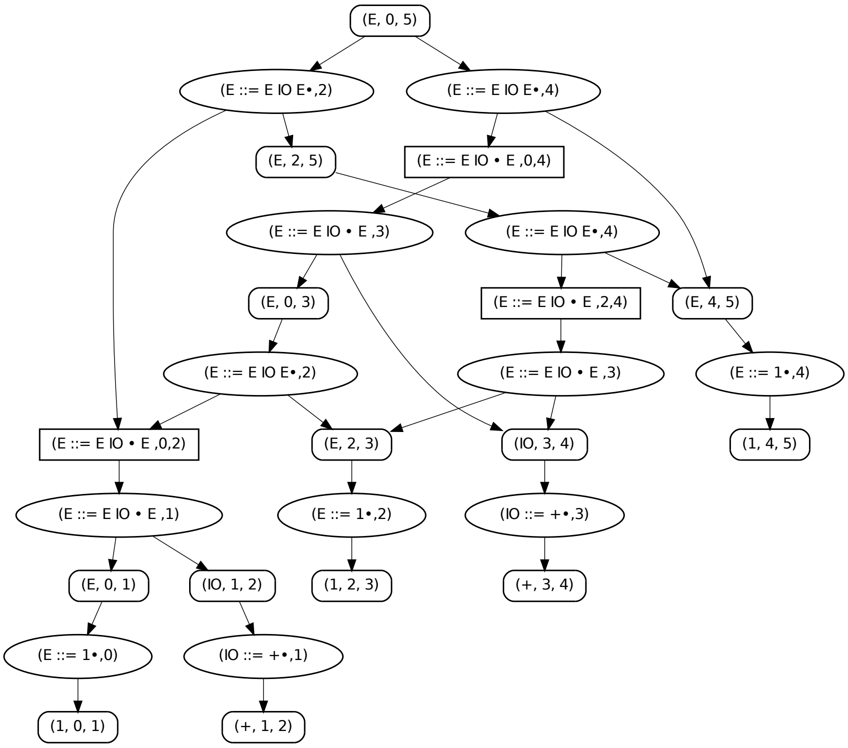


Figure 4.11: SPPF for Grammar $\Gamma_{4.5}$ and input sentence $1 + 1 + 1$

As an example consider Grammar $\Gamma_{4.5}$, where the infix operators and prefix operators are specified using separate nonterminals (IO and PO respectively).

$$\begin{aligned} E &::= E\ IO\ E \ \mid\ PO\ E \ \mid\ 1 \\ IO &::= `+\text{'} \mid `-\text{'} \mid `*\text{'} \\ PO &::= `-\text{'} \end{aligned} \tag{$\Gamma_{4.5}$}$$

Now, a packed node will still indicate whether nonterminal E is expanded as a binary infix operator or as an unary prefix operator. However, in order to find the specific operator, we need to look below the symbol node for $IO$ or $PO$ in the SPPF. Figure 4.11 shows an example of this.

In order to remove all SPPF parse trees where the binary $+$-operator is right-associative, we need to specify a subtree rather than a path. Another approach would be to integrate the use of attribute grammars in the parser, and add an attribute to the packed nodes that indicates the operator.

The design of algorithms for removing all parse trees containing some subtree is left as future work.

# Chapter 5

# Disambiguation of Expression Grammars

## 5.1 Expression Grammars

Expression grammars with unary and binary operators are an interesting class of grammars. These kind of grammars are typically used to define arithmetic expressions, and are present in a wide range of programming languages. Disambiguation is needed to select the right parse tree using relative priorities of the productions and associativity information in the case of binary operators.

Before looking into the disambiguation, we will first formally introduce the set of expression grammars with unary and binary operators. We will follow the format introduced by Brink [16], but use a slightly different notation. Let $A$ be a finite set of unary prefix operators, $B$ a finite set of binary operators, and $C$ a finite set of unary postfix operators. The *type* of an operator is either unary prefix, binary infix, or unary postfix. Any expression grammar $\Gamma_{expr}$ can be defined as a quadruple $\langle T, N, P, S\rangle$, where $T \subseteq A \cup B \cup C \cup \{\Box\}$, $N = \{E\}$, $S = E$, and

$$P \subseteq \{\langle E, a_i E\rangle \mid a_i \in A\} \cup \{\langle E, E b_i E\rangle \mid b_i \in B\} \cup \{\langle E, E c_i\rangle \mid c_i \in C\}.$$

The symbol $\Box$ is used to denote a kind of "bottom" terminal such as a variable name or a value. An example of an expression grammar is shown in Grammar $\Gamma_{5.1}$.

$$\begin{aligned} E &::= E + E \\ E &::= -E \\ E &::= E * E \end{aligned} \qquad (\Gamma_{5.1})$$

Note that there are no productions that include parentheses, e.g. $E ::= (E)$. These are omitted since they do not introduce additional ambiguities.

Note that the filters that will be described in this chapter are able to deal with these parenthesis productions.

## 5.2 Precedence correct parse trees

In order to disambiguate expression grammars, we need to have a definition of the associativities and precedences of the operators in the grammar. Therefore we will introduce two functions $assoc$ and $prio$ that give the associativity and precedence of an operator respectively. We will refer to a grammar together with precedence and associativity rules as a *precedence grammar*.

Given a valid input sentence according to a precedence grammar, we want to throw away parse trees that *do not* adhere to the precedence and associativity rules. Parse trees that *do* adhere to these rules are called *precedence correct* parse trees.

In Chapter 3 we have seen that precedence correct trees are often defined in terms of a specific parsing method, for example in YACC. In most cases, only an appeal is made to the precedence of mathematical operators like multiplication and addition or an unambiguous grammar with integrated precedence levels is given.

A formal definition of precedence correct parse trees is given by Aasa [1], that introduces a predicate $Pc_\Gamma$. Given a precedence grammar $\Gamma$, $Pc_\Gamma$ specifies when a tree $t$ is precedence correct. The predicate is defined in such a way that the parse trees constructed by an operator precedence parser [22] are precedence-correct. The definition introduces two different kinds of precedence weight of a parse tree; a left weight $Lw$, and a right weight $Rw$. Prefix operators have precedence only to the right, and postfix have precedence only to the left. Infix operators have precedences in both directions.

To disambiguate expression grammars we will use a variant of precede follow filtering. This filter is parser independent and correctly removes the trees which are not precedence correct, while keeping trees that are precedence correct. It seems that the precedence correct parse trees according to our precede follow filter are the same as those according to predicate $Pc_\Gamma$. A formal proof is left as future work.

In the next section we will show that two-level filtering is not sufficient. With two-level filtering we mean that we remove trees where some production occurs directly below another production. To completely disambiguate expression grammars we will introduce a precede-follow filter. Complete disambiguation means that we obtain exactly one parse tree for any input sentence given a precedence grammar.

## 5.3 Why two-level filtering is not sufficient

In Section 4.3.3 we have introduced the semantics of the priority and associativity rules in SDF. This mechanism uses a type of two-level filtering, meaning that it looks for a derivation with some production rule where one nonterminal in this rule is expanded into a new production rule. Two-level filtering is not sufficient for removing all invalid derivations. The following counterexample will show why this is the case.

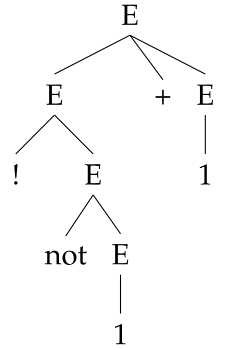
Given is Grammar $\Gamma_{5.2}$, with priorities $prio(E ::=! \ E) > prio(E ::= E + E) > prio(E ::= \text{not } E)$. Production $E ::= E + E$ is left-associative.

$$E ::= !\, E \mid E + E \mid \text{not } E \mid 1 \tag{$\Gamma_{5.2}$}$$

Consider input sentence "! not 1 + 1". For this input sentence it is not immediately obvious what the correct derivation is given the priorities. Derivation $(!(\text{not}(1 + 1)))$ does not adhere to the high priority of !, whereas derivation $(!(\text{not}(1)) + 1)$, does not adhere to the low priority of $+$.

We consider $(!(\text{not}(1 + 1)))$ to be the correct derivation. This example has also been described by Aasa [1], where the same derivation is chosen.

Now, consider the parse tree of the incorrect derivation:



In this parse tree there is no two-level filter applicable. Production $E ::=!E$ below production $E ::= E + E$ poses no problem since it has a higher priority. A similar argument applies to production rules $E ::=!E$ and $E ::= \text{not } E$. Therefore two-level filtering is not able to resolve the ambiguity in this case. In order to correctly remove these kind of derivations we will look at precede and follow restrictions. This disambiguation method allows us to disambiguate all expression grammars with unique single-character operators.

## 5.4 Precede and Follow Restrictions

The precede-follow restrictions mechanism has been introduced as a parse tree filter in Section 3.3.3. This mechanism can also be used as an SPPF filter to disambiguate any valid expression grammar $\Gamma_{expr}$. The precede restrictions ensure that a production $E ::= E\alpha$ may not be preceded by higher-priority operators. For instance, production $E ::= E * E$ is not allowed to precede $E ::= E + E$, since this would mean that the addition-operator would bind stronger. Productions of the form $E ::= \beta E$ may not be followed by higher-priority operators. The restrictions are also used to enforce the associativities of the operators.

Brink has given a formal proof for the situation where each operator has a unique priority, and the same operator does not occur as multiple types. We will refer to these assumptions as the *unique priority assumption*, and the *unique operator assumption*. Due to these assumptions, using only the PRECEDE$(u)$ and FOLLOW$(u)$ information of a node $u$ in an SPPF, a decision can be made whether $u$ is valid or invalid. If the node is invalid, all parse trees containing this node need to be removed from the SPPF. Finally we have the *single character assumption*, stating that an operator consists of a single character. This means that "+", or "!" is valid operator, but "++" or "&&" is not. Since operators cannot have the same priority, we can for instance not specify that addition and substraction have the same priority. Therefore after giving the original set of restriction, we will relax the assumptions to allow these kind of situations.

In order to perform disambiguation we need information about the relative priorities among the productions, and associativity information for the binary operators.

Assume a total function $assoc$ from each binary operator to $\{left, right\}$, representing an associativity assignment:

$$assoc : x \to \{left, right\} \quad \text{with } x \in B.$$

Assume a total function *prio* from each operator to $\mathbb{N}^+$, where a higher priority means that the operator has a higher precedence and hence binds stronger. Each operator has a unique priority:

$$prio : x \to \mathbb{N}^+ \quad \text{with } x \in A \cup B \cup C.$$

The precede and follow restrictions are defined in the following way:

$$\begin{aligned}
\mathcal{P} = \{(\langle E, Eb_iE\rangle, b_j) \quad & \mid b_i, b_j \in B : prio(b_i) < prio(b_j) \lor \\
& (b_i = b_j \land assoc(b_i) = left))\} \\
\cup\ \{(\langle E, Eb_iE\rangle, x) \quad & \mid b_i \in B, x \in A \cup C : prio(b_i) < prio(x)\} \\
\cup\ \{(\langle E, Ec_i\rangle, x) \quad & \mid c_i \in C, x \in A \cup B : prio(c_i) < prio(x)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F} = \{(\langle E, Eb_iE\rangle, b_j) \quad & \mid b_i, b_j \in B : prio(b_i) < prio(b_j) \lor \\
& (b_i = b_j \land assoc(b_i) = right))\} \\
\cup\ \{(\langle E, Eb_iE\rangle, x) \quad & \mid b_i \in B, x \in A \cup C : prio(b_i) < prio(x)\} \\
\cup\ \{(\langle E, a_iE\rangle, x) \quad & \mid a_i \in A, x \in B \cup C : prio(a_i) < prio(x)\}.
\end{aligned}$$

**Implementation** These restrictions can be implemented as an SPPF-filter. Assume we have an SPPF $s$ with $yield(s) = x_1 \dots x_m$. Given the input sentence, we can retrieve the terminal at position $i$ with $1 \leq i \leq m$ in constant time. Otherwise we have to determine the yield from the SPPF, which takes at most $\mathcal{O}(m^3)$.

Now we can traverse the SPPF in a depth-first of breath-first manner. When we visit a symbol node $u$ with label $(A, i, j)$, $\text{PRECEDE}(u) = x_{i-1}$ and $\text{FOLLOW}(u) = x_{j+1}$. In order to check the precede and follow restrictions for this node, we need the production of $A$. A symbol node has a packed node child for each possible alternative. Given a packed node $c$, which is a child of $u$, we extract the grammar slot from $c$ and obtain a production $p$, with $head(p) = A$ and $body(p) = \alpha$. Now we check whether $(\langle A, \alpha\rangle, x_{i-1}) \in \mathcal{P}$ or $(\langle A, \alpha\rangle, x_{j+1}) \in \mathcal{F}$. If a restriction applies, node $c$ becomes invalid and all parse trees containing $c$ need to be removed from the SPPF. This removal can be done using the node removal algorithm given in Section 4.3.1. After removing the parse trees containing $c$, the result will be a single smaller SPPF.

### 5.4.1 Disambiguate all single character operators

In order to support a wider set of expression grammars, we will remove the *unique operator assumption* and the *unique priority assumption*. Because we relax the assumptions, the precede and follow restrictions have to be adapted. Before showing the adapted restrictions, we will first show how to find the precede and follow values in an SPPF, and introduce the concepts of *precede production* and *follow production*.

Assume a total function $assoc$ from each binary production to the set $\{left, right\}$, representing an associativity assignment:

$$assoc : \langle E, Eb_iE\rangle \rightarrow \{left, right\} \quad \text{with } \langle E, Eb_iE\rangle \in P.$$

Assume a total function $prio$ from each production rule to $\mathbb{N}^+$, where a higher priority means that the operators binds stronger:

$$prio : \langle E, \gamma\rangle \rightarrow \mathbb{N}^+ \quad \text{with } \langle E, \gamma\rangle \in P.$$

Figure 5.1 shows how to obtain $\text{PRECEDE}\ (x)$ for some node $x$ in a parse tree with production $E ::= E\alpha$. Suppose $x$ has production $E ::= E\beta$. Then we first have to ignore a chain of ancestors of $x$ that also have a production of the form $E ::= E\gamma_i$, until we get to a node $y$ with a production of the form $E ::= \alpha E$. The value of $\text{PRECEDE}\ (x)$ is given by the rightmost symbol in $\alpha$ (i.e. $\text{LAST}(\alpha)$), and the precede production of $x$ is $y$. Note that $x$ is in the subtree of the rightmost $E$ of $y$, and in the subtree of the leftmost $E$ for nodes with a production of the shape $E :: E\gamma_i$.

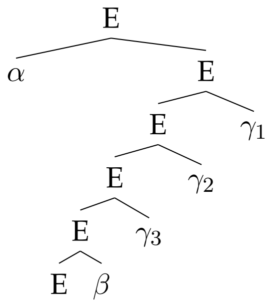
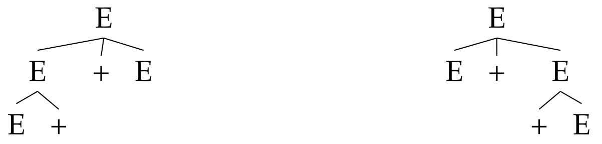


Figure 5.1: Example showing that the precede production of $E ::= E\beta$ is $E ::= \alpha E$).

As stated before, we allow the same symbol, say $+$, to occur both as unary prefix, unary postfix, and binary operator. This means that a sentence of the form $\Box + + + \Box$ can be interpreted as $(((\Box+)+) + \Box)$, $((\Box+) + (+\Box))$, and $(\Box + (+(+\Box)))$. In order to correctly disambiguate the additional ambiguities that might occur now, we need to add additional precede and follow restrictions.

Suppose we have an input sentence containing substring "++". Then we can parse this in two different ways. Either as prefix and binary operator, or as binary and postfix operator. Figure 5.2 shows these two situations.



(a) $prio(\langle E, E+\rangle) > prio(\langle E, +E\rangle)$ (b) $prio(\langle E, +E\rangle) > prio(\langle E, E+\rangle)$

Figure 5.2

To disambiguate this ambiguity we introduce the following two additional precede and follow restrictions, $\mathcal{P}_{op}$ and $\mathcal{F}_{op}$:

$$\begin{aligned} \mathcal{P}_{op} = \{ (\langle E, a_i E\rangle, \langle E, E b_i E\rangle) \mid a_i \in A, b_i \in B :\ a_i = b_i \wedge \\ (\exists c_i : c_i \in C : c_i = a_i\ \wedge prio(\langle E, a_i E\rangle) < prio(\langle E, E c_i\rangle)\ )\} \\ \mathcal{F}_{op} = \{ (\langle E, E c_i, \rangle, \langle E, E b_i E\rangle) \mid b_i \in B, c_i \in C :\ b_i = c_i \wedge \\ (\exists a_i : a_i \in A : a_i = b_i\ \wedge prio(\langle E, E c_i\rangle) < prio(\langle E, a_i E\rangle)\ )\} \end{aligned}$$

To get a unique derivation tree we need the requirement that an operator occurring as multiple types in the grammar has a different priority for each type. We will refer to this assumption as the *multiple type unique priority assumption*. Without this assumption, we cannot completely disambiguate certain ambiguities, like the one given in Figure 5.2.

Another assumption that we need is the *same priority same associativity assumption*, stating that productions with the same priority have the same associativity. Note that associativity only plays a role for productions of the form $\langle E, Eb_iE\rangle$ with $b_i \in B$. To see why we need this assumption consider two productions $p_+ = \langle E, E+E\rangle$ and $p_- = \langle E, E-E\rangle$ with $prio(p_+) = prio(p_-)$. Suppose that $assoc(p_+) = left$ and $assoc(p_-) = right$. Then given input sentence $E+E-E$, derivation $((E+E)-E)$ would violate the right associativity of $p_-$, and derivation $(E+(E-E))$ would violate the left associativity of $p_+$. Enforcing the precede and follow restrictions would mean that we would end up with no tree at all. If we require that these productions have the same associativity, there is a unique valid derivation for this input sentence.

