Delft University of Technology
Software Engineering Research Group
Technical Report Series

Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking

Eric Bouwers, Martin Bravenboer, Eelco Visser

Report TUD-SERG-2007-004

TU Delft

SERG

TUD-SERG-2007-004

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking

Eric Bouwers[a,1] Martin Bravenboer[b,2] Eelco Visser[b,3]

[a] *Department of Information and Computing Sciences*
Utrecht University, The Netherlands

[b] *Department of Software Technology*
Delft University of Technology, The Netherlands

Abstract

A wide range of parser generators are used to generate parsers for programming languages. The grammar formalisms that come with parser generators provide different approaches for defining operator precedence. Some generators (e.g. YACC) support precedence declarations, others require the grammar to be unambiguous, thus encoding the precedence rules. Even if the grammar formalism provides precedence rules, a particular grammar might not use it. The result is grammar variants implementing the same language. For the C language, the GNU Compiler uses YACC with precedence rules, the C-Transformers uses SDF without priorities, while the SDF library does use priorities. For PHP, Zend uses YACC with precedence rules, whereas PHP-front uses SDF with priority and associativity declarations.
The variance between grammars raises the question if the precedence rules of one grammar are compatible with those of another. This is usually not obvious, since some languages have complex precedence rules. Also, for some parser generators the semantics of precedence rules is defined operationally, which makes it hard to reason about their effect on the defined language. We present a method and tool for comparing the precedence rules of different grammars and parser generators. Although it is undecidable whether two grammars define the same language, this tool provides support for comparing and recovering precedence rules, which is especially useful for reliable migration of a grammar from one grammar formalism to another. We evaluate our method by the application to non-trivial mainstream programming languages, such as PHP and C.

Keywords: Precedence, precedence rules, disambiguation, priorities, associativity, grammar engineering, grammar recovery, parsing, YACC, SDF.

1 Introduction

Defining the syntax of a programming language using a context-free grammar is one of the most established practices in the software industry and computer science. For various reasons a wide range of parser generators are used to generate parsers from context-free grammars. For almost every mainstream programming language there exists a series of parser generators, not only featuring different parsing algorithms, but also different grammar formalisms. These grammar formalisms often provide methods for declaring the precedence of operators, since the notions of priority and associativity are pervasive in the definition of the syntax of programming languages.

[1] Email: embouwer@cs.uu.nl
[2] Email: martin@st.ewi.tudelft.nl (corresponding author)
[3] Email: visser@acm.org

As early as 1975 Aho and Johnson recognized [1] that for many languages the most natural grammar is not accepted by the parser generators that are used in practice, since the grammar does not fall in the class of context-free grammars for which the parser generator can produce an efficient parser. Aho and Johnson proposed to define the syntax of a programming languages as an ambiguous grammar combined with disambiguation rules that tell the parser how to resolve a *parsing action conflict*, a method that was implemented in the now still dominant YACC parser generator [5]. Unfortunately, most of the work on separate precedence declarations has been guided by the underlying parsing technique and not by an analysis of the requirements and fundamentals of precedence declarations. Indeed, parser generators only support precedence rules that can efficiently be implemented in the parser. This is understandable from a practical point of view, yet the result is that there is little known about the actual requirements for separate precedence declarations. Indeed, the semantics of separate precedence declarations is apparently so ill-defined that it is still not used in language specifications. Rather, language specifications prefer to encode precedence rules in the productions of the grammar. Sadly, it is difficult to disagree with this approach, since an encoding in productions is still the most precise, formal, and parsing technology independent way of defining precedences!

In this paper, we argue that precedence rules need to be liberated from the idiosyncrasies of specific parser generators. The reasons for this are closely related to the efforts to work towards an engineering discipline for grammarware [6,11,13,9]. Liberating grammars from concrete parser generators is not a new idea [8], however precedence rules have never been studied fundamentally outside of the context of specific parsing technologies or parser generators. Indeed, there is currently, for example, no solid methodology to

- recover precedence rules from 'legacy' grammar formalisms. For example, for PHP there is no language specification, only a YACC grammar. Due to the conflict resolution semantics of YACC precedence declarations, the exact precedence rules of PHP are currently very difficult to determine.
- compare the precedence rules of two grammars, whether they are defined in the same grammar formalism or not. For example, for the C language, the GNU Compiler uses YACC with precedence rules, the C-Transformers [2] uses SDF [15] without priorities, while the SDF library does use priorities. For PHP, Zend uses YACC with precedence rules, whereas PHP-front uses SDF with priority and associativity declarations. However, there is no way to check that the precedence rules of one grammar are compatible with those of another.
- reliably migrate a grammar from one grammar formalism to another including its precedence rules. This does not necessarily have to be completely automatic, but at least there can be support for recovering precedence rules and generating precedence declarations in the new formalism.

In this paper we present a method and its implementation for recovering precedence rules from grammars. Our method is based on a core formalism for defining precedence rules, which is independent of specific parser generators. Based on this formalism and the recovery of precedence rules, we can compare precedence rules of different grammars, defined in different grammar formalism, and using different precedence declaration mechanisms. We have implemented support for recovering precedence rules from YACC [5] and SDF [4,15] (parser generators using different parsing algorithms) and present the details

of an algorithm to check precedence rules against LR parsers. Although it is undecidable whether two grammars define the same language, this tool provides support for comparing and recovering precedence rules, which is especially useful for reliable migration of a grammar from one grammar formalism to another. Also, the method can be used to analyze the precedence rules of a language, for example to determine if they can be defined using a certain grammar formalism specific precedence declaration mechanism. We evaluate our method by the application to the non-trivial mainstream programming languages C and PHP. For both languages we compare the precedence rules of three grammars defined in SDF or YACC. The evaluation was most successful and revealed several differences and bugs in the precedence rules of the grammars. The YACC and SDF implementations of the method that we present are implemented in Stratego/XT [16] and available as open source software as part of the Stratego/XT Grammar Engineering Tools [4].

Contributions. The contributions of this paper are: *(1)* A core formalism for precedence rules. *(2)* A novel method for recovering precedence rules from grammars *(3)* A method for checking the compatibility of precedence rules across grammars *(4)* Implementations of the recovery method for YACC and SDF and an evaluation for non-trivial programming languages C and PHP.

