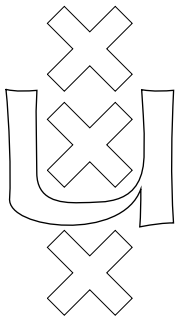
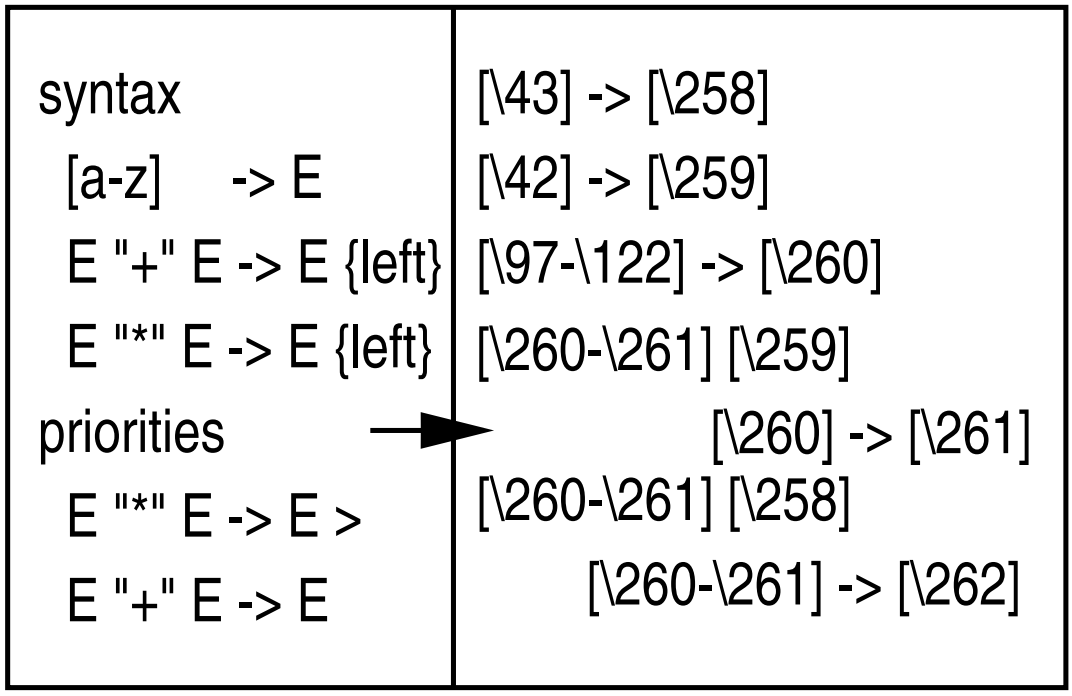


University of Amsterdam

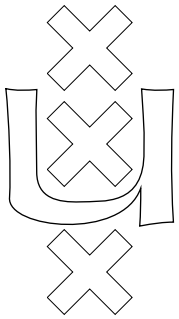
Programming Research Group



From Context-free Grammars with Priorities to Character Class Grammars

Eelco Visser

Report P9717 October 1997



University of Amsterdam

Department of Computer Science

Programming Research Group

From context-free grammars with priorities to character class grammars

Eelco Visser

Report P9717 October 1997

E. Visser

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7590
e-mail: visser@wins.uva.nl

This research was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO). Project 612-317-420: Incremental parser generation and context-sensitive disambiguation: a multi-disciplinary perspective.

Universiteit van Amsterdam, 1997

From Context-free Grammars with Priorities to Character Class Grammars

Eelco Visser

Priority and associativity declarations are used to disambiguate ambiguous fragments of context-free grammars. Usually this concerns expression grammars. It is possible to describe the same language by means of an unambiguous context-free grammar only, but using auxiliary non-terminals and extra chain productions resulting in a grammar that generates different and larger trees and extra parse steps.

In this paper we introduce a grammar transformation that translates a context-free grammar with priorities to a character class grammar that does only generate trees without priority conflicts. The transformed grammar has the property that each production corresponds to a production in the original grammar and that no extra productions are used. The parse trees over the transformed grammar are therefore isomorphic to parse trees over the original grammar.

1 Introduction

1.1 Priority and Associativity

Priority and associativity are notions that exist ever since formal languages have been used in mathematics. To read the mathematical expression $x \oplus y \ominus z$ it is necessary to know the priority relation between the operators $\oplus$ and $\ominus$. For instance, if $\oplus$ has higher priority than $\ominus$ the expression is read as $(x \oplus y) \ominus z$. In order to use mathematical expressions in a programming language the notions of priority and associativity have to be formalized in the description of the syntax of the language. Several formalizations have been developed.

Floyd (1962) introduced *operator precedence grammars* that allow the declaration of precedence and associativity among the operators of a context-free grammar over a single non-terminal in which no two non-terminals are adjacent. A restricted version of operator precedence can still be encountered in most reference manuals of programming languages that give a list of operators ordered by precedence. However, operator precedence is a very restricted method. It is not defined for grammars with ϵ-productions and cannot cope with 'invisible' operators such as the application operator in functional languages.

Another way to express priority and associativity is to encode the information in the context-free grammar of the language. This produces a considerable overhead (1) in the number of non-terminals, since each priority level is encoded by means of a non-terminal and (2) in the number of productions, since extra chain productions are needed. The effect is more parse steps (reductions

with chain rules) and parse trees that are different from the desired abstract syntax. Moreover, this encoding 'hardwires' priority information in the productions making the grammar harder to understand and to extend.

To overcome these drawbacks a number of more declarative methods have been proposed to separate the priority rules from the grammar rules and directly use the notions of priority and associativity. An ambiguous grammar of expressions is combined with a declaration of priority and associativity between the productions of the grammar. This results in better abstract syntax because no auxiliary non-terminals and chain rules have to be used. Earley (1975) and Aho *et al.* (1975) independently introduced priority declarations as a number of binary relations (priority, and left-, right-, and non-associativity) on productions of context-free grammars. These relations were restricted to range over unary and binary operators.