For compactness in the definitions of $\mathcal{P}$ and $\mathcal{F}$ we will use $\gamma$ to denote the production $\langle E, \gamma\rangle$ (i.e. we omit the left hand side, which is always equal to $E$ in the case of expression grammars).

$$
\begin{array}{ll}
\mathcal{P} = \{(Eb_iE, Eb_jE) & \mid b_i, b_j \in B : prio(Eb_iE) < prio(Eb_jE) \vee \\
 & (prio(Eb_iE) = prio(Eb_jE) \\
 & \wedge\, assoc(Eb_iE) = left))\} \\
\quad\cup\ \{(Eb_iE, a_jE) & \mid b_i \in B, a_j \in A : prio(Eb_iE) < prio(a_jE)\} \\
\quad\cup\ \{(Ec_i, a_jE) & \mid c_i \in C, a_j \in A : prio(Ec_i) < prio(a_jE)\} \\
\quad\cup\ \{(Ec_i, Eb_iE) & \mid c_i \in C, b_j \in B : prio(Ec_i) < prio(Eb_jE)\} \\
\quad\cup\ \{a_iE, Eb_iE) & \mid a_i \in A, b_i \in B :\ a_i = b_i \wedge \\
 & (\exists c_i : c_i \in C : c_i = a_i\ \wedge prio(a_iE) < prio(Ec_i)\ )\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{F} = \{(Eb_iE, Eb_jE) & \mid b_i, b_j \in B : prio(Eb_iE) < prio(Eb_jE) \vee \\
 & (prio(Eb_iE) = prio(Eb_jE) \\
 & \wedge\, assoc(Eb_iE) = right))\} \\
\quad\cup\ \{(Eb_iE, Ec_j) & \mid b_i \in B, c_j \in C : prio(Eb_iE) < prio(Ec_j)\} \\
\quad\cup\ \{(a_iE, Eb_iE) & \mid a_i \in A, b_j \in B : prio(a_iE) < prio(Eb_jE)\} \\
\quad\cup\ \{(a_iE, Ec_i) & \mid a_i \in A, c_j \in C : prio(a_iE) < prio(Ec_j)\} \\
\quad\cup\ \{(Ec_i, Eb_iE) & \mid b_i \in B, c_i \in C :\ b_i = c_i \wedge \\
 & (\exists a_i : a_i = b_i\ \wedge a_i \in A : prio(Ec_i) < prio(a_iE)\ )\}
\end{array}
$$

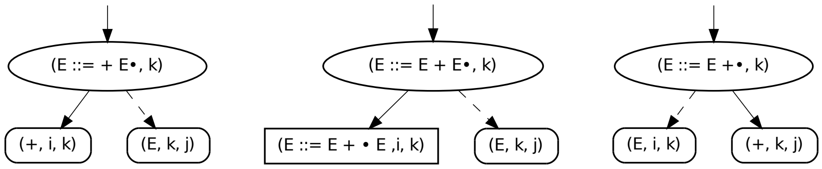
This set of precede and follow restrictions can correctly disambiguate all valid expression grammars adhering to the required assumptions. To prove this, we need to show that the relaxing the assumptions does not introduce new ambiguities.

In the relaxed situation, operators are allowed to occur as various types. The definitions for $\mathcal{P}$ and $\mathcal{F}$ can cope with this due to the fact that they use the precede production, rather than the precede symbol. Therefore a distinction can be made between operators occurring as different types. For instance, we can distinguish between a unary minus and binary minus when the precede symbol is a minus.

The assumption that two operators of the same type have a different priority is no longer needed. Given two productions $p_1, p_2 \in P$, all precede and follow rules look for a violation of $p_1 < p_2$, so introducing productions with the same priority is no problem. We do however need the *multiple type unique priority assumption*. Consider for example an operator, say $+$, occurring as prefix and binary operator. Then given input sentence $+1 + 1$ we cannot choose between $+(1 + 1)$ and $(+1) + 1$. As explained before we also need the *same priority same associativity assumption* for productions with binary operators.

**Implementation** The precede and follow restrictions that use precede and follow productions can also be implemented as an SPPF filter. The only difference is that we need to keep track of the precede and follow productions when traversing the SPPF. For this reason, we introduce the concept of an SPPF-walker $\mathcal{W}(p, f)$, with $p, f \in P$, that keeps track of these values while traversing the SPPF.

Whenever we follow an outgoing edge of a packed node to a symbol node, we update the precede or follow production. If the edge is the first child edge, we update $f$. If the edge is the last child edge, we update $p$. Figure 5.3 shows the various scenarios that are possible when considering expression grammars. For example, Figure 5.3a shows the situation where we follow the right arrow to nonterminal $E$. The precede production is updated to $\langle E, +E \rangle$, since the precede value is $+$.



(a) update $p$ (b) update $p$ (c) update $f$

Figure 5.3: Walker $\mathcal{W}(p, f)$ is updated when traversing the dashed edge.

Since a node can have multiple parents, we have to remember the path from the precede or follow production to the node. Applying a precede or follow filter means removing this path from the SPPF. After removing the path we obtain a new set of SPPFs. Part of the precede-follow violations will be present in each of these newly created SPPFs. This means that the same path filter can be applied multiple times, one time for each SPPF.

A solution to this problem is to separate the collection of the invalid paths from the actual path removal. First, the algorithm creates a list of all invalid paths by traversing the SPPF. Paths that are lower in the SPPF are appended to the end of the list. In the second step the invalid paths are removed. Using a list rather than a set improves the performance of the filter. If a path higher in the SPPF is removed, the part that may become unreachable can be larger than removing a path lower in the SPPF, which means that less copying will be needed.

The main bottleneck in the performance of the filter is the copying of the SPPF. By using persistent data structures, unnecessary copying of edges can be avoided.

### 5.4.2 Disambiguate Java expressions

In the previous section we have looked at disambiguating expression grammars with operators consisting of a single character. This assumption is however too rigid when we consider typical expressions in programming languages. Consider for instance the binary boolean operators $||$ and $\&\&$, or increment and decrement operators $++$ and $--$ in the C-family of programming languages.

$$\begin{array}{rllllll}
InfixOp ::= & \text{“}||\text{”} & \mid \text{“}\&\&\text{”} & \mid \text{“}|\text{”} & \mid \text{“}\hat{}\text{”} & \mid \text{“}\&\text{”} & \mid \\
 & \text{“}\ ==\ \text{”} & \mid \text{“}!=\ \text{”} & \mid \text{“}\ <\ \text{”} & \mid \text{“}\ >\ \text{”} & \mid \text{“}\ <=\ \text{”} & \mid \\
 & \text{“}\ >=\ \text{”} & \mid \text{“}\ <<\ \text{”} & \mid \text{“}\ >>\ \text{”} & \mid \text{“}\ >>>\ \text{”} & \mid \text{“}\ +\ \text{”} & \mid \\
 & \text{“}\ -\ \text{”} & \mid \text{“}\ *\ \text{”} & \mid \text{“}/\text{”} & \mid \text{“}\%\text{”} & & \\
PrefixOp ::= & \text{“}\ ++\text{”} & \mid \text{“}\ --\text{”} & \mid \text{“}!\text{”} & \mid \text{“}\ \sim\ \text{”} & \mid \text{“}\ +\ \text{”} & \mid \\
 & \text{“}\ \_\ \text{”} & & & & & \\
PostfixOp ::= & \text{“}\ ++\text{”} & \mid \text{“}\ --\text{”} & & & &
\end{array} \qquad (\Gamma_{5.3})$$

Grammar $\Gamma_{5.3}$ shows the operators that are supported in Java 7, taken from the Java Language Specification [25]. We can see that $+$ and $-$ occur as binary and prefix operator, and $++$ and $--$ as prefix and postfix operator. Allowing an operator to be multiple characters introduces new ambiguities. Consider as an example the prefix operators $+$ and $++$. Then we can parse an input sentence containing $+++$ in three ways, namely $+(++E)$, $+(+(+E))$, and $++(+E)$.

At this point we still have the *multiple type unique priority assumption*. This means that productions $\langle E, oE\rangle$, $\langle E, ooE\rangle$, $\langle E, EoE\rangle$ and $\langle E, Eoo\rangle$ all have unique priorities for $o \in \{+, -\}$. We will remove the single character assumption in order to allow disambiguation of Java expressions. The new ambiguities that arise given the Java operators can be classified into two different categories. The additional precede and follow restrictions are all introduced to disambiguate expressions containing $Eo^iE$ where $o^1 = o$, and $o^{n+1} = oo^n$.

**Same type operators** Assume we have two operators of the same type. We consider the case where both operators are postfix. We will use $*$ to denote any arbitrary production.

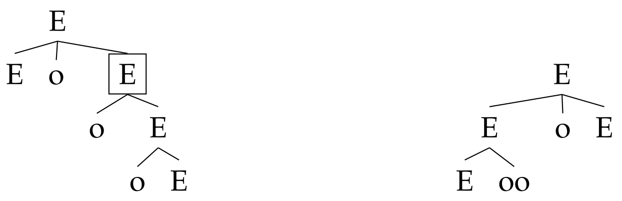
- If $prio(\langle E, ooE\rangle) > prio(\langle E, oE\rangle)$, then $(\langle E, oE\rangle, \langle E, oE\rangle) \in \mathcal{P}$.
- If $prio(\langle E, oE\rangle) > prio(\langle E, ooE\rangle)$, then $(\langle E, ooE\rangle, *) \in \mathcal{P}$.

**Different type operators** Assume we have two operators, where one is a prefix and one is a postfix operator. Let $|\langle A, \alpha\rangle|$ denote the number of symbols in production $A ::= \alpha$. Then given $prio(p) > prio(q)$ for $p, q \in P$, we consider the cases $|p| \leq |q|$ and $|p| > |q|$.

- case $|p| \leq |q|$:
  If $prio(\langle E, Eoo\rangle) > prio(\langle E, ooE\rangle)$, then $(\langle E, ooE\rangle, \langle E, EoE\rangle) \in \mathcal{P}$.
  If $prio(\langle E, ooE\rangle) > prio(\langle E, Eoo\rangle)$, then $(\langle E, Eoo\rangle, \langle E, EoE\rangle) \in \mathcal{P}$.

- case $|p| > |q|$:
  Suppose $prio(\langle E, Eoo\rangle) > prio(\langle E, oE\rangle$. Then we want to remove parse trees of the form shown in Figure 5.4a. The two o-characters that are parsed as postfix operators can also be parsed as a single prefix operator, see Figure 5.4b. Note that between the postfix productions there can be an arbitrary number of binary and prefix operators. If we look at the marked E in Figure 5.4a then we know that we have a production $p = \langle E, oE\rangle$ with precede production $\langle E, EoE\rangle$. Suppose that the $o$-character of $p$ is at position $i$ of the input string. Then we can check whether the character at position $i + 1$ is an $o$, but not to which production it belongs using only the precede and follow productions of $E$. This may either be a binary operator or a prefix operator. Therefore we cannot create a precede-follow restriction that can handle this case.

  In the same way, restriction $prio(\langle E, ooE\rangle) > prio(\langle E, Eo\rangle$ can also not be enforced using precede-follow filtering. Note that these two cases can be filtered using a custom path filter.



(a) invalid

(b) valid

Figure 5.4: Invalid and valid parse trees for $prio(\langle E, Eoo\rangle) > prio(\langle E, oE\rangle)$.

The primary Java compiler, `javac` [37], uses a hand-written LL parser for parsing Java code. As stated in the preamble of the parser code, for efficiency reasons, an operator precedence scheme is used for parsing binary operation expressions. If we look at expressions of the shape $E +^i E$, then the Java parser will correctly parse $E + E$ and $E + + + E$, where $E + + + E$ is parsed as $(E + +) + E$. For $i \neq 1$ and $i \neq 3$, $E +^i E$ will result in a syntax error. So, the precede-follow mechanism with the given restrictions can parse all valid Java expressions (that is, with respect to by the grammar and the language specification), except $E + + + E$ and $E - - - E$. As mentioned before we can introduce a custom filter for these cases using an SPPF-walker.

# Chapter 6

# Disambiguation of Mixfix Expressions

In the previous chapter we have looked at the disambiguation of expression grammars with unary and binary operators. This chapter illustrates the applicability of the SPPF filters described in Chapter 4 by applying it on a more general class of expressions that includes *mixfix* operators. A mixfix (also known as distfix) operator can have several name parts, and multiple operator holes. Each hole is a position in which an argument expression is expected. For instance, $if\ \_\ then\ \_\ else\ \_$ is a prefix mixfix operator.

To illustrate the applicability of the SPPF filters, we will look at ambiguities that arise in a grammar containing mixfix operators. For this purpose we look at the mCRL2 language [20], a formal specification language for describing concurrent discrete event systems.

## 6.1 Mixfix expressions

Before delving into causes of ambiguity in mixfix expressions, we will first describe our concept of mixfix operators, and the mixfix expressions that can be formed using these operators. We will follow the definitions given by Wieland [54].

**Definition 6.1.1.** Every mixfix operator pattern is a sequence of $n$ separators, interleaved by $n-1$ operand placeholders $s_0\ \_\ s_1\ \ldots\ \_\ s_n$.

**Definition 6.1.2.** If a separator between two operand placeholders is the empty sequence, these operands are called *adjacent*.

As an example consider the following mixfix operator pattern: $and\ \_\ \_$. Here the two operands following $and$ are called *adjacent*.

**Definition 6.1.3.** The *arity* of a mixfix operator pattern is equal to the number of placeholders in the pattern.

**Definition 6.1.4.** If a separator at the beginning of a pattern is empty, the operator is called *left-open*, otherwise it is called *left-closed*. An operator with an empty last separator is called *right-open*. If the last separator is non-empty, the operator is called *left-closed*.

Left-closed operators are also called *prefix* operators, while right-closed operators are also called *postfix* operators. Operators which are both prefix and postfix are called *closed* operators. *Infix* operators are both left-open, and right-open. To illustrate the fixity of operators, consider the following examples:

- infix: _ ⊢ _ : _,
- prefix: `if _ then _ else _`,
- postfix: _ [ _ ],
- closed: { _ }.

In the previous chapter we have seen unary prefix and unary postfix operators, as well as binary infix operators. In this chapter we will also look at operators that have a higher arity, which are part of the mCRL2 grammar.

A *mixfix expression* is a sequence of the separators of the operator pattern, interleaved with operand expressions between the separators.

## 6.2 Causes of ambiguity in mixfix expressions

In mixfix expressions, there are various causes of ambiguity [54]. In this section we will look at syntactical ambiguities. Besides syntactical ambiguities, there can also be semantic ambiguities arising from type ambiguities, for instance when the language allows some form of polymorphism. Since these ambiguities need to be resolved by a type inference system, we will not consider them further here.

For the types of ambiguity that occur in mCRL2, we will investigate how they can be resolved.

### 6.2.1 Shared separator tokens

The first cause of ambiguity in mixfix expressions is mixfix operators sharing one or more separator tokens. The notorious "dangling else" ambiguity is an example of an ambiguity that falls in this category. This ambiguity occurs in nested `if-then-else` expressions, where the inner `else` can be interpreted as belonging to either the inner `if`, or the outer `if`. Given mixfix operators `if _ then _`, and `if _ then _ else _`, there are two derivations for expression `if _ then _ if _ then _ else`:

1. `(if _ then (if _ then _ ) else _ )`
2. `(if _ then (if _ then _ else _ ))`

Suppose we have two productions $p = \langle A, \alpha \rangle$ and $q = \langle A, \alpha\beta \rangle$ with $\alpha, \beta \neq \varepsilon$. Then we can prefer one of the productions over the other by using a prefer filter. In our example, if we want the second derivation, we prefer `if _ then _`over `if _ then _ else _`.

### 6.2.2 Adjacent operands

The second cause of ambiguity can occur when one or more mixfix expressions have adjacent operands. In these situations, it is not clear where the left operand ends and where the right operand starts. For instance consider expression *abc*, with operator `a _ _ c`, where the first and second operand expressions can be *b* or $\varepsilon$. Then we have derivations `a _ _ c` $\Rightarrow$ `a b` $\varepsilon$ `c` $\Rightarrow$ `a b c`, and `a _ _ c` $\Rightarrow$ `a` $\varepsilon$ `b c` $\Rightarrow$ `a b c`.

In the mCRL2 grammar there are some adjacent operands in production rules containing $A+$ for some $A \in N$. This could potentially lead to ambiguities. However, for each instance of $A$ we have that the production rule is *right-closed* with a semicolon. This means that each occurrence of $A$ is well separated, and no adjacent operand ambiguity is introduced.

### 6.2.3 Left-open vs. right-open operators

When both left-open and right-open operators are present, another type of ambiguity can arise. This type of ambiguity is related to the relative precedence and associativity of the operators. In Chapter 5, we have already seen various examples of this type of ambiguity. As an example, consider the binary infix operators $\_+\_$ and $\_*\_$, which are both left-open and right-open. Then expression $1 + 2 * 3$ can be derived in two ways, namely $(1 + (2 * 3))$, and $((1 + 2) * 3)$. In the first derivation, the $*$-operator has precedence over the $+$-operator, while in the second derivation this is the other way round.

The attentive reader might have noticed a pattern in the set of precede and follow restrictions given in Chapter 5. If we have two rules, where one is left-open and the other right-open, we introduce a precede or follow restriction depending on the relative priorities.

Assume a partial function *prio* from production rules to $\mathbb{N}^+$, where a higher priority means that the operator binds stronger.

$$prio : \langle E, \gamma \rangle \rightarrow \mathbb{N}^+ \quad \text{with } \langle E, \gamma \rangle \in P$$

Assume a partial function *assoc* from production rules to $\mathbb{N}^+$, specifying the associativity of infix operators.

$$prio : \langle E, \gamma \rangle \rightarrow \{left, right, non\text{-}assoc\} \quad \text{with } \langle E, \gamma \rangle \in P$$

**Associativity** Let $p$ and $q$ be production rules (possibly the same) of an infix operator, satisfying the following properties.

1. $p = \langle E, A\beta B\rangle$, with $\beta \neq \varepsilon$
2. $q = \langle E, C\gamma D\rangle$, with $\gamma \neq \varepsilon$
3. $p$ and $q$ do not contain adjacent operands
4. $prio(p) = prio(q)$
5. $assoc(p) = assoc(q)$
6. $A \overset{*}{\Rightarrow} E$, $B \overset{*}{\Rightarrow} E$
7. $C \overset{*}{\Rightarrow} E$, $D \overset{*}{\Rightarrow} E$

Depending on the associativity of $p$, given by $assoc(p)$, we obtain the following precede and following restrictions.

1. left: $(\langle E, A\beta B\rangle, \langle E, C\gamma D\rangle) \in \mathcal{P}$
2. right: $(\langle E, A\beta B\rangle, \langle E, C\gamma D\rangle) \in \mathcal{F}$
3. non-assoc: $(\langle E, A\beta B\rangle, \langle E, C\gamma D\rangle) \in \mathcal{P}$, $(\langle E, A\beta B\rangle, \langle E, C\gamma + D\rangle) \in \mathcal{F}$

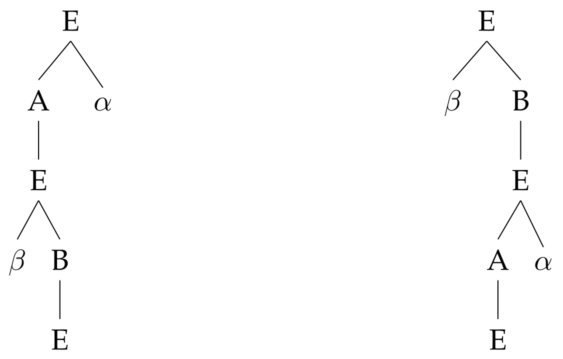
Note that we do not require that the outer nonterminals are the same as the head of the production rule. We do however need that they derive the head of the production rule. By this specification, the filtering will also apply to injections (or chain rules), where $E$ is derived after a number of derivation steps. Recall that the associativity of two productions must be the same if the priority is also the same, otherwise we could end up with no derivation at all (see Section 5.4.1).