Organization. In Section 2 we introduce notations for context-free grammars and tree patterns. In Section 3 we introduce a running example and explain the precedence mechanisms of YACC and SDF. Section 4 is the body of the paper, where we present our precedence rule recovery method. Section 5 discusses compatibility checking. In Section 6 we present our evaluation, and we conclude with a discussion of related work.

2 Grammars and Tree Patterns

In this section we define the notions and notations for context-free grammars and tree patterns as we will use them in this paper.

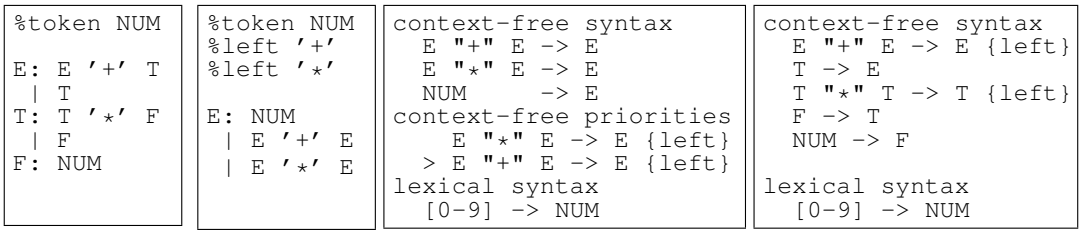
A **context-free grammar** G is a tuple (Σ, N, P), with Σ a set of terminals, N a set of non-terminals, and P a set of productions of the form $A \rightarrow \alpha$, where we use the following notation: V for $N \cup \Sigma$; A, B, C for variables ranging over N; X, Y, Z for variables ranging over V; a, b for variables ranging over Σ; v, w, x for variables ranging over Σ^*; and α, β, γ for variables ranging over V^*. Context-free grammars are usually written in some concrete grammar formalism. Figure 1 gives examples of grammars for the same language in different grammar formalisms. The underlying structure is that of context-free grammars just defined. The augmentation of grammars with precedence mechanisms will be discussed in the next section.

The family of valid **parse trees** T_G over a grammar G is a mapping from V to a set of trees, and is defined inductively as follows:

- if a is a terminal symbol, then $a \in T_G(a)$
- if $A_0 \rightarrow X_1 ... X_n$ is a production in G, and $t_i \in T_G(X_i)$ for $1 \leq i \leq n$, then $\langle A_0 \rightarrow t_1 ... t_n \rangle \in T_G(A_0)$.

For example, the tree $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow \langle F \rightarrow \mathit{NUM} \rangle \rangle \rangle + \langle T \rightarrow \langle F \rightarrow \mathit{NUM} \rangle \rangle \rangle$ is a parse tree for the addition of two numbers according to the left-most grammar in Figure 1.

[4] http://www.strategoxt.org/GrammarEngineeringTools



```
%token NUM

E: E '+' T
 | T
T: T '*' F
 | F
F: NUM
```

```
%token NUM
%left '+'
%left '*'

E: NUM
 | E '+' E
 | E '*' E
```

```
context-free syntax
  E "+" E -> E
  E "*" E -> E
  NUM     -> E
context-free priorities
    E "*" E -> E {left}
  > E "+" E -> E {left}
lexical syntax
  [0-9] -> NUM
```

```
context-free syntax
  E "+" E -> E {left}
  T -> E
  T "*" T -> T {left}
  F -> T
  NUM -> F

lexical syntax
  [0-9] -> NUM
```

Fig. 1. Grammars for a small arithmetic expressions language. Left to right: YACC using encoded precedence ($YACC_1$), YACC using precedence declarations ($YACC_2$), SDF using precedence declarations (SDF_1), SDF using a mixture of encoding and precedence declarations (SDF_2).

The family TP_G of **parse tree patterns** (or *tree patterns* for short) over a grammar G, is a mapping from grammar symbols in V to sets of parse trees over G extended with non-terminals as trees, which we define inductively as follows:

- if X is a terminal or non-terminal symbol in V, then $X \in TP_G(X)$
- if $A_0 \rightarrow X_1 ... X_n$ is a production in G, and $t_i \in TP_G(X_i)$ for $1 \leq i \leq n$, then $\langle A_0 \rightarrow t_1 ... t_n \rangle \in TP_G(A_0)$.

A parse tree pattern p denotes a *set* of parse trees, namely the set obtained by replacing each non-terminal A in p by the elements of $T_G(A)$. Basically, a parse tree pattern corresponds to the derivation tree for a sentential form. For example, the tree pattern $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow F \rangle \rangle + T \rangle$ denotes the set of trees for summation expressions where the first summand is a 'factor'. We denote a tree pattern with root $A \in N$ and yield α by $\langle A \rightsquigarrow \alpha \rangle$

We use the notation $\langle A \sim B \rightarrow t^* \rangle$ to denote an **injection chain** from a tree pattern with root A to a node with non-terminal B and leaves t^*. Formally, $\langle A \sim B \rightarrow t^* \rangle$ is the subset of $TP_G(A)$ such that

- if $A \rightarrow B$ is a production in G, and $\langle B \rightarrow t^* \rangle \in TP_G(B)$, then $\langle A \rightarrow \langle B \rightarrow t^* \rangle \rangle \in \langle A \sim B \rightarrow t^* \rangle$
- if $A \rightarrow C$ is a production in G, and $\langle C \sim B \rightarrow t^* \rangle \in TP_G(C)$, then $\langle A \rightarrow \langle C \sim B \rightarrow t^* \rangle \rangle \in \langle A \sim B \rightarrow t^* \rangle$

For example, the expression $\langle E \rightarrow \langle E \sim F \rightarrow \mathit{NUM} \rangle + T \rangle$ abbreviates the injection chain in the tree pattern $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow \langle F \rightarrow \mathit{NUM} \rangle \rangle \rangle + T \rangle$.

