Using Filters for the Disambiguation of Context-free Grammars*

Paul Klint§¶
paulk@cwi.nl

Eelco Visser§
visser@fwi.uva.nl

Abstract

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

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

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

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

1 Introduction

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

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

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

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

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

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

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

the parse forest (rather than the single parse tree) that will be produced by the parser. Disambiguation encompasses the whole spectrum from simple priority declarations for resolving ambiguities in arithmetic expressions to the use of semantic (e.g., type) information for pruning the parse forest. As a last resort, the *user* of the parser may have to resolve ambiguities interactively.

In this paper we explore disambiguation mechanisms for general context-free grammars and their impact on parsing. We will concentrate on "logical" disambiguation that can be modeled by a *filter* on sets of parse trees. This excludes disambiguation methods that are inherently intertwined with a specific parsing technique. We study the expressiveness of various filters and their interaction with parsing: as a general rule simpler filters can be applied earlier (during parsing or even during parser generation).

This research was motivated by our experience with the modular syntax definition formalism SDF (Syntax Definition Formalism) [HHKR92] and its implementation based on generalized LR parsing [Rek92]. Although quite elaborate disambiguation techniques are being used (local conflict detection based on priority and associativity, and a multi-set ordering for pruning the parse forest) we keep encountering examples where more fine-tuned filtering would be useful. This suggests an approach based on extensible, user-defined, disambiguation filters. For efficiency reasons, it will be advantageous to apply these filters as early as possible.

The rest of this paper is structured as follows. In section 2 we consider several characteristics of disambiguation methods. In section 3 we introduce some preliminary terminology about context-free grammars and parsing. In section 4 we define the notion *filter* on sets of parse trees, the disambiguation of a context-free grammar by a filter and several properties of filters. In section 5 we discuss several examples illustrating the expressive power of filters. In section 6 we investigate how filters can be used in generating parsers for disambiguated languages by considering the implementation of parsers for context-free grammars with priorities. Finally in section 7 we discuss related work and related issues.

2 Disambiguation

A disambiguation mechanism for context-free languages is a procedure that chooses from a range of possible parses for a sentence the most appropriate one according to some criterium. The architecture we propose to use for disambiguation consists of three parts (see Figure 1):

Grammar description: a *context-free grammar* and a set of *disambiguation rules.* Disambiguation rules concern *lexical disambiguation rules* (e.g., preference for a longest match, preference for keywords over identifiers), *context-free disambiguation rules* (e.g., precedence relations between operators), and *static semantic disambiguation rules* (e.g., type or declaration dependent rules).

Generation phase: a grammar transformer and a parser generator. Typical grammar transformations are the elimination of left/right recursion, and the coding of priority and associativity information in grammar rules. Parser generation is most likely based on standard GLR techniques [Tom85, Rek92].

Parsing phase: a parser/filter pipeline that transforms input sentences into a single (unambiguous) parse tree.

Given this architecture, we can classify disambiguation methods according to the following characteristics:

Interference of context-free grammar and disambiguation rules. In Figure 1, we suggest that the given context-free grammar and disambiguation rules are completely

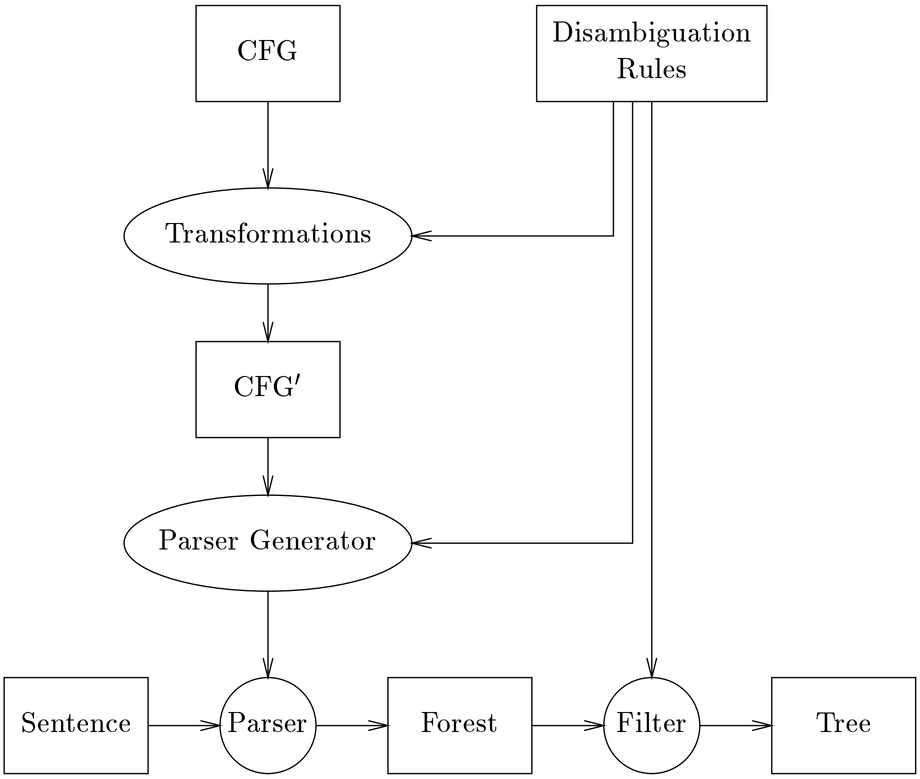


Figure 1: Phases in parsing with ambiguous grammars

disjoint. In many cases, however, they will interfere with each other. For instance, disambiguation rules may be embedded in grammar rules, or the order of grammar rules may have a significance for disambiguation. In this paper, we will keep them disjoint.

Dependence on parsing method. Disambiguation can be defined in terms of parse actions (and is then closely intertwined with parsing) or it can be understood independently from the parsing method used. We will adopt this latter view and we consider the first approach only when it is an implementation method of the latter.

Moment of disambiguation. Disambiguation can take place during grammar transformation, during parser generation, and during post-parse filtering. As a general rule, deferring disambiguation is expensive but can be used to implement very general methods. Our strategy will be to define *all* disambiguation methods as post-parse filters and to seek implementation techniques that apply them (transparently but more efficiently) at an earlier moment.

Semantic assumptions. An issue in the disambiguation of grammars is the question whether the derivations of an ambiguous sentence should all have the same meaning. In natural language parsing, this is clearly not true. In some other approaches like, e.g., [Tho94a, Tho94b] this seems an essential assumption. In addition, it is not clear whether each sentence generated by the underlying CFG should also be a sentence of the disambiguated grammar. This property is called ‘completeness’ in [Tho94a]. But if we consider the language of type-correct Pascal programs we see that this property does not hold.

3 Preliminaries

3.1 Context-free Grammars

DEFINITION 3.1
A *context-free grammar (CFG)* $\mathcal{G}$ is a triple $\langle V_N, V_T, \mathcal{P}\rangle$, where V_N is a set of nonterminal symbols, V_T a set of terminal symbols, V the set of symbols of $\mathcal{G}$ is $V_N \cup V_T$, and $\mathcal{P} \subseteq V_N \times V^*$ a set of productions. We write $A \to \alpha$ for a production $p = \langle A, \alpha\rangle \in \mathcal{P}$. □

We will sometimes refer to a production by a number or by an abbreviation of its symbols, e.g., $E \to E + E$ is abbreviated as $+$.

We will characterize the language generated by a CFG by the parse trees it generates instead of by derivations as is usually done. This method is as clear as derivations and has the advantage that the semantics of filters is easily definable.

DEFINITION 3.2 A CFG $\mathcal{G}$ generates a family of sets of *parse trees* $\mathcal{T}^{\mathcal{G}} = (\mathcal{T}^{\mathcal{G}}_X \mid X \in V)$, which contains the minimal sets $\mathcal{T}^{\mathcal{G}}_X$ such that

$$\begin{array}{ll} X \in \mathcal{T}^{\mathcal{G}}_X & \Leftarrow X \in V \\ [A \to t_\alpha] \in \mathcal{T}^{\mathcal{G}}_A & \Leftarrow A \to \alpha \in \mathcal{P} \wedge t_\alpha \in \mathcal{T}^{\mathcal{G}}_\alpha \end{array}$$

where $\mathcal{T}^{\mathcal{G}}_{X_1 \ldots X_n} = \{t_1 \ldots t_n \mid t_i \in \mathcal{T}^{\mathcal{G}}_{X_i}\}$. The *signature* of a tree is the production used to construct the root of a tree; $sign([A \to t_\alpha]) = A \to \alpha$ □

We omit the superscript $\mathcal{G}$ from $\mathcal{T}^{\mathcal{G}}$ when the grammar $\mathcal{G}$ is clear from context. We will identify $\mathcal{T}$ and $\bigcup_{X \in V} \mathcal{T}_X$ when appropriate.

According to this definition we should write $[E \to [E \to a] + [E \to b]]$ for a tree with yield $a + b$. When no ambiguity arises we will often write this as $[a + b]$, using only brackets to indicate the tree structure.

DEFINITION 3.3 The *yield*[1] of a tree t is the string containing all leaves from left to right, i.e.

$$\begin{array}{lcl} \mathit{yield}(X) & = & X, \text{ if } X \in V_T \\ \mathit{yield}([A \to t_1 \ldots t_n]) & = & \mathit{yield}(t_1) \\ & & \cdots\ \mathit{yield}(t_n) \end{array}$$

The function can be lifted to sets of parse trees by

$$\mathit{yield}(\Phi) = \{\mathit{yield}(t) \mid t \in \Phi\}$$ □

DEFINITION 3.4 The *language* $L(\mathcal{G})$ generated by a CFG $\mathcal{G}$ is the set of strings $\mathit{yield}(\mathcal{T}^{\mathcal{G}})$. The language $L(\mathcal{G})_A$ generated by nonterminal A is the set $\mathit{yield}(\mathcal{T}^{\mathcal{G}}_A)$. □

A CFG is *ambiguous* if it generates at least two different trees t and s such that $\mathit{yield}(t) = \mathit{yield}(s)$.