Aho *et al.* (1975) interpret such declarations by solving conflicts in an LR(1) parse table. Certain patterns of shift/reduce conflicts produced by ambiguous binary expressions can be solved by considering priority declarations. For instance, a shift/reduce conflict between the items $E \bullet + E \rightarrow E$ and $E + E \bullet \rightarrow E$ is solved in favor of reduce if $+$ is declared as left-associative. This implementation method is used in the YACC parser generator of Johnson (1975).

1.2 Disambiguation Filters

Heering *et al.* (1989) introduce a more general definition of priorities in the syntax definition formalism SDF. Priority declarations can range over arbitrary productions in the grammar. The YACC approach cannot be used for this more general definition of priorities, because it cannot deal with mixfix operators or binary expressions with an implicit operator (e.g., application in functional languages). Furthermore, SDF parsers are required to cope with arbitrary context-free grammars because this eases grammar development. These problems are solved by the following approach to parsing: A string is parsed using the context-free grammar part of a syntax definition using Generalized-LR parsing (Tomita, 1985; Rekers, 1992). The result is a parse forest representing all possible parses for the string. This forest is pruned using a filter that interprets the priority declarations. In fact two consecutive filters are used, one that selects trees without priority conflicts and a second one that selects the smallest tree in a multi-set ordering on parse trees based on priorities. In this paper we will concentrate on the first phase. The method is formalized by Klint (1988) who also gives a large number of examples.

Aasa (1992) and Thorup (1994) explore variants of priorities interpreted as filters. Aasa (1992) gives an alternative interpretation of priorities that is more complete than the definition of SDF. It is defined as a filter on sets of parse trees and in a variation of Earley's parsing algorithm. Thorup (1994) introduces disambiguation by means of tree rewrite rules. This can be used for instance to translate a right-associative tree to a left-associative one, thereby declaring that the construct should be interpreted as left-associative. A special case is a rewrite rule that rewrites a tree to $*$, meaning any other tree. The effect is that the tree in the rule is excluded as a subtree. This method is interpreted by an algorithm that tries to solve conflicts in an LR(k) parse table based on the rewrite rules. An important consideration is that the method should be complete, i.e., the resulting parser should be able to parse all strings generated

by the original grammar.

Klint and Visser (1994) generalize the notion of a filter on sets of parse trees providing a framework of *disambiguation filters* for formalization and comparison of disambiguation methods independent of parsing algorithms. Implementation of disambiguation is achieved by composing a Generalized-LR parser for the context-free part of the grammar with a filter expressing the disambiguation method. The filter prunes the parse forest produced by the GLR parser, deleting all trees that are not selected by the disambiguation method. This framework gives a very general account of disambiguation of context-free grammars. In Klint and Visser (1994) and Visser (1997d) several applications are discussed.

As an implementation paradigm disambiguation filters are not adequate for all disambiguation methods. If the number of possible trees generated by the grammar is very large, filtering after parsing leads to a bad performance of the resulting parsers. By partially evaluating the composition of a GLR parser and a filter this performance might be improved. In Visser (1997a) a case study in such a partial evaluation is discussed: a parser generation time interpretation of priorities is derived from the composition of the SLR(1) algorithm and a filter based on priority conflicts. Parsers produced by this partial evaluation do not construct parse trees with priority conflicts, thus saving computation time and memory. The method is used in the implementation of a parser generator for SDF2 (Visser, 1997c, 1997d), which is a redesign of SDF.

1.3 Grammar Transformation

In this paper we introduce a grammar transformation to interpret priorities. The advantage of this method is that it simplifies the parser generator by shifting the computation with priority rules from the parser generator to a grammar transformer. The main idea is to use character classes, compact representations of sets of numbers, to represent the priority levels in the grammar. By using such sets, no chain productions are needed and the exact structure of the original grammar can be preserved.

The paper is structured as follows. In §2 context-free grammars with priorities are introduced. A grammar generates a family of sets of parse trees. Priorities give a restriction on this set of parse trees by excluding trees matching certain patterns. In §3 character class grammars are introduced. Grammar symbols are restricted to sets of characters represented by numbers. By means of such character classes a non-terminal can be represented by the set of production numbers that define the non-terminal. It is shown how context-free grammars can be expressed in terms of character class grammars and vice versa In §4 priorities are expressed as an operation on a character class grammar. The result is a character class grammar that represents the original context-free grammar with priorities.

2 Context-free Grammars with Priorities

In this section we introduce context-free grammars, the interpretation of context-free grammars as tree generators, and the framework of priority and associativity declarations for the disambiguation of a certain class of ambiguous context-free grammars.

2.1 Context-Free Grammars

Context-free grammars were introduced by Chomsky (1956) to describe the *phrase structure* of natural language sentences. In most later work this aspect of context-free grammars is often ignored and the emphasis is on the string rewriting interpretation of context-free productions. Here we will use the phrase structure view to define context-free grammars because we are not just interested in the language defined by a grammar, but particularly in the structure assigned to sentences, since that will be the input for compilers and interpreters.

A commonly used notation for context-free grammars is BNF introduced by Naur *et al.* (1960) for the definition of Algol60. We use the notation of the syntax definition formalism SDF (Heering *et al.*, 1989). The most significant difference is the $\alpha \to A$ notation for productions, instead of the traditional $A \to \alpha$ or $A ::= \alpha$, to emphasize the use of productions as mixfix function declarations.

Formally, we have the following definition of context-free grammars.

Definition 2.1 (CFG) A *context-free grammar* $\mathcal{G}$ is a triple $\langle V_N, V_T, \mathcal{P} \rangle$, with V_N a finite set of nonterminal symbols, V_T a finite set of terminal symbols, V the set of symbols of $\mathcal{G}$ is $V_N \cup V_T$, and $\mathrm{P}(\mathcal{G}) = \mathcal{P} \subseteq V^* \times V_N$ a finite set of productions. We write $\alpha \to A$ for a production $p = \langle \alpha, A \rangle \in \mathcal{P}$.