Finally, to define the notion of precedence, we will need **one-level tree patterns**, which we define as follows:

- if $A \rightarrow \alpha B \gamma$ and $B \rightarrow \beta$ are productions, then $\langle A \rightarrow \alpha \langle B \rightarrow \beta \rangle \gamma \rangle \in TP^1_G(A)$
- if $A \rightarrow \alpha B \gamma$ is a production and $\langle B \sim C \rightarrow \beta \rangle \in TP_G(B)$ then $\langle A \rightarrow \alpha \langle B \sim C \rightarrow \beta \rangle \gamma \rangle \in TP^1_G(A)$

That is, one-level tree patterns are productions with a single subtree, with possibly an injection chain from the root production to the child production. Observe that $TP^1_G(A) \subseteq TP_G(A)$. The tree pattern $\langle E \rightarrow E + \langle T \rightarrow T * F \rangle \rangle$ is one-level, and so is $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow T * F \rangle \rangle + T \rangle$. However, $\langle E \rightarrow \langle E \rightarrow \langle T \rightarrow T * F \rangle \rangle + \langle T \rightarrow T * F \rangle \rangle$ is not a one-level tree pattern, since it has two non-chain subtrees.

3 Precedence Mechanisms

In this paper we focus on two grammar formalisms, their parser generators, and their precedence mechanisms. The first is YACC (Yet Another Compiler-Compiler) and the second is SDF (Syntax Definition Formalism). The parser targeted by the SDF parser generator has a different name: SGLR (Scannerless Generalized-LR). Considering the combination of SDF and YACC is interesting for three reasons. First, the two grammar formalisms provide very different precedence declaration mechanisms. Second, the grammar formalisms are implemented using different parsing techniques. Third, the conversion of YACC to SDF is a very common use case. We introduce the basics of the YACC and SDF precedence declaration mechanisms with a few grammars for a small arithmetic language, see Figure 1.

YACC [5] is *the* classic parser generator. It accepts grammars of the LALR(1) class of context-free grammars with optionally additional disambiguation rules. For our YACC-based tools we use Bison, the GNU version of YACC, however, we will refer to our use of Bison as YACC (on most systems `yacc` is actually an alias of `bison`). The first and the second grammar of Figure 1 are YACC grammars. The first grammar encodes the precedence rules of the arithmetic language in the productions of the grammar. The operators + and $\star$ are left-associative, since the grammar does not allow an occurrence of + at the right-hand side of a +. The operator $\star$ takes precedence over the operator +, since it is not possible at all to have a + at left or right-hand side of a $\star$. The second grammar uses separate YACC precedence declarations [1]. Without disambiguation rules (and implicit conflict resultion), this grammar is ambiguous, e.g `1 + 2 * 3` has two different parse trees: $\langle E \rightarrow \langle E \rightarrow \langle E \rightarrow 1\rangle + \langle E \rightarrow 2\rangle\rangle * \langle E \rightarrow 3\rangle\rangle$ and $\langle E \rightarrow \langle E \rightarrow 1\rangle + \langle E \rightarrow \langle E \rightarrow 2\rangle * \langle E \rightarrow 3\rangle\rangle\rangle$ are both elements of $T_G(E)$.

As disambiguation rules, YACC allows declarations of the precedence of operators, which can be `%left`, `%right`, or `%nonassoc`. After the associativity comes a list of tokens. All tokens on the same line have the same precedence. The relative precedence of the operators is defined by the order of the precedence declarations. The operators in the first precedence declaration have lower precedence than the next. The semantics of the precedence declarations of YACC are defined in terms of parser generation. YACC produces an LALR parse table in which the action has to be deterministic for each state and lookahead. If there are multiple possible actions, then this results in shift/reduce or reduce/reduce conflicts. The precedence declarations are used by YACC to select the appropriate action if there is a conflict between two actions. If there is no precedence declaration for the involved tokens, then YACC will resolve the conflict by preferring a shift over a reduce. For a reduce/reduce conflict, YACC resolves the conflict by selecting the reduce of the first production in the input grammar. Later we will see in more detail what the consequence of this is for the precedence rules.

The main weakness of precedence declarations of YACC is that it is not really a precedence declaration mechanism, i.e. YACC has no notion of precedence of operators. Precedence declarations are a mechanism to resolve conflicts in the parse table, which can be used to *implement* operator precedence. Unfortunately, this requires understanding of LALR parsing and the way YACC generates a parser.

SDF [4,15] is a feature rich grammar formalism that integrates lexical and context-free syntax. SDF supports arbitrary context-free grammars, so grammars are not restricted to subclasses of context-free grammar, such as LL or LALR. The SDF parser generator

generates a parse table for a scannerless generalized-LR parser. For disambiguation, SDF supports various disambiguation filters [15,14], some of which are used to define precedence rules. The third grammar of Figure 1 uses the precedence declarations of SDF[5]. Similar to the second YACC grammar, the productions of this grammar define an ambiguous language. A separate definition of *priorities* is used to define that $\star$ takes precedence over +. Also, both operators are defined to be left associative by using the associativity attribute `left`.

The semantic of SDF priorities is well-defined in terms of the grammar, as opposed to operationally in the parser generator. SDF applies the transitive closure to the declared priority relation over productions (which introduces some limitations). Priority declarations generate a set $\mathit{conflicts}(\mathcal{G})$ of tree patterns of the form $\langle A \rightarrow \alpha\langle B \rightarrow \beta\rangle\gamma\rangle$. Note that this pattern has the same form as patterns from the set of one-level tree patterns, excluding injection chains. If $A \rightarrow \beta B\gamma > B \rightarrow \beta$ is in the closure of the priority relation, then $\langle A \rightarrow \alpha\langle B \rightarrow \beta\rangle\gamma\rangle \in \mathit{conflicts}(\mathcal{G})$. The generated parser will never create a parse tree that matches one of the tree patterns in $\mathit{conflicts}(\mathcal{G})$.

The fourth grammar of Figure 1 illustrates that encoding precedence in productions is possible in all grammar formalisms, even if they provide separate precedence declarations. To make the example a bit more interesting, this grammar defines the priority of operators in productions, but uses associativity definitions for individual operators.

4 Precedence Rule Recovery

In previous sections, we have argued that there is a need for methods and tools for determining the precedence rules of a grammar. In this section, we present such a method for recovering the precedence rules as encoded in productions or defined using separate precedence declarations.

A Core Formalism for Precedence Rules. The recovered precedence rules need to be expressed in a certain formalism. To liberate the precedence rules from the idiosyncrasies of specific grammar formalisms, we need a formalization that is independent of specific parsing techniques. The formalism for precedence rules does not need to be concise or notationally convenient. Rather, it serves as a core representation of precedence rules of programming languages.