Derivation in the classical semantics of CFGs and parse trees are similar notions as is witnessed by the following proposition.

PROPOSITION 3.5 *For any CFG $\mathcal{G}$ and any $A \in V_N$, $\alpha \in V^*$: $A \to^*_{\mathcal{G}} \alpha \iff \alpha \in L(\mathcal{G})_A$* □

3.2 Parse Forests

A parse forest is a compact representation of a set of parse trees. Compaction is achieved by sharing common subtrees and by packing different trees for the same yield in one node. Parse forests can be described by contexts and sets of contexts.

DEFINITION 3.6 A *context* $C[\bullet]$ is a parse tree with exactly one occurrence of a hole $\bullet$. The instantiation $C[t]$ of a context $C[\bullet]$ is constructed by replacing the hole $\bullet$ by t. We denote the set $\{C[t] \mid t \in \Phi\}$ by $C[\Phi]$. Similarly, $\Gamma[\bullet]$ denotes a set of contexts and its instantiation $\Gamma[t]$ is defined as $\{C[t] \mid C[\bullet] \in \Gamma[\bullet]\}$. □

Sharing of a tree t by a set of trees is represented by the instantiation $\Gamma[t]$ of a set of contexts. Packing of a set of trees in a single node is represented by the instantiation $C[\Phi]$ of a context with a set of trees. Sharing of a packed node by a set of trees is denoted by $\Gamma[\Phi]$.

[1] Some authors use frontier instead of yield.

3.3 Parsing

Definition 3.7 A *parser* is a function Π that maps each string $w \in V_T^*$ to a set of parse trees. A parser Π *accepts* a string w if $|\Pi(w)| > 0$. A parser Π is *deterministic* if $|\Pi(w)| \leq 1$ for all strings w. A parser for a CFG $\mathcal{G}$ that accepts exactly the sentences in $L(\mathcal{G})$ is defined by

$$\Pi_{\mathcal{G}}(w) = \{t \in \mathcal{T} \mid \mathit{yield}(t) = w\} \qquad \Box$$

We restrict our attention to pure parsers that do not modify parse trees during parsing.

An example of an implementation of parsers for arbitrary CFGs is Tomita's generalized LR algorithm [Lan74, Tom85, Rek92]. Such a generalized parser produces a parse forest as representation of a set of trees by packing all trees for a subsentence v, the set $\Phi = \Pi_{\mathcal{G}}(v)$, in a single node and sharing that node in all trees built for the sentence in which v is embedded, i.e., if uvw is a sentence and parsing the sentence $u \bullet w$ produces $\Pi_{\mathcal{G}}(u \bullet w) = \Gamma[\bullet]$ then the forest for the whole sentence can be constructed as $\Pi_{\mathcal{G}}(uvw) \supseteq \Gamma[\Phi]$.

4 Filters

Ambiguous context-free grammars produce multiple interpretations for some of the sentences they generate. A language definition should unambiguously assign to each string a single interpretation. Therefore, if a language definition is based on a context-free grammar, it should select from the multiple interpretations given by the grammar the most appropriate one. We formalize the specification of selection of an appropriate interpretation by the notion of *parse tree filters*. It will turn out that most 'logical' disambiguation methods can be expressed by means of filters.

Definition 4.1 A *filter* $\mathcal{F}$ for a CFG $\mathcal{G}$ is a function $\mathcal{F} : \wp(\mathcal{T}) \rightarrow \wp(\mathcal{T})$ that maps sets of parse trees to sets of parse trees, where $\mathcal{F}(\Phi) \subseteq \Phi$ for any $\Phi \subseteq \mathcal{T}$. The *disambiguation* of a CFG $\mathcal{G}$ by a filter $\mathcal{F}$ is denoted by $\mathcal{G}/\mathcal{F}$. The *language* $L(\mathcal{G}/\mathcal{F})$ generated by $\mathcal{G}/\mathcal{F}$ is the set

$$\{w \in V_T^* \mid \exists \Phi \subseteq \mathcal{T}^{\mathcal{G}} : \mathit{yield}(\Phi) = \{w\} \\ \wedge \mathcal{F}(\Phi) = \Phi\}$$

The *interpretation* of a string w by $\mathcal{G}/\mathcal{F}$ is the set of trees $\mathcal{F}(\Pi_{\mathcal{G}}(w))$. A filter $\mathcal{F}_2$ is also applicable to a disambiguated grammar $\mathcal{G}/\mathcal{F}_1$, which is denoted by $(\mathcal{G}/\mathcal{F}_1)/\mathcal{F}_2$ and is equivalent to $\mathcal{G}/(\mathcal{F}_2 \circ \mathcal{F}_1)$. $\Box$

Given a set of parse trees Φ for some sentence w, a filter selects the 'correct' parse tree(s) in Φ yielding a reduced set of trees $\Phi' \subseteq \Phi$. The condition $\mathcal{F}(\Phi) \subseteq \Phi$ ensures that filters do indeed *reduce* the set of trees instead of inventing new ones. A trivial example of a filter that satisfies this condition is the identity function on sets of parse trees.

Often we will define a filter in negative terms by specifying which trees are 'wrong' and then throw away the wrong trees from a set of trees.

A disambiguated CFG $\mathcal{G}/\mathcal{F}$ generates a subset of the language generated by $\mathcal{G}$, i.e., a string w is only in the language generated by $\mathcal{G}/\mathcal{F}$ if there is at least one tree with yield w that is not rejected by the filter.

This is a very general definition allowing arbitrary functions as filters. Later in this paper we will consider several classes of filters that use less powerful functions.

4.1 Properties of filters

We can now investigate several properties of filters.

Definition 4.2 A filter is *completely disambiguating* when $|\mathcal{F}(\Pi_{\mathcal{G}}(w))| \leq 1$ for all $w \in V^*$. $\Box$

This is a useful property if the parse trees are input for a next, semantic, processing phase; no provisions have to be made for sets of trees in such a phase. A more restrictive property is completeness:

Definition 4.3 [Tho94a] A filter $\mathcal{F}$ is *complete* for a CFG $\mathcal{G}$ if $w \in L(\mathcal{G}) \Rightarrow |\mathcal{F}(\Pi_{\mathcal{G}}(w))| = 1$. □

Actually, Thorup defines a *parser* to be complete if it produces exactly one 'canonical' parse tree for each sentence in the language of its underlying CFG.

Corollary 4.4 *If $\mathcal{F}$ is complete for $\mathcal{G}$ then* $L(\mathcal{G}) = L(\mathcal{G}/\mathcal{F})$ □

Definition 4.5 A filter $\mathcal{F}$ for a CFG $\mathcal{G}$ is *local* if for each set of contexts $\Gamma[\bullet] \subseteq \mathcal{T}[\bullet]$ and each $\Phi \subseteq \mathcal{T}$

$$\mathcal{F}(\Gamma[\Phi]) \subseteq \Gamma[\mathcal{F}(\Phi)]$$

A filter is *global* if it is not local. □

Global filters are counter intuitive: rejection by a global filter of a tree for some substring of a sentence does not imply that that tree can not be a subtree of a parse tree for the sentence. A local filter is transparent: a rejected tree can not be a subtree of any larger tree. This means that a local filter can be applied to a local ambiguity instead of to the entire set of complete parse trees for a sentence. It seems that a disambiguation method that can be defined in terms of a local filter is both intuitive and easy to implement.

Definition 4.6 A filter $\mathcal{F}$ is *incremental* if for each pair of sets of parse trees Φ_1, Φ_2

$$\mathcal{F}(\mathcal{F}(\Phi_1) \cup \mathcal{F}(\Phi_2)) = \mathcal{F}(\Phi_1 \cup \Phi_2)$$ □

A generalized parser constructs sets of parse trees for local ambiguities in an incremental fashion. If a filter is incremental, it can be applied to a set whenever an element is added; if it is not incremental application to a set is only legal if the set is completed.

Definition 4.7 Two filters $\mathcal{F}_1$ and $\mathcal{F}_2$ are *commutative* if for each set of trees Φ

$$\mathcal{F}_1(\mathcal{F}_2(\Phi)) = \mathcal{F}_2(\mathcal{F}_1(\Phi))$$

i.e, if their composition commutes:

$$\mathcal{G}/\mathcal{F}_1 \circ \mathcal{F}_2 = \mathcal{G}/\mathcal{F}_2 \circ \mathcal{F}_1$$ □

Definition 4.8 A filter $\mathcal{F}$ for a CFG $\mathcal{G}$ is *context-free* if there is an unambiguous CFG $\mathcal{G}'$ and a function $tr : \mathcal{T}^{\mathcal{G}'} \rightarrow \mathcal{T}^{\mathcal{G}}$ such that $L(\mathcal{G}') = L(\mathcal{G}/\mathcal{F})$, i.e., $\mathcal{G}'$ generates the same language as $\mathcal{G}/\mathcal{F}$, and $tr(\Pi_{\mathcal{G}'}(w)) \in \mathcal{F}(\Pi_{\mathcal{G}}(w))$. A filter is *context-dependent* if it is not context-free. □

4.2 Specification of Filters

Filters can be defined in many ways. We will consider two special classes of filters that are defined in terms of predicates and relations on trees.