**Priority** Let $p$ and $q$ be production rules, where $p$ is left-open and $q$ is right open, satisfying the following properties.

1. $p = \langle E, A\alpha\rangle$, with $\alpha \neq \varepsilon$
2. $q = \langle E, \beta B\rangle$, with $\beta \neq \varepsilon$
3. $A \overset{*}{\Rightarrow} E$
4. $B \overset{*}{\Rightarrow} E$

Depending on the relative priorities of the production rules, we obtain the following precede and follow restrictions.

1. If $prio(p) < prio(q)$, then $\langle E, A\alpha\rangle, \langle E, \beta B\rangle \in \mathcal{P}$.



(a) Valid tree when $prio(\langle E, \beta B\rangle) > prio(\langle E, A\alpha\rangle)$

(b) Valid tree when $prio(\langle E, A\alpha\rangle) > prio(\langle E, \beta B\rangle)$

Figure 6.1: Derivation trees when a grammar contains both a left-open and right-open rule.

2. If $prio(p) > prio(q)$, then $\langle E, \beta B\rangle, \langle E, A\alpha\rangle \in \mathcal{F}$.

Given two such rules $\langle E, A\alpha\rangle$ and $\langle E, \beta B\rangle$, there are two different derivation trees, shown in Figure 6.1.

To illustrate the generation of the restrictions, consider the following grammar:

$$\begin{array}{lll} DataExpr & ::= \text{“}forall\text{”}\ Id\ \text{“.”}\ DataExpr & \{1\} \\ DataExpr & ::= DataExpr\ \text{“==”}\ DataExpr & \{\text{left}, 5\} \\ DataExpr & ::= Id & \\ Id & ::= [a\text{-}z]+ & \end{array} \qquad (\Gamma_{6.1})$$

The first production rule is right-open, and the second production rule is left-open. Since the open nonterminal is $DataExpr$ in both cases, we need to add a restriction. Because the first production rule has a lower priority we need a precede restriction.

$$(\langle DataExpr, \text{“}forall\text{”}\ Id\ \text{“.”}\ DataExpr\rangle,$$
$$\langle DataExpr, DataExpr\ \text{“==”}\ DataExpr\rangle) \in \mathcal{P}$$

Since the second production rule is left-associative, we obtain the following precede restriction:

$$(\langle DataExpr, DataExpr\ \text{“==”}\ DataExpr\rangle,$$
$$\langle DataExpr, DataExpr\ \text{“==”}\ DataExpr\rangle) \in \mathcal{P}$$

Note that the restrictions are only applicable if a priority between the rules is defined. We do not require that every production rule has a priority.

For instance, closed expressions are not part of any ambiguity and therefore do not need a priority. In the case of two distinct infix operators, we obtain two restrictions if their relative priorities are defined.

The precede and follow restrictions are used by a new filter, the left-open right-open filter (LORO-filter). This filter removes ambiguities related to the relative precedence and associativity of left- and right-open operators. The LORO-filter resembles the precede/follow production filter, but uses an additional check to avoid filtering below closed operators. The implementation of this filter is discussed in Section 6.5. We will now look at the mCRL2 language in more detail, and the types of ambiguities that occur in the various parts of the language.

## 6.3 mCRL2 specification language

As described in the introduction of this chapter, mCRL2 is a formal specification language that can be used for describing concurrent discrete event systems. The behavioral part of the language is based on process algebra (Algebra of Communicating Processes [7]). The specified behavior can be simulated, visualized or verified against its requirements using the mCRL2 toolkit. In this section we will give a brief introduction of the various components present in the mCRL2 language.

The basis for mCRL2 specifications are abstract data types, with operations defined on them. These abstract data types are present in processes, and actions that may carry a number of parameters. Processes are the most important entities in a process specification. They are used to describe the behavior of some component or system. In the remainder of this section we will look at data, actions, and processes in more detail.

### 6.3.1 Data types

The underlying theory of mCRL2 data types is abstract data types. These types consists of:

- sorts and operations on these sorts;
- equations on terms, composed from operations and variables, where the terms are of the same sort.

New sorts are declared using the keyword **sort**. For a sort, constructor functions can be defined using the keyword **cons**. These functions specify exactly all elements in the sort. For instance, we can construct the sort *Nat* representing the natural numbers in the following way:

**sort** $Nat$;
**cons** $zero : Nat$;
    $successor : Nat \to Nat$;

Any natural number can now be denoted by an expression of the form:

$$successor(successor(\ldots successor(zero)\ldots)).$$

Auxilary functions can be defined using the keywords **map** and **eqn**. Using **map**, we can define the typing of the functions, and the actual equations are given using **eqn**. Variables are introduced by the keyword **var**, or keyword **glob** for global variables. To illustrate this, we extend our definition of the natural numbers with a plus operator:

```
map plus, times : Nat × Nat → Nat;
var  n, m : Nat;
eqn plus(n, zero) = n;
      plus(n, successor(m)) = successor(plus(n, m));
```

### 6.3.2 Processes and Actions

The data types and their equations can be used in the specifications of processes. A process is defined using the keyword **proc**. Processes can perform actions, which are defined using the keyword **act**. Processes can be composed to form new processes. A system usually consists of several processes in parallel.

As an example, consider the specification of a clock that counts its ticks:

```
act tick;
proc Clock(n : Nat) = tick.Clock(n + 1);
 init  Clock(0);
```

Here, **init** is used to denote the initial state of the clock.

## 6.4 Parsing mCRL2 specifications

When looking at the mCRL2 grammar, one of the first observations is that the grammar actually consists of various components. The grammar contains the mCRL2 specification grammar, but also the grammar for specifying requirements on the modeled system. The requirements can be formulated using an action formula (`ActFrm` in the grammar), a Boolean equation system (`BesSpec`), or a Parametrized boolean equation system (`PbesSpec`). We will focus our attention only on the mCRL2 specifications. This corresponds to lines 1-258 and 390-400 in the DParser grammar for mCRL2.

At several points, the grammar has been modified over time in order to remove certain ambiguities. To find the intended meaning behind the grammar with respect to precedences and associativities, we use the original syntax definition given by Groote and Mousavi [26] and the mCRL2

SVN repository [1].

The current toolset of mCRL2 uses DParser [38] for parsing specifications. DParser is a scannerless GLR parser based on the Tomita algorithm. It allows specification of priorities and associativities. Operator priorities and associativities in DParser are specified on the reduction which creates the token. Besides operator priorities, rule priorities can be specified. Rule priorities specify the priority of the reduction itself. The actual implementation of the disambiguation method for these disambiguation rules is not very clear. According to the manual it is a combination of shift/reduce conflict resolving, stack comparisons, and heuristics. Appendix A contains the full DParser grammar for mCRL2.

Because the mCRL2 grammar is in EBNF format and the GLL implementation we are using requires BNF, we transform the grammar to BNF. The following rules describe this transformation.

1. Replace a group ($\alpha$), where $\alpha$ denotes a sequence of symbols, by a nonterminal $X$, and add the production rule $\langle X, \alpha\rangle$ to the set of production rules.

2. Replace a symbol $X?$ by a nonterminal $Y$, and add production rules $\langle Y, X\rangle$ and $\langle Y, \varepsilon\rangle$ to the set of production rules.

3. Replace a symbol $X*$ by a nonterminal $Y$, and add production rules $\langle Y, XY\rangle$ and $\langle Y, \varepsilon\rangle$ to the set of production rules.

4. Replace a symbol $X+$ by a nonterminal $Y$, and add production rules $\langle Y, XY\rangle$ and $\langle Y, X\rangle$ to the set of production rules.

By replacing $X*$ by a nonterminal $Y$, and production rules $\langle Y, XY\rangle$ and $\langle Y, \varepsilon\rangle$ we have made the choice for right recursion. Another option would be to replace $X*$ by a nonterminal $Y$ and production rules $\langle Y, YX\rangle$ and $\langle Y, \varepsilon\rangle$. This decision might have influence on whether precede or follow restrictions will be generated containing the newly introduced production rules.

The BNF grammar can be found in Appendix B.

In the remainder of this section, we will look at the various types of ambiguities that occur in mCRL2 specifications, and how they can be resolved. It is important to keep in mind that the order of applying the filters to resolve the ambiguities might yield different results. This will be described in more detail in Section 6.6.

[1] https://svn.win.tue.nl/trac/MCRL2/browser/trunk/doc/specs/mcrl2-syntax.g

### 6.4.1 Restricted keywords

The following set $K$ contains all predefined keywords in the mCRL2 language. These keywords cannot be used as identifiers.

$$\begin{aligned}K = \{&\texttt{sort, cons, map, var, eqn, act, proc, init, nil,}\\ &\texttt{delta, tau, sum, block, allow, hide, rename, comm,}\\ &\texttt{struct, Bool, Pos, Nat, Int, Real, List, Set, Bag,}\\ &\texttt{FSet, FBag, true, false, whr, end, lambda, forall,}\\ &\texttt{exists, div, mod, in}\}\end{aligned}$$

In the mCRL2 grammar all identifiers are derived from $Id$, which is defined by the following production:

$$Id ::= [A\text{-}Za\text{-}z\_][A\text{-}Za\text{-}z\_0 - 9']* \qquad (\Gamma_{6.2})$$

All parse trees where a restricted keyword is derived from an identifier must to be removed. This can be done by removing all parse trees having the path $\langle (Id, i, j), (Id ::= [A\text{-}Za\text{-}z\_][A\text{-}Za\text{-}z\_0 - 9'] * \cdot, k), (keyw, i, j)\rangle$ with $keyw \in K$. In our GLL parser for mCRL2 we filter these trees during parsing, by not executing the *pop* if the next terminal is present in set $K$.

### 6.4.2 Ambiguities in sort expressions

Sort expressions are used to declare new sorts. In the grammar, `SortSpec` is used for specifying the typing of a sort. Consider the following example:

**sort** A = $Nat \times Nat \to Nat \to Nat$;

In mCRL2, → operator is right-associative and has a lower priority than the × operator. Therefore, brackets in the example should be added in the following way:

**sort** A = $(Nat \times Nat) \to (Nat \to Nat)$;

The ambiguities that occur in sort expressions relate to the precedences and associativities of these two operators. Since the ambiguity occurs inside $SortExpr$, we will focus only on this part of the grammar. In our example, $SortExpr$ derives the part $Nat \times Nat \to Nat \to Nat$.

We have distilled the essentials needed from the grammar to allow a compact description of the ambiguity related to the precedence of the → and × operator. The compact grammar is shown in Grammar $\Gamma_{6.3}$.

$$\begin{aligned}SortExpr &::= \text{`}Nat\text{'} \mid Domain \text{ `} \to \text{' } SortExpr\\ Domain &::= SortExprList\\ SortExprList &::= SortExpr\ (\text{`}\#\text{'}\ SortExpr)*\end{aligned} \qquad (\Gamma_{6.3})$$

After conversion to BNF we obtain the following grammar:

$$\begin{aligned} \mathit{SortExpr} &::= \text{`}\mathit{Nat}\text{'} \mid \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr} \\ \mathit{Domain} &::= \mathit{SortExprList} \\ \mathit{SortExprList} &::= \mathit{SortExpr}\ \mathit{SortExprStar} \\ \mathit{SortExprStar} &::= \text{`\#'}\ \mathit{SortExpr}\ \mathit{SortExprStar} \mid \varepsilon \end{aligned} \qquad (\Gamma_{6.4})$$

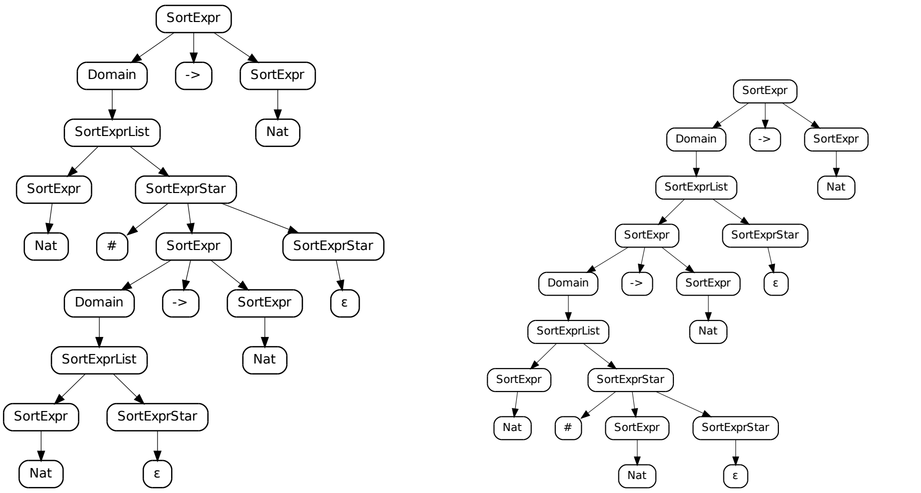
In order to correctly disambiguate sort expressions, we need filters that enforce the precedence of # over → and the right-associativity of →. Figure 6.2 shows the invalid parse trees for our example. We can make the following observations:

- Production $\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle$ may never have a precede production containing the #-operator (i.e. $\langle \mathit{SortExprStar}, \text{`\#'}\ \mathit{SortExpr}\ \mathit{SortExprStar}\rangle$ in our BNF grammar). This case corresponds to the higher precedence of # over →, filtering trees where → is nested below a #. Furthermore, production $\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle$ may not have a follow production containing the #-operator, since this would also violate the higher precedence of # over →.
- Production $\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle$ may never have a follow production equal to $\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle$. This case corresponds to the right-associativity of →, filtering trees where → is left-associative.

Figures 6.2a and 6.2b show parse trees corresponding to these cases in our example. There are no other ambiguities in sort expressions. So, in order to remove all invalid parse trees that have an incorrect sort expression we can use the left-open right-open filter that uses these following restrictions.

$$\begin{aligned} &(\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle, \\ &\langle \mathit{SortExprStar}, \text{`\#'}\ \mathit{SortExpr}\ \mathit{SortExprStar}\rangle) \in \mathcal{P} \\ &(\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle, \\ &\langle \mathit{SortExprStar}, \text{`\#'}\ \mathit{SortExpr}\ \mathit{SortExprStar}\rangle) \in \mathcal{F} \\ &(\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle, \\ &\langle \mathit{SortExpr}, \mathit{Domain}\ \text{`} \to \text{'}\ \mathit{SortExpr}\rangle) \in \mathcal{F} \end{aligned}$$

Note: In our definition of precede and follow productions we will ignore productions that derive $\varepsilon$. The $\varepsilon$-productions are introduced by our conversion to BNF. Ignoring these productions can be done by looking at the extent or pivot of the nodes in the path. Figure 6.3 shows an example of this.



(a) Parse tree corresponding to derivation $((Nat \# (Nat \to Nat)) \to Nat)$

(b) Parse tree corresponding to derivation $((Nat \# Nat) \to Nat) \to Nat)$

Figure 6.2: Two invalid parse trees corresponding to input $Nat \times Nat \to Nat \to Nat$.

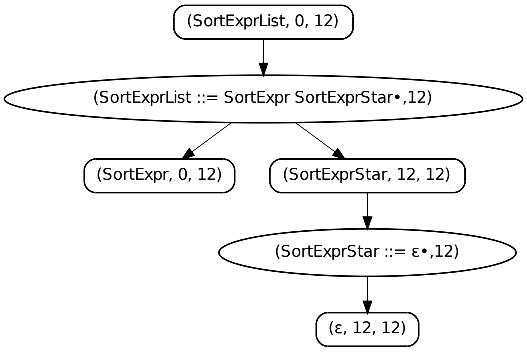


Figure 6.3: Symbol node (SortExprStar,12,12) derives $\varepsilon$, which can be inferred from the extend of node $(SortExprList, 0, 12)$ and the pivot of $(SortExprList ::= SortExprSortExprStar\cdot, 12)$.

### 6.4.3 Ambiguities in data expressions

Data expressions are used to describe data types with their associated functions and operators. The developer documentation of mCRL2 gives the priorities and associativities of the operators used in data expressions [34]:

> The prefix operators have the highest priority, followed by infix operators, followed by the lambda operator together with universal and existential quantification, followed by the where clause.

The precedences and associativities of the infix operators are shown in Table 6.1.

| operators | associativity |
|---|---|
| `*, .` | left |
| `/, div, mod` | left |
| `+, -` | left |
| `\|>` | right |
| `<\|` | left |
| `++` | left |
| `<, >, <=, >=, in` | none |
| `==, !=` | right |
| `&&, \|\|` | right |
| `=>` | right |

Table 6.1: Precedence of infix operators

Given this information and the DParser grammar of mCRL2, we can derive a set of precede and follow production restrictions. Using these restrictions we can remove all parse trees from the SPPF that violate one of the precedence or associativity rules, again by using the left-open right-open filter. There are no ambiguities arising from shared separator tokens.

### 6.4.4 Ambiguities in process expressions

Processes are used to describe the behavior of some system or component. Process expressions allow the description of these processes. In these expressions data expressions and action expressions are used.

The process expressions also have ambiguities related to the associativity and priority of operators. They can be resolved by generating the precede and follow restrictions as described in Section 6.2.3 and using a left-open right-open filter for removing undesired derivations.

In process expressions, we also have an ambiguity arising from shared separator tokens. More specifically, it is an instance of the dangling else ambiguity. The ambiguity arises because of the following two rules:

1. $ProcExpr ::= DataExpr \rightarrow ProcExpr \diamond ProcExpr$
2. $ProcExpr ::= DataExpr \rightarrow ProcExpr$

According to the semantics of mCRL2, the dangling else should be attached to the nearest by $\rightarrow$ operator. Therefore, we add a *prefer* attribute to the production without the else (in our case the diamond), and a *avoid* attribute to the production with the else. The prefer filter looks for a symbol node with label $(ProcExpr, i, j)$ for some $i, j$, that has a packed node child with label $(ProcExpr ::= DataExpr \rightarrow ProcExpr, l)$ for some $k$. All other packed node children will be removed, which have labels of the form $(ProcExpr ::= DataExpr \rightarrow ProcExpr \diamond ProcExpr, k)$ for some $l$.

## 6.5 Left-open right-open filter

Associativity and priority ambiguities occurring in mCRL2 are resolved by using a left-open right-open (LORO) filter. This filter is implemented by an SPPF-walker (cf. Section 5.4.1). SPPF-walker $\mathcal{W}(node, p, f, rightNullable)$ keeps track of four variables:

- $node \in V$: current node in SPPF $\mathcal{S}(V, E)$
- $p$: path to precede production
- $f$: path to follow production
- $rightNullable \in \mathbb{B}$: indicates whether the current nonterminal in a production is right-open. The purpose of this variable is explained in more detail below.