Inspired by previous work on SDF conflict sets defined by priorities [4,15], we use parse tree patterns to define precedence rules. Parse tree patterns denote a set of parse trees. Thus, a parse tree pattern can be used to define a set of invalid parse trees. For example, for the grammar SDF_1 in Figure 1 the tree pattern $\langle E \rightarrow \langle E \rightarrow E + E\rangle \star E\rangle$ denotes a set of invalid parse trees according to the precedence rules of this grammar. However, the precedence rules for a grammar $\mathcal{G}$ can not just be defined as a subset of $\mathit{TP}_{\mathcal{G}}$. The reason for this is that for grammars that encode precedence in productions, there will be no tree patterns that denote invalid parse trees. Such grammars have a series of expression non-terminals that are only allowed at specific places. For example, in grammar YACC_1 of Figure 1, the expression E is not allowed at the right-hand side of the operator $+$ in the production $E \rightarrow E + T$. Nevertheless, we are interested in precedence rules over such

[5] SDF uses a reversed notation for production rules. We will only use this notation in verbatim examples of SDF. All other productions are written in conventional $A \rightarrow \alpha$ notation.

$\langle T \to \langle T \sim E \to E + T\rangle \star F\rangle$ $\langle T \to T \star \langle F \sim T \to T \star F\rangle\rangle$ $\langle T \to T \star \langle F \sim E \to E + T\rangle\rangle$ $\langle E \to E + \langle T \sim E \to E + T\rangle\rangle$	$\langle E \to \langle E \to E + E\rangle \star E\rangle$ $\langle E \to E \star \langle E \to E \star E\rangle\rangle$ $\langle E \to E \star \langle E \to E + E\rangle\rangle$ $\langle E \to E + \langle E \to E + E\rangle\rangle$
$\langle E \to \langle E \to E + E\rangle \star E\rangle$ $\langle E \to E \star \langle E \to E \star E\rangle\rangle$ $\langle E \to E \star \langle E \to E + E\rangle\rangle$ $\langle E \to E + \langle E \to E + E\rangle\rangle$	$\langle T \to \langle T \sim E \to E+ E\rangle \star T\rangle$ $\langle T \to T \star \langle T \to T \star T\rangle\rangle$ $\langle T \to T \star \langle T \sim E \to E +E\rangle\rangle$ $\langle E \to E + \langle E \to E + E\rangle\rangle$

Fig. 2. Precedence rules for grammar of Figure 1. First row: $YACC_1$, $YACC_2$, second row: SDF_1, SDF_2

grammars. Therefore, we define the set of precedence rules for $\mathcal{G} = (\Sigma, N, P)$ to be a subset of $TP_{\underrightarrow{\mathcal{G}}(N_E)}$, where $\underrightarrow{\mathcal{G}}(N_E)$ is an **extended context-free grammar** of the grammar $\mathcal{G}$ where $N_E \subseteq N$ and $\underrightarrow{\mathcal{G}}(N_E) = (\Sigma, N, P')$ where $P' = P \cup \{A \to B | A \in N_E, B \in N_E, A \neq B\}$. For example, for the grammar $YACC_1$ in Figure 1, $\underrightarrow{YACC_1}(\{E, T, F\})$ contains the injections $E \to F, T \to E, F \to E$, and $F \to T$ in addition to the productions of $YACC_1$.

Based on this definition we can now introduce the precedence rules for the grammars of Figure 1 that are presented in Figure 2. First, note that an injection chain $\langle A \to \alpha\langle B \sim C \to \beta\rangle\gamma\rangle$ is used when the symbol C of the nested production is not equal to the symbol B at the place where the nested production is used. Second, note that for the grammar $YACC_1$ the tree pattern $\langle T \sim E \to E + T\rangle$ is not actually valid. This is exactly where $\underrightarrow{YACC_1}$ comes in, since the injection $T \to E$ is present in $\underrightarrow{YACC_1}$.

There is no relation defined between the tree patterns that are members of the precedence rule set, e.g we do not take the transitive closure of a precedence relation over productions. If a precedence declaration for operators needs to be transitively closed for a language, then this should be expressed by having all combinations in the set. A precedence rule set is not by definition required to be minimal. This means that some tree patterns can define precedence rules that are already implied by other tree patterns.

Precedence rules defined by tree patterns are closely related to the set of conflicts $conflicts(\mathcal{G})$ defined by SDF priority and associativity declarations. One important difference is that the set of conflicts of SDF is transitively closed, since it is defined by a priority relation that is a strict partial ordering between productions. Another difference is that we do not restrict the tree patterns used in the precedence rule sets to trees of two productions. As mentioned before, we do not assume anything about (the feasibility of) a concise notation for the set of tree patterns.

Tree Pattern Generation. We recover precedence rules from grammars by generating a set of tree patterns involving expression productions and checking if a parse is possible that will result in a parse tree matching the tree patterns. By default, we generate the set of one-level tree patterns $TP^1_{\underrightarrow{\mathcal{G}}}(N_E)$ for a grammar $\mathcal{G}$ with P restricted to $P = \{A \to \alpha \in P | A \in N_E\}$, i.e a set of tree patterns involving two productions for all combinations of expression productions. For example, the set of one-level tree patterns for two productions $E \to E + E$ and $E \to \& E$ is $\langle E \to \& \langle E \to \& E\rangle\rangle$, $\langle E \to \langle E \to \& E\rangle + E\rangle$, $\langle E \to \langle E \to E + E\rangle + E\rangle$, $\langle E \to \& \langle E \to E + E\rangle\rangle$, $\langle E \to E + \langle E \to \& E\rangle\rangle$, $\langle E \to E + \langle E \to E + E\rangle\rangle$. One-level tree patterns are sufficient to express the precedence rules of most operator languages. Indeed, our case studies

in Section 6 are based on one-level tree patterns. However, some languages require precedence rules that include 3 or more productions. For this, the precedence recovery tool supports configuration of the number of levels that is to be generated.

Next, the question is how to check if a grammar allows a parse that matches a tree pattern. If the pattern is accepted, then there are valid parse trees for this pattern. If not, then it denotes invalid parse trees and it will be an element of the resulting precedence rule set. Clearly, checking tree patterns is parser generator specific, since we need intimate knowledge about the semantics of the grammar formalism that is used by the parser generator. Based on the requirements for our case studies and our practical needs, we implemented the validation of tree patterns for YACC and SDF. However, the algorithm and the approach that is used can easily be ported to different (Generalized) LR parser generators.