Observe that we do not distinguish a start symbol from which sentences are derived. Each nonterminal in V_N generates a set of phrase structures or parse trees as is defined in the following definition.

Definition 2.2 (Parse Trees) A context-free grammar $\mathcal{G}$ generates a family of sets of *parse trees* $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(X) \mid X \in V)$ that contains the minimal sets $\mathcal{T}(\mathcal{G})(X)$ such that

$$\frac{X \in V_T}{X \in \mathcal{T}(\mathcal{G})(X)}$$

$$\frac{A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(A_1),\ \ldots,\ t_n \in \mathcal{T}(\mathcal{G})(A_n)}{[t_1 \ldots t_n \to A] \in \mathcal{T}(\mathcal{G})(A)}$$

We will write t_α for a list $t_1 \ldots t_n$ of trees where α is the list of symbols $X_1 \ldots X_n$ and $t_i \in \mathcal{T}(\mathcal{G})(X_i)$ for $1 \leq i \leq n$. Correspondingly we will denote the set of all lists of trees of type α as $\mathcal{T}(\mathcal{G})(\alpha)$. Using this notation $[t_1 \ldots t_n \to A]$ can be written as $[t_\alpha \to A]$ and the concatenation of two lists of trees t_α and t_β is written as $t_\alpha t_\beta$ and yields a list of trees of type $\alpha\beta$.

The *yield* of a tree is the concatenation of its leaves. The language $L(\mathcal{G})$ defined by a grammar $\mathcal{G}$ is the family of sets of strings $L(\mathcal{G})(A) = \mathrm{yield}(\mathcal{T}(\mathcal{G})(A))$.

Definition 2.3 (Parsing) 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 context-free grammar $\mathcal{G}$ that accepts exactly the sentences in $L(\mathcal{G})$ is defined by

$$\Pi(\mathcal{G})(w) = \{t \in \mathcal{T}(\mathcal{G})(A) \mid A \in V_N, \mathrm{yield}(t) = w\}$$

Example 2.4 As an example consider the following ambiguous grammar of expressions:

```
[\+]       -> "+"
[\*]       -> "*"
[a-z]      -> E
E "+" E    -> E
E "*" E    -> E
```

In this examples character classes are used to denote a number of alternative terminal characters. According to this grammar the string $a + a * a$ has two parses:

$$\begin{aligned}\Pi(\mathcal{G})(a + b * c) = \{&[[[a \to E] + [b \to E] \to E] * [c \to E] \to E] \\ &[[a \to E] + [[b \to E] * [c \to E] \to E] \to E]\}\end{aligned}$$

2.2 Priorities

Earley (1975) defines a priority declaration by means of the binary relations L, R and N that declare left-, right- and non-associativity, respectively, between productions and the relation $>$ that declares priority between productions. In SDF (Heering *et al.*, 1989) a formalism with the same underlying structure but with a less Spartan and more concise syntax is used. In SDF one writes `left` for L, `right` for R and `non-assoc` for N. We will use both notations, the formal definition is the following:

Definition 2.5 (Priority Declaration) A *priority declaration* $\Pr(\mathcal{G})$ for a context-free grammar $\mathcal{G}$ 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. □

A priority declaration is interpreted as the declaration of a set of parse tree patterns that cannot be used as subtrees of parse trees.

Definition 2.6 (Priority Conflict) The set $\mathrm{conflicts}(\mathcal{G})$ generated by the priority declaration of a grammar $\mathcal{G}$ is the smallest set of parse tree patterns of the form $[\alpha[\beta \to B]\gamma \to A]$ defined by the following rules.

$$\frac{\alpha B \gamma \to A > \beta \to B \in \Pr(\mathcal{G})}{[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

$$\frac{\gamma \neq \epsilon,\ \beta \to B\ (\texttt{right} \cup \texttt{non-assoc})\ B\gamma \to A \in \Pr(\mathcal{G})}{[[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

$$\frac{\alpha \neq \epsilon,\ \beta \to B\ (\texttt{left} \cup \texttt{non-assoc})\ \alpha B \to A \in \Pr(\mathcal{G})}{[\alpha[\beta \to B] \to A] \in \mathrm{conflicts}(\mathcal{G})}$$

This set defines the patterns of trees with a priority conflict. □

Using the definition of priority conflict we can define a filter on sets of parse trees that selects parse trees without a conflict.

Definition 2.7 (Priority Conflict Filter) A tree t has a *root priority conflict* if its root matches one of the tree patterns in $\mathrm{conflicts}(\mathcal{G})$. A tree t has a *priority conflict*, if t has a subtree s that has a root priority conflict. The filter $\mathcal{F}^{\Pr}$ is now defined by $\mathcal{F}^{\Pr}(\Phi) = \{t \in \Phi \mid t \text{ has no priority conflict}\}$. The pair $\langle \mathcal{G}, \Pr \rangle$ defines the disambiguated grammar $\mathcal{G}/\mathcal{F}^{\Pr}$. □

Example 2.8 Consider the following grammar with priority declaration

```
syntax
  [a-z]   -> E
  E "*" E -> E {left}
  E "+" E -> E {left}
priorities
  E "*" E -> E >
  E "+" E -> E
```

Here the attribute `left` of a production p abbreviates the declaration $p \mathrm{L} p$. The tree

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

has a priority conflict over this grammar—it violates the first priority condition since multiplication has higher priority than addition. The tree

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

does not have a conflict. These trees correspond to the (disambiguated) strings $(a + a) * a$ and $a + (a * a)$, respectively. The implication operator in logic is an example of a right associative operator: $a \to a \to a$ should be read as $a \to (a \to a)$. Non-associativity can be used to exclude unbracketed nested use of the equality operator in expressions using the production `E "=" E -> E`. □

3 Character Class Grammars

In the next section we will define a grammar transformation that embeds the definition of priorities in the productions of the grammar. For this purpose we introduce the notion of character class grammars in this section. In character class grammars the grammar symbols are character classes. Character classes are compact representations of sets of characters, i.e., numbers. Character classes originate from the definition of lexical syntax (for example, in LEX (Lesk and Schmidt, 1986)), where they are used to summarize a large number of chain productions. For instance, `[a-z]` denotes the set of all lowercase letters.