### 6.5.1 Hidden openness with nullable nonterminals

In Section 6.4.2 we have seen a situation where nonterminals that derive $\varepsilon$ may introduce new ambiguities. If we have a rule of the shape $\langle E, \alpha AB \rangle$, and $B \overset{*}{\Rightarrow} \varepsilon$, then the production rule becomes right-open. In our EBNF to BNF conversion there are several cases of this. For instance

$$SortExprList ::= SortExpr(\text{‘}\#\text{’}\ SortExpr\ SortExprStar)* \qquad (\Gamma_{6.5})$$

is translated into

$$\begin{aligned} &SortExprList ::= SortExpr\ SortExprStar \\ &SortExprStar ::= \text{‘}\#\text{’}\ SortExpr\ SortExprStar \mid \varepsilon. \end{aligned} \qquad (\Gamma_{6.6})$$

If nonterminal $SortExprStar$ derives $\varepsilon$, $SortExprList$ becomes right-open (besides being already left-open). Furthermore, nonterminal $SortExpr$ in

$SortExprList$ will have the same follow production as $SortExprList$ instead of $SortExprList$ itself.

During the traversal of the SPPF we keep track of variable $rightNullable$. If we visit a nonterminal symbol node $A$, the value of $rightNullable$ indicates whether the part of the production rule right from $A$ derives epsilon. If this is the case, $A$ becomes right-open. We do not need a variable $leftNullable$, since the binarization of the production is done from the left. Whenever a nonterminal symbol node is the left child, it is left-open. If it is the right child, we can look at the extent of the left symbol node child. If the left extent is equal to the right extent, then it derives $\varepsilon$.

To show how the variable $rightNullable$ is updated, consider a production rule of the shape $\langle E, \alpha\gamma\delta\beta\rangle$ with $\gamma, \delta \in T \cup N$. Assume that we have a packed node with two children, shown in Figure 6.4. If the packed node has only one child, the value of $rightNullable$ does not change.

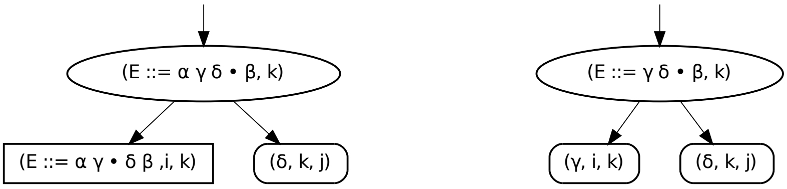


Figure 6.4: A packed node with two children.

The packed node has a pivot between $\delta$ and $\beta$. Variable $rightNullable$ indicates whether $\beta$ is nullable. If we now go down to the left child, we have to check whether $\delta\beta \overset{*}{\Rightarrow} \varepsilon$:

$$\delta\beta \overset{*}{\Rightarrow} \varepsilon \iff \delta \overset{*}{\Rightarrow} \varepsilon \land \beta \overset{*}{\Rightarrow} \varepsilon \iff k = j \land rightNullable.$$

If we go down to the right child, $rightNullable$ indicates whether $\beta$ is nullable.

To see whether $\alpha\gamma$ is nullable, we only have to look whether $i = k$.

### 6.5.2 Applicability of restrictions

Compared to the expression grammars discussed in Chapter 5, we will use the precede and follow restrictions in a slightly different way. This is due to the fact that mCRL2 has operators of higher arity, where a production rule can have both an open and closed nonterminal. An example of this is the function application operator; $DataExpr$ ::= $DataExpr(DataExprList)$. Here $DataExpr$ is open, but $DataExprList$ is guarded by braces.

Now consider the following grammar, where the associativity and priority of the operator is given between curly brackets:

$$\begin{aligned} E &::= E(E) \quad \{\text{left, } 13\} \\ E &::= E + E \quad \{\text{left, } 10\} \\ E &::= 1 \end{aligned} \qquad (\Gamma_{6.7})$$

Given this grammar, we would generate two restrictions:

$$(\langle E, E+E\rangle, \langle E, E(E)\rangle) \in \mathcal{F}, \quad (\langle E, E+E\rangle, \langle E, E+E\rangle) \in \mathcal{P}.$$

Implementing the filter in the same way as in Chapter 5 removes all invalid parse trees. However, it will also remove *valid* parse trees. Consider the input $1(1+1)$. Clearly this input has only one left-most derivation ($E \Rightarrow E(E) \Rightarrow 1(E) \Rightarrow 1(E+E) \overset{+}{\Rightarrow} 1(1+1)$). However, the follow production of $\langle E, E+E\rangle$ in the tree is $\langle E, E(E)\rangle$ due to the closing brace. This means that the tree corresponding to $1(1+1)$ will be incorrectly removed.

The reason that the restriction should not be applied is that the second $E$ in $\langle E, E(E)\rangle$ is guarded by braces. In the case that a nonterminal is guarded, we do not want to enforce the restrictions. If both $leftNullable$ and $rightNullable$ are false, the nonterminal is guarded and the precede and follow production paths need to be reset. This ensures that valid trees are not removed by the filter.

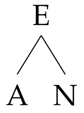
### 6.5.3 Updating the walker

During the traversal of the SPPF we have a function *UpdateWalker* that updates the SPPF-walker given the current edge that is taken.

```
**Algorithm** *UpdateWalker*($\mathcal{W}, u, v$)
**Input:** walker $\mathcal{W}$, packed node $u$, node $v$ with $(u,v) \in E$ (in SPPF $\mathcal{S}(V,E)$)
1.  $\mathcal{W}' \leftarrow \mathcal{W}$
2.  if $|parent.children| = 1$ (* precede/follow productions do not change *)
3.    then return $\mathcal{W}'$
4.    else
5.        determine $leftNullable$ and $rightNullable$
6.        if $v.type = symbol$
7.          then if $\neg leftNullable \wedge \neg rightNullable$
8.                 then (* guarded literal *)
9.                      $\mathcal{W}'.p = [\,]; \mathcal{W}'.f = [\,]$
10.                else if $\neg leftNullable$
11.                       then $\mathcal{W}'.p = [u]$;
12.                     if $\neg rightNullable$
13.                       then $\mathcal{W}'.f = [u]$;
14.       if $\mathcal{W}'.p \neq [\,]$
```

```
            then $\mathcal{W}'.p \leftarrow \mathcal{W}'.p \mathbin{+\!\!+} v$
        if $\mathcal{W}'.f \neq [\,]$
            then $\mathcal{W}'.f \leftarrow \mathcal{W}'.f \mathbin{+\!\!+} v$
        update $\mathcal{W}'.rightNullable$
        return $\mathcal{W}'$
```

*Note:* For the left-open right-open filter we have made the decision that a production not having any terminals can be a precede or follow production of a symbol node. In the following tree, $A$ has follow production $\langle E, AN\rangle$ under the condition that $N \overset{*}{\not\Rightarrow} \varepsilon$:



This is what one would expect, since given the extent of $A$ in the input string, the next character in the input string is derived from $N$.

### 6.5.4 Left-open right-open filter pseudo code

The left-open right-open filter consists of two parts. Algorithm *RemoveFirstInvalidPath* traverses the SPPF with an SPPF-walker to find paths that need to be removed. As soon as such a path is found, the algorithm invokes Algorithm *RemoveParseTreesP* to remove all parse trees containing this path, and returns the set of resulting SPPFs. Algorithm *LeftOpenRightOpenFilter* iteratively invokes Algorithm *RemoveFirstInvalidPath* until the set of resulting SPPFs does not change anymore.

**Algorithm** *LeftOpenRightOpenFilter*($\mathcal{S}, \mathcal{P}, \mathcal{F}$)
**Input:** $\mathcal{S}$: SPPF to be filtered

```
$Q \leftarrow [S]$ (* queue of SPPFs to be filtered *)
$outputSet \leftarrow \emptyset$
$S \leftarrow \{\mathcal{S}\}$
while $Q \neq [\,]$
    do $\mathcal{S}_x \leftarrow Dequeue(Q)$
        $resultSet \leftarrow RemoveFirstInvalidPath(\mathcal{S}_x, \mathcal{P}, \mathcal{F})$
        if $resultSet = \{\mathcal{S}_x\}$
            then (* filter did not change the SPPF *)
                $outputSet \leftarrow outputSet \cup \mathcal{S}_x$
            else for $\mathcal{S}_i$ in $resultSet$
                    do $Enqueue(Q, \mathcal{S}_i)$
        return $outputSet$
```

Algorithm *RemoveFirstInvalidPath* finds the first invalid path in the SPPF $\mathcal{S}$ given a set of precede restrictions $\mathcal{P}$ and a set of follow restrictions $\mathcal{F}$. As soon as such a path is found, all parse trees containing this path are removed from the SPPF and the resulting set of SPPFs is returned. In this way invalid paths can be removed as soon as possible, and exploring parts of the SPPF that will be removed anyway is avoided. Each walker is handled at most once by using a set *visited*. When a packed node is visited, a check is done whether a precede or follow restriction applies. If this is the case, the path removal algorithm is invoked and return the set of resulting SPPFs. After visiting a node the walker is updated using Algorithm *UpdateWalker* to update the SPPF-walker. Recall that an SPPF-walker contains four fields: $\mathcal{W}(node, p, f, rightNullable)$.

**Algorithm** *RemoveFirstInvalidPath*($\mathcal{S}, \mathcal{P}, \mathcal{F}$)
**Input:** $\mathcal{S}$: SPPF to be filtered

```
$w \leftarrow \mathcal{W}(\mathcal{S}.root, \emptyset, \emptyset, true)$
$visited \leftarrow \{w\}$
$Q \leftarrow [w]$ (* queue with SPPF walkers *)
while $q \neq [\,]$
    do $w \leftarrow$Dequeue($Q$) (* remove element from $Q$ *)
        $node = w.node$
        if $node.type = packed$
            then if $(node, w.p[0]) \in \mathcal{P}$
                    then return RemoveParseTreesP($\mathcal{S}, w.p$)
                if $(node, w.f[0]) \in \mathcal{F}$
                    then return RemoveParseTreesP($\mathcal{S}, w.f$)
        for $(node, child) \in E$
            do $newWalker \leftarrow$UpdateWalker($w, node, child$)
                if $newWalker \notin visited$
                    then Enqueue($Q, newWalker$)
                        $visited \leftarrow visited \cup \{newWalker\}$
return $\mathcal{S}$
```

Another strategy to apply left-open right-open filtering is to perform the two parts consecutively. Then, we first find all invalid paths, and then iteratively removing all invalid paths in each of the SPPFs that is outputted by the filter. In this strategy each path handled only once. In the variant discussed above, the same path may need to be removed multiple times.

### 6.5.5 Apply left-open right-open filtering on parse time

It is possible to partially apply left-open right-open filtering on parse time in GLL parsing. Consider a production of the form $E ::= \alpha_1 \mid \ldots \mid \alpha_n$. Then the corresponding lines of the algorithm will look as follows:

$L_E$: if $(test(I[c_I], E, \alpha_1))$ { add($L_{E_1}, c_U, c_I, \$$) }
$\ldots$

```
    if ($test(I[c_I], E, \alpha_n)$) { add($L_{E_n}, c_U, c_I, \$$) }
    goto $L_0$
```

Given GSS node $c_U$, with some label $(R_X, i)$, we know the return label $R_X$, which corresponds to a grammar slot directly after some nonterminal. Given this return label, it is possible to infer the parent production, and which nonterminal we try to expand in the derivation. For instance, if the parent production is $E ::= E + E$, label $R_X$ could correspond to grammar slot $E ::= E \cdot +E$ or $E ::= E + E \cdot$.

Now, in the code corresponding to label $L_E$, we are adding descriptors to schedule work for each applicable alternate $\alpha_i$ by executing the *add* method. Before executing this method, we can check whether the nonterminal that we try to expand is left-open or right-open given the grammar and the grammar slot corresponding to label $R_X$. Furthermore, we have the label corresponding to each alternate $\alpha_i$, namely $L_{\alpha_i}$.

Before executing the *add* function, we invoke Algorithm *CheckRestrictions*. This algorithm returns whether a restriction applies. If the algorithm returns *true*, we do not execute the *add* function.

**Algorithm** *CheckRestrictions*($L_{E_i}, c_U$)
**Output:** returns *true* if and only if a restriction applies

```
Assume $(c_U) \equiv (R_X, i)$
if $R_X$ corresponds to a grammar slot $E ::= E \cdot \alpha$
  then if $L_{E_i}$ corresponds to an grammar slot $E ::= \beta E \cdot$
        then return $(\langle \beta E \rangle, \langle E \alpha \rangle) \in \mathcal{F}$
if $R_X$ corresponds to a grammar slot $E ::= \beta E \cdot$
  then if $L_{E_i}$ corresponds to an grammar slot $E ::= E \cdot \alpha$
        then return $(\langle E \alpha \rangle, \langle \beta E \rangle) \in \mathcal{P}$ return false
```

To show the relation between the LORO parse forest filter and the parse time filter, consider Figure 6.5. In this figure we have that $R_X$ corresponds to $E ::= E + E \cdot$ (packed node below $(E, i, j)$, and $L_{E_i}$ corresponds to $(E ::= E + E \cdot)$ (packed node below $(E, k, j)$. Given these grammar slots, we can infer that the parser is about to add a left-open rule directly below a right-open rule. Therefore we have to check the precede restrictions to check whether this action of the parser will result in a parse tree where a precede violation would result. Since the addition operator is left-associative, there is indeed a precede restriction that applies. The result is that the $add$ function will not be executed by the parser, and prevents the right-associative derivation of the addition operator. This mechanism also works in a similar way for follow restrictions.

Note that we are pruning the parse forest that is being generated. We check the direct nesting of a left-open rule below a right-open rule, where the parent nonterminal is open. Figure 6.6 shows the two situations in which we check the restrictions. These situations are simplified variants

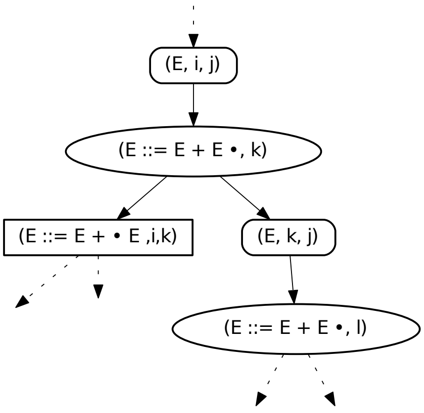
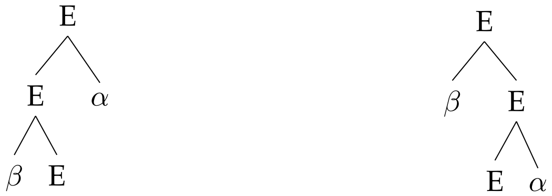


Figure 6.5: Case when $R_X$ corresponds to $E ::= E+E\cdot$ (packed node below $(E, i, j)$, and $L_{E_i}$ corresponds to $(E ::= E + E\cdot$.



(a) Valid tree when $prio(\langle E, \beta E\rangle) > prio(\langle E, E\alpha\rangle)$

(b) Valid tree when $prio(\langle E, E\alpha\rangle) > prio(\langle E, \beta E\rangle)$

Figure 6.6: Derivation trees when a grammar contains both a left-open and right-open rule.

of the situations shown in Figure 6.1. By not executing the *add* function, we prevent undesired parse trees from being generated. We do not remove any node or edge that is already present in the SPPF.

## 6.6 Order of filtering is of importance

Given the prefer filter and left-open right-open filter, one might wonder whether the order of application of these filters on an SPPF can yield to different results. This indeed turns out to be the case. To illustrate this, consider Grammar $\Gamma_{6.8}$ and input string $a \to b + c \to d \diamond e$. Recall that the

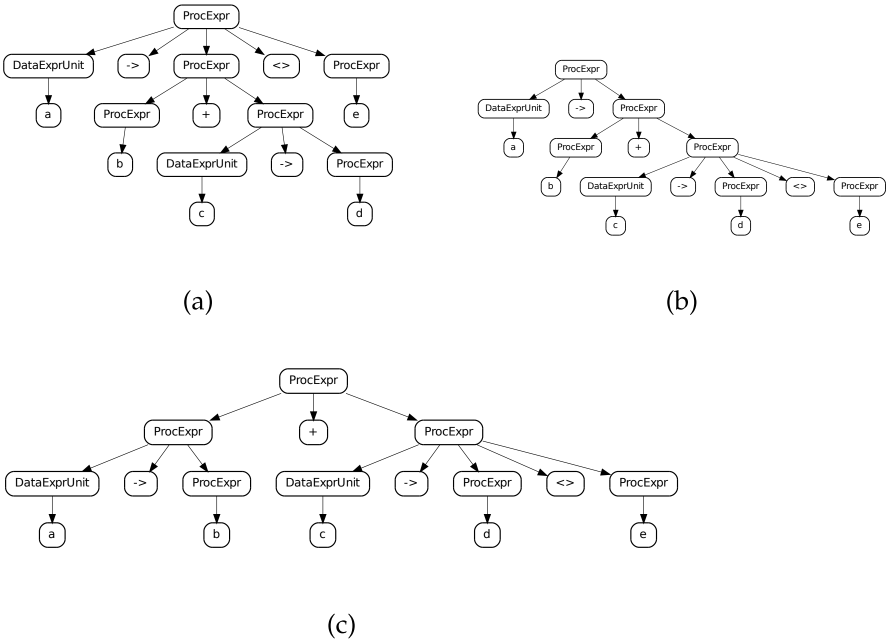


Figure 6.7: Parse trees for input string $a \to b + c \to d \diamond e$ and Grammar $\Gamma_{6.8}$.

$+$-operator has the lowest priority.

$$\begin{aligned} ProcExpr &::= DataExprUnit \to ProcExpr \\ ProcExpr &::= DataExprUnit \to ProcExpr \ \diamond ProcExpr \\ ProcExpr &::= ProcExpr + ProcExpr \\ ProcExpr &::= a \mid b \mid c \mid d \mid e \\ DataExprUnit &::= a \mid b \mid c \mid d \mid e \end{aligned} \tag{$\Gamma_{6.8}$}$$

For our example, there are three possible parse trees, given in Figure 6.7. In the resulting SPPF, the top productions for each of these trees are added as packed nodes below the root node.

Suppose that we first apply the prefer filter and consecutively the left-open right-open filter. The prefer filter will prefer trees starting with $DataExprUnit \to ProcExpr$ over trees starting with $DataExprUnit \to ProcExpr \diamond ProcExpr$. This means that the parse tree shown in Figure 6.7a will be removed. The left-open right-open filter will discard the parse tree shown in Figure 6.7b, since

$$(\langle ProcExpr + ProcExpr \rangle, \langle DataExprUnit \to ProcExpr \rangle) \in \mathcal{P}.$$

After applying both filters we are left with the parse tree shown in Figure 6.7c.