Precedence Rule Recovery: YACC. For YACC, the precedence rules are difficult to determine from the grammar directly, since the semantics of precedence declarations in YACC is defined operationally. The precedence declarations are used to resolve conflicts during parser generation, which means that precedence rules are only applied if there is actually a conflict. Also, YACC applies implicit conflict resolution mechanisms, i.e. preference for a shift over a reduce, and preference for a reduce of the first production in the grammar. Furthermore, grammars can encode the precedence rules in productions and combine this with precedence declarations, an issue that is not YACC specific. Hence, checking the *grammar* for possible matches of tree patterns is complex and requires intimate knowledge of YACC parser generation and conflict resolution. A much more general solution is to validate tree patterns against the *parse table* generated by YACC. Of course, a parser generated by YACC can not parse tree patterns. To check if a tree pattern is valid, we simulate the parsing of a sentential form that results in a parse tree matching the tree pattern. If this is possible, then the tree pattern is valid, otherwise it is invalid.

A shift reduce parser is a transition system with as configuration a stack and an input string. The configuration is changed by shift and reduce actions. A shift action moves a symbol from the input to the stack, which corresponds to a step of one symbol in the right-hand sides of a set of productions that is currently expected. A reduce removes a number of elements from the stack and replaces them by a single element, which corresponds to the application of a grammar production. In an LR parser [7], the information on the actions to perform is stored in an action table. Both a shift and a reduce introduce state transitions, which are recorded on the stack and are based on information in the action and goto table. After popping elements from the stack in a reduce, the goto entries of the state on top of the stack are consulted to find the new state to push on the stack.

To recognize tree patterns, we change the input of the LR parser to a string of tree patterns and symbols. The tree patterns are translated into LR actions and all changes in the configuration of the parser are checked against the actions that are allowed to derive a parse tree that matches the tree pattern. Figure 3 lists the transition rules that implement the modified LR parser for recognizing tree patterns. The configuration of the parser, denoted by $\mid stack \mid input \mid$, is rewritten by the transition rules. The stack grows to the right, the input grows to the left. The variable e ranges over all possible input symbols, which is the set $N \cup \Sigma \cup \mathit{TP}_{\underrightarrow{\mathcal{G}}}(N_E) \cup \mathcal{R}(P) \cup \overrightarrow{\mathcal{R}}(N)$. Hence, the input of the parser consists of a sequence of non-terminals, terminals, tree patterns, and two special elements for representing reduces. $\mathcal{R}(A \rightarrow \alpha)$ represents a reduction of the production $A \rightarrow \alpha$. $\overrightarrow{\mathcal{R}}(A)$ represents a reduction of any chain production $B \rightarrow C$, until B is A. The function $head$ finds the first

$$\frac{action(s_m, a) = \texttt{shift}(s_{m+1})}{\mid s_0 \dots s_m \mid a, e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m, s_{m+1} \mid e_i \dots e_n \mid} \quad (1)$$

$$\frac{}{\mid s_0 \dots s_m \mid \langle A \to \alpha \rangle \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid \alpha, \mathcal{R}(A \to \alpha) \dots e_n \mid} \quad (2)$$

$$\frac{}{\mid s_0 \dots s_m \mid \langle A \sim B \to \alpha \rangle \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid \langle B \to \alpha \rangle, \overrightarrow{\mathcal{R}}(A) \dots e_n \mid} \quad (3)$$

$$\frac{goto(s_m, A) = s_{m+1}}{\mid s_0 \dots s_m \mid A, e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m, s_{m+1} \mid e_i \dots e_n \mid} \quad (4)$$

$$\frac{action(s_{m+k}, head(e_i \dots e_n)) = \texttt{reduce}(A \to X_1 \dots X_k)}{\mid s_0 \dots s_m \dots s_{m+k} \mid \mathcal{R}(A \to X_1 \dots X_k), e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid A, e_i \dots e_n \mid} \quad (5)$$

$$\frac{action(s_{m+1}, head(e_i \dots e_n)) = \texttt{reduce}(A \to B)}{\mid s_0 \dots s_m, s_{m+1} \mid \overrightarrow{\mathcal{R}}(A), e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid A, e_i \dots e_n \mid} \quad (6)$$

$$\frac{action(s_{m+1}, head(e_i \dots e_n)) = \texttt{reduce}(B \to C), B \neq A}{\mid s_0 \dots s_m, s_{m+1} \mid \overrightarrow{\mathcal{R}}(A), e_i \dots e_n \mid \Rightarrow \mid s_0 \dots s_m \mid B, \overrightarrow{\mathcal{R}}(A), e_i \dots e_n \mid} \quad (7)$$

Fig. 3. Transition rules for checking tree patterns for a YACC parser

non-reduce element in its list of arguments.

Equation 1 defines a shift of a terminal a. This definition is not different from a shift in a normal LR parser. A terminal is removed from the input and a new state is pushed on the stack. Equation 2 defines the unfolding of a tree pattern $\langle A \to \alpha \rangle$. This transition rule does not exist for an LR parser, since the normal input is a sequence of terminals. The unfolding of a tree pattern involves adding α and a reduce of $\langle A \to \alpha \rangle$ to the input. The reduction is denoted by $\mathcal{R}(A \to \alpha)$. Equation 3 defines the unfolding of a tree pattern $\langle A \sim B \to \alpha \rangle$. After the unfolding of $\langle B \to \alpha \rangle$, a reduce $\overrightarrow{\mathcal{R}}(A)$ for arbitrary chain productions is inserted. Thanks to the unfolding of productions, the input of the system can now contain non-terminals. This is the reason for a separate transition rule 4 for performing a goto, which is usually considered to be a part of the reduce action. The goto transition rule removes a non-terminal from the input and pushes a new state on the stack, determined by the *goto* function. The reason why this works is that we can assume that the non-terminal A is productive, which means that there will always be a production for A that will finally reduce to state s_m, which would lead to exactly the same goto.