Definition 4.9 The filter $\mathcal{F}^{\mathcal{E}}$ *generated by the unary predicate* $\mathcal{E}$ (exclude) on trees is defined by

$$\mathcal{F}^{\mathcal{E}}_A(\Phi) = \{t \in \Phi \mid \neg\mathcal{E}(t)\}$$

A predicate $\mathcal{E}$ is *compositional* if for each tree t and each context $C[\bullet]$ $\mathcal{E}(t) \Rightarrow \mathcal{E}(C[t])$. □

A filter $\mathcal{F}^{\mathcal{E}}$ selects all trees which do *not* have property $\mathcal{E}$. The predicate characterizes, for instance, trees with a *conflict*. Compositionality of a filter-predicate ensures that if a tree has a conflict, any tree composed from it has a conflict as well. This implies that to understand a conflict in a sentence one only has to consider the smallest part of the sentence that has the conflict.

Proposition 4.10 *A filter $\mathcal{F}^{\mathcal{E}}$ is local iff $\mathcal{E}$ is compositional.*

Proof. ($\Leftarrow$) Let $\Gamma[\bullet]$ a set of contexts and Φ a set of trees. If $t \in \mathcal{F}(\Gamma[\Phi])$, then $\neg\mathcal{E}(t)$ and $t = C[t']$ such that $C[\bullet] \in \Gamma[\bullet]$ and $t' \in \Phi$. Since $\mathcal{E}$ is compositional we have that $\neg\mathcal{E}(t')$ and therefore $t' \in \mathcal{F}(\Phi)$ and thus $t \in \Gamma[\mathcal{F}(\Phi)]$.

($\Rightarrow$) Assume $\mathcal{E}$ is not compositional, i.e., there is some tree t and context $C[\bullet]$, such that $\mathcal{E}(t)$ and not $\mathcal{E}(C[t])$. Then $\mathcal{F}(\{C[t]\}) = \{C[t]\} \not\subseteq \emptyset = C[\mathcal{F}(\{t\})]$ and thus $\mathcal{F}^{\mathcal{E}}$ is not local. □

Definition 4.11 The predicate $\overline{\mathcal{E}}$ is defined in terms of $\mathcal{E}$ by $\overline{\mathcal{E}}(t) = \bigvee_{s \in sub(t)} \mathcal{E}(s)$, where $sub(t)$ denotes the set of all subtrees of t. $\square$

Note that $\overline{\mathcal{E}}$ is always compositional.

Definition 4.12 The filter $\mathcal{F}^{\prec}$ *generated by the relation* $\prec$ is defined by

$$\mathcal{F}^{\prec}(\Phi) = \{t \in \Phi \mid \neg \exists t' \in \Phi : t' \prec t\}$$

A relation $\prec$ is *compositional* if

$$\forall s, t, C[\bullet] : s \prec t \Rightarrow C[s] \prec C[t] \qquad \square$$

A filter $\mathcal{F}^{\prec}$ selects the minimal trees in a set according to the order $\prec$. Note that if $\prec$ is reflexive or symmetric, the filter $\mathcal{F}^{\prec}$ rejects all trees. For instance, given any CFG $\mathcal{G}$, $\mathcal{G}/\mathcal{F}^{=}$ defines the empty language. The notation $\prec$ suggests that the most useful filters of this kind are based on strict partial orders, i.e., if $\prec$ is transitive, irreflexive and antisymmetric. If $\prec$ is a strict partial order, $\mathcal{F}^{\prec}$ is monotonous, i.e., $\mathcal{F}^{\prec}(\Phi_1) \subseteq \mathcal{F}^{\prec}(\Phi_2)$ if $\Phi_1 \subseteq \Phi_2$, which adds to the clarity of a disambiguation method.

Proposition 4.13 *A filter* $\mathcal{F}^{\prec}$ *is local iff* $\prec$ *is compositional.*

Proof. ($\Leftarrow$) Assume $\mathcal{F}$ not local, i.e., there are $\Gamma[\bullet]$, Φ and $s = C[s'] \in \Gamma[\Phi]$ such that $s \in \mathcal{F}(\Gamma[\Phi])$ but $s \notin \Gamma[\mathcal{F}(\Phi)]$. Thus $\neg\exists t \in \Gamma[\Phi] : t \prec s$, i.e., $\forall t \in \Gamma[\Phi] : \neg t \prec s$ and especially $\forall t' \in \Phi : \neg C[t'] \prec C[s']$ then, by compositionality of $\prec$, $\forall t' \in \Phi : \neg t' \prec s'$ which is equivalent to $\neg\exists t' \in \Phi : t' \prec s'$ but this is in contradiction with $\exists t' \in \Phi : t' \prec s'$ which follows from $s \notin \Gamma[\mathcal{F}(\Phi)]$.

($\Rightarrow$) Assume that $\prec$ is not compositional, i.e., there are some s, t and $C[\bullet]$ such that $s \prec t \wedge \neg C[s] \prec C[t]$. But then $\mathcal{F}(C[\{s,t\}]) = C[\{s,t\}] \not\subseteq C[\{s\}] = C[\mathcal{F}(\{s,t\})]$, which contradicts the fact that $\mathcal{F}$ is local. $\square$

4.3 Parsers for $\mathcal{G}/\mathcal{F}$

By definition a filter can always be used as a post-parse procedure to prune the parse forest, i.e., $\Pi_{\mathcal{G}/\mathcal{F}} = \mathcal{F} \circ \Pi_{\mathcal{G}}$. For efficiency reasons it is attractive to apply the disambiguation rules described by a filter as early in the parse process as possible.

The problem of producing the most efficient parser from an abstract specification of a filter is probably undecidable. However, for certain classes of filters efficient parsers are possible. By considering many disambiguation methods in this one framework of filters crossovers between implementation strategies might arise.

Definition 4.14 An *approximation* of a parser for $\mathcal{G}/\mathcal{F}$ is a parser Π such that for any string w

$$\mathcal{F}(\Pi_{\mathcal{G}}(w)) \subseteq \Pi(w) \subseteq \Pi_{\mathcal{G}}(w) \qquad \square$$

If $\mathcal{F}$ is a local filter for a CFG $\mathcal{G}$, we can construct an approximation Π for $\mathcal{G}/\mathcal{F}$ by filtering any local ambiguity as soon as it is constructed. Formally, if $\Pi_{\mathcal{G}}(v)_A = \Phi$ and $\Pi_{\mathcal{G}}(u \bullet_A w)_B = \Gamma[\bullet_A]$ then $\Pi(uvw) \subseteq \Gamma[\mathcal{F}(\Phi)]$ If there are no trees left in a local ambiguity the parser that corresponds to it can be stopped, yielding the empty set of trees.

Parsing schemata are abstract specifications of parsing algorithms. In section 6 we will start an investigation of the implementation of parsers for grammars disambiguated by filters based on parsing schemata.

5 Case Studies

In order to assess the feasibility of using filters for the disambiguation of context-free grammars we present case studies that illustrate the expressive power of our method.

Priorities are a conventional tool for disambiguation and have been proposed in many forms. In sections 5.1 and 5.2 we study the disambiguation mechanism of SDF which consists of a filter for priority conflicts and a filter for priority comparisons, both derived from a single priority declaration.

Extensible languages are typical examples of languages that are not in the scope of context-free grammars disambiguated by filters. The definition of a filter presumes a set of possible trees from which it selects appropriate ones. A grammar for an extensible language must somehow describe how new productions, i.e., new tree forms, can be introduced. However, restricted forms of extensibility, like Prolog's user-defined operators, are in the range of filters (section 5.3).

Landin's offside rule is a disambiguation method based on indentation. In section 5.4 we define this method by a filter.

A restricted class of filters based on pattern matching is described in 5.5.

5.1 Priority Conflicts

Disambiguation by *precedences* or *priorities* is used by many grammar formalisms in various instantiations [Ear75, AJU75, Joh75, HHKR92, Aas92]. In this and the next section we study priorities in the syntax definition formalism SDF [HHKR92]. An SDF priority declaration induces a strict partial order on grammar productions combined with associativity declarations. From the priority and associativity declarations $\mathcal{R}$ two filters $\mathcal{F}^{\mathcal{E}^{\mathcal{R}}}$ and $\mathcal{F}^{\prec^{\mathcal{R}}}$ are derived. The first removes trees with priority conflicts and the second selects trees which are minimal with respect to a multiset ordering on trees.

We do not use the notation of SDF for the declaration of priorities but a notation similar to Earley's notation for precedence rules in [Ear75] that is more suitable for theoretical exposition as in this paper. The concrete notation of SDF can be translated to the abstract notation used here. There have been many proposals for the interpretation of SDF priorities; here we follow [Kli88].

DEFINITION 5.1 A *priority declaration* $\mathcal{R}$ for a CFG $\mathcal{G}$ is a tuple $\langle \mathrm{L}, \mathrm{R}, \mathrm{N}, > \rangle$, where $\oplus \subseteq \mathcal{P} \times \mathcal{P}$ for $\oplus \in \{\mathrm{L}, \mathrm{R}, \mathrm{N}, >\}$, such that L, R and N are symmetric and $>$ is irreflexive and transitive. □

The relations L, R and N declare left-, right- and non-associativity, respectively, between productions. The relation $>$ declares priority between productions. A tree with signature p_1 can not be a child of a tree with signature p_2 if $p_2 > p_1$.