Now suppose that we first apply the left-open right-open filter. Then we will again remove the parse tree shown in Figure 6.7b. Next, we apply the prefer filter which will not remove any tree. This results from the observation, that we cannot prefer $DataExprUnit \to ProcExpr$ over $DataExprUnit \to ProcExpr \diamond ProcExpr$ since the tree corresponding to the former has already been removed. In the resulting SPPF, there will therefore be two embedded parse trees, where one tree is undesired.

In our example we can solve this problem by postponing enforcing the restrictions related to productions $DataExprUnit \to ProcExpr$ and $DataExprUnit \to ProcExpr \diamond ProcExpr$, until after prefer filtering in the left-open right-open post-parse filter.

# Chapter 7

# Experimental Evaluation

## 7.1 Experimental Setup

All disambiguation filters that are described in Chapters 5 and 6 have been implemented in Java. The generation of the SPPF is done by a generalized LL parser that has also been implemented in Java. The original implementation originates from previous work [17], but has been improved substantially in order to obtain better running times and add support for disambiguation filters. We have chosen to use GLL as parsing algorithm because in this algorithm it is relatively easy to add parse-time filters. The GLL parsers generated by our implementation support a restricted form of left-open right-open filtering on parse-time. The restricted keywords filter has also been implemented as a parse-time filter. Chapter 8 describes the implementation of the parser and filters in more detail.

For the experimental evaluation of our disambiguation filters, we use the set of example mCRL2 files available in the source release [1]. In our analysis, we have excluded files that have less than 200 tokens. The number of tokens is determined by parsing the file and counting the number of leaves in the SPPF. Furthermore some files differ in only one token. Therefore we have also excluded files `fischer-100`, `fischer-1000`, and `fischer-10000`.

The experiments have been run on an Intel© Core i7-3630QM processor (2.4GHz, 6-MB L3), with 8GB 1600 DDR3 memory. The following Java VM settings have been used:

```
-Xms4048m  -Xmx4048m
```

The set of example mCRL2 files are very heterogeneous, both in size as in the level of ambiguity. In order to evaluate the performance of our disambiguation filters on files with lots of ambiguities, we need a measure

[1] `http://www.mcrl2.org/release/user_manual/download.html` (mCRL2 toolkit version 201310.0)

to express the level of ambiguity in an SPPF. As a measure we could for instance use:

- *The number of embedded parse trees, given by function $emb_{\mathcal{S}}(r)$ where $r$ is the root of SPPF $\mathcal{S}$.* Define $emb_{\mathcal{S}}(u)$ for $u \in V(\mathcal{S})$ as follows:

  $$emb_{\mathcal{S}}(u) = \begin{cases} 1 & \text{if } |u.children| = 0 \\ \prod\limits_{c \in u.children} emb_{\mathcal{S}}(c) & \text{if } |u.children| > 0 \wedge u.type = packed \\ \sum\limits_{c \in u.children} emb_{\mathcal{S}}(c) & \text{if } |u.children| > 0 \wedge (u.type = symbol \\ & \vee\ u.type = intermediate) \end{cases}$$

  The main problem with this metric is that the number of embedded parse trees becomes very large as the number of ambiguities increases. For instance, if a packed node $v$ has two children which each have 100 possible subtrees, then there are 10,000 subtrees with $v$ as root.

- *The disambiguation time divided by the parse time.*

  If the disambiguation time is very long compared to the parse time, the number of ambiguities is typically also large. However, the size of the SPPF indirectly influences this metric since the amount of bookkeeping becomes bigger when the SPPF grows.

- *The number of packed node children below ambiguous nodes versus the total number of packed nodes.* Let $\mathcal{N}_{\mathcal{S}}$ and $\mathcal{I}_{\mathcal{S}}$ be the set of nonterminal symbol nodes and intermediate nodes in $\mathcal{S}$. Define $\Xi(\mathcal{S})$ as follows:

  $$\Xi(\mathcal{S}) = \frac{\sum_{u \in \mathcal{N}_{\mathcal{S}} \cup \mathcal{I}_{\mathcal{S}}} |children(u)|}{|\mathcal{N}_{\mathcal{S}}| + |\mathcal{I}_{\mathcal{S}}|}$$

  If there are no ambiguities in $\mathcal{S}$, $\Xi(\mathcal{S})$ will be equal to 0. If the number of ambiguities increases, the value of $\Xi(\mathcal{S})$ will grow. If each packed node in $\mathcal{S}$ is part of an ambiguity, $\Xi(\mathcal{S})$ will be equal to 1. A disadvantage of this metric is that it does not take the relative depths of the ambiguities into account. Ambiguities that occur close to the root will be found earlier during filtering, than ambiguities occurring deep in the SPPF if we use a top-down filtering strategy.

We have chosen to use the last two metrics, and select the 10 files with the highest values. This gives us the input files that have the most ambiguous SPPFs and where disambiguation takes a lot of time.

## 7.2 Hypotheses

For the experimental evaluation of the algorithms, we have constructed the following hypotheses to be tested.

(H1) *Prefer filtering is significantly faster than left-open right-open filtering.*

The left-open right-open (LORO) filter uses an SPPF-walker to keep a number of variables. Prefer filter on the other hand does not need to keep track of any variables, and can make a decision based on local information. So, prefer filtering should be much faster than left-open right-open filtering.

(H2) *Post-parse filtering will not have a higher running time than the parse time.*

The generalized parser will construct an SPPF containing all derivations. Filtering this SPPF can be done in a top-down fashion, disregarding invalid subtrees as early as possible. Only in the worst case do we need to traverse the whole SPPF to find ambiguities that lie far from the root of the SPPF. Therefore we assume that post-parse filtering will not take longer than the parse time.

(H3) *Parsing with parse-time filtering will on average lead to a slightly higher running time compared to the parsing without parse-time filtering.*

By augmenting the parser with parse-time filters, a little extra work needs to be done on parse-time. Therefore the parse time with parse-time filtering will be slightly higher.

(H4) *Parse-time filtering will resolve most of the ambiguities related to associativities and ambiguities.*

On parse-time we have only local information that can be used by the filter. For instance the parent of a production is known by a GLL parser. For many left-open right-open ambiguities this information is sufficient.

(H5) *Parsing ambiguous files with parse-time and post-parse time filtering will be faster than parsing with only post-parse time filtering.*

If the input file is ambigous, filtering is needed to remove undesired derivations. On parse-time part of invalid derivations can be avoided by keeping them out of to the SPPF.

## 7.3 Results and Discussion

### 7.3.1 Results

Table C.1 in Appendix C contains the number of tokens and value of ambiguity metric $\Xi$ for each file in our data set. This table also shows the total

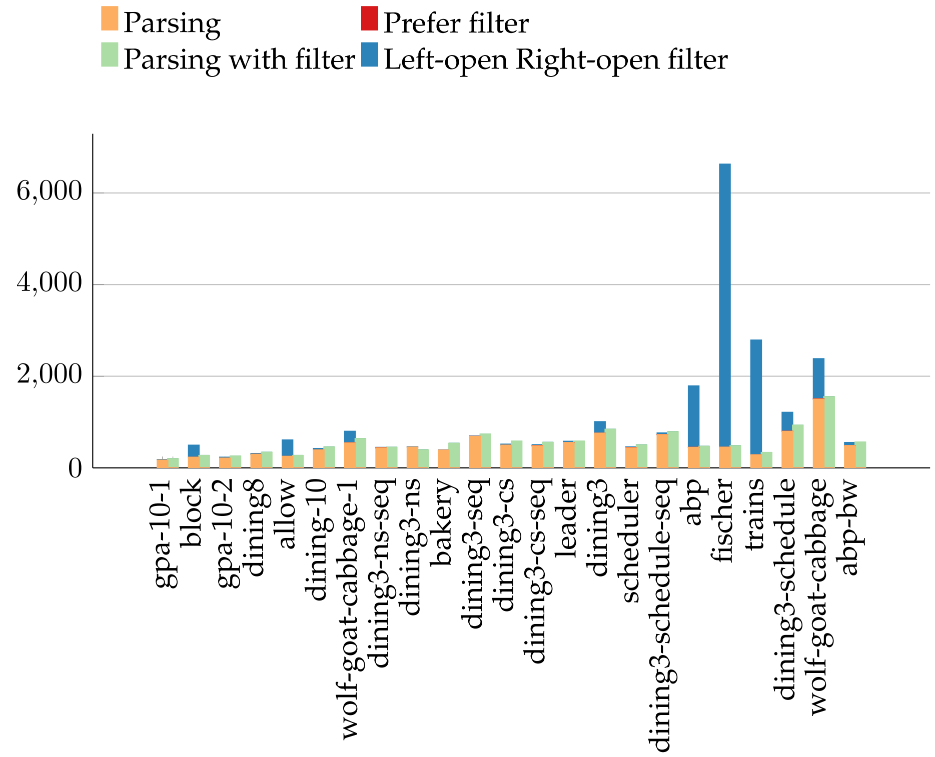


Figure 7.1: Post-parse filtering versus filtering while parsing for small mCRL2 files (time in ms).

parsing time versus the parse time with parse-time and post-parse time filtering. The longest parsing time, 27377 ms, is needed for `garage-ver`, which also has the largest number of tokens. On average the total execution time of parsing with parse-time LORO filtering versus parsing without LORO filtering is 9.4% higher. However, the standard deviation is 16.2%. For highly ambiguous files, parse-time filtering is faster because many invalid derivations are not explored in the first place.

Figure 7.1 shows the execution time of parsing with post-parse filtering versus parse-time filtering for small files. For larger input files, post-parse filtering leads to an out of memory exception due to an excessive amount of garbage collection. Table 7.1 also shows the number of tokens and the value of Ξ for these files. Files *magic-square*, *bakery*, *rational* are omitted because they lead to an out of memory exception, as did files *fischer-10* and *cellular-automata*.

Table 7.2 shows the execution time of parsing with and without parse-time filtering for the most ambiguous files, having the highest Ξ-values, in our dataset.

| | | | **Post-Parse Time** | | | **Parse Time** | |
|---|---|---|---|---|---|---|---|
| Input File | Tokens | $\Xi(\mathcal{S})$ | Parsing | PR+LORO | Total | Parsing | Difference |
| gpa-10-1 | 206 | 0.091 | 170 | 0 + 11 | 181 | 202 | 0.116 |
| block | 222 | 0.189 | 231 | 0 + 264 | 495 | 270 | -0.455 |
| gpa-10-2 | 222 | 0.095 | 213 | 0 + 20 | 233 | 259 | 0.112 |
| dining8 | 275 | 0.180 | 295 | 0 + 18 | 313 | 341 | 0.089 |
| allow | 277 | 0.216 | 250 | 0 + 360 | 610 | 270 | -0.557 |
| dining-10 | 287 | 0.257 | 391 | 0 + 31 | 422 | 458 | 0.085 |
| wolf-goat-cabbage-1 | 291 | 0.148 | 544 | 1 + 253 | 798 | 639 | -0.199 |
| dining3-ns-seq | 300 | 0.103 | 436 | 1 + 10 | 447 | 451 | 0.009 |
| dining3-ns | 300 | 0.104 | 454 | 0 + 8 | 462 | 395 | -0.145 |
| bakery | 328 | 0.058 | 386 | 0 + 7 | 393 | 537 | 0.366 |
| dining3-seq | 352 | 0.079 | 685 | 1 + 11 | 697 | 739 | 0.060 |
| dining3-cs | 366 | 0.141 | 496 | 0 + 22 | 518 | 585 | 0.129 |
| dining3-cs-seq | 366 | 0.140 | 482 | 1 + 23 | 506 | 557 | 0.101 |
| leader | 399 | 0.187 | 552 | 1 + 27 | 580 | 585 | 0.009 |
| dining3 | 400 | 0.119 | 758 | 2 + 248 | 1008 | 843 | -0.164 |
| scheduler | 424 | 0.133 | 438 | 1 + 20 | 459 | 503 | 0.096 |
| dining3-schedule-seq | 434 | 0.132 | 725 | 1 + 37 | 763 | 790 | 0.035 |
| abp | 439 | 0.133 | 446 | 1 + 1342 | 1789 | 471 | -0.737 |
| fischer | 439 | 0.109 | 451 | 0 + 6178 | 6629 | 484 | -0.927 |
| trains | 459 | 0.184 | 284 | 0 + 2506 | 2790 | 335 | -0.880 |
| dining3-schedule | 482 | 0.155 | 796 | 1 + 415 | 1212 | 933 | -0.230 |
| wolf-goat-cabbage | 490 | 0.094 | 1499 | 9 + 875 | 2383 | 1554 | -0.348 |
| abp-bw | 496 | 0.079 | 486 | 1 + 66 | 553 | 564 | 0.020 |

Table 7.1: Parsing and filtering times for small mCRL2 files with post-parse time filtering or both parse-time and post-parse time filtering.

| Input File | Tokens | $\Xi(\mathcal{S})$ | Parsing | Parsing & Parse-time Filtering | Difference |
|---|---:|---|---:|---:|---:|
| SMS | 4206 | 0.453 | 5371 | 6350 | 0.182 |
| food-package | 657 | 0.458 | 1898 | 1928 | 0.016 |
| sets-bags | 764 | 0.568 | 1069 | 1124 | 0.051 |
| garage-ver | 11545 | 0.669 | 27377 | 29114 | 0.063 |
| 11073 | 4428 | 0.674 | 8321 | 9142 | 0.099 |
| mpsu | 325 | 0.701 | 672 | 692 | 0.030 |
| magic-square | 227 | 0.870 | 1197 | 655 | -0.453 |
| chatbox | 3840 | 0.912 | 18786 | 14105 | -0.249 |
| WMS | 5987 | 0.925 | 15512 | 10927 | -0.296 |
| numbers | 882 | 0.971 | 8143 | 1776 | -0.782 |

Table 7.2: Parsing and filtering times for ambiguous mCRL2 files with post-parse time filtering or both parse-time and post-parse time filtering.

### 7.3.2 Discussion

In this section we will analyze the performance of the disambiguation filters, and see whether our hypotheses are correct. Figure 7.1 shows the results for post-parse filtering versus parse-time filtering. Several highly ambiguous input files are omitted because they gave an out-of-memory exception. From the results for the small files we can see that prefer filtering is significantly faster than left-open right-open filtering. This stems in accordance with hypothesis *H1*. For ambiguous input files, the time needed for post-parse filtering becomes much larger than for parsing. Therefore, we must reject hypothesis *H2*. The bottleneck in our implementation is caused by cloning SPPFs. As future work, an efficient persistent data structure is desired for storing and modifying the SPPFs.

If we look at the performance of the parser with parse-time filtering, we see that for almost all input files parse-time filtering leads to a higher running time. If the input file is not ambiguous, many of the checks done by the parse-time filter are redundant. By checking only in those cases where an ambiguity might arise, the overhead of the parse-time filter can be significantly reduced. Concluding, we can say that hypothesis *H3* is only true for files that are not ambiguous. For ambiguous files, the parse time with parse-time filtering is lower than parsing and post-parse filtering. This observation stems in accordance with hypothesis *H5*. For input files with many ambiguities, parse-time filtering is much faster than post-parse filtering.

From Table C.1 we can see that parse-time filtering in most cases resolves *all* ambiguities. In the entire dataset, there are only two files where post-parse filtering is needed to remove the remaining ambiguities. In file *onebit*, there is an ambiguity in a sort expression described in Section 6.4.2. File *game-of-goose*, contains an ambiguity in an process expression (see Section 6.4.4) that requires post-parse filtering. Most ambiguities can be resolved, or better put avoided, on parse-time, which is in accordance with hypothesis *H4*. Many ambiguities that are present in mCRL2 files are between binary operators. These kind of ambiguities can often be resolved by looking at the direct nesting of parent and child productions in a derivation.

To check the correctness of the filtering we perform the following steps. Firstly, we create a set of ambigous mCRL2 files containing specific types of ambiguities. Then we feed each mCRL2 file into the parser currently used by mCRL2 to obtain a correctly bracketed expression. Given the ambiguous and bracketed files, we generate three SPPFs using the following setup:

1. parse bracketed file without filtering;
2. parse ambigous file with parse-time and post-parse time filtering;
3. parse ambigous file with post-parse time filtering.

Then we have to compare the resulting SPPFs and check whether they have the same structure.

Concluding, we can say that the combination of parse-time filtering and post-parse filtering is preferable over filtering all ambiguities after parsing. Compared to parsing, the total execution time increases on average for about 10%, while ambiguous files are processed much faster. By checking the LORO restrictions only if a restriction might apply, the overhead of parse-time filtering can be reduced.

# Chapter 8

# Implementation

For the experimental evaluation of our disambiguation filters, we have implemented a parser generator for generating GLL-based Java parsers in previous work [17]. We have improved upon this implementation to obtain better running times. Furthermore different data structures are used for representing SPPFs, to allow the copying of SPPFs. The disambiguation filters are also implemented in Java and are compatible with the SPPF that is constructed by the generated parsers. In this chapter we describe the parser generator, and implementation notes on the disambiguation filters.

## 8.1 GLL parser generator

The parser generator that we have implemented generates object-oriented GLL-based Java parsers. Instead of using labels and goto-statements, we use a functional decomposition. In GLL, there are two types of functions: match functions, and alternate functions.

*Match functions* are responsible for actual parsing of the input string with respect to parts of the grammar. Given an alternate in the grammar, we parse this alternate in parts. We refer to these parts as *GLL blocks*. Before parsing a nonterminal occurring in an alternate, say the first $E$ in $E ::= abEEeF$, we create a descriptor with the return location corresponding to the grammar slot directly after the nonterminal, i.e. $E ::= abE \cdot EeF$. This descriptor continues parsing the rest of the alternate after the first $E$ has been parsed. After parsing the entire alternate, the GLL algorithm invokes the *pop* routine.

To illustrate the concept of GLL blocks in more detail, consider the alternate $E ::= abEEeF$ and the corresponding GLL blocks:
$E ::= \underbrace{abE}_{0}\underbrace{E}_{1}\underbrace{eF}_{2}\underbrace{\ \ }_{3}.$

The alternate consists of four GLL blocks, namely "abE", "F", "eE", and " ". Note the empty GLL block at the end, that performs a *pop* statement.

Definition 8.1.1 gives a formal definition of a GLL block.

**Definition 8.1.1.** A GLL block $b$ in alternate $\alpha$ is a sequence of literals $s$, such that $s$ is of the shape $t +\!\!+ [last(s)]$. If $s$ is $\varepsilon$, the GLL block is also $\varepsilon$.

- $s$ is a subsequence of $\alpha$
- $head(s)$ is *not* preceded by a terminal.
- $last(s)$ is either:
  - a nonterminal symbol, or
  - a terminal symbol that is *not* followed by any literal
- $t$ is a sequence of terminal symbols

*Alternate functions* are used to determine which alternates to parse next. Given a nonterminal in the grammar, we have a set of alternates that correspond to this nonterminal. For each alternate, we invoke the *test*-function. The *test*-function looks at the current input position, and the FIRST and FOLLOW sets to determine whether the alternate is applicable. If an alternate is applicable, a descriptor is created scheduling the function corresponding to the first GLL block of the alternate.