Equation 5 defines a reduce action. The transition system only allows reduces if a reduce is explicitly identified in the input. This method of *checked reduces* is used to enforce the structure of the tree pattern on the parser, i.e. it is not possible to recognize the leafs of the tree pattern with a parse tree that has a different internal structure. The definition of the reduce action reuses the separate transition rule of *goto* by inserting a non-terminal in front of the list. The equations 6 and 7 define the reduction of chain productions, which is allowed if there is an $\overrightarrow{\mathcal{R}}(A)$ in front of the input. If the reduce is applied for $A \to B$ then $\overrightarrow{\mathcal{R}}(A)$ is removed from the input and A is added. If the chain production does not produce A, then more chain productions might be necessary. Therefore, the $\overrightarrow{\mathcal{R}}(A)$ is preserved and B is pushed in front of the input to trigger a goto.

Using the extended LR parser that operates on tree patterns, the parsing of an actual input of the form of a tree pattern is simulated in detail. To illustrate the validation of

unfold $\mid 0 \mid \langle E \rightarrow E + \langle E \rightarrow E \star E \rangle \rangle \mid$	
goto $\mid 0 \mid E,\ +,\ \langle E \rightarrow E \star E \rangle,\ \mathcal{R}(+) \mid$	
shift $\mid 0,3 \mid +,\ \langle E \rightarrow E \star E \rangle,\ \mathcal{R}(+) \mid$	
unfold $\mid 0,3,5 \mid \langle E \rightarrow E \star E \rangle,\ \mathcal{R}(+) \mid$	unfold $\mid 0 \mid \langle E \rightarrow \langle E \rightarrow E + E \rangle \star E \rangle \mid$
goto $\mid 0,3,5 \mid E,\star,\ E,\ \mathcal{R}(\star),\ \mathcal{R}(+) \mid$	unfold $\mid 0 \mid \langle E \rightarrow E + E \rangle,\ \star,\ E,\ \mathcal{R}(\star) \mid$
shift $\mid 0,3,5,7 \mid \star,\ E,\ \mathcal{R}(\star),\ \mathcal{R}(+) \mid$	goto $\mid 0 \mid E,+,\ E,\ \mathcal{R}(+),\star,\ E,\ \mathcal{R}(\star) \mid$
goto $\mid 0,3,5,7,6 \mid E,\ \mathcal{R}(\star),\ \mathcal{R}(+) \mid$	shift $\mid 0,3 \mid +,\ E,\ \mathcal{R}(+),\star,\ E,\ \mathcal{R}(\star) \mid$
reduce $\mid 0,3,5,7,6,8 \mid \mathcal{R}(\star),\ \mathcal{R}(+) \mid$	goto $\mid 0,3,5 \mid E,\ \mathcal{R}(+),\ \star,\ E,\ \mathcal{R}(\star) \mid$
goto $\mid 0,3,5 \mid E,\ \mathcal{R}(+) \mid$	error $\mid 0,3,5,7 \mid \mathcal{R}(+),\ \star,\ E,\ \mathcal{R}(\star) \mid$
reduce $\mid 0,3,5,7 \mid \mathcal{R}(+) \mid$	
goto $\mid 0 \mid E \mid$	
accept $\mid 3,0 \mid \ \mid$	

Fig. 4. LR configuration sequences for a valid and invalid tree pattern

tree patterns, Figure 4 shows the configuration of a parser generated from grammar $YACC_2$ (Figure 1) for every application of a transition rule. The $\mathcal{R}(\star)$ and $\mathcal{R}(+)$ inputs are abbreviations for the complete productions of these operators. The tree pattern on the left is valid. The tree pattern on the right is invalid, since in the last configuration the lookahead is the terminal $\star$. For this lookahead, a reduce of the + operator is not allowed, since that would give the + operator precedence over $\star$. Thus, parsing fails and the tree pattern is invalid.

By working on the parse table generated by YACC, the recovery supports all precedence rules of a YACC grammar: encoded in productions, defined using precedence declarations, and even implicit conflict resolution. Indeed, if we remove the precedence declarations from $YACC_2$, then the precedence rule recovery returns $\langle E \rightarrow \langle E \rightarrow E \star E \rangle \star E \rangle$, $\langle E \rightarrow \langle E \rightarrow E + E \rangle \star E \rangle$, $\langle E \rightarrow \langle E \rightarrow E \star E \rangle + E \rangle$, $\langle E \rightarrow \langle E \rightarrow E + E \rangle + E \rangle$, which illustrates that YACC prefers a shift over a reduce.

Bison has a detailed report function that provides information about the generated LR parse table, item sets, shifts, gotos, reduces, and conflicts. We parse this output to get a representation of the parse table. The tree pattern parser is implemented in Stratego. The transition rules of Figure 3 directly correspond to rewrite rules in the Stratego implementation, which are applied using a rewriting strategy. The configurations of the parser can be inspected, which was used to produce the examples of configuration sequences of Figure 4. The implementation of the transition system takes 55 lines of code.

Precedence Rule Recovery: SDF. For recovering precedence rules from SDF grammars, an analysis of the grammar would be feasible, since the precedence declarations of SDF are not operationally defined in terms of parser generation. Yet, supporting a mixture of encoded and separately defined precedence declarations can still be rather involved. Based on the success of the approach that we used for recovering YACC precedence rules, we chose the same method for SDF grammars. Thus, precedence rules are recovered by checking generated tree patterns up to a certain level against the parse table generated from an SDF grammar.

We cannot reuse the transition system (a modified LR parser) that we defined for checking tree patterns against YACC parse tables, since SDF is implemented using a scannerless generalized-LR parser, called SGLR. Because the parser uses the generalized-LR algorithm, there will be cases where multiple actions are possible in some configuration, for example a shift as well as a reduce action. To handle the alternatives, the GLR configuration

needs to be forked, where in the end one of the alternatives has to succeed to make a tree pattern valid. Furthermore, the scannerless generalized-LR parser uses a different method for applying precedence declarations to the parse table. Whereas YACC uses precedence declarations to resolve conflicts between shift and reduce actions, SDF effectively prunes the goto table of a parse table. SGLR refines the goto table from gotos based on symbols to gotos based on productions, i.e the goto table is now a table of states and productions instead of states and symbols [15]. This slightly complicates the definition of the transition system, since the system applies gotos that are not introduced by a reduce, but by a non-terminal in the tree pattern. For this reason, we distinguish such a goto from a goto induced by a reduce. In the case of a goto caused by a non-terminal in the input, we consider all possible gotos for this non-terminal. We determine the set of possible states where the parser can goto from the current configuration and fork the GLR configuration to check all alternatives. Our method supports ambiguous grammars, which is illustrated by the case studies of Section 6, where two ambiguous grammars for C are compared.