DEFINITION 5.2 A tree t has a *root priority conflict* $\mathcal{E}^{\mathcal{R}}(t)$ if it violates a right- or non-associativity rule

$$\frac{A \to B\alpha \, \mathrm{R}^{\mathcal{R}} \, B \to \beta \vee A \to B\alpha \, \mathrm{N}^{\mathcal{R}} \, B \to \beta}{\mathcal{E}^{\mathcal{R}}([A \to [B \to t_\beta] \; s_\alpha])}$$

or violates a left- or non-associativity rule

$$\frac{A \to \alpha B \, \mathrm{L}^{\mathcal{R}} \, B \to \beta \vee A \to \alpha B \, \mathrm{N}^{\mathcal{R}} \, B \to \beta}{\mathcal{E}^{\mathcal{R}}([A \to s_\alpha \; [B \to t_\beta]])}$$

or violates a priority rule

$$\frac{A \to \alpha B \gamma >^{\mathcal{R}} B \to \beta}{\mathcal{E}^{\mathcal{R}}([A \to s_\alpha \; [B \to t_\beta] \; s_\gamma])}$$

A tree t has a *priority conflict*, if $\overline{\mathcal{E}}^{\mathcal{R}}(t)$. □

According to definition 4.9 we can now construct the filter $\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}$. Thus the semantics of the pair $\langle \mathcal{G}, \mathcal{R} \rangle$ is the disambiguated CFG $\mathcal{G}/\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}$. By definition of $\overline{\mathcal{E}}$ in terms of $\mathcal{E}$ we have the following:

COROLLARY 5.3 $\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}$ *is a local filter.* □

EXAMPLE 5.4 The usual example for priorities is the following grammar $\mathcal{G}_{exp}$ for arithmetic expressions that is completely disambiguated by the priority relation $\mathcal{R}_{exp}$

$$\begin{array}{rcl} \mathcal{G}_{exp} & = & E \to E + E \mid E - E \mid E * E \\ & & \quad \mid \; E \uparrow E \mid \; -E \mid (E) \\ & & \quad \mid \; a \mid b \mid \ldots \end{array}$$

$$\begin{array}{rcl} \mathcal{R}_{exp} & = & -E > \uparrow > * > \{+, -\} \\ & & + \, \mathrm{L} \, +, \; - \, \mathrm{L} \, -, \; + \, \mathrm{L} \, -, \; * \, \mathrm{L} \, * \\ & & \uparrow \mathrm{R} \uparrow, \end{array}$$

Now we have, for instance,

$$\begin{array}{l}\mathcal{F}^{\mathcal{E}^{\mathcal{R}_{exp}}}(\Pi_{\mathcal{G}_{exp}}(a+b+c)) \\ \quad = \mathcal{F}^{\mathcal{E}^{\mathcal{R}_{exp}}}(\{[a+[b+c]],[[a+b]+c]\}) \\ \quad = \{[[a+b]+c]\}\end{array}$$

because $+ \,\mathrm{L}\, +$ □

According to the definition above a root priority conflict of a tree can be detected by looking at the signature of the tree and at the signatures of its children. The following version of the predicate is somewhat stronger in that it looks through chain rules.

DEFINITION 5.5 The function *ecr* (*chain rule elimination*) yields the first subtree that is not an application of a chain rule:

$$\begin{array}{lcl} ecr(X) & = & X \\ ecr([A \rightarrow t_B]) & = & ecr(t_B) \\ ecr([A \rightarrow t_\alpha]) & = & [A \rightarrow ecr{*}(t_\alpha)], \\ & & \quad \text{if } |\alpha| \neq 1 \end{array}$$ □

DEFINITION 5.6 A tree t has a root priority conflict *modulo chain rules* if $\mathcal{E}_c^{\mathcal{R}}(t)$:

$$\mathcal{E}_c^{\mathcal{R}}(t) \iff \mathcal{E}^{\mathcal{R}}(ecr(t))$$ □

5.2 Multiset Ordering

After selecting the conflict-free trees from a set there might still be more than one tree in the set. The next filter that is used by SDF selects trees by comparing trees with respect to a multiset ordering $\prec$ on trees.

DEFINITION 5.7 A *multiset* is a function $M : \mathcal{P} \rightarrow \mathbf{N}$ that maps productions to the number of their occurrences in the set. The union $M \uplus N$ of two multisets M and N is defined as $M \uplus N(p) = M(p) + N(p)$. The empty multiset is denoted by $\emptyset$, i.e., $\emptyset(p) = 0$ for any p. We write $p \in M$ for $M(p) > 0$. A multiset with a finite number of elements with a finite number of occurrences can be written as $M = \{p_1, p_1, \ldots, p_2, \ldots\}$, where $M(p)$ is the number of occurrences of p in the list. □

DEFINITION 5.8 A tree t is translated to a multiset by $\overline{\bullet} : \mathcal{T} \rightarrow (\mathcal{P} \rightarrow \mathbf{N})$ as

$$\begin{array}{rcl} \overline{X} & = & \emptyset \\ \overline{[A \rightarrow t_\alpha]} & = & \{A \rightarrow \alpha\} \uplus \overline{t_\alpha} \\ \overline{\epsilon} & = & \emptyset \\ \overline{t_\alpha t_\beta} & = & \overline{t_\alpha} \uplus \overline{t_\beta} \end{array}$$

DEFINITION 5.9 Given some priority declaration $\mathcal{R}$, the order $\prec^{\mathcal{R}}$ on multisets is defined as

$$\begin{array}{l} M \prec^{\mathcal{R}} N \iff M \neq N \wedge \\ \quad \forall y \in M : M(y) > N(y) \Rightarrow \\ \qquad \exists x \in N : y >^{\mathcal{R}} x \wedge M(x) < N(x) \end{array}$$ □

The motivation for this ordering is that it prefers parse trees that are constructed with the smallest possible number of productions of the highest possible priority.

Given a priority declaration $\mathcal{R}$, we can now construct the filter $\mathcal{F}^{\prec^{\mathcal{R}}}$ using definition 4.12 that selects those trees which are minimal with respect to the multiset ordering induced by the priority declarations.

PROPOSITION 5.10 [Kli88] *The multiset ordering $\prec^{\mathcal{R}}$ on trees is compositional.*
PROOF. a) If $t_1 \prec^{\mathcal{R}} t_2$ then $t_1 \neq t_2$ and thus $T_1 = C[t_1] \neq C[t_2] = T_2$. b) Assume $\overline{T_1}(y) > \overline{T_2}(y)$, then $\overline{t_1}(y) > \overline{t_2}(y)$. Since $t_1 \prec^{\mathcal{R}} t_2$, $\exists x \in t_2 : y >^{\mathcal{R}} x \wedge \overline{t_1}(x) < \overline{t_2}(x)$, then also $\exists x \in T_2 : y >^{\mathcal{R}} x \wedge \overline{T_1}(x) < \overline{T_2}(x)$. From a) and b) we conclude that $C[t_1] \prec^{\mathcal{R}} C[t_2]$. □

EXAMPLE 5.11 The following grammar is a typical example of the working of the multiset order for the disambiguation of overloaded operators.

$$\begin{array}{rcl} \mathcal{G}_{bexp} & = & R \rightarrow R+R \mid R*R \mid N \mid r \\ & & N \rightarrow N+N \mid N*N \mid n \end{array}$$

$$\begin{array}{rcl} \mathcal{R}_{bexp} & = & *^N > *^R > +^N > +^R, \\ & & +^N \,\mathrm{L}\, +^N, +^R \,\mathrm{L}\, +^R \end{array}$$

Given the string $n+n$ the following trees are generated by the grammar (with number of occurrences of productions):

tree	$\# +^N$	$\# +^R$
$[N \rightarrow [N \rightarrow n] + [N \rightarrow n]]$	1	0
$[R \rightarrow [R \rightarrow n] + [R \rightarrow n]]$	0	1

Some other combinations of strings and trees:

$$\begin{array}{ll} n+n+n & [N \rightarrow [N \rightarrow +]n] \\ n+n+r & [R \rightarrow [N \rightarrow n+n] + r] \\ n+n*r & [R \rightarrow n + [R \rightarrow n*r]] \end{array}$$ □

The following examples give illustrations of grammars that can not or not appropriately be disambiguated with priority rules.

EXAMPLE 5.12 Another well-known example is the following grammar for statements with a *dangling-else* construct.

$$\begin{array}{rcl} \mathcal{G}_{ie} & = & S \rightarrow iSeS \\ & & S \rightarrow iS \\ \mathcal{R}_{ie} & = & ie > i \end{array}$$

This disambiguation is correct according to the conventional solution of this problem in that it forbids an i as first descendant of ie, as we can see from the parses of the sentence $iiSeS$

$$\begin{array}{l} \mathcal{R}_{ie}(\Pi_{\mathcal{G}_{ie}}(iiSeS)) \\ \quad = \mathcal{R}_{ie}(\{[i[iS]eS]), [i[iSeS]]\}) \\ \quad = \{[i[iSeS]]\} \end{array}$$

where the first parse is filtered out because it contains a priority conflict against $ie > i$. However, the sentence $iSeiS$ is not a member of $L(\mathcal{G}_{ie}/\mathcal{R})$ since $[iSe[iS]]$ is the only tree in $\mathcal{T}_S$ with the right yield and it has a priority conflict against $ie > i$. □

EXAMPLE 5.13 A more serious problem of precedences is posed by the following grammar that defines arithmetic expressions by one *generic production for binary operators.*

$$\begin{array}{rcl} \mathcal{G}_{gop} & = & E \rightarrow E\ O\ E \\ & & O \rightarrow + \mid * \mid \cdots \end{array}$$

This grammar can not be disambiguated like the grammars in example 5.4, although it is useful when generic operations on the trees have to be defined. □

5.3 Operators in Prolog

Several languages have mechanisms for introducing new infix, prefix and postfix operators and declaring their precedence and associativity. Here we study a mechanism that allows the user to introduce new operators with relative priority instead of with absolute priority as in Prolog [Bra90]. The meaning of the priorities is the same as in the previous sections, but since the priority declarations are part of the tree, the definition of the filter is more complicated.