Character class grammars are used to reduce the complexity in the number of productions and non-terminals. Character classes denote an arbitrary collection of productions. Any character class grammar can be translated to a context-free grammar as we will show below, but this can lead to an exponential amount of non-terminals and for each non-terminal a number of chain productions. We also explore the reverse translation from context-free grammars to character class grammars. This translation will be the basis for the interpretation of priorities.

3.1 Character Classes

A character class is a compact representation of a set of characters.

Definition 3.1 (Character Class) A character class is a list $[cr_1 \dots cr_n]$ of numbers and pairs of numbers that represents the set of those numbers that

are either contained in the list or that are contained in one of the intervals. Operations on character classes are union ($\vee$), intersection ($\wedge$), difference ($/$) and complement with respect to the interval `\0` – `\TOP`.

Proposition 3.2 (Normal Form) *For each finite set of characters there is a unique, most compact character class representing it.*

Character classes can be specified using symbolic characters. These are translated to numbers using some character encoding. In this paper we use the ASCII encoding for characters in the ASCII range. An example character class is `[a-zA-Z]` denoting the set of all lower and uppercase characters. It's normal form is `[\65-\90\97-\122]`. Usually the characters in a character class are restricted to a fixed range, for instance, the 127 ASCII characters, the 256 characters that can be represented by a byte or the 16 bit characters of UniCode. However, there is no fundamental reason for this restriction. We can just as easily work with character classes in which there is no upperbound for characters or character ranges.

Visser (1997b) gives a specification of character classes in ASF+SDF including the normalization to the most compact normal form using rewrite rules. In the rest of this paper we will consider character classes modulo equivalence, i.e., assume that character classes are in normal form.

3.2 Character Class Grammars

Character classes are usually used in grammars to abbreviate a set of terminal characters. For example, the productions

```
[a-z]    -> Id
Id [a-z] -> Id
```

define the syntax of identifiers as a sequence of one or more lowercase letters. We generalize the use of character classes in grammars by allowing characters as non-terminals. In fact in character class grammars only character classes are used as grammar symbols.

Definition 3.3 (Character Class Grammar) A *character class grammar* is a finite set of productions $\alpha \to A$ such that $\alpha \in CC^*$ and $A = [c]$ is a singleton character class.

A character is a non-terminal in a CCG $\mathcal{G}$ if it is defined by at least one of the productions. Otherwise it is a terminal character.

Definition 3.4 (Parse Trees for CCGs) Given a character class grammar $\mathcal{G}$, the family of sets of parse trees $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(cc) \mid cc \in CC)$ contains the minimal sets $\mathcal{T}(\mathcal{G})(cc)$ such that

$$\frac{c \in cc,\ c \text{ is a terminal in } \mathcal{G}}{c \in \mathcal{T}(\mathcal{G})(cc)} \quad \text{(Ch)}$$

$$\frac{cc_1 \ldots cc_n \to cc_0 \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(cc_1),\ \ldots,\ t_n \in \mathcal{T}(\mathcal{G})(cc_n)}{[t_1 \ldots t_n \to cc] \in \mathcal{T}(\mathcal{G})(cc)} \quad \text{(App)}$$

$$\frac{t \in \mathcal{T}(\mathcal{G})(cc),\ cc \subseteq cc'}{t \in \mathcal{T}(\mathcal{G})(cc')} \quad \text{(Sub)}$$

3.3 From CCG to CFG

Any character class grammar can be expressed by means of a context-free grammar. This translation shows us why character class grammars are useful.

Algorithm 3.5 (CCG to CFG) Given a character-class grammar $\mathcal{G}$ construct the context-free grammar $\mathrm{cfg}(\mathcal{G})$according to the following algorithm:

(0) Define the terminal alphabet V_T of $\mathrm{cfg}(\mathcal{G})$ as the terminal characters of $\mathcal{G}$.

(1) Assign to each character class cc used in some of the productions a non-terminal $\mathrm{nt}(cc)$.

(2) For each production $cc_1 \dots cc_n \to cc_0$ in the grammar $\mathcal{G}$ define the production $\mathrm{nt}(cc_1) \dots \mathrm{nt}(cc_n) \to \mathrm{nt}(cc_0)$.

(3) For each non-terminal $\mathrm{nt}(cc)$ define the chain productions $\mathrm{nt}([n]) \to \mathrm{nt}(cc)$ for each $n \in cc$. □

Theorem 3.6 *The trees genererated by a character class grammar $\mathcal{G}$ and its corresponding context-free grammar* $\mathrm{cfg}(\mathcal{G})$ *are isomorphic, that is,* $\mathcal{T}(\mathcal{G}) \cong \mathcal{T}(\mathrm{cfg}(\mathcal{G}))$.

Proof. Extend the translation cfg to trees, i.e., define the function $\mathrm{cfg} : \mathcal{T}(\mathcal{G}) \to \mathcal{T}(\mathrm{cfg}(\mathcal{G}))$ as follows:

$$\frac{c \in V_T}{\mathrm{cfg}(c) = c}$$

$$\frac{cc_1 \dots cc_n \to cc_0 \in \mathrm{P}(\mathcal{G})}{\mathrm{cfg}([t_1 \dots t_n \to cc_0]) = [[\mathrm{cfg}(t_1) \to \mathrm{nt}([cc_1])] \dots [\mathrm{cfg}(t_n) \to \mathrm{nt}([cc_n])] \to \mathrm{nt}(cc_0)]}$$

This translation is clearly a bijection. □

In this translation we see that character classes are more concise than context-free grammars because of the chain rules implied by the character classes.

3.4 From CFG to CCG

Conversely, and more interestingly for our intended application, character class grammars can be used to describe context-free grammars. The following algorithm constructs a character class grammar for a given context-free grammar such that it generates trees with the same structure. This time the structure is more faithfully copied.