The implementation of the precedence recovery tool for SDF is a very basic and somewhat naive GLR parser. However, for the size of tree patterns this is not an issue at all. Again, the checker is implemented in Stratego using rewrite rules that rewrite the GLR configuration. Due to space constraints we cannot present the transition system for the SDF implementation.

5 Precedence Compatibility

Comparing the language defined by two grammars is undecidable, but this does not mean that nothing can be said about the compatibility of two grammars. Static analysis tools, such as our precedence rule recovery tool, can be used to extract information from different grammars and compare the results, even if they are written in different grammar formalisms.

While the precedence rules are represented in a grammar formalism independent formalism, this does not imply that precedence rules can be compared directly in a useful way after recovering them from two different grammars. Grammars usually have different naming conventions, different names for lexical symbols, and often also have a different structure at some points. The recovered precedence rules can still be compared by first applying grammar transformations to the precedence rules to achieve a common representation. After this, the comparison of precedence rules is a simple set comparison.

Grammar Transformation. Precedence rule recovery usually results in rather big sets of tree patterns. Trying to transform this huge set of tree patterns to a common representation is usually not a good idea. To avoid working with this big set of precedences, it is usually a good idea to first extract the productions from the precedence rules and compare and transform the set productions in order to find the required set of grammar transformations that achieves a common representation. Also, this is the most convenient way to identify language extensions that are only present in one of the two grammars.

The relationship between two grammars is something that has to be custom defined for a particular combination of grammars. Typically, one of the grammar transformations that needs to be applied to the precedence rules is the renaming of all expression symbols to a single expression symbol. Note that it is essential that this renaming is applied to the precedence rules and not to the original grammar, since that would most likely change the

precedence rules of the language or even make it impossible to generate a parser.

Similar to the renaming of expression symbols, injections caused by the application of chain productions are no longer useful. To achieve a common representation, all injection chain nodes $\langle B \sim C \rightarrow \beta \rangle$ are transformed to $\langle C \rightarrow \beta \rangle$

In the comparison of a YACC grammar and an SDF grammar a common issue is that the YACC precedence rules use names for the operators of the language (e.g. `ANDAND` instead of `&&`). This is usually a straightforward renaming where the lexical specification can be consulted if necessary.

6 Evaluation

We have evaluated the method for precedence rule recovery and compatibility checking by applying the implementation for YACC and SDF to a set of grammars for the C and PHP languages. Both languages have a large number of operators and non-obvious precedence rules. The size and complexity of the languages makes this compatibility check a good benchmark for our method.

C99. We have compared three grammars for C99:

- The C compiler of the the GNU Compiler Collection uses a parser generated from a YACC grammar [6]. The YACC grammar uses a mixture of precedence declarations and encoding of priorities in productions.
- The Transformers project provides a C99 SDF grammar [2]. This grammar is a direct translation of the standard to SDF [7]. The grammar does not use SDF precedence declarations. Instead, it uses an encoding of precedence in productions as specified by the standard. The grammar is designed to be ambiguous where the C syntax is ambiguous.
- The SDF Library provides an ANSI C SDF grammar [8]. Unlike C-Transformers, this grammar uses SDF precedence declarations. The grammar is designed to be ambiguous.

The precedence tools reported various differences between the grammars. All the reports have been verified as being real differences, i.e there were no false positives. Examples of the reported differences are:

$\langle E \rightarrow$ **sizeof** $\langle E \rightarrow$ **(** *TypeName* **)** $E \rangle \rangle$
A cast as an argument of `sizeof` is forbidden in GCC and C-Transformers, which is correct, but it is allowed in the SDF Library, which is a bug.

$\langle E \rightarrow$ **++** $\langle E \rightarrow$ **(** *TypeName* **)** $E \rangle \rangle$ $\langle E \rightarrow$ **--** $\langle E \rightarrow$ **(** *TypeName* **)** $E \rangle \rangle$
GCC and SDF Library allow a cast as an argument of ++ and −−. The C-Transformers do not, which corresponds to the standard. The standard defines ++ and −− separate from unary operators, while GCC and the SDF Library ignore this difference.

$E \rightarrow$ **sizeof(***TypeName***)**
Though not a precedence problem, our tools reported this missing production in the SDF library grammar. This means that some `sizeof` expressions that should be parsed ambiguously are currently unambiguous.

[6] In GCC 4.1 the Bison-generated C parser has been replaced with a hand-written recursive-decent parser. We use the Bison grammar for GCC 4.03.

[7] We used revision 1611 of the transformers-c-tools package. The one bug we found has been fixed in revision 1613.

[8] We used revision 20649 of the sdf-library package for our evaluation.

$\langle E \rightarrow E \texttt{ ? } E \texttt{ : } \langle E \rightarrow E \texttt{ = } E \rangle \rangle$

This tree pattern of an assignment in the else branch of the conditional is forbidden in GCC and the SDF Library, but is allowed in C-Transformers. This is a bug in C-Transformers: the else branch of the conditional operator uses the wrong non-terminal

$\langle E \rightarrow \langle E \rightarrow E \texttt{ ? } E \texttt{ : } E \rangle \texttt{ = } E \rangle \quad \langle E \rightarrow \langle E \rightarrow \texttt{ (} \textit{TypeName} \texttt{) } E \rangle \texttt{ = } E \rangle$

A conditional or a cast in the left-hand side of an assignment is allowed by GCC and the SDF Library. For GCC this is a legacy feature that now produces a semantic error. C-Transformers forbids this, which is correct. The same issue holds for many more binary operators (`||`, `&&`, `|`, `^`, `&`, `!=`, `==`, `>=`, `<=`, `>`, `<`, `<<`, `>>`, `-`, `+`, `%`, `/`, `*`). The C standard only supports unary operators in the left-hand side of an assignment.

PHP 5. We compared three grammars for PHP:

- The official PHP distribution comes with a YACC grammar for PHP, as part of the Zend engine. The grammar makes heavy use of YACC precedence declarations [9].
- The open source PHP compiler PHC comes with a YACC grammar that has been forked from the PHP distribution [10].
- PHP-front provides a syntax definition for PHP 4.0 and 5.0 in SDF.