Grammar The CFG $\mathcal{G}_{prolog}$ describes a language of programs P that consist of a list of clauses C that are either operator declarations D or expressions E. There is an infinite supply of operators O and priority between operators can be declared by the relations R, L and $>$ which have the same meaning as before. A declaration is valid from the point of introduction until the end of the program unless overruled by a new declaration.

$$\begin{array}{rcl} \mathcal{G}_{prolog} & = & O \rightarrow + \mid * \mid \ldots \\ & & A \rightarrow [a-z]+ \\ & & D \rightarrow O\ \mathrm{R}\ O \mid O\ \mathrm{L}\ O \mid O > O \\ & & E \rightarrow EOE \mid A \mid (E) \\ & & C \rightarrow D \mid E \\ & & P \rightarrow C.P \mid \epsilon \end{array}$$

Global filter A filter for these programs selects those trees that have expression trees that do not violate the priority declarations earlier in the tree. The first method checks a program tree by traversing it from left to right, checking each expression tree with the priority information it has collected earlier in the traversal.

DEFINITION 5.14 The predicate $\mathcal{E}^{\mathcal{R}}$ is defined as follows on program trees

$$\begin{aligned}\mathcal{E}^{\mathcal{R}}([e.p]) &\iff \overline{\mathcal{E}}^{\mathcal{R}}(e) \vee \mathcal{E}^{\mathcal{R}}(p)\\ \mathcal{E}^{\mathcal{R}}([d.p]) &\iff \mathcal{E}^{\mathcal{R}\cup\{d\}}(p)\end{aligned}$$

and for expression trees

$$\begin{aligned}\mathcal{E}^{\mathcal{R}}([[e_1 \oplus e_2] \otimes e_3]) &\Leftarrow \oplus \mathrm{R} \otimes \vee \otimes > \oplus\\ \mathcal{E}^{\mathcal{R}}([e_1 \oplus [e_2 \otimes e_3]]) &\Leftarrow \oplus \mathrm{L} \otimes \vee \oplus > \otimes\end{aligned}$$

The filter for sets of P-trees over $\mathcal{G}_{prolog}$ can now be defined as

$$\mathcal{F}(\Phi) = \{t \in \Phi \mid \neg\mathcal{E}^{\emptyset}(t)\} \qquad \square$$

Local filter Another approach to selecting the right tree is by means of a local filter. The basic idea of the filter as defined below is that it infers the priority constraints posed by each subtree of a tree. If these constraints form an inconsistent statement the subtree can never be correct with respect to any priority declaration.

DEFINITION 5.15 The function pr maps trees in $\mathcal{T}^{\mathcal{G}_{prolog}}$ to first-order logical formulas.

$$\begin{aligned}
& pr([A \to x]) = \forall \otimes, \oplus, \ominus :\\
& \qquad \otimes > \oplus \Rightarrow \neg \oplus > \otimes \wedge \neg \oplus \mathrm{L} \otimes \wedge \neg \oplus \mathrm{R} \otimes\\
& \quad \wedge\ \otimes > \oplus \wedge \oplus > \ominus \Rightarrow \otimes > \ominus\\
& \quad \wedge\ \otimes \mathrm{L} \oplus \Rightarrow \oplus \mathrm{L} \otimes \wedge \neg \otimes \mathrm{R} \oplus\\
& \quad \wedge\ \otimes \mathrm{R} \oplus \Rightarrow \oplus \mathrm{R} \otimes \wedge \neg \otimes \mathrm{L} \oplus\\
& pr([E \to E \to e_1 \otimes e_2]) =\\
& \quad (op(e_1) > \otimes\ \vee\ op(e_1)\,\mathrm{L} \otimes)\ \wedge\ pr(e_1)\ \wedge\\
& \quad (op(e_2) > \otimes\ \vee\ op(e_2)\,\mathrm{R} \otimes)\ \wedge\ pr(e_2)\\
& pr([P \to e.p]) = pr(e) \wedge pr(p)\\
& pr([P \to d.p]) = d \wedge pr(p)
\end{aligned}$$

where $op([A \to x]) = x$ and $op([E \to e_1 \oplus e_2]) = \oplus$. The filter can now be defined as

$$\mathcal{F}^{pr}(\Phi) = \{t \in \Phi | \exists \mathcal{R} : \mathcal{R} \models pr(t)\} \qquad \square$$

EXAMPLE 5.16 Sentence:

$$* > +.a + b * c.$$

Trees:

$$\begin{aligned}
pr((* > +).((a + b) * c).) &= * > + \wedge + > *\\
&= \bot\\
pr((* > +).(a + (b * c)).) &= * > +
\end{aligned}$$

$\square$

EXAMPLE 5.17 Sentence:

$$a + b * c + d.$$

Trees:

$$\begin{aligned}
pr(((a + b) * (c + d))) &= + > *\\
pr((((a + b) * c) + d)) &= + \mathrm{L} *\\
pr(((a + (b * c)) + d)) &= * > + \wedge + \mathrm{L} +\\
pr((a + ((b * c) + d))) &= * > + \wedge + \mathrm{R} +\\
pr((a + (b * (c + d)))) &= + \mathrm{R} *
\end{aligned} \qquad \square$$

It is clear that this disambiguation method can not be applied at parser-generation time, but can very well be applied at parse-time.

Aasa [Aas91, Aas92] describes a disambiguation method for a limited class of context-free grammars with distfix operators based on a predicate on trees. This filter is used to transform CFGs into disambiguated CFGs which generate the same trees.

5.4 Offside Rule

Several languages use the offside rule to enforce uniform indentation and at the same time reduce the number of keywords for separating constructs. The rule was first formulated by Landin [Lan66] and later (but shorter) by Richards [Ric84] as:

> None of an expression's tokens can
> lie to the left of its first token.

In the following definition disambiguation by the offside rule is defined by means of a filter.

DEFINITION 5.18 Associate with each occurrence of a terminal $X \in V_T$ its horizontal position $h(X)$. Associate with each tree $t = [A \to t_1 \ldots t_n]$ its horizontal position $h(t) = h(t_1)$ and its minimal horizontal position $hm(t) = \min_{i=1}^{n}(hm(t_i))$. A tree t is *offside* ($o(t)$) if $hm(t) < h(t)$. The grammar $\mathcal{G}/\mathcal{F}^{\overline{o}}$ is disambiguated by the offside rule. $\square$

5.5 Pattern Matching Filters

In section 5.1 we saw how priorities can be defined in terms of a unary predicate that checks every node of a tree for a priority violation, i.e., if it matches some pattern that indicates a priority conflict. This method is part of a larger class of disambiguation methods based on pattern matching. This class is attractive since it is weak enough to implement efficiently and it is strong enough to resolve ambiguities in the area of precedence and associativity in an elegant way.

DEFINITION 5.19 A tree t *matches* a tree (pattern) q, if $q \diamond t$:

$$\begin{array}{rl} X \diamond X & \\ A \diamond [A \to t_\alpha] & \\ [A \to q_1 \dots q_n] \diamond [A \to t_1 \dots t_n] & \\ \Leftarrow \bigwedge_{i=1}^{n} q_i \diamond t_i & \end{array}$$

If Q is a set of patterns then $Q \diamond t$ if there is some $q \in Q$ such that $q \diamond t$. □

(This definition can easily be extended such that $\diamond$ yields a substitution of the variables—indexed nonterminals–in the pattern, if patterns are linear. We will write $\sigma = q \diamond t$ to indicate that σ is a substitution such that $q\sigma = t$.)

Subtree Exclusion Thorup [Tho94b] describes a disambiguation method that consists of specifying a set of tree patterns that are excluded from trees produced by a parser. In terms of filters this works according to the following

DEFINITION 5.20 Given a set Q of tree patterns, the *subtree exclusion filter* $\mathcal{F}^Q$ is defined by

$$\mathcal{F}^Q(\Phi) = \{t \in \Phi \mid \neg\exists s \in sub(t) : Q \diamond s\} \quad \square$$

Disambiguation by priority conflicts as defined in section 5.1 can be defined in terms of subtree exclusion by translating the rules in a priority declaration $\mathcal{R}$ to a set of tree patterns $Q_\mathcal{R}$ that characterize trees with priority conflicts. For example, if $* > + \in \mathcal{R}$, then the pattern $((E + E) * E)$ is illegal and therefore the tree $((a + b) * c)$ is illegal.