Algorithm 3.7 (CFG to CCG) Given a context-free grammar $\mathcal{G}$, construct the character-class grammar $\mathrm{ccg}(\mathcal{G})$ according to the following algorithm:

(0) Assign a character $\mathrm{num}(X)$ to each terminal symbol X in $\mathcal{G}$.

(1) Assign a unique number $\mathrm{num}(\alpha \to A)$ to each production $\alpha \to A$ in the grammar such that the smallest production number is larger than the largest character number.

(2) Assign to each non-terminal A in the grammar a character class $\mathrm{nums}(A)$ containing the numbers of the productions for that non-terminal, i.e., such that if $\alpha \to A$ is a production then $\mathrm{num}(\beta \to A) \in \mathrm{nums}(A)$.

(3) For each production $\alpha \to A$: (3a) Replace the result A by the character class $[\mathrm{num}(\alpha \to A)]$ containing the number of the production. (3b) Replace each non-terminal A_i in α by the character class $\mathrm{nums}(A_i)$. □

Theorem 3.8 *The trees genererated by a context-free grammar $\mathcal{G}$ and its corresponding character class grammar* $\mathrm{ccg}(\mathcal{G})$ *are isomorphic, that is,* $\mathcal{T}(\mathcal{G}) \cong \mathcal{T}(\mathrm{ccg}(\mathcal{G}))$.

Proof. Extend the translation ccg to trees, i.e., define the function $\mathrm{ccg} : \mathcal{T}(\mathcal{G}) \to \mathcal{T}(\mathrm{ccg}(\mathcal{G}))$ as follows:

$$\frac{X \in V_T}{\mathrm{ccg}(X) = \mathrm{num}(X)}$$

$$\frac{A_1 \ldots A_n \to A_0 \in \mathrm{P}(\mathcal{G})}{\mathrm{ccg}([t_1 \ldots t_n \to A_0]) = [\mathrm{ccg}(t_1) \ldots \mathrm{ccg}(t_n) \to \mathrm{num}(A_0)]}$$

It is clear that $\mathrm{ccg}(t) \in \mathcal{T}(\mathrm{ccg}(\mathcal{G}))$ and that this translation is a bijection. □

Example 3.9 Take the following grammar of expressions

```
[\+]      -> "+"
[\*]      -> "*"
[a-z]     -> E
E "*" E   -> E
E "+" E   -> E
```

Applying the algorithm to this grammar leads to the following steps. (0) The terminals of this grammar are already character classes. Note that the normal form of `[\+]` is `[\43]`. (1) Assign numbers to the productions

```
num([\43]       -> "+") = \258
num([\42]       -> "*") = \259
num([\97-\122] -> E)    = \260
num(E "*" E     -> E)   = \261
num(E "+" E     -> E)   = \262
```

(2) Assign character classes to the non-terminals.

```
nums(E)   = [\260-\262]
nums("*") = [\259]
nums("+") = [\258]
```

(3) Replace the non-terminals.

```
[\43]                            -> [\258]
[\42]                            -> [\259]
[\97-\122]                       -> [\260]
[\260-\262] [\259] [\260-\262] -> [\261]
[\260-\262] [\258] [\260-\262] -> [\262]
```

Observe that this is not the most compact encoding possible for this grammar. A more compact encoding of the example grammar is:

```
[\97-\122]                          -> [\260]
[\260-\261] [\42-\43] [\260-\261] -> [\261]
```

But the point of the encoding is not the compact representation for pure context-free grammars. In the next section we will use the encoding above as basis for the grammar transformation to embed priorities.

4 Priorities as Grammar Transformation

Given a context-free grammar with priorities we can derive a character class grammar (without priorities) that exactly describes the parse trees of the original grammar. The idea is to remove from a character class at some position in a left-hand side the numbers of productions that would cause a priority conflict at that position. For instance, to express that an addition should not occur as the child of a multiplication the production number for the addition can be removed from the class `[\260-\262]` resulting in the adapted production

```
[\260-\261] [\259] [\260-\261] -> [\261]
```

for multiplication. The functions $\mathbf{L}$, $\mathbf{M}$ and $\mathbf{R}$ in the algorith refer to left positions, middle positions and right positions in the left-hand side of a production.

Algorithm 4.1 (Priority CFG to CCG) Given a context-free grammar $\mathcal{G}$ with priorities $\mathrm{Pr}(\mathcal{G})$, construct the character-class grammar $\mathrm{pccg}(\mathcal{G})$ according to the following algorithm:

(1) Construct the CCG for the context-free part of the grammar according to Algorithm 3.7.

(2) Translate the priorities to three functions $\mathbf{L}$, $\mathbf{M}$ and $\mathbf{R}$ mapping productions numbers to sets of numbers as follows:

$$\frac{p_1\ (> \cup \texttt{right} \cup \texttt{non-assoc})\ p_2}{\mathrm{num}(p_2) \in \mathbf{L}(\mathrm{num}(p_1))} \quad \text{(L)}$$

$$\frac{p_1 > p_2}{\mathrm{num}(p_2) \in \mathbf{M}(\mathrm{num}(p_1))} \quad \text{(M)}$$

$$\frac{p_1\ (> \cup \texttt{left} \cup \texttt{non-assoc})\ p_2}{\mathrm{num}(p_2) \in \mathbf{R}(\mathrm{num}(p_1))} \quad \text{(R)}$$

(3) Filter the character classes $cc_1 \ldots cc_n$ in the left-hand side of each production $cc_1 \ldots cc_n \to [p]$ as follows:

(a) If $n = 0$ (ϵ-production) or $n = 1$ (empty production), then do nothing.

(b) If $n > 1$: take $cc'_1 := cc_1 / \mathbf{L}(p)$, take for $1 < i < n$: $cc'_i := cc_i / \mathbf{M}(p)$, and take $cc'_n := cc_n / \mathbf{R}(p)$. Replace the production by $cc'_1 \ldots cc'_n \to [p]$. □

Theorem 4.2 *The trees genererated by a character class grammar* $\mathrm{pccg}(\mathcal{G})$ *do not contain priority conflicts.*

Proof. For each of the clauses of Definition 2.6 we have to check that the excluded parse tree patterns are not generated by $\mathrm{pccg}(\mathcal{G})$.