### 8.1.1 Abstract parser

The GLL algorithm contains a set of functions that are grammar-independent, for instance the *pop*, *create*, *add*, *test*, *getNodeP*, and *getNodeT* functions. Therefore, each parser that is generated inherits from an abstract parser. This abstract parser also contains functions to interact with the data structures used by GLL.

### 8.1.2 Scanner

In our GLL implementation are separate lexer is used, which is driven by the parser. Each time the *test*-function is invoked, the parser gives a set of tokens to the scanner to check whether one of the tokens can be matched against the substring starting from the current input pointer. These tokens can be character-level tokens, but it is also possible to be a regular expression. In this way the grammar can be specified at the character-level, or by using a combination of terminals and lexical rules containing a regular expression. Using regular expressions can be used to implement a longest-match rule while parsing.

Another advantage of using a separate scanner is that a white space scanner can be used that skips white space in the input string. This avoids the need to augment the grammar with lots of nonterminals for matching layout between terminals. When matching a token, we skip the white space occurring after this token. In the *create* statement, the GLL algorithm retrieves the right extent of a node and uses this position when creating descriptors. Before creating this descriptor, we need to skip the white space and give this input position to the descriptor.

For the matching of the tokens against the input string, the `dk.brics.automaton` [36] regular expression library is used. This library is based on deterministic finite automatons (DFA), rather than nondeterministic finite automatons (NFA). Using DFA-based matching rather than NFA-based matching means that features like capturing groups are not available, but these kind of features are not needed for a lexer. The advantage of using DFAs compared to NFAs is that for each state and alphabet the transition relation has exactly one state, meaning that DFAs can match patterns linear in the length of the input string, independently of the complexity of the pattern. By using the `dk.brics.automaton` library, instead of the Java NFA-based library, the performance gain has been increased significantly.

## 8.2 Parse tree removal

In Chapter 4 we have described a set of algorithms for removing parse trees from an SPPF. When all parse trees are removed from the SPPF that contain some path, the result might be a set of SPPFs rather than a single smaller SPPF. To avoid deep copying of the entire SPPF, we represent each SPPF by two adjacency lists; one for the incoming edges, and one for the outgoing edges to allow fast edge lookups. A persistent hash table is used to keep references from each node to its list of outgoing or incoming edges. In our implementation we use the persistent hash map provided by the CLOJURE language [28], which can be imported in Java. Another persistent hash map that can be used in Java is included in the PCOLLECTIONS library [47]. This variant unfortunately gave even worse results than deep cloning. After testing the performance of the persistent hash map we found that the hash code proved to be a bottleneck. Therefore we use a Java `BitSet` that indicates whether a certain edge is present in the SPPF. Equality of SPPFs is checked by comparing the corresponding BitSets, which is very fast.

Because the set of SPPF nodes cannot grow after parsing, we never need to copy SPPF nodes. To create a smaller copy of the SPPF, we give a set $E_r$ of edges to remove. If the list for a certain node changes because one or more edges are removed, we make a deep copy of the list without these edges. Otherwise we just use a reference to the old list. If one or more edges are altered, the hash table must also be updated. After creation of the SPPF, it can not be altered anymore, meaning that it becomes immutable.

# Chapter 9

# Conclusions

In this thesis we have looked into disambiguation in shared packed parse forests that are generated by generalized parsing algorithms like GLL and GLR. We have looked at filters that allow the removal of parse trees from the parse forest containing some invalid construct, like an invalid node or path. Invalid constructs are specified by filters that address certain kinds of ambiguity. For instance the restriction keyword filter specifies that certain nonterminals may never yield a word that is contained in a set of restricted keywords.

In Chapter 2 we have introduced the SPPF that is generated by parsing algorithms like GLL. The current practices of disambiguation have been described in Chapter 3. Chapters 4 and 5 in this thesis deal with the research questions posed in the introduction. Chapter 8 gives some notes on the implementation that we have used for testing our filters. This chapter concludes the thesis by summarizing our contributions to these questions, and gives directions for future work.

## 9.1 Contributions to Research Questions

**Research Question 1** How can we remove undesired parse trees from an SPPF while keeping desired parse trees?

One of the main challenges in SPPF filtering is the fact that two types of sharing are used; subtree sharing and local ambiguity packing. When removing certain nodes or edges in the SPPF, one must be very cautious not to remove valid parse trees. In Chapter 4 we have described a set of SPPF filters that allows the removal of embedded parse trees in an SPPF containing some node, edge, or path. It turns out that removing all parse trees from an SPPF containing some invalid path can be done by first splitting the SPPF into multiple copies, and then apply the edge removal filter on these copies. One of the consequences of this is that path removal cannot be done on parse time in generalized-parsing, without using a different structure to store all parse trees. Otherwise, valid parse trees might be removed unintentionally.

**Research Question 2** What kind of ambiguities occur in expression grammars and how to implement parse forest filters to resolve these ambiguities?

In Chapter 5 we have looked into ambiguities that occur in expression grammars. The kind of ambiguities that occur are caused by the associativity and precedence of operators. For resolving these ambiguities, we have developed the precede/follow production filter. This filter looks at the precede and follow productions given some production, and specifies a set of restrictions that describe which combinations lead to invalid derivations. The filter has been implemented as a parse-forest filter, and uses the filters described in Chapter 4 to remove all parse trees where one or more restrictions apply.

**Research Question 3** Are the parse forest filters able to resolve all ambiguities occurring in mixfix grammars?

When extending the class of expression grammar with mixfix expressions, there are additional types of ambiguities that might occur. These types of ambiguities are described in Chapter 6. As a case study of ambiguity resolving in mixfix expressions, we have looked into the disambiguation of the mCRL2 grammar. In order to completely and correctly disambiguate mCRL2 input files, we need several filters. For resolving an instance of the dangling-else ambiguity we use a prefer filter and a not-follow filter. Restricted keywords are enforced by applying a restricted keywords filter, which can be efficiently implemented as a parse-time filter. The precedences and associativities are enforced by the left-open right-open filter, which is generalization of the precede/follow production filter. This generalized variant is needed to prevent filtering below closed operands, and to ignore nonterminals that derive epsilon.

The left-open right-open filter and prefer filter have been implemented as parse forest filters, and use the SPPF filters described in Chapter 4 to remove all invalid parse trees. Furthermore, we have implemented the left-open right-open filter in a restricted form as a parse-time filter in GLL. This filter avoids expanding a nonterminal in a derivation, if a precede/follow restriction applies to the parent production and the production of the expanded nonterminal.

Chapter 7 contains an experimental evaluation of the filters used for filtering the mCRL2 grammar. The parse-time filter is able to resolve almost all ambiguities arising from left-open and right-open operators, which means that the resulting parse forest is much smaller. Furthermore, by using parse-time filtering in addition to post-parse filtering on ambiguous sentences, the total execution time for parsing and disambiguation can be dramatically reduced compared to post-parse filtering.

For mCRL2 we have been able to resolve all ambiguities that occur in the grammar. However, as described in Section 6.4.4, ambiguities that arise from shared separator tokens require other filters. These kind of filters are left as future work.

## 9.2 Future work

This thesis provides a set of filters that allow disambiguation on SPPFs. While the filters are sufficiently expressive to disambiguate the set of expression grammars, and simple mixfix grammars, there are still a number of open questions. In this section we will pose some directions for future work.

### Removing all parse trees containing some subtree

If a generic production for binary operators is used, path filtering is no longer sufficient. To illustrate this, consider the following example.

$$\begin{aligned} &E ::= EOE \\ &O ::= + \mid * \mid \cdots \end{aligned} \tag{$\Gamma_{9.1}$}$$

In order to check whether a nesting of two productions of the form $EOE$ is allowed, we need information about the specific operators that are specified by nonterminal $O$. Therefore we need an algorithm that can remove all parse trees containing a specific subtree.

### Filtering ambiguities caused by shared separator tokens

In Section 6.4.4, we have seen an example of an ambiguity that is caused by operators sharing one or more separator tokens. For this specific instance, the ambiguities could be resolved by using a prefer-filter, and a follow restriction filter. In general however, a more generic filter is desired that is specifically aimed at resolving these kind of ambiguities. Ideally, first an investigation is done when such ambiguities might occur, and then a filter must be specified resolving the ambiguities.

### Persistent data structure for SPPFs

As described in Section 8.2, the current data structure used to represent SPPFs needs to be improved. At the moment the bottleneck in parse-forest filtering is the copying of SPPFs and removing some edges in the copy. By using a persistent data structure, parts that are not changed in the new SPPF can be shared, because after creation the SPPF becomes immutable. Current implementations of persistent hash maps proved to be inadequate at solving the problem. A persistent data structure is needed that improves the amount of sharing. This data structure must be efficient to update, and it must be efficient to query the incoming and outgoing edges of some node. An observation that might be useful in the design process is that the removal of edges in a graph is always local, and that the set of edges that is removed are always connected.

### Parsing language embeddings

Context-free languages that are combined again form a context-free language. This allows languages to be embedded in other languages. For each language there might be a unique set of disambiguation rules. In order to correctly disambiguate the entire input sentence, various parts of the SPPF have to be disambiguated by different disambiguation rules. A topic of future research is how to apply parse forest filtering in an SPPF containing different languages, and which kind of filters are needed.

### SPPF Visualization

If the SPPF for a certain input sentence contains ambiguities, it is often desired to give feedback about these ambiguities to the user. This is for instance useful when designing a language, to see which kind of ambiguities can occur and need to be disambiguated. At the moment we use GraphViz `dot` [23] for visualizing SPPFs, but for large SPPFs (for instance the large mCRL2 files in our data set), the time needed to visualize the SPPF becomes impractical (more than 10 minutes). Ideally, one would like an interactive environment where parts of the SPPF can be collapsed, and places where ambiguities occur can be highlighted.

### GLL Production Parser

Although our GLL implementation is usable for files of reasonable size, a production parser needs to be a whole lot faster to be usable in a real-life setting. The parse time of GLL can be improved in several ways. In the GLL algorithm, many checks are done whether a certain descriptor has already been handled, or is present in the set of pending descriptors. By improving the hash codes, an increase in parsing speed can be obtained. Furthermore, before starting the parser the FIRST and FOLLOW sets are determined. By generating this information during parser generation, this information does not need to be recomputed each time the parser is started.

# Bibliography

[1] Aasa, A.: Precedences in specifications and implementations of programming languages. In: Maluszynski, J., Wirsing, M. (eds.) PLILP'91, Lecture Notes in Computer Science, vol. 528, pp. 183–194 (1991)

[2] Afroozeh, A.: Gtext: A Language Workbench based on GLL and Term Rewriting. Master's thesis, Eindhoven University of Technology, the Netherlands (2012)

[3] Afroozeh, A., Bach, J.C., van den Brand, M., Johnstone, A., Manders, M., Moreau, P.E., Scott, E.: Island Grammar-Based Parsing Using GLL and Tom. In: Czarnecki, K., Hedin, G. (eds.) Software Language Engineering, 5th International Conference, SLE 2012. Lecture Notes in Computer Science, vol. 7745, pp. 224–243. Springer Berlin Heidelberg, Dresden, Germany (Sep 2012), `http://dx.doi.org/10.1007/978-3-642-36089-3_13`

[4] Aho, A.V., Johnson, S.C., Ullman, J.D.: Deterministic parsing of ambiguous grammars. Commun. ACM 18(8), 441–452 (1975)

[5] Arnoldus, B.J.: An Illumination of the Template Enigma: Software Code Generation with Templates. Ph.D. thesis, Technische Universiteit Eindhoven (2010)

[6] Aycock, J.: Why bison is becoming extinct. Crossroads 7(5), 3–3 (2001)

[7] Baeten, J.C.M., Weijland, W.P.: Process algebra. Cambridge University Press, New York, NY, USA (1990)

[8] Basten, H.: Ambiguity Detection for Programming Language Grammars. Ph.D. thesis, University of Amsterdam (2011)

[9] Blasband, D.: Parsing in a hostile world. In: WCRE'01. pp. 291–300. IEEE Computer Society (2001)

[10] Bouwers, E., Bravenboer, M., Visser, E.: Grammar engineering support for precedence rule recovery and compatibility checking. Electronic Notes in Theoretical Computer Science 203(2), 85 – 101

(2008), http://www.sciencedirect.com/science/article/pii/S1571066108001515, proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)

[11] Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. Tech. Rep. RS-06-09, BRICS, University of Aarhus (May 2006), http://www.brics.dk/~brabrand/grambiguity/

[12] van den Brand, M.G.J., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient annotated terms. Software: Practice and Experience 30(3), 259–291 (2000), http://dx.doi.org/10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y

[13] van den Brand, M., Klusener, S., Moonen, L., Vinju, J.J.: Generalized parsing and term rewriting: Semantics driven disambiguation. Electronic Notes in Theoretical Computer Science 82(3), 575–591 (2003)

[14] van den Brand, M., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Horspool, R.N. (ed.) CC'02. Lecture Notes in Computer Science, vol. 2304, pp. 143–158. Springer (2002), http://www.springerlink.com/content/03359k0cerupftfh/

[15] van den Brand, M.G.J., Sellink, A., Verhoef, C.: Current parsing techniques in software renovation considered harmful. In: IWPC '98. pp. 108–117. IEEE Computer Society (1998)

[16] ten Brink, A.: Disambiguation mechanisms and disambiguation strategies. Master's thesis, Eindhoven University of Technology, the Netherlands (2013)

[17] Cappers, B., Mengerink, J., van der Sanden, B.: Object Oriented GLL with Error Handling. Tech. rep., University of Eindhoven - Department of Mathematics and Computer Science (2014)

[18] Cheung, B.S.N., Uzgalis, R.C.: Ambiguity in context-free grammars. In: SAC'95. pp. 272–276. ACM Press (1995)

[19] Chomsky, N., Schützenberger, M.P.: The algebraic theory of context-free languages. In: Braffort, P., Hirshberg, D. (eds.) Computer Programming and Formal Systems, pp. 118–161. Studies in Logic, North-Holland Publishing (1963)

[20] Cranen, S., Groote, J., Keiren, J., Stappers, F., Vink, E., Wesselink, W., Willemse, T.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S. (eds.) Tools and Algorithms for

the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 7795, pp. 199–213. Springer Berlin Heidelberg (2013)

[21] Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13(2), 94–102 (1970)

[22] Floyd, R.W.: Syntactic analysis and operator precedence. J. ACM 10(3), 316–333 (1963)

[23] Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Softw. Pract. Exper. 30(11), 1203–1233 (Sep 2000), `http://dx.doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.3.CO;2-E`

[24] Ginsburg, S., Harrison, M.A.: Bracketed context-free languages. J. Comput. Syst. Sci. 1, 1–23 (1967)

[25] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 7 Edition. Java Series, Pearson Education (2013)

[26] Groote, J., Mousavi, M.: Modelling and Analysis of Communicating Systems (2013), lecture notes for System Validation course

[27] Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF–Reference Manual—. ACM SIGPLAN Notices 24(11), 43–75 (1989)

[28] Hickey, R.: Clojure. `http://clojure.org/`, [Online; accessed 10-June-2014]

[29] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Series in Computer Science, Addison-Wesley (1979)

[30] Johnson, S.C.: YACC — yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, Murray Hill, New Jersey (July 1975)

[31] Kats, L.C., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: ACM Sigplan Notices. vol. 45, pp. 918–932. ACM (2010)

[32] Klein, D., Manning, C.D.: Accurate unlexicalized parsing. In: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1. pp. 423–430. ACL '03, Association for Computational Linguistics, Stroudsburg, PA, USA (2003), `http://dx.doi.org/10.3115/1075096.1075150`

[33] Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In: Pighizzini, G., San Pietro, P. (eds.) ASMICS Workshop on Parsing Theory. pp. 89–100. Technical Report 126-1994, Università di Milano (1994)

[34] Mathijssen, A.: Data types for mCRL2. http://www.mcrl2.org/dev/developer_manual/_downloads/mcrl2data.pdf (2014), [Online; accessed 18-April-2014]

[35] McPeak, S., Necula, G.C.: Elkhound: A fast, practical GLR parser generator. In: Duesterwald, E. (ed.) CC'04. Lecture Notes in Computer Science, vol. 2985, pp. 73–88. Springer (2004)

[36] Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010), http://www.brics.dk/automaton/

[37] Oracle: javac - Java Programming Language Compiler. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html, [Online; accessed 11-April-2014]

[38] Plevyak, J.: DParser. http://dparser.sourceforge.net/, [Online; accessed 18-April-2014]

[39] Rekers, J.: Parser Generation for Interactive Environments. Ph.D. thesis, University of Amsterdam (1992), http://homepages.cwi.nl/~paulk/dissertations/Rekers.pdf

[40] Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. In: PLDI'89. pp. 170–178. ACM Press (1989)

[41] Salomon, D., Cormack, G.: The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Tech. Rep. 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada (1995)

[42] Schröer, F.W.: AMBER, an ambiguity checker for context-free grammars. Tech. rep., compilertools.net (2001), http://accent.compilertools.net/Amber.html

[43] Scott, E., Johnstone, A.: Right nulled glr parsers. ACM Trans. Program. Lang. Syst. 28(4), 577–618 (Jul 2006), http://doi.acm.org/10.1145/1146809.1146810

[44] Scott, E., Johnstone, A.: GLL parsing. Electronic Notes in Theoretical Computer Science 253(7), 177–189 (2010), http://dblp.uni-trier.de/db/journals/entcs/entcs253.html#ScottJ10

[45] Scott, E., Johnstone, A.: Recognition is not parsing - SPPF-style parsing from cubic recognisers. Science of Computer Programming 75(1-2), 55 – 70 (2010), `http://www.sciencedirect.com/science/article/pii/S0167642309000951`, special issue on {ETAPS} 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07)

[46] Scott, E., Johnstone, A.: GLL parse-tree generation. Sci. Comput. Program. 78(10), 1828–1844 (Oct 2013), `http://dx.doi.org/10.1016/j.scico.2012.03.005`

[47] Theodorou, J.: pcollections. `http://pcollections.org/`, [Online; accessed 10-June-2014]

[48] Thorup, M.: Controlled grammatic ambiguity. ACM Trans. Progr. Lang. Syst. 16(3), 1024–1050 (1994)

[49] Thorup, M.: Disambiguating grammars by exclusion of sub-parse trees. Acta Informatica 33(5), 511–522 (1996), `http://dx.doi.org/10.1007/BF03036460`

[50] Tomita, M.: Graph-structured stack and natural language parsing. In: Hobbs, J.R. (ed.) ACL. pp. 249–257

[51] Tomita, M.: Efficient Parsing for Natural Language. Kluwer Academic Publishers (1986)

[52] Visser, E.: Scannerless generalized-LR parsing. Tech. Rep. P9707, University of Amsterdam (Jul 1997), `http://citeseer.ist.psu.edu/visser97scannerles.html`