DEFINITION 5.21 A priority declaration $\mathcal{R}$ derives a pattern set $Q_\mathcal{R}$ as follows:

$$\begin{array}{l} A \to \alpha B \ \mathrm{L} \ B \to \beta \in \mathcal{R} \\ \quad \Rightarrow \quad [A \to \alpha[B \to \beta]] \in Q_\mathcal{R} \\ A \to B\alpha \ \mathrm{R} \ B \to \beta \in \mathcal{R} \\ \quad \Rightarrow \quad [A \to [B \to \beta]\alpha] \in Q_\mathcal{R} \\ A \to B\alpha B \ \mathrm{N} \ B \to \beta \in \mathcal{R} \\ \quad \Rightarrow [A \to [B \to \beta]\alpha B] \in Q_\mathcal{R} \\ \quad \wedge (A \to B\alpha(B \to \beta)) \in Q_\mathcal{R} \\ A \to \alpha B\beta > B \to \gamma \in \mathcal{R} \\ \quad \Rightarrow [A \to \alpha[B \to \gamma]\beta] \in Q_\mathcal{R} \end{array} \quad \square$$

PROPOSITION 5.22 *A tree has a root priority conflict (Definition 5.2) according to a priority declaration $\mathcal{R}$ iff it matches one of the patterns in $Q_\mathcal{R}$, i.e., $\mathcal{E}^\mathcal{R}(t) \iff Q_\mathcal{R} \diamond t$* □

Subtree exclusion is strictly more expressive than priorities as a disambiguation mechanism: Proposition 5.22 proves that each priority declaration can be expressed as a subtree exclusion filter. Example 5.13 showed the grammar with generic syntax for infix operators $E \to EOE$ that could not be disambiguated with priorities. By excluding patterns like

$$[E \to E \ [O \to *] \ [E \to E \ [O \to +] \ E]]$$

the intended disambiguation can be achieved. This higher expressivity of subtree exclusion is due to the fact that arbitrarily deep patterns can be specified, while priorities provide fixed pattern templates—corresponding to associativity and precedence—that are always 2 levels deep. Like priorities, subtree exclusion is not expressive enough for a correct disambiguation of the dangling-else grammar in example 5.12. This is due to the fact that this problem can not be solved with a finite number of fixed depth patterns. Below we will propose to solve this problem by the use of higher-order patterns.

Rewrite Rules LaLonde and Des Rivieres [LR81] describe a disambiguation method for operator grammars—with productions of the form $E \to E \oplus E$—that works by translating a grammar to an unambiguous right-associative CFG—with productions $E \to T\ O\ E$ and $O \to \oplus$—and defining *tree transformations* that transform a tree over the unambiguous grammar to the correct tree over the ambiguous grammar. Such a transformed grammar is implemented by a deterministic parser that yields right-associative trees that are transformed after parsing to the correct form by generic rules like

$$\begin{array}{l}[E \to [E \to E_1[O \to \oplus]E_2][O \to \otimes]E_3] \\ \ \Rightarrow [E \to E_1[O \to \oplus] \\ \qquad\quad [E \to E \to E_2[O \to \otimes]E_3]], \\ \text{if } \otimes > \oplus\end{array}$$

The transformation system is specialized for operator precedence information. A generalization of this technique is achieved by applying an arbitrary *tree rewrite system* instead of operator transformations; for instance, to express that $* > +$, the rewrite system contains a rule

$$\begin{array}{l}[E \to [E \to E_1 + E_2] * E_3] \\ \qquad \to [E \to E_1 + [E \to E_2 * E_3]]\end{array}$$

Thorup [Tho94a] uses this idea in a method for the disambiguation of CFGs by TRSs:

DEFINITION 5.23 A *tree* or *term rewrite system* (TRS) is a set E of tree pairs (s,t). A tree t *rewrites in one step* to a tree s in a TRS E ($t \to_E s$) if $t = C[t']$, $s = C[s']$ and there is a pair $(q,p) \in E$ such that $\sigma = q \ \diamond\ t'$ and $p\sigma \diamond s'$. A tree t *rewrites* to a tree s if $t \to_E^+ s$. □

DEFINITION 5.24 If E is a TRS, then $\mathcal{F}^E$ is the filter defined by

$$\mathcal{F}^E(\Phi) = \{t \in \Phi \mid \neg\exists s \in \Phi : t \to_E^+ s\} \quad \square$$

CONJECTURE 5.25 *Rewrite filters $\mathcal{F}^E$ are local.* □

The grammar $\mathcal{G}/\mathcal{F}^E$ is not implemented by post-parse filtering, but the TRS is used for the solution of conflicts in LR parse tables. The input for the algorithm is a CFG $\mathcal{G}$ and a TRS E, the output is a complete, linear time parser Π and a TRS $E' = E \cup E''$ if such a pair exists, indication of failure otherwise. The parser Π is a deterministic parser for $L(\mathcal{G})$, that produces for each sentence w a tree t in normal form with respect to E', i.e., there is no tree s such that $t \to_{E'}^+ s$.

Disambiguating rewrite rules can be derived from *semantic equations* $s = t$ that express that two trees (patterns) s and t have the same meaning. If the yields of the lhs and rhs of such an equation are the same, i.e. $\mathit{yield}(s) = \mathit{yield}(t)$, a disambiguation rule choosing either one can be derived. This is especially appropriate for associative operators as in $a \oplus (b \otimes c) = (a \oplus b) \otimes c$. Thorup [Tho94a] assumes $s = t$ if $\mathit{yield}(s) = \mathit{yield}(t)$ and neither $s \to^+ t$ nor $t \to^+ s$.

Higher-Order Patterns Several disambiguation problems can not be described by fixed-depth patterns. We propose a language of higher-order patterns that adds expressive power to pattern matching; it allows the correct specification of the disambiguation of the dangling-else grammar from example 5.12.

DEFINITION 5.26 A *higher-order tree pattern* is an element from the set

$$\mathcal{H} = \mathcal{T} \cup \{\dot\alpha, \dot\beta, \dot\gamma, \ldots\} \cup (V_N \times \mathcal{H}^*)$$

We write $(A \to^* q_1 \ldots q_n)$ for an element of $V_N \times \mathcal{H}^*$. A tree t *matches* with a higher order pattern q if they are in the relation $q \diamondsuit t$:

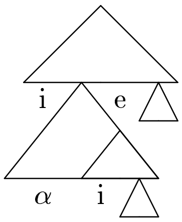
$$\begin{array}{rcl} X \diamondsuit X & & \\ A \diamondsuit A \to \alpha(t_\alpha) & & \\ [A \to q_\alpha] \diamondsuit [A \to t_\alpha] & \Leftarrow & \bigwedge_{X \in \alpha} q_X \diamondsuit t_X \\ [A \to^* \vec{q}] \diamondsuit [A \to \vec{t}] & \Leftarrow & \vec{q} \diamondsuit \vec{t} \\ \dot\alpha\ \vec{q} \diamondsuit \vec{t}_1\vec{t}_2 & \Leftarrow & \vec{q} \diamondsuit \vec{t}_2 \\ q\vec{q} \diamondsuit t_1\vec{t}_2 & \Leftarrow & q \diamondsuit t_1 \wedge \vec{q} \diamondsuit \vec{t}_2 \\ \vec{q} \diamondsuit \vec{t}_0[A \to \vec{t}_1]\vec{t}_2 & \Leftarrow & \vec{q} \diamondsuit \vec{t}_0\vec{t}_1\vec{t}_2 \end{array}$$

□

EXAMPLE 5.27 The dangling-else grammar can now be disambiguated by excluding all subtrees that match with

$$[S \to i\ [S \to^* \dot{\alpha}\ \ i\ S]\ e\ S]$$

This pattern matches any tree of the form



6 Implementation

Filters are an attractive method for the disambiguation of context-free grammars because they specify the interpretation of a sentence in a logical manner and can be implemented as post-parse filter. An implementation consisting of a standard generalized parser in combination with a post-parse filter allows fast prototyping of, and experimentation with, new disambiguation methods. However, deferring filtering until parsing is complete can be expensive, because many trees built during parsing are thrown away afterwards. If a tree is rejected by a filter after parsing we can look at the parse steps that created it and see at which point the reason for rejection is introduced. These facts can be used to apply filter rules during parsing or even when constructing the parser. We start an investigation into the derivation of efficient parsers for context-free grammars disambiguated with filters by refining the parsing schema for the Earley parsing algorithm.

Parsing Schemata Parsing schemata are defined by Sikkel in [Sik93, Sik94] as 'an intermediate level of abstraction between context-free grammars and parsers'. We will not give a lengthy introduction to parsing schemata but refer the reader to the above mentioned literature.

Parsing schema 6.1 describes Earley's parsing algorithm for arbitrary context-free grammars. 'Parsing' with this schema consists of finding the items derivable from the set of hypotheses $H = \{[a_i, i-1, i] \mid 1 \leq i \leq n\}$ containing the tokens for the string $a_1 \ldots a_n$ to be parsed. The declaration of $\mathcal{I}_{Earley}$ defines *parse items* of the form $[A \to \alpha \bullet \beta, i, j]$ for each production $A \to \alpha\beta$ and for each pair of positive integers i and j. The items from the set $\mathcal{I}_{Earley}$ can be deduced from H according to the rules in $\mathcal{D}_{Earley}$. Step D^{Init} predicts a sentence for start symbol S.[2] The D^{Pred} step predicts a production $B \to \gamma$ at position j if it is needed for the completion of an item $[A \to \alpha \bullet B\beta, i, j]$. The D^{Scan} and D^{Compl} steps finalize the recognition of a subconstruct. The rules are defined such that an item $[A \to \alpha \bullet \beta, i, j]$ can only be deduced if there is some $t_S \in \mathcal{T}_S$ such that $t_S = C[[A \to t_\alpha\beta]]$, $yield(t_\alpha) = a_{i+1} \ldots a_j$ and $yield(C[A]) = a_1 \ldots a_i\ A\ \gamma$ for some γ.