(1) $[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})$ due to the priority $\alpha B\gamma \to A > \beta \to B$. Consider the translation of the production: First the non-terminals are replaced by character classes

$$\mathrm{ccg}(\alpha B\gamma \to A) \mapsto \mathrm{nums}(\alpha)\,\mathrm{nums}(B)\,\mathrm{nums}(\gamma) \to \mathrm{num}(\alpha B\gamma \to A)$$

Then these character classes are filtered. In particular, because of the priority rule, $\mathrm{num}(\beta \to B) \notin cc'_B$. Hence, $\mathrm{ccg}([t_\alpha[t_\beta \to B]t_\gamma \to A]) \notin \mathcal{T}(\mathrm{pccg}(\mathcal{G}))(A)$ for any t_α, t_β and t_γ.

(2,3) The other cases are similar. □

Example 4.3 (1) Take the character class grammar obtained in Example 3.9.
(2) We derive the following encoding of the priority rules:

```
L(\261) = [\262]        L(\262) = []
M(\261) = []            M(\262) = []
R(\261) = [\261-\262]   R(\262) = [\262]
```

(3) Filter the non-terminal classes using this encoding

```
[\43]                           -> [\258]
[\42]                           -> [\259]
[\97-\122]                      -> [\260]
[\260-\261] [\259] [\260]       -> [\261]
[\260-\262] [\258] [\260-\261] -> [\262]
```

Observe that the transformed grammar has the exact same structure as the original context-free grammar, i.e., each production in the transformed grammar corresponds to a production in the original grammar and the left-hand sides of productions also have the same structure. The only difference is a more fine-grained specification of usage of productions at specific positions in left-hand sides. After parsing the production numbers can be used to construct parse trees over the original grammar.

Compare this to the usual encoding using extra non-terminals and extra chain productions. To help the comparison the production numbers have been added.

```
[\43]   -> "+"       [\258]
[\42]   -> "*"       [\259]
[a-z]   -> F         [\260]
T "*" F -> T         [\261]
F       -> T
E "+" T -> E         [\262]
T       -> E
```

In this grammar chains $F \to T \to E$ are built to include 'simple' expressions into sums. The length of such chains grows with the number of priority levels.

Now consider again our example string `a+b*c`. According to the transformed grammar above, this string has only one parse tree, which is the following:

```
[[\97 -> \260] [\43 -> \258]
 [[\98 -> \260] [\42 -> \259] [\99 -> \260] -> \261]
 -> \262]
```

This corresponds to the tree

$$[[a \to E] + [[b \to E] * [c \to E] \to E] \to E]$$

that is declared by the priority rules.

5 Discussion

We have defined a transformation on context-free grammars that compiles priority and associativity declarations into the productions of the grammar by using character classes to concisely encode sets of productions.

Even though transformed grammars do not generate trees with priority conflict this does not mean that they are unambiguous or do not cause conflicts in parse tables. Conflicts and ambiguities can have been overlooked or caused by constructs that can not be dealt with by means of priorities. See (Visser, 1997d) for further discussions and solutions.

A first prototype of the transformation algorithm has been implemented as part of the implementation of a parser generator for the syntax definition formalism SDF2. The usage of character classes in context-free grammars came natural in this setting because SDF2 integrates lexical and context-free syntax of languages by combining them into a single context-free grammar. Parsers for such grammars do not need separate lexical analyzers and are thus called *scannerless* parsers (Visser, 1997d).

Although the parser generator spends no time on the lookup of information in the priority table, more time is spent on character class computations. An efficient implementation of character classes is therefore essential for a successful implementation.

Further optimizations can be achieved by further transforming the derived character class grammars. An obvious candidate is chain rule elimination. If $[n] \to [m]$ is a chain production, replace everywhere m by n, effectively removing a production from the grammar.

References

Aasa, A. (1992). *User Defined Syntax*. Ph.D. thesis, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden.

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

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, **2**, 113–124.

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

Floyd, R. W. (1962). Syntactic analysis and operator precedence. *Communications of the ACM*, **5**(10), 316–333.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, **24**(11), 43–75.

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

Klint, P. (1988). Definitie van prioriteiten in SDF. Unpublished technical note (in dutch).

Klint, P. and Visser, E. (1994). Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Lesk, M. E. and Schmidt, E. (1986). *LEX — A lexical analyzer generator*. Bell Laboratories. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

Naur, P. *et al.* (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, **3**(5), 299–314.

Rekers, J. (1992). *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

Thorup, M. (1994). Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, **16**(3), 1024–1050.

Tomita, M. (1985). *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

Visser, E. (1997a). A case study in optimizing parsing schemata by disambiguation filters. In *International Workshop on Parsing Technology (IWPT'97)*, pages 210–224, Boston, USA. Massachusetts Institute of Technology.

Visser, E. (1997b). Character classes. Technical Report P9708, Programming Research Group, University of Amsterdam.

Visser, E. (1997c). A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam.

Visser, E. (1997d). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

A Specification of the Transformation

This appendix contains the specification in ASF+SDF of the transformation to character class grammars described in the paper. The transformation works on SDF2 syntax definitions in normal form. That is, it transforms the productions

in the 'syntax' sections interpreting the priorities in the priorities section assuming that the latter are normalized according to the specification in Visser (1997c).