For PHP YACC versus PHC YACC the precedence tool reported several major bugs in the PHC YACC grammar: several operator precedences have been inverted since the fork of the grammar. For example, in PHC the `||`, `OR`, and `XOR` operators had precedence over respectively `&&`, `AND`, and `AND`. This issue was reported by our tools as a missing precedence rule $\langle E \rightarrow \langle E \rightarrow E \texttt{ || } E \rangle \texttt{ && } E \rangle$ in PHC. For each precedence rule in PHC that was not in PHP, there was a corresponding rule in PHC that was not in PHP. For example, the rule corresponding to the previous pattern is $\langle E \rightarrow E \texttt{ || } \langle E \rightarrow E \texttt{ && } E \rangle \rangle$.

For PHP versus the PHP-front SDF grammar we expected many differences in the precedence rules. We were already aware of various issues in the precedence of operators of the PHP-front grammar. Actually, the uncertainty about the exact precedence rules of PHP was the primary motivation to develop this method of precedence rule recovery. One of the questions that we want to answer in this project is if the PHP precedence rules can actually be expressed in SDF. The PHP operators are a bit unusual since PHP has very weak as well as very strong binding unary operators. The transitive closure of priorities in SDF results in various cases where we could not find a solution by hand. In future work, we plan to analyse precedence rule sets to extract characteristics and hopefully determine automatically if these precedence rules can be expressed using grammar formalism specific precedence declaration mechanisms, in this case SDF priorities.

7 Related Work

Grammar Engineering Vision. Several researchers have suggested that there is a strong need for proper foundations and practices for grammar engineering [11,6,13,9]. In particular, [6] presents an extensive research agenda for grammar engineering. Our method for

[9] We used PHP 5.2.0 for our evaluation.

[10] We used PHC 0.1.7 for our evaluation. All bugs have been fixed by the developers of PHC after our reports.

recovery and compatibility checking of precedence rules is highly related to several of the presented research challenges, such as maintaining consistency between the incarnations of conceptually the same grammar. Also, our precedence rules help to abstract from the idiosyncratic precedence mechanisms provided by the various parser generators in use. Our precedence rule recovery method is very useful in the semi-automatic grammar recovery process [11] from language references and existing compilers. In particular, more automation of grammar recovery is now possible, since precedence declarations can be checked during the life-time of a grammar.

Grammar Engineering Tools. The Grammar Deployment Kit (GDK) [8] targets the process of producing a working parser from a specification. An important goal is parser generator independence. The GDK provides tools to generate parser generator specific grammars from a universal grammar formalisms, called LLL. The GDK does not provide more advanced grammar analysis tools, such as our precedence recovery tool. Parser generator independence can be increased by our representation of precedence rules, for which there is no comparable concept available in the GDK. Sellink and Verhoef [13] present the vision and implementation of a set of tools for grammar reengineering, such as assessment (metrics) and conversion tools. BNF2SDF automatically improves the resulting grammar by using EBNF list notations, but does not consider precedence rules. Early versions of Stratego/XT provided a similar tool `yacc2sdf` [3]. Our precedence recovery for YACC should be integrated in such a tool. Lämmel [9] discusses a formal approach to grammar transformation based on a concise set of primitives and combinators for refactoring, extension, and restriction of grammars. This work does not consider grammars that use separate disambiguation mechanisms. Also, the authors define some equivalency notions for grammars. Comparing precedence rules is a very restricted form of structural equivalence, which is much easier than comparing grammars in general. Schatborn [12] presents a feature rich grammar transformation language for SDF, providing modules, functions, variables, types, and patterns to facilitate the development of grammar independent grammar transformations, based on a detailed case study of the transformation from ANSI C from YACC to SDF. Advanced grammar analysis tools, such as our precedence recovery, would be valuable in combination with such a language.

Grammar Testing. Lämmel [10] contributes grammar coverage analysis techniques, combined with test set generation, applied to grammar recovery. Checking the correct implementation of precedences could be implemented using test set generation accompanied by code coverage requirements. Our method just exercises the parsing of operators using sentential forms, ignoring the actual values of the expression. Also, tests need a description of the expected result, which is usually parser specific (not just parser *generator* specific, like our method). Our method does not actually run the parser, which makes it easier in practice to test expressions in isolation. In this way, we also have very precise control over the correct behaviour of the parser, which makes a comparison to the result of the parser unnecessary.

8 Conclusion

We have presented a method for recovering precedence rules from grammars. We have presented the algorithm for YACC and implemented the method in tools for YACC and SDF. As far as we know, this is the first effort to develop methods and tools for reliably

assisting grammar developers with the recovery of precedence rules, migration of grammars with precedence rules, and compatibility checking of grammars. Although there are many open issues and opportunities for further research, the evaluation of our current prototypes has already clearly demonstrated the value of the tools that we have presented.

Acknowledgments. This research was supported by the NWO/JACQUARD project *TraCE: Transparent Configuration Environments* (638.001.201). The development of the PHP-front SDF grammar was sponsored by the *Google Summer of Code 2006*. We thank Mark van den Brand, Giorgios Robert Economopoulos, Jurgen Vinju, and the rest of the SDF/SGLR team at CWI for their work on SDF. We thank Valentin David and the Transformers team at the EPITA Research & Development Laboratory for their work on the SDF grammar for C99. We thank the anonymous reviewers of LDTA 2007 for providing useful feedback on an earlier version of this paper.

References

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

[2] A. Borghi, V. David, and A. Demaille. C-transformers: a framework to write c program transformations. *Crossroads*, 12(3):3–3, 2006.

[3] M. de Jonge and R. Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings: International Conference on Software Maintenance (ICSM 2001)*, pages 240–249, Los Alamitos, CA, USA, Nov. 2001. IEEE Computer Society.

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

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

[6] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[7] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[8] J. Kort, R. Lämmel, and C. Verhoef. The grammar deployment kit. In M. van den Brand and R. Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[9] R. Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[10] R. Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[11] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[12] E. P. Schatborn. GTL, a grammar transformation language for SDF specifications. Master's thesis, Programming Research Group, Faculty of Science, University of Amsterdam, Amsterdam, September 2005.

[13] M. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000.

[14] M. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

[15] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[16] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

TUD-SERG-2007-004
ISSN 1872-5392