Earley with Trees An Earley item $[A \to \alpha\bullet\beta, i, j]$ is an abstraction of the set of partial parse trees

$$\{[A \to t_\alpha\beta] \mid yield(t_\alpha) = a_{i+1} \ldots a_j\}$$

In parsing schema 6.2 Earley items have been replaced by Earley tree-items where the list of symbols before the dot is replaced by a list of trees, i.e., $[A \to \alpha \bullet \beta, i, j]$ becomes $[A \to t_\alpha \bullet \beta, i, j]$. Note how trees are built as a result of the steps D^{Scan} and D^{Compl}. Schema 6.2 is an item refinement of schema 6.1; each item $[A \to \alpha \bullet \beta, i, j]$ in the latter represents the set of items

$$\{[A \to t_\alpha \bullet \beta, i, j] \mid yield(t_\alpha) = a_{i+1} \ldots a_j\}$$

in the former.

[2] Up to this point we did not use CFGs with start symbols; any symbol of a CFG can be the start symbol. This situation can be created in CFGs with start symbol by adding a production $S \to A$ for each $A \in V$.

PARSING SCHEMA 6.1 (Earley)

$$
\begin{array}{lcl}
\mathcal{I}_{Earley} & = & \{[A \to \alpha \bullet \beta, i, j] \mid A \to \alpha\beta \in \mathcal{P} \wedge 0 \le i \le j\}, \\
D^{Init} & = & \{\vdash [S \to \bullet\gamma, 0, 0]\}, \\
D^{Pred} & = & \{[A \to \alpha \bullet B\beta, i, j] \vdash [B \to \bullet\gamma, j, j]\}, \\
D^{Scan} & = & \{[A \to \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1], \\
D^{Compl} & = & \{[A \to \alpha \bullet B\beta, h, i], [B \to \gamma\bullet, i, j] \vdash [A \to \alpha B \bullet \beta, h, j]\}, \\
D_{Earley} & = & D^{Init} \cup D^{Pred} \cup D^{Scan} \cup D^{Compl}
\end{array}
\qquad \square
$$

PARSING SCHEMA 6.2 (Earley with Trees **ET**)

$$
\begin{array}{lcl}
\mathcal{I}_{ET} & = & \{[A \to t_\alpha \bullet \beta, i, j] \mid A \to \alpha\beta \in \mathcal{P} \wedge 0 \le i \le j \wedge t_\alpha \in \mathcal{T}_\alpha\}, \\
D^{Init} & = & \{\vdash [S \to \bullet\gamma, 0, 0]\}, \\
D^{Pred} & = & \{[A \to t_\alpha \bullet B\beta, i, j] \vdash [B \to \bullet\gamma, j, j]\}, \\
D^{Scan} & = & \{[A \to t_\alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1], \\
D^{Compl} & = & \{[A \to t_\alpha \bullet B\beta, h, i], [B \to t_\gamma\bullet, i, j] \vdash [A \to t_\alpha[B \to t_\gamma] \bullet \beta, h, j]\}, \\
D_{ET} & = & D^{Init} \cup D^{Pred} \cup D^{Scan} \cup D^{Compl}
\end{array}
\qquad \square
$$

Schema **ET** recognizes a string if an item of the form $[S \to t_\gamma\bullet, 0, n]$ can be deduced. Therefore, we can define a parser for arbitrary CFGs as

$$
\Pi_{\mathcal{G}}^{\mathbf{ET}}(w) = \{[S \to t_\gamma] \mid \mathbf{ET}(\mathcal{G})(w) \vdash [S \to t_\gamma\bullet, 0, n]\}
$$

(where n is the length of w). An implementation of a disambiguated grammar $\mathcal{G}/\mathcal{F}$ is the parser $\mathcal{F} \circ \Pi_{\mathcal{G}}^{\mathbf{ET}}$. This parser can be optimized if more information about a specific (class of) filter(s) is available. As an example we study the optimization for the priority conflict filter $\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}$ as defined in section 5.1.

Earley with Priorities By the definitions in sections 4 and 5.1 we have

$$
\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}(\Phi) = \{t \in \Phi \mid \forall s \in sub(t) : \neg\mathcal{E}^{\mathcal{R}}(s)\}
$$

Combining this with the specification of $\Pi_{\mathcal{G}}^{\mathbf{ET}}$ gives

$$
\begin{array}{l}
\mathcal{F}^{\overline{\mathcal{E}}^{\mathcal{R}}}(\Pi_{\mathcal{G}}^{\mathbf{ET}}(w)) = \\
\quad \{[S \to t_\gamma] \mid \mathbf{ET}(\mathcal{G})(w) \vdash [S \to t_\gamma\bullet, 0, n] \wedge \\
\qquad\qquad \forall s \in sub([S \to t_\gamma]) : \neg\mathcal{E}^{\mathcal{R}}(s)\}
\end{array}
$$

Following this specification we have to check every subtree of a final tree for a conflict violation with predicate $\mathcal{E}^{\mathcal{R}}$. In parsing schema 6.3 (**ETP**) this check is formulated in the declaration of the set of items $\mathcal{I}_{ETP}$ by forbidding items—and therefore parse steps—that represent a tree with a priority conflict.

An extra check has been built in the predict rule D^{Pred} to prevent the prediction of productions that would lead to an attempt to a complete with items $[A \to t_\alpha \bullet B\beta, h, i]$ and $[B \to t_\gamma\bullet, i, j]$ that would be forbidden because the resulting item $[A \to t_\alpha[B \to \gamma] \bullet \beta, h, j]$ has a priority conflict.

The restriction on items in $\mathcal{I}_{ETP}$ is rather general and it is not clear how to implement it; it seems that at each step all items involved have to be checked completely for priority conflicts. Predicate $\mathcal{E}^{\mathcal{R}}$ considers only the signatures of the root and the direct descendants of a tree. Furthermore, for each kind of violation just one of the descendants of the roots is considered. These two considerations lead to parsing schema **EP** (6.4). The items are normal Earley items again and the priority check is transferred to step D^{Compl}

PARSING SCHEMA 6.3 (Earley with trees and priorities **ETP**)

$$
\begin{array}{lcl}
\mathcal{I}_{ETP} & = & \{[A \to t_\alpha \bullet \beta, i, j] \mid A \to \alpha\beta \in \mathcal{P} \wedge 0 \leq i \leq j \wedge t_\alpha \in \mathcal{T}_\alpha \wedge \neg\overline{\mathcal{E}}^{\mathcal{R}}([A \to t_\alpha\beta])\},\\
D^{Init} & = & \{\vdash [S \to \bullet\gamma, 0, 0]\},\\
D^{Pred} & = & \{[A \to t_\alpha \bullet B\beta, i, j] \vdash [B \to \bullet\gamma, j, j] \mid \neg\overline{\mathcal{E}}^{\mathcal{R}}([A \to t_\alpha[B \to \gamma]\beta])\},\\
D^{Scan} & = & \{[A \to t_\alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1],\\
D^{Compl} & = & \{[A \to t_\alpha \bullet B\beta, h, i], [B \to t_\gamma\bullet, i, j] \vdash [A \to t_\alpha[B \to t_\gamma] \bullet \beta, h, j]\},\\
D_{ETP} & = & D^{Init} \cup D^{Pred} \cup D^{Scan} \cup D^{Compl}
\end{array}
\qquad \square
$$

PARSING SCHEMA 6.4 (Earley with priorities **EP**)

$$
\begin{array}{lcl}
\mathcal{I}_{EP} & = & \{[A \to \alpha \bullet \beta, i, j] \mid A \to \alpha\beta \in \mathcal{P} \wedge 0 \leq i \leq j\},\\
D^{Init} & = & \{\vdash [S \to \bullet\gamma, 0, 0]\},\\
D^{Pred} & = & \{[A \to \alpha \bullet B\beta, i, j] \vdash [B \to \bullet\gamma, j, j] \mid \neg\mathcal{E}^{\mathcal{R}}([A \to \alpha[B \to \gamma]\beta])\},\\
D^{Scan} & = & \{[A \to \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1],\\
D^{Compl} & = & \{[A \to \alpha \bullet B\beta, h, i], [B \to \gamma\bullet, i, j] \vdash [A \to \alpha B \bullet \beta, h, j]\\
 & & \mid \neg\mathcal{E}^{\mathcal{R}}([A \to \alpha[B \to \gamma]\beta])\},\\
D_{EP} & = & D^{Init} \cup D^{Pred} \cup D^{Scan} \cup D^{Compl}
\end{array}
\qquad \square
$$

where trees are built. The check in rule D^{Pred} is still useful to prevent unnecessary predictions. The lhs of an item no longer records the trees that are recognized, since all trees in items that are constructed have passed the priority conflict filter and have no influence on conflicts caused by their right siblings.

Implementation Currently we are implementing schema **ETP** by a standard generalized LR parser that checks for priority conflict violation at each reduction. This implementation is not only useful for disambiguation by priorities but can also be used for, e.g., subtree exclusion. It seems interesting to build an LR parse-table generator based on schema **EP**—possibly refined to SLR(1). The goto and predict functions would not have a single symbol as input but a tree skeleton of one level deep, i.e., a terminal or a production. The increase in the number of states is a problematic factor which requires further study.

7 Discussion

Many disambiguation methods for context-free grammars for programming languages have been proposed since the early seventies. We can only briefly sketch here some of the related work.