The specification is divided into two modules. Module ProdSym-Numbers defines a table for storing the assignment of characters to productions and character classes to symbols. For convenience an extra grammar section with the keyword 'numbers' is introduced for storing this table. Module PCFG-to-CCG defines the transformation proper.

module ProdSym-Numbers
imports CC-Sdf-Syntax Character-Class-Normalization Kernel-Sdf-Projection
Grammar-Normalization
exports
sorts PSNumbers ProdNum ProdNumbers SymNum SymNumbers
context-free syntax

PSNumbers "[" Production "]"	→ Character
PSNumbers "[" Symbol "]"	→ CharClass
new-numbers	→ PSNumbers
number(Productions, PSNumbers)	→ PSNumbers
"<" ProdNumbers "," SymNumbers "," Character ">"	→ PSNumbers
"<" Character "," Production ">"	→ ProdNum
"[" ProdNum* "]"	→ ProdNumbers
"<" CharClass "," Symbol ">"	→ SymNum
"[" SymNum* "]"	→ SymNumbers
numbers PSNumbers	→ Grammar
"N"(Grammar)	→ PSNumbers

variables

"*psn*"[*0-9'*]*	→ PSNumbers
"*pn*"[*0-9'*]*	→ ProdNum
"*sn*"[*0-9'*]*	→ SymNum
"*pn**"[*0-9'*]*	→ ProdNum*
"*sn**"[*0-9'*]*	→ SymNum*

hiddens
context-free syntax
mk-charclass(Symbol) → CharClass
equations

[1] $\mathrm{N}(\text{numbers } psn) = psn$
[2] $\mathrm{N}(\mathcal{G} \text{ numbers } psn) = psn$
[3] $\mathrm{N}(\emptyset) = \text{new-numbers}$
[4] $\mathrm{N}(\mathcal{G}_1\ \mathcal{G}_2) = \mathrm{N}(\mathcal{G}_1)$ **otherwise**

Assigning numbers to symbols and productions

[5] $\text{new-numbers} = \langle [], [], \text{succ}(\text{succ}(\texttt{\textbackslash TOP}))\rangle$

[6] $\text{number}(, psn) = psn$

$$[7]\quad \frac{\begin{array}{c}\mathrm{succ}(\mathsf{c}_1) = \mathsf{c}_2,\\ [pn_2^*] = [pn_1^*\ \langle \mathsf{c}_2,\ \alpha \to \mathcal{A}\ \$\rangle],\\ [sn_2^*] = [\langle [\mathsf{c}_2],\ \mathcal{A}\rangle\ sn_1^*]\end{array}}{\begin{array}{l}\mathrm{number}(\alpha \to \mathcal{A}\ \$\ p^*,\ \langle [pn_1^*],\ [sn_1^*],\ \mathsf{c}_1\rangle)\ =\\ \mathrm{number}(p^*,\ \langle [pn_2^*],\ [sn_2^*],\ \mathsf{c}_2\rangle)\end{array}}$$

$$[8]\quad [\langle cc_1,\ \mathcal{A}\rangle\ sn_1^*\ \langle cc_2,\ \mathcal{A}\rangle\ sn_2^*]\ =\ [sn_1^*\ \langle cc_1 \vee cc_2,\ \mathcal{A}\rangle\ sn_2^*]$$

Looking up numbers

$$[9]\quad \langle [pn_1^*\ \langle \mathsf{c},\ p_1\rangle\ pn_2^*],\ [sn^*],\ \mathsf{c}'\rangle[p_2] = \mathsf{c}\quad \textbf{when}\ \ p_1 \cong p_2 = \top$$
$$[10]\quad \langle [pn^*],\ [sn_1^*\ \langle cc,\ \mathcal{A}\rangle\ sn_2^*],\ \mathsf{c}'\rangle[\mathcal{A}] = cc$$
$$[11]\quad \langle [pn^*],\ [sn^*],\ \mathsf{c}'\rangle[\mathcal{A}] = \text{mk-charclass}(\mathcal{A})\quad \textbf{otherwise}$$

$$[12]\quad \text{mk-charclass}(cc) = cc$$
$$[13]\quad \text{mk-charclass}(\mathcal{A}) = []\quad \textbf{otherwise}$$

module PCFG-to-CCG
imports CC-Sdf-Syntax Priority-Sdf-Syntax Character-Class-Normalization
Kernel-Sdf-Projection Priority-Sdf-Projection
Restrictions-Sdf-Projection ProdSym-Numbers^A
exports
sorts PrioRel NumPrior NumPriors NumPriority NumPriorities
context-free syntax
Character "[" CharClass "," CharClass "," CharClass "]" → NumPriority
{NumPriority ","}* → NumPriorities
NumPriorities "." Character → NumPriority
NumPriorities "++" NumPriorities → NumPriorities {**right**}
variables
"*npr*"[*0-9'*]* → NumPriority
"*npr**"[*0-9'*]* → {NumPriority ","}*
context-free syntax
ccg(Grammar) → Grammar
rejected(Attributes) → Attributes
prods(PSNumbers, Productions) → Productions
syms(PSNumbers, Symbols) → Symbols
restrs(PSNumbers, Restrictions) → Restrictions
union(PSNumbers, Symbols) → CharClass
priors(PSNumbers, Priorities) → NumPriorities
filter(NumPriorities, Productions) → Productions
filter(NumPriority, Symbols, Symbols) → Symbols

restr(PSNumbers, Restrictions) → Restrictions
equations
From CFG to CCG

$$[1]\quad \frac{\mathrm{P}(\mathcal{G}) = p^*,\ \ \mathrm{number}(p^*, \text{new-numbers}) = psn}{\begin{array}{lll}\mathrm{ccg}(\mathcal{G}) &=& \text{numbers } psn\\ & & \text{syntax filter}(\mathrm{priors}(psn, \mathrm{Pr}(\mathcal{G})), \mathrm{prods}(psn, p^*))\\ & & \text{restrictions restrs}(psn, \mathrm{R}(\mathcal{G}))\end{array}}$$

[2] $\text{prods}(psn,) =$

[3] $\text{prods}(psn, \alpha \to \mathcal{A}\ \$) = \text{syms}(psn, \alpha) \to [psn[\alpha \to \mathcal{A}\ \$]]\ \text{rejected}(\$)$

[4] $\text{prods}(psn, p_1^+\ p_2^+) = \text{prods}(psn, p_1^+) +\!\!+ \text{prods}(psn, p_2^+)$