[53] Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis (1997), `http://citeseer.ist.psu.edu/visser97syntax.html`

[54] Wieland, J.: Parsing Mixfix Expressions. Ph.D. thesis, Technische Universitat Berlin (2009)

# Appendix A

# DParser grammar for mCRL2

Source: https://svn.win.tue.nl/trac/MCRL2/browser/trunk/doc/specs/mcrl2-syntax.g, revision 12508.

```
// Author(s): Wieger Wesselink
// Copyright: see the accompanying file COPYING or copy at
// https :// svn.win.tue. nl / trac /MCRL2/browser/trunk/COPYING
//
// Distributed under the Boost Software License, Version 1.0.
// (See accompanying file LICENSE_1_0.txt or copy at
// http :// www.boost.org/LICENSE_1_0.txt)
//
/// \file mcrl2-syntax.g
/// \brief dparser grammar of the mCRL2 language

${declare tokenize}
${declare longest_match}

//--- Sort expressions and sort declarations

SimpleSortExpr
  : ’Bool’                                           // Booleans
  | ’Pos’                                            // Positive numbers
  | ’Nat’                                            // Natural numbers
  | ’ Int ’                                          // Integers
  | ’Real’                                           // Reals
  | ’ List ’ ’(’ SortExpr ’)’                        // List sort
  | ’Set’ ’(’ SortExpr ’)’                           // Set sort
  | ’Bag’ ’(’ SortExpr ’)’                           // Bag sort
  | ’FSet’ ’(’ SortExpr ’)’                          // Finite set sort
  | ’FBag’ ’(’ SortExpr ’)’                          // Finite bag sort
  | Id                                               // Sort reference
  | ’(’ SortExpr ’)’                                 // Sort expression with parentheses
  | ’ struct ’ ConstrDeclList                        // Structured sort
  ;

SortExpr
  : SimpleSortExpr
  | HashArgs ’->’ SortExpr ;                         // Function sort

SortExprList: (SortExpr ’#’)* SortExpr ;             // Product sort

HashArgs: SimpleSortExpr (’#’ SimpleSortExpr)* ;     // Simple product sort
```

```
SortSpec: ’sort’ SortDecl+ ;                                    // Sort specification

SortDecl
  : IdList ’;’                                                  // List of sort identifiers
  | Id ’=’ SortExpr ’;’                                         // Sort alias
  ;

ConstrDecl: Id ( ’(’ ProjDeclList ’)’ )? ( ’?’ Id )? ;          // Constructor declaration

ConstrDeclList: ConstrDecl ( ’|’ ConstrDecl )∗ ;                // Constructor declaration list

ProjDecl: ( Id ’:’ )? SortExpr ;                                // Domain with optional projection

ProjDeclList: ProjDecl ( ’,’ ProjDecl )∗ ;                      // Declaration of projection functions

//−−− Constructors and mappings

IdsDecl: IdList ’:’ SortExpr ;                                  // Typed parameters

ConsSpec: ’cons’ ( IdsDecl ’;’ )+ ;                             // Declaration of constructors

MapSpec: ’map’ ( IdsDecl ’;’ )+ ;                               // Declaration of mappings

//−−− Equations

GlobVarSpec: ’glob’ ( VarsDeclList ’;’ )+ ;                     // Declaration of global variables

VarSpec: ’var’ ( VarsDeclList ’;’ )+ ;                          // Declaration of variables

EqnSpec: VarSpec? ’eqn’ EqnDecl+ ;                              // Definition of equations

EqnDecl: (DataExpr ’−>’)? DataExpr ’=’ DataExpr ’;’ ;           // Conditional equation

//−−− Data expressions

VarDecl: Id ’:’ SortExpr ;                                      // Typed variable

VarsDecl: IdList ’:’ SortExpr ;                                 // Typed variables

VarsDeclList: VarsDecl ( ’,’ VarsDecl )∗ ;                      // Individually typed variables

DataExpr
  : Id                                                          // Identifier
  | Number                                                      // Number
  | ’true’                                                      // True
  | ’false’                                                     // False
  | ’[’ ’]’                                                     // Empty list
  | ’{’ ’}’                                                     // Empty set
  | ’{”:”}’                                                     // Empty bag
  | ’[’ DataExprList ’]’                                        // List enumeration
  | ’{’ BagEnumEltList ’}’                                      // Bag enumeration
  | ’{’ VarDecl ’|’ DataExpr ’}’                                // Set/bag comprehension
  | ’{’ DataExprList ’}’                                        // Set enumeration
  | ’(’ DataExpr ’)’                                            // Brackets
  | DataExpr ’[’ DataExpr ’−>’ DataExpr ’]’  $unary_left 13     // Function update
  | DataExpr ’(’ DataExprList ’)’            $unary_left 13     // Function application
  | ’!’ DataExpr                             $unary_right 12    // Negation, set complement
  | ’−’ DataExpr                             $unary_right 12    // Unary minus
  | ’#’ DataExpr                             $unary_right 12    // Size of a list
  | ’forall’ VarsDeclList ’.’ DataExpr       $unary_right 1     // Universal quantifier
  | ’exists’ VarsDeclList ’.’ DataExpr       $unary_right 1     // Existential quantifier
  | ’lambda’ VarsDeclList ’.’ DataExpr       $unary_right  1    // Lambda abstraction
  | DataExpr (’=>’ $binary_op_right 2) DataExpr                // Implication
  | DataExpr (’||’ $binary_op_right 3) DataExpr                // Conjunction
  | DataExpr (’&&’ $binary_op_right 4) DataExpr                // Disjunction
  | DataExpr (’==’ $binary_op_left 5) DataExpr                 // Equality
  | DataExpr (’!=’ $binary_op_left 5) DataExpr                 // Inequality
  | DataExpr (’<’ $binary_op_left 6) DataExpr                  // Smaller
  | DataExpr (’<=’ $binary_op_left 6) DataExpr                 // Smaller equal
  | DataExpr (’>=’ $binary_op_left 6) DataExpr                 // Larger equal
  | DataExpr (’>’ $binary_op_left 6) DataExpr                  // Larger
  | DataExpr (’in’ $binary_op_left 6) DataExpr                 // Set, bag, list membership
  | DataExpr (’|>’ $binary_op_right 7) DataExpr                // List cons
  | DataExpr (’<|’ $binary_op_left 8) DataExpr                 // List snoc
  | DataExpr (’++’ $binary_op_left 9) DataExpr                 // List concatenation
  | DataExpr (’+’ $binary_op_left 10) DataExpr                 // Addition, set/bag union
  | DataExpr (’−’ $binary_op_left 10) DataExpr                 // Subtraction, set/bag difference
  | DataExpr (’/’ $binary_op_left 11) DataExpr                 // Division
  | DataExpr (’div’ $binary_op_left 11) DataExpr               // Integer div
  | DataExpr (’mod’ $binary_op_left 11) DataExpr               // Integer mod
  | DataExpr (’∗’ $binary_op_left 12) DataExpr                 // Multiplication , set/bag intersection
  | DataExpr (’.’ $binary_op_left 12) DataExpr                 // List element at position
  | DataExpr ’whr’ AssignmentList ’end’      $unary_left 0      // Where clause
  ;

DataExprUnit
  : Id                                                         // Identifier
  | Number                                                     // Number
  | ’true’                                                     // True
  | ’false’                                                    // False
  | ’(’ DataExpr ’)’                                           // Bracket
  | DataExprUnit ’(’ DataExprList ’)’        $unary_left  14    // Function application
  | ’!’ DataExprUnit                         $unary_right 13    // Negation, set complement
  | ’−’ DataExprUnit                         $unary_right 13    // Unary minus
  | ’#’ DataExprUnit                         $unary_right 13    // Size of a list
  ;

Assignment: Id ’=’ DataExpr ;                                  // Assignment

AssignmentList: Assignment ( ’,’ Assignment )∗ ;               // Assignment list

DataExprList: DataExpr ( ’,’ DataExpr )∗ ;                     // Data expression list

BagEnumElt: DataExpr ’:’ DataExpr ;                            // Bag element with multiplicity

BagEnumEltList: BagEnumElt ( ’,’ BagEnumElt )∗ ;               // Elements in a finite bag

//−−− Communication and renaming sets

ActIdSet: ’{’ IdList ’}’ ;                                     // Action set

MultActId: Id ( ’|’ Id )∗ ;                                    // Multi−action label

MultActIdList: MultActId ( ’,’ MultActId )∗ ;                  // Multi−action labels

MultActIdSet: ’{’ MultActIdList? ’}’ ;                         // Multi−action label set

CommExpr: Id ’|’ MultActId ’−>’ Id ;                           // Action synchronization

CommExprList: CommExpr ( ’,’ CommExpr )∗ ;                     // Action synchronizations

CommExprSet: ’{’ CommExprList? ’}’ ;                           // Action synchronization set

```

```
RenExpr: Id '->' Id ;                                      // Action renaming

RenExprList: RenExpr ( ',' RenExpr )* ;                    // Action renamings

RenExprSet: '{' RenExprList? '}' ;                         // Action renaming set

//--- Process expressions

ProcExpr
  : Action                                                 // Action or process instantiation
  | Id '(' AssignmentList? ')'                             // Process assignment
  | 'delta'                                                // Delta, deadlock, inaction
  | 'tau'                                                  // Tau, hidden action, empty multi-action
  | 'block' '(' ActIdSet ',' ProcExpr ')'                  // Block or encapsulation operator
  | 'allow' '(' MultActIdSet ',' ProcExpr ')'              // Allow operator
  | 'hide' '(' ActIdSet ',' ProcExpr ')'                   // Hiding operator
  | 'rename' '(' RenExprSet ',' ProcExpr ')'               // Action renaming operator
  | 'comm' '(' CommExprSet ',' ProcExpr ')'                // Communication operator
  | '(' ProcExpr ')'                                       // Brackets
  | ProcExpr ('+' $binary_op_left 1) ProcExpr              // Choice operator
  | ('sum' VarsDeclList '.' $unary_op_right 2) ProcExpr    // Sum operator
  | ProcExpr ('||' $binary_op_right 3) ProcExpr            // Parallel operator
  | ProcExpr ('||_' $binary_op_right 4) ProcExpr           // Leftmerge operator
  | (DataExprUnit '->' $unary_op_right 5) ProcExpr         // If-then operator
  | (DataExprUnit IfThen $unary_op_right 5) ProcExpr       // If-then-else operator
  | ProcExpr ('<<' $binary_op_left 6) ProcExpr             // Until operator
  | ProcExpr ('.' $binary_op_right 7) ProcExpr             // Sequential composition operator
  | ProcExpr ('@' $binary_op_left 8) DataExprUnit          // At operator
  | ProcExpr ('|' $binary_op_left 9) ProcExpr              // Communication merge
  ;

ProcExprNoIf
  : Action                                                 // Action or process instantiation
  | Id '(' AssignmentList? ')'                             // Process assignment
  | 'delta'                                                // Delta, deadlock, inaction
  | 'tau'                                                  // Tau, hidden action, empty multi-action
  | 'block' '(' ActIdSet ',' ProcExpr ')'                  // Block or encapsulation operator
  | 'allow' '(' MultActIdSet ',' ProcExpr ')'              // Allow operator
  | 'hide' '(' ActIdSet ',' ProcExpr ')'                   // Hiding operator
  | 'rename' '(' RenExprSet ',' ProcExpr ')'               // Action renaming operator
  | 'comm' '(' CommExprSet ',' ProcExpr ')'                // Communication operator
  | '(' ProcExpr ')'                                       // Brackets
  | ProcExprNoIf ('+' $binary_op_left 1) ProcExprNoIf      // Choice operator
  | ('sum' VarsDeclList '.' $unary_op_right 2) ProcExprNoIf  // Sum operator
  | ProcExprNoIf ('||' $binary_op_right 3) ProcExprNoIf    // Parallel operator
  | ProcExprNoIf ('||_' $binary_op_right 3) ProcExprNoIf   // Leftmerge operator
  | (DataExprUnit IfThen $unary_op_right 4) ProcExprNoIf   // If-then-else operator
  | ProcExprNoIf ('<<' $binary_op_left 5) ProcExprNoIf     // Until operator
  | ProcExprNoIf ('.' $binary_op_right 6) ProcExprNoIf     // Sequential composition operator
  | ProcExprNoIf ('@' $binary_op_left 7) DataExprUnit      // At operator
  | ProcExprNoIf ('|' $binary_op_left 8) ProcExprNoIf      // Communication merge
  ;

IfThen: '->' ProcExprNoIf '<>' $left 0 ;                   // Auxiliary if-then-else

//--- Actions

Action: Id ( '(' DataExprList ')' )? ;                     // Action, process instantiation

ActDecl: IdList ( ':' SortExprList )? ';' ;                // Declarations of actions

ActSpec: 'act' ActDecl+ ;                                  // Action specification
```

```

MultAct
  : 'tau'                                                   // Tau, hidden action, empty multi-action
  | ActionList                                              // Multi-action
  ;

ActionList: Action ( '|' Action )* ;                        // List of actions

//--- Process and initial state declaration

ProcDecl: Id ( '(' VarsDeclList ')' )? '=' ProcExpr ';' ;   // Process declaration

ProcSpec: 'proc' ProcDecl+ ;                                // Process specification

Init: 'init' ProcExpr ';' ;                                 // Initial process

//--- Data specification

DataSpec: ( SortSpec | ConsSpec | MapSpec | EqnSpec )+ ;    // Data specification

//--- mCRL2 specification

mCRL2Spec: mCRL2SpecElt* Init mCRL2SpecElt* ;               // MCRL2 specification

mCRL2SpecElt
  : SortSpec                                                // Sort specification
  | ConsSpec                                                // Constructor specification
  | MapSpec                                                 // Map specification
  | EqnSpec                                                 // Equation specification
  | GlobVarSpec                                             // Global variable specification
  | ActSpec                                                 // Action specification
  | ProcSpec                                                // Process specification
  ;

//--- Boolean equation system

BesSpec: BesEqnSpec BesInit ;                               // Boolean equation system

BesEqnSpec: 'bes' BesEqnDecl+ ;                             // Boolean equation declaration

BesEqnDecl: FixedPointOperator BesVar '=' BesExpr ';' ;     // Boolean fixed point equation

BesVar: Id ;                                                // BES variable

BesExpr
  : 'true'                                                  // True
  | 'false'                                                 // False
  | BesExpr ('=>' $binary_op_right 2) BesExpr               // Implication
  | BesExpr ('||' $binary_op_right 3) BesExpr               // Disjunction
  | BesExpr ('&&' $binary_op_right 4) BesExpr               // Conjunction
  | '!' BesExpr            $unary_right 5                   // Negation
  | '(' BesExpr ')'                                         // Brackets
  | BesVar                                                  // Boolean variable
  ;

BesInit: 'init' BesVar ';' ;                                // Initial BES variable

//--- Parameterized Boolean equation systems

PbesSpec: DataSpec? GlobVarSpec? PbesEqnSpec PbesInit ;     // PBES specification

PbesEqnSpec: 'pbes' PbesEqnDecl+ ;                          // Declaration of PBES equations
```

```

PbesEqnDecl: FixedPointOperator PropVarDecl '=' PbesExpr ';' ;  // PBES equation

FixedPointOperator
  : 'mu'                                                          // Minimal fixed point operator
  | 'nu'                                                          // Maximal fixed point operator
  ;

PropVarDecl: Id ( '(' VarsDeclList ')' )? ;                       // PBES variable declaration

PropVarInst: Id ( '(' DataExprList ')' )? ;                       // Instantiated PBES variable

PbesInit: 'init' PropVarInst ';' ;                                // Initial PBES variable

DataValExpr: 'val' '(' DataExpr ')';                              // Marked data expression

PbesExpr
  : DataValExpr                                                   // Data expression
  | 'true'                                                        // True
  | 'false'                                                       // False
  | 'forall' VarsDeclList '.' PbesExpr      $unary_right  0       // Universal quantifier
  | 'exists' VarsDeclList '.' PbesExpr      $unary_right  0       // Existential quantifier
  | PbesExpr ('=>' $binary_op_right 2) PbesExpr                   // Implication
  | PbesExpr ('||' $binary_op_right 3) PbesExpr                   // Disjunction
  | PbesExpr ('&&' $binary_op_right 4) PbesExpr                   // Conjunction
  | '!' PbesExpr                            $unary_right  5       // Negation
  | '(' PbesExpr ')'                                              // Brackets
  | PropVarInst                                                   // Propositional variable
  ;

//--- Action formulas

ActFrm
  : MultAct                                 $left 10              // Multi-action
  | '(' ActFrm ')'                          $left 11              // Brackets
  | DataValExpr                             $left 20              // Boolean data expression
  | 'true'                                                        // True
  | 'false'                                                       // False
  | '!' ActFrm                              $unary_right  6       // Negation
  | 'forall' VarsDeclList '.' ActFrm        $unary_right  0       // Universal quantifier
  | 'exists' VarsDeclList '.' ActFrm        $unary_right  0       // Existential quantifier
  | ActFrm ('@' $binary_op_left 5) DataExpr                       // At operator
  | ActFrm ('&&' $binary_op_right 4) ActFrm                       // Intersection of actions
  | ActFrm ('||' $binary_op_right 3) ActFrm                       // Union actions
  | ActFrm ('=>' $binary_op_right 2) ActFrm                       // Implication
  ;

//--- Regular formulas

RegFrm
  : ActFrm                                  $left 20              // Action formula
  | '(' RegFrm ')'                          $left 21              // Brackets
  | 'nil'                                                         // Empty regular formula
  | RegFrm ('+' $binary_op_left 1) RegFrm                         // Alternative composition
  | RegFrm ('.' $binary_op_right 2) RegFrm                        // Sequential composition
  | RegFrm '*'                              $unary_right  3       // Iteration
  | RegFrm '+'                              $unary_right  3       // Nonempty iteration
  ;

//--- State formulas

StateFrm
```

```
  : DataValExpr                                  $left 20         // Data expression
  | '(' StateFrm ')'                             $left 20         // Brackets
  | 'true'                                                        // True
  | 'false'                                                       // False
  | 'mu' StateVarDecl '.' StateFrm               $unary_right 1   // Minimal fixed point
  | 'nu' StateVarDecl '.' StateFrm               $unary_right 1   // Maximal fixed point
  | 'forall' VarsDeclList '.' StateFrm           $unary_right 2   // Universal quantification
  | 'exists' VarsDeclList '.' StateFrm           $unary_right 2   // Existential quantification
  | StateFrm ('=>' $binary_op_right 3) StateFrm                   // Implication
  | StateFrm ('||' $binary_op_right 4) StateFrm                   // Disjunction
  | StateFrm ('&&' $binary_op_right 5) StateFrm                   // Conjunction
  | '[' RegFrm ']' StateFrm                      $unary_right 6   // Box modality
  | '<' RegFrm '>' StateFrm                      $unary_right 6   // Diamond modality
  | '!' StateFrm                                 $unary_right 7   // Negation
  | Id ( '(' DataExprList ')' )?                                  // Instantiated PBES variable
  | 'delay' ( '@' DataExpr )?                                     // Delay
  | 'yaled' ( '@' DataExpr )?                                     // Yaled
  ;

StateVarDecl: Id ( '(' StateVarAssignmentList ')' )? ;            // PBES variable declaration

StateVarAssignment: Id ':' SortExpr '=' DataExpr ;                // Typed variable with initial value

StateVarAssignmentList: StateVarAssignment ( ',' StateVarAssignment )* ; // Typed variable list

//--- Action Rename Specifications

ActionRenameSpec: (SortSpec | ConsSpec | MapSpec | EqnSpec | ActSpec | ActionRenameRuleSpec)+ ;
                                                                  // Action rename specification

ActionRenameRuleSpec: VarSpec? 'rename' ActionRenameRule+ ; // Action rename rule section

ActionRenameRule: (DataExpr '->')? Action '=>' ActionRenameRuleRHS ';' ; // Conditional action renaming

ActionRenameRuleRHS
  : Action                                                        // Action
  | 'tau'                                                         // Tau, hidden action, empty multi-action
  | 'delta'                                                       // Delta, deadlock, inaction
  ;

//--- Identifiers

IdList: Id ( ',' Id )* ;                                          // List of identifiers

Id: "[A-Za-z_][A-Za-z_0-9']*" $term -1 ;                          // Identifier

Number: "0|([1-9][0-9]*)" $term -1 ;                              // Number

//--- Whitespace

whitespace: "([ \t\n\r]|(%[^\n\r]*))*" ;                          // Whitespace
```