Lexical Disambiguation Disambiguation of the lexical tokens of a language is based on disambiguation rules that differ from those used at the context-free level. Typical rules are to prefer the *longest match*, to *prefer keywords*, or to disambiguate using type information obtained from a symbol table lookup. Experiments like, for instance [SC89], show that reasonably efficient lexical analysis can be implemented using techniques for parsing of context-free languages. It is therefore conceivable that our filtering approach can also be applied at the lexical level.

Disambiguation of operators, precedences Disambiguation of arithmetic operators is most commonly done by assigning a priority to each operator and to resolve conflicting priorities during parsing. Traditionally, resolution of priority conflicts and parsing are closely intertwined. Techniques for disambiguation have been applied in all phases discussed earlier in Section 2: grammar transformations, heuristic resolution of table conflicts during parser generation, rule based resolution, and post-parse filtering of parse trees.

Typical grammar transformations are the elimination of left/right recursion, and the coding of priority and associativity information in grammar rules.

Aho et al. [AJU75] describe how parsers for ambiguous grammars of binary expressions can be disambiguated with associativity and precedence declarations. This technique is applied by Johnson in the parser generator YACC [Joh75].

Earley [Ear75] describes a general scheme of precedence relations on context-free productions but only indicates how these could be used in static disambiguation. Precedences in the definition of programming languages are also discussed in [Aas91].

User definable disambiguation is, for instance, used in Prolog by declaring the absolute priority and associativity of operators.

The order of the productions in a context-free grammar is used in [Wha76] (backtracking) and YACC [Joh75] (resolution of shift/ reduce conflicts).

Wharton [Wha76] defines a backtracking parser that is guided by an ordering on parse steps. This ensures a single parse for any sentence over any grammar. However, this resolution of ambiguity is not based on the language being defined but on properties of the grammar productions.

In [HHKR92] (SDF) a strict partial order on productions is used as well as relative associativity of productions. This involves the detection of priority conflicts, and a multiset ordering on trees.

Thorup [Tho92, Tho94a, Tho94b] describes a technique of resolving LR and LL conflicts based on a set of rewrite rules over parse trees. A consequence of this work is disambiguation by exclusion of a set of tree patterns from the set of legal trees generated by a grammar.

Semantic Disambiguation Disambiguation can also be combined with the further semantic processing of parse trees. For instance, during static semantic checking (type checking) of a tree disambiguation can be done using type information. Examples of this approach based on attribute grammars can be found in [Aas92, Vel88a, Vel88b, OS92]. Van den Brand [Bra92] describes parse time application of semantic predicates in affix grammars (a variant of attribute grammars). His technique can also be applied to lexical disambiguation. Parr and Quong [PQ94] describe a disambiguation method that mixes syntactic and semantic disambiguation in LL parsers. Static semantic restrictions on parse trees are also used in [BC93, MS93]

An even stronger form of semantics-directed disambiguation can be found in languages like, e.g., APL where execution and parsing of a program occur simultaneously and decisions regarding parsing can depend on the outcome of execution.

Filters The notion of "filtering" as a means of disambiguation has been proposed by other authors as well. A separation between disambiguation and parsing is described in [LR81] where post-parse transformations on trees are used to produce the right parse tree. The idea appears also in e.g., [Aas92]. In our approach, the treatment of filters and their properties is more abstract and completely independent from the underlying parsing techniques.

In the framework of parsing schemata [Sik93] the notion of filtering is used for describing refinement relations between parsing algorithms.

In several approaches the *user* is queried interactively to filter ambiguities. An application of user-directed filtering is described in [Sha88] where a modification of YACC is used that reports parse conflicts during parsing (instead of during parser generation) and lets the user solve them. This technique is proposed as a solution of parsing documents in various ambiguous mark-up languages. Tomita [Tom85] also describes resolution of ambiguities by the user. The implementation of SDF ([HHKR92]) uses interactive dialogs to filter ambiguities that could not be resolved by priorities.

We did not propose a *formalism* for the specification of filters, since we mainly explored their semantics. Section 4.2, however, already suggests an approach to the specification of filters using predicates or partial orders, thus abstracting from the application of these to sets of trees or to parse forests.

Unparsing Disambiguation does not only play a role for parsing but also for unparsing, i.e., generating a string for some semantical value. If parse trees are mapped to abstract syntax trees and in this process bracket functions are considered as identity functions (e.g., (x) and x are identified at the level of abstract syntax trees), there is a problem during the reverse mapping of abstract syntax trees to parse trees since the right brackets may have to be introduced. For disambiguation methods based on precedence relations there is some body of knowledge how to do this. For arbitrary filters new theory is needed indicating how to unparse in this general case. Blikle [Bli89] describes the derivation of concrete syntax from abstract syntax. The equations that translate abstract to concrete syntax remind us of the algebraic specification of pretty printers in [Bra93]. Blikle's method breaks down when the syntax becomes too concrete, i.e., when brackets become optional. Thorup's disambiguation method [Tho94a] is aimed at solving that problem. However, unparsing is not addressed by him.

8 Conclusions

We have presented filters as a unifying framework for a large class of existing disambiguation methods. This framework can handle all 'logical' disambiguation methods but is not suited for defining parser-specific methods.

All disambiguation methods expressed as filter can be implemented by post-parse filtering. This gives a way of experimenting easily with new methods without having to adapt a given parser generator.

For us, the main merit of this framework is an increased understanding of the relationship between parsing and disambiguation. This insight may help during the design of new disambiguation methods and their integration with grammar definition formalisms.

Our initial ideas sketched in Section 6 suggest that a separation of parsing and filtering at the conceptual level does not exclude the use of efficient parsing techniques at the implementation level.

Acknowledgements

Arie van Deursen suggested considering higher-order patterns in filter definitions and commented on a draft of this paper.

References

[Aas91] Annika Aasa. Precedences in specifications and implementations of programming languages. In J. Maluzynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag, 1991.

[Aas92] Annika Aasa. *User Defined Syntax.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden, 1992.

[AJU75] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, 1975.

[BC93] Paul A. Bailes and Trevor Chorvat. Facet grammars: Towards static semantic analysis by context-free parsing. *Computer Languages*, 18(4):251–271, 1993.

[Bli89] A. Blikle. Denotational engineering. *Science of Computer Programming*, 12(3):207–253, 1989.

[Bra90] Ivan Bratko. *Prolog. Programming for Artificial Intelligence*. Addison-Wesley, second edition, 1990.

[Bra92] M.G.J. van den Brand. *Pregmatic, A generator for incremental programming environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[Bra93] M. G. J. van den Brand. Generation of language independent prettyprinters. Technical Report P9327, Programming Research Group, University of Amsterdam, Amsterdam, October 1993.

[Ear75] J. Earley. Ambiguity and precedence in syntax description. *Acta Informatica*, 4(1):183–192, 1975.

[HHKR92] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - Reference Manual,* version 6 December, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. Available by *ftp* from ftp.cwi.nl:/pub/gipe as SDFmanual.ps.Z.

[Joh75] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[Kli88] Paul Klint. Definitie van prioriteiten in SDF. (unpublished technical notition, in dutch), June 11, 1988.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[Lan74] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[LR81] Wilf R. LaLonde and Jim des Rivieres. Handling operator precedence in arithmetic expressions with tree transformations. *ACM Transactions on Programming Languages and Systems*, 3(1):83–103, January 1981.

[MS93] Carl McCrosky and Ken Sailor. A synthesis of type-checking and parsing. *Computer Languages*, 18(4):241–250, 1993.

[OS92] P. Oude Luttighuis and K. Sikkel. Attribute evaluation during generalized parsing. Memoranda Informatica 92-85, Universiteit Twente, Faculteit der Informatica, December 1992.

[PQ94] Terence J. Parr and Russel W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In Peter A. Fritzson, editor, *Compiler Construction, 5th International Conference, CC'94*, volume

786 of *LNCS*, pages 263–277, Edinburgh, U.K., April 1994. Springer-Verlag.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Rek92.ps.Z.

[Ric84] H. Richards. An overview of ARC SASL. *SIGPLAN Notices*, 19(10), October 1984.

[SC89] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178, 1989.

[Sha88] Michael Share. Resolving ambiguities in the parsing of translation grammars. *ACM SIGPLAN Notices*, 23:103–109, 1988.

[Sik93] Klaas Sikkel. *Parsing Schemata.* PhD thesis, Universiteit Twente, Enschede, December 1993.

[Sik94] Klaas Sikkel. How to compare the structure of parsing algorithms. In *First Workshop on Algebraic and Syntactic Methods in Computer Science*, Milano, October 13 & 14 1994.

[Tho92] Mikkel Thorup. Ambiguity for incremental parsing and evaluation. Technical Report PRG-TR-24-92, Program Research Group, Oxford University, Oxford, U.K., 1992.

[Tho94a] Mikkel Thorup. Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, 16(3):1024–1050, May 1994.

[Tho94b] Mikkel Thorup. Disambiguating grammars by exclusion of sub-parse trees. Technical Report 94/11, Dept. of Computer Science, University of Copenhagen, Denmark, 1994.

[Tom85] Masura Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, 1985.

[Vel88a] G. E. Veldhuijzen van Zanten. An attributed-LALR-parser generator for syntactically ambiguous grammars. Master's thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands, 1988.

[Vel88b] G. E. Veldhuijzen van Zanten. SABLE: A parser generator for ambiguous grammars. Memoranda Informatica INF 88-62, Department of Computer Science, University of Twente, Enschede, The Netherlands, December 1988.

[Wha76] R. Wharton. Resolution of ambiguity in parsing. *Acta Informatica*, 6(4):387–395, 1976.