[5] $\text{syms}(psn,) =$

[6] $\text{syms}(psn, \mathcal{A}) = psn[\mathcal{A}]$

[7] $\text{syms}(psn, \alpha^+\ \beta^+) = \text{syms}(psn, \alpha^+) +\!\!+ \text{syms}(psn, \beta^+)$

[8] $\text{rejected}(\{attr_1^*, \text{reject}, attr_2^*\}) = \{\text{reject}\}$

[9] $\text{rejected}(\$) =$ **otherwise**

[10] $\text{restrs}(psn,) =$

[11] $\text{restrs}(psn, restr_1^+\ restr_2^+) = \text{restrs}(psn, restr_1^+) +\!\!+ \text{restrs}(psn, restr_2^+)$

[12] $\text{restrs}(psn, \alpha +\!\!- cc) = \text{union}(psn, \alpha) +\!\!- cc$

[13] $\text{union}(psn,) = []$

[14] $\text{union}(psn, \alpha^+\ \beta^+) = \text{union}(psn, \alpha^+) \vee \text{union}(psn, \beta^+)$

[15] $\text{union}(psn, cc) = cc$

[16] $\text{union}(psn, \mathcal{A}) = psn[\mathcal{A}]$ **otherwise**

Translating a priority relation to a numeric priority relation

[17] $\text{priors}(psn,)\ =$

[18] $$\frac{psn[p_1] = \mathsf{c}_1,\ psn[p_2] = \mathsf{c}_2}{\text{priors}(psn, p_1 > p_2)\ =\ \mathsf{c}_1[[\mathsf{c}_2], [\mathsf{c}_2], [\mathsf{c}_2]]}$$

[19] $$\frac{psn[p_1] = \mathsf{c}_1,\ psn[p_2] = \mathsf{c}_2}{\text{priors}(psn, p_1\ \text{assoc}\ p_2)\ =\ \mathsf{c}_1[[], [], [\mathsf{c}_2]], \mathsf{c}_2[[], [], [\mathsf{c}_1]]}$$

[20] $$\frac{psn[p_1] = \mathsf{c}_1,\ psn[p_2] = \mathsf{c}_2}{\text{priors}(psn, p_1\ \text{left}\ p_2)\ =\ \mathsf{c}_1[[], [], [\mathsf{c}_2]], \mathsf{c}_2[[], [], [\mathsf{c}_1]]}$$

[21] $$\frac{psn[p_1] = \mathsf{c}_1,\ psn[p_2] = \mathsf{c}_2}{\text{priors}(psn, p_1\ \text{right}\ p_2)\ =\ \mathsf{c}_1[[\mathsf{c}_2], [], []], \mathsf{c}_2[[\mathsf{c}_1], [], []]}$$

[22] $$\frac{psn[p_1] = \mathsf{c}_1,\ psn[p_2] = \mathsf{c}_2}{\text{priors}(psn, p_1\ \text{non-assoc}\ p_2)\ =\ \mathsf{c}_1[[\mathsf{c}_2], [], [\mathsf{c}_2]], \mathsf{c}_2[[\mathsf{c}_1], [], [\mathsf{c}_1]]}$$

[23] $\text{priors}(psn, pr, pr^+)\ =\ \text{priors}(psn, pr^+) +\!\!+ \text{priors}(psn, pr)$

[24] $npr_1^* +\!\!+ = npr_1^*$

[25] $npr_1^* +\!\!+ npr, npr_2^* = npr_1^*, npr +\!\!+ npr_2^*$

Joining entries

[26] $$npr_1^*, \mathsf{c}[cc_{11}, cc_{12}, cc_{13}], npr_2^*, \mathsf{c}[cc_{21}, cc_{22}, cc_{23}]$$
$$= npr_1^*, \mathsf{c}[cc_{11} \vee cc_{21}, cc_{12} \vee cc_{22}, cc_{13} \vee cc_{23}], npr_2^*$$

Looking up priorities

[27] $$npr_1^*, \mathsf{c}[cc_1,\ cc_2,\ cc_3],\ npr_2^* \cdot \mathsf{c} = \mathsf{c}[cc_1,\ cc_2,\ cc_3]$$

[28] $$npr^* \cdot \mathsf{c} = \mathsf{c}[[],\ [],\ []] \quad \textbf{otherwise}$$

Filtering the CCG

[29] $$\text{filter}(npr^*,\ ) =$$

[30] $$\text{filter}(npr^*,\ p_1^+\ p_2^+) = \text{filter}(npr^*,\ p_1^+) +\!\!+ \text{filter}(npr^*,\ p_2^+)$$

[31] $$\text{filter}(npr^*,\ \alpha \rightarrow [\mathsf{c}]\ \$) = \text{filter}(npr^* \cdot \mathsf{c},\ ,\ \alpha) \rightarrow [\mathsf{c}]\ \$$$

[32] $$\text{filter}(npr,\ ,\ ) \ =$$

[33] $$\text{filter}(npr,\ ,\ \mathcal{A}) \ =\ \mathcal{A}$$

[34] $$\frac{npr = \mathsf{c}[cc_1,\ cc_2,\ cc_3]}{\text{filter}(npr,\ ,\ cc\ \alpha^+) \ =\ \text{filter}(npr,\ cc\ /\ cc_1,\ \alpha^+)}$$

[35] $$\frac{npr = \mathsf{c}[cc_1,\ cc_2,\ cc_3]}{\text{filter}(npr,\ \alpha^+,\ cc\ \beta^+) \ =\ \text{filter}(npr,\ \alpha^+\ cc\ /\ cc_2,\ \beta^+)}$$

[36] $$\frac{npr = \mathsf{c}[cc_1,\ cc_2,\ cc_3]}{\text{filter}(npr,\ \alpha^+,\ cc) \ =\ \alpha^+\ cc\ /\ cc_3}$$

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (URL `http://www.wins.uva.nl/research/prog/reports/`) or using anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[P9717] E. Visser. *From Context-free Grammars with Priorities to Character Class Grammars.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9714] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification of Rewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*