# Appendix B

# BNF grammar for mCRL2

The GLL implementation used for this thesis only supports BNF. This appendix contains the BNF variant of the EBNF grammar of mCRL2. Parts of the grammar that were introduced to avoid ambiguities have been removed. For instance, this grammar does not contain the nonterminals $ProcExprNoIf$ and $IfThen$, because a precede/follow filter is used to resolve the dangling else ambiguity. The translation from EBNF to BNF is done according to the rules described in Section 6.2.3. In the BNF grammar, $\varepsilon$ is denoted by an empty alternate.

```
mCRL2Spec ::= mCRL2SpecStar Init mCRL2SpecStar;

mCRL2SpecStar ::= mCRL2SpecElt mCRL2SpecStar | ;

mCRL2SpecElt
::= SortSpec
  | ConsSpec
  | MapSpec
  | EqnSpec
  | GlobVarSpec
  | ActSpec
  | ProcSpec
  ;

SortExpr
::= ”Bool”
  | ”Pos”
  | ”Nat”
  | ”Int”
  | ”Real”
  | ”List” ”(” SortExpr ”)”
  | ”Set” ”(” SortExpr ”)”
  | ”Bag” ”(” SortExpr ”)”
  | ”FSet” ”(” SortExpr ”)”
  | ”FBag” ”(” SortExpr ”)”
  | Id
  | ”(” SortExpr ”)”
  | Domain ”−>” SortExpr
  | ”struct” ConstrDeclList
  ;

```

```
Domain ::= SortExprList;

SortExprList  ::=  SortExpr SortExprStar;
SortExprStar ::=  ”#”  SortExpr SortExprStar | ;

SortSpec ::= ”sort”  SortSpecPlus;

SortSpecPlus
 ::=  SortDecl
   |  SortSpecPlus SortDecl
   ;

SortDecl
 ::=  IdList   ”;”
   |  IdList  ”=”  SortExpr  ”;”
   ;

ConstrDecl ::= Id  ConstrDeclOptOne ConstrDeclOptTwo;
ConstrDeclOptOne ::= ”(” ProjDeclList  ”)”   |  ;
ConstrDeclOptTwo ::= ”?” Id |  ;

ConstrDeclList ::=  ConstrDecl ConstrDeclListStar;
ConstrDeclListStar ::=  ”|”  ConstrDecl ConstrDeclListStar |  ;

ProjDecl ::=  ProjDeclOpt SortExpr;
ProjDeclOpt ::=  Id   ”:”   |  ;

ProjDeclList  ::=  ProjDecl ProjDeclListStar;
ProjDeclListStar  ::=   ”,”  ProjDecl ProjDeclListStar  |  ;




IdsDecl ::=  IdList   ”:”  SortExpr;

ConsSpec ::= ”cons” ConsSpecPlus;
ConsSpecPlus ::= IdsDecl ”;” ConsSpecPlus | IdsDecl ”;” ;

MapSpec ::= ”map” MapSpecPlus;
MapSpecPlus ::= IdsDecl ”;” MapSpecPlus | IdsDecl ”;” ;


GlobVarSpec ::= ”glob” GlobVarSpecPlus ;

GlobVarSpecPlus ::= VarsDeclList ”;” GlobVarSpec | VarsDeclList  ”;”;

VarSpec ::= ”var”  VarSpecPlus;
VarSpecPlus ::= VarsDeclList  ”;”  VarSpecPlus | VarsDeclList  ”;”;

EqnSpec ::= EqnSpecOpt ”eqn” EqnSpecPlus;
EqnSpecOpt ::= VarSpec | ;
EqnSpecPlus ::= EqnDecl EqnSpecPlus | EqnDecl;

EqnDecl ::= EqnDeclOpt DataExpr ”=” DataExpr ”;”;
EqnDeclOpt ::= DataExpr ”−>” | ;


VarDecl ::=  Id   ”:”  SortExpr;
VarsDecl ::=  IdList   ”:”  SortExpr;
VarsDeclList  ::=  VarsDecl VDLStar;
VDLStar ::=  ”,”  VarsDecl VDLStar | ;

```

```
DataExpr ::= Id
   | Number
   | ”true”
   | ”false”
   | ”[” ”]”
   | ”{” ”}”
   | ”{” ”:” ”]”
   | ”[” DataExprList ”]”
   | ”{” BagEnumEltList ”}”
   | ”{” VarDecl ”|” DataExpr ”}”
   | ”{” DataExprList ”}”
   | ”(” DataExpr ”)”
   | DataExpr ”[” DataExpr ”−>” DataExpr ”]”
   | DataExpr ”(” DataExprList ”)”
   | ”!” DataExpr
   | ”−” DataExpr
   | ”#” DataExpr
   | ” forall ” VarsDeclList ”.” DataExpr
   | ”exists” VarsDeclList ”.” DataExpr
   | ”lambda” VarsDeclList ”.” DataExpr
   | DataExpr ”=>” DataExpr
   | DataExpr ”||” DataExpr
   | DataExpr ”&&” DataExpr
   | DataExpr ”==” DataExpr
   | DataExpr ”!=” DataExpr
   | DataExpr ”<” DataExpr
   | DataExpr ”<=” DataExpr
   | DataExpr ”>=” DataExpr
   | DataExpr ”>” DataExpr
   | DataExpr ”in” DataExpr
   | DataExpr ”|>” DataExpr
   | DataExpr ”<|” DataExpr
   | DataExpr ”++” DataExpr
   | DataExpr ”+” DataExpr
   | DataExpr ”−” DataExpr
   | DataExpr ”/” DataExpr
   | DataExpr ”div” DataExpr
   | DataExpr ”mod” DataExpr
   | DataExpr ”∗” DataExpr
   | DataExpr ”.” DataExpr
   | DataExpr ”whr” AssignmentList ”end”
   ;

DataExprUnit ::= Id
   | Number
   | ”true”
   | ”false”
   | ”(” DataExpr ”)”
   | DataExprUnit ”(” DataExprList ”)”
   | ”!” DataExprUnit
   | ”−” DataExprUnit
   | ”#” DataExprUnit
   ;

Assignment ::= Id ”=” DataExpr ;

AssignmentList ::= Assignment ALS;
ALS ::= ”,” Assignment ALS | ;

DataExprList ::= DataExpr DELS;
DELS ::= ”,” DataExpr DELS | ;
```

```
BagEnumElt ::= DataExpr ”:” DataExpr;
BagEnumEltList ::= BagEnumElt BEELS;
BEELS ::= ”,” BagEnumElt BEELS | ;


ActIdSet ::= ”{” IdList ”}” ;

MultActId ::= Id MultActIdStar;
MultActIdStar ::= ”|” Id MultActIdStar | ;

MultActIdList ::= MultActId MultActIdListStar;
MultActIdListStar ::= ”,” MultActId MultActIdListStar | ;


MultActIdSet ::= ”{” MultActIdSetOpt ”}” ;
MultActIdSetOpt ::= MultActIdList | ;

CommExpr ::= Id ”|” MultActId ”−>” Id;

CommExprList ::= CommExpr CommExprListStar;
CommExprListStar ::= ”,” CommExpr CommExprListStar | ;

CommExprSet ::= ”{” CommExprSetOpt ”}”;
CommExprSetOpt ::= CommExprList | ;

RenExpr ::= Id ”−>” Id ;
RenExprList ::= RenExpr RenExprListStar;
RenExprListStar ::= ”,” RenExpr RenExprListStar | ;

RenExprSet ::= ”{” RenExprSetOpt ”}” ;
RenExprSetOpt ::= RenExprList | ;




ProcExpr
   ::= Action
   | Id ”(” AssignmentList ”)”
   | Id ”(” ”)”
   | ”delta”
   | ”tau”
   | ”block” ”(” ActIdSet ”,” ProcExpr ”)”
   | ”allow” ”(” MultActIdSet ”,” ProcExpr ”)”
   | ”hide” ”(” ActIdSet ”,” ProcExpr ”)”
   | ”rename” ”(” RenExprSet ”,” ProcExpr ”)”
   | ”comm” ”(” CommExprSet ”,” ProcExpr ”)”
   | ”(” ProcExpr ”)”
   | ProcExpr ”+” ProcExpr
   | ”sum” VarsDeclList ”.” ProcExpr
   | ProcExpr ”||” ProcExpr
   | ProcExpr ”||_” ProcExpr
   | DataExprUnit ”−>” ProcExpr
   | DataExprUnit ”−>” ProcExpr ”<>” ProcExpr
   | ProcExpr ”<<” ProcExpr
   | ProcExpr ”.” ProcExpr
   | ProcExpr ”@” DataExprUnit
   | ProcExpr ”|” ProcExpr
   ;



Action ::= Id ”(” DataExprList ”)” | Id;
```

```

ActDecl ::= IdList ”:” SortExprList ”;” | IdList ”;” ;

ActSpec ::= ”act” ActSpecPlus ;

ActSpecPlus ::= ActDecl ActSpecPlus | ActDecl;

MultAct ::= ”tau” | ActionList ;

 ActionList ::= Action ActionListStar ;
 ActionListStar ::= ”|” Action ActionListStar | ;

ProcDecl ::= Id ”(” VarsDeclList ”)” ”=” ProcExpr ”;” | Id ”=” ProcExpr ”;” ;

ProcSpec ::= ”proc” ProcSpecPlus ;

ProcSpecPlus ::= ProcDecl ProcSpecPlus | ProcDecl;

 Init ::= ” init ” ProcExpr ”;” ;

Id ::= ’[ A−Za−z_][A−Za−z_0−9]∗’;

 IdList ::= Id IdListStar ;
 IdListStar ::= ”,” Id IdListStar | ;

Number ::= ’0|([1−9][0−9]∗)’;
```

# Appendix C

# Experimental data

Table C.1 contains the data that we obtained after running the parser without filtering, and the parser with parse-time filtering and post-parse time filtering. All execution times are in milliseconds. Column *Parsing & Filters* contains the running time of the parser with parse-time filtering. The total execution time of parsing and parse-time filtering versus only parsing is shown in column *Difference*. Note that the Prefer and LORO filters are used for post-parse filtering, and are only run if the ambiguity cannot be resolved on parse-time. If all ambiguities are resolved on parse time, the running time of the post-parse filters is zero.

Table C.1: Experimental data.

| Input file | Tokens | $\Xi(\mathcal{S})$ | Parsing | Parsing & filters | Prefer | LORO | Difference |
|---|---|---|---|---|---|---|---|
| gpa-10-1 | 206 | 0.091 | 170 | 202 | 0 | 0 | 0.188 |
| block | 222 | 0.189 | 231 | 270 | 0 | 0 | 0.169 |
| gpa-10-2 | 222 | 0.095 | 213 | 259 | 0 | 0 | 0.216 |
| magic-square | 227 | 0.870 | 1197 | 655 | 0 | 0 | -0.453 |
| dining8 | 275 | 0.180 | 295 | 341 | 0 | 0 | 0.156 |
| allow | 277 | 0.216 | 250 | 270 | 0 | 0 | 0.080 |
| dining-10 | 287 | 0.257 | 391 | 458 | 0 | 0 | 0.171 |
| wolf-goat-cabbage-1 | 291 | 0.148 | 544 | 639 | 0 | 0 | 0.175 |
| dining3-ns-seq | 300 | 0.103 | 436 | 451 | 0 | 0 | 0.034 |
| dining3-ns | 300 | 0.104 | 454 | 395 | 0 | 0 | -0.130 |
| mpsu | 325 | 0.701 | 672 | 692 | 0 | 0 | 0.030 |
| bakery | 328 | 0.058 | 386 | 537 | 0 | 0 | 0.391 |
| dining3-seq | 352 | 0.079 | 685 | 739 | 0 | 0 | 0.079 |
| dining3-cs | 366 | 0.141 | 496 | 585 | 0 | 0 | 0.179 |
| dining3-cs-seq | 366 | 0.140 | 482 | 557 | 0 | 0 | 0.156 |
| leader | 399 | 0.187 | 552 | 585 | 0 | 0 | 0.060 |
| dining3 | 400 | 0.119 | 758 | 843 | 0 | 0 | 0.112 |
| rational | 406 | 0.411 | 835 | 931 | 0 | 0 | 0.115 |
| scheduler | 424 | 0.133 | 438 | 503 | 0 | 0 | 0.148 |
| dining3-schedule-seq | 434 | 0.132 | 725 | 790 | 0 | 0 | 0.090 |

Continued on next page

**Table C.1 – continued from previous page**

| Input file | Tokens | $\Xi(\mathcal{S})$ | Parsing | Parsing & filters | Prefer | LORO | Difference |
|---|---|---|---|---|---|---|---|
| abp | 439 | 0.133 | 446 | 471 | 0 | 0 | 0.056 |
| fischer | 439 | 0.109 | 451 | 484 | 0 | 0 | 0.073 |
| trains | 459 | 0.184 | 284 | 335 | 0 | 0 | 0.180 |
| dining3-schedule | 482 | 0.155 | 796 | 933 | 0 | 0 | 0.172 |
| wolf-goat-cabbage | 490 | 0.094 | 1499 | 1554 | 0 | 0 | 0.037 |
| abp-bw | 496 | 0.079 | 486 | 564 | 0 | 0 | 0.160 |
| fischer-10 | 519 | 0.115 | 568 | 648 | 0 | 0 | 0.141 |
| cellular-automata | 622 | 0.257 | 1311 | 1536 | 0 | 0 | 0.172 |
| food-package | 657 | 0.458 | 1898 | 1928 | 0 | 0 | 0.016 |
| cabp | 674 | 0.146 | 617 | 655 | 0 | 0 | 0.062 |
| swp-func | 733 | 0.078 | 1116 | 1397 | 0 | 0 | 0.252 |
| par | 734 | 0.083 | 591 | 690 | 0 | 0 | 0.168 |
| onebit | 760 | 0.253 | 835 | 939 | 1 | 4 | 0.131 |
| sets-bags | 764 | 0.568 | 1069 | 1124 | 0 | 0 | 0.051 |
| numbers | 882 | 0.971 | 8143 | 1776 | 0 | 0 | -0.782 |
| swp-lists | 899 | 0.074 | 1913 | 2110 | 0 | 0 | 0.103 |
| swp-fgpbp | 921 | 0.107 | 1643 | 1747 | 0 | 0 | 0.063 |
| domineering | 999 | 0.265 | 2561 | 2847 | 0 | 0 | 0.112 |
| game-of-goose | 1019 | 0.203 | 2303 | 2566 | 2 | 28 | 0.127 |
| hex | 1248 | 0.264 | 3186 | 3624 | 0 | 0 | 0.137 |
| brp | 1259 | 0.069 | 1685 | 1936 | 0 | 0 | 0.149 |
| clobber | 1362 | 0.239 | 3504 | 3994 | 0 | 0 | 0.140 |
| snake | 1419 | 0.125 | 3497 | 4175 | 0 | 0 | 0.194 |
| knights | 1426 | 0.313 | 3403 | 3759 | 0 | 0 | 0.105 |
| othello | 1496 | 0.110 | 3256 | 3753 | 0 | 0 | 0.153 |
| wafer-stepper | 1508 | 0.121 | 2602 | 3036 | 0 | 0 | 0.167 |
| rubiks-cube | 1811 | 0.075 | 3702 | 4293 | 0 | 0 | 0.160 |
| alma | 1843 | 0.178 | 2652 | 3004 | 0 | 0 | 0.133 |
| lift3-init | 1847 | 0.058 | 3227 | 3666 | 0 | 0 | 0.136 |
| lift3-final | 2105 | 0.059 | 3588 | 4259 | 0 | 0 | 0.187 |
| four-in-a-row | 2121 | 0.131 | 5519 | 6106 | 0 | 0 | 0.106 |
| peg-solitaire | 2525 | 0.157 | 6870 | 7626 | 0 | 0 | 0.110 |
| swp-with-tanenbaums-bug | 2739 | 0.280 | 4768 | 5159 | 0 | 0 | 0.082 |
| bke | 3255 | 0.100 | 5636 | 6227 | 0 | 0 | 0.105 |
| chatbox | 3840 | 0.912 | 18786 | 14105 | 0 | 0 | -0.249 |
| commprot | 4019 | 0.109 | 6939 | 7557 | 0 | 0 | 0.089 |
| SMS | 4206 | 0.453 | 5371 | 6350 | 0 | 0 | 0.182 |
| 11073 | 4428 | 0.674 | 8321 | 9142 | 0 | 0 | 0.099 |
| 1394-fin | 4452 | 0.111 | 6863 | 8092 | 0 | 0 | 0.179 |
| WMS | 5987 | 0.925 | 15512 | 10927 | 0 | 0 | -0.296 |
| garage-r3 | 9323 | 0.040 | 19592 | 24071 | 0 | 0 | 0.229 |
| garage-r2-error | 9405 | 0.038 | 20944 | 23718 | 0 | 0 | 0.132 |
| garage-r2 | 9459 | 0.040 | 19991 | 23654 | 0 | 0 | 0.183 |
| garage | 9815 | 0.049 | 21401 | 24380 | 0 | 0 | 0.139 |
| garage-r1 | 9818 | 0.049 | 21792 | 24648 | 0 | 0 | 0.131 |
| garage-ver | 11545 | 0.669 | 27377 | 29114 | 0 | 0 | 0.063 |