Parsing in a Broad Sense

Vadim Zaytsev[1] and Anya Helene Bagge[2]

[1] Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net
[2] Universitetet i Bergen, Norway, anya@ii.uib.no

Abstract. Having multiple representations of the same instance is common in software language engineering: models can be visualised as graphs, edited as text, serialised as XML. When mappings between such representations are considered, terms "parsing" and "unparsing" are often used with incompatible meanings and varying sets of underlying assumptions. We investigate 12 classes of artefacts found in software language processing, present a case study demonstrating their implementations and state-of-the-art mappings among them, and systematically explore the technical research space of bidirectional mappings to build on top of the existing body of work and discover as of yet unused relationships.

Keywords: parsing, unparsing, pretty-printing, model synchronisation, technical space bridging, bidirectional model transformation

1 Introduction

Parsing is a well established research field [1] — in fact, its maturity has already become its own enemy: new results are specialised refinements published at a handful of venues with a critical mass of experts to appreciate them. Unparsing is a less active field, there are no books on *unparsing techniques* and there is no general terminological agreement (printing, pretty-printing, unparsing, formatting), but this family of mappings has nevertheless been studied well [27,13,41,47,29,11,2,4,18,33]. Parsing research concerns recognising grammatically formed sentences, providing error-correcting feedback, constructing graph-based representations, as well as optimising such algorithms on time, memory and lookahead. Research questions in the domain of unparsing include conservatism (in BX related to hippocraticness [44] and resourcefulness [8]), metalanguage completeness and designing a universal way to specify pretty-printers. It was suggested that unparsing in a broad sense should comprise advanced techniques like syntax highlighting [51, §3.8.3] and adjustment to screen width [33, §2.1]. Methods have been proposed to infer parsers from unparsers [33], unparsers from parsers [10,23] or both from annotated syntactic definitions [9,39,31]. Our attempt is about *modelling parsing and unparsing together as bidirectional transformation*, essentially by breaking up big step parsing into smaller steps for the sake of bidirectionalisation [52] and analysis. There are no general BX frameworks [44] to give a solution. This modelling approach can provide useful insights in bridging the gap between structural editors and text-based IDEs, which is known to be one of the open problems of software language engineering [5].

2 Motivation

Bidirectional transformations are traditionally [44] useful for situations when we need consistency restoration among two or more entities that share information in such a way that if one is updated, the other(s) need to coevolve. For example, a language instance (the term we use to avoid committing to "model", "program", "table", etc) can be expected to be defined in a specific way (i.e., to belong to an algebraic data type), but its concrete implementation can take a form of a textual file, or a graph, or an XML tree — one can think of scenarios when we would like to freely choose which instance to update so that the rest get co-updated automatically. (This can be seen correponding to the classic PIM/PSM distinction in MDA or to Ast/Cst from the next sections).

When two instances represent the same information in different ways, the mapping between them is bijective, and the solution is trivial. However, it is often the case that each kind of an instance is only a "view" on the complete picture. In the database technical space, this complete picture is formally defined, and we know exactly what the view is lacking. This allows an asymmetrical view/update or get/putback solution with one function taking just one argument and replacing its target with a new result, and one function taking two arguments and updating one of them [25,8,17]. However, in software language engineering we often end up having bits of specific locally significant information scattered among many artefacts. For example, a text of a program (Str) can contain indentation preferred by programmers, while the graphical model of the same instance (Dra) can contain colours and coordinates of its visualisation, and the abstract internal representation (Ast) can have precalculated metrics values cached in its nodes. Solving the problem by creating one huge structure to cover everything is undesirable: this solution is neither modular nor extendable.

In our endeavour to stay unbiased, we adopt a symmetrical approach that treats all views uniformly, as will be described in detail in § 3. To model all existing work we will also need a more demanding definition than the one used for lenses [25], since many techniques in the grammarware technological space are error-correcting, and thus can update both entities for consistency.

In § 4, we will propose a megamodel of mappings between strings, tokenised strings, layout-free lists of tokens, lexical source models, parse forests, parse trees, concrete syntax trees, abstract syntax trees, pictures, graphs and models of different kinds. We will also give examples of such mappings and argue how they fit our framework and what can we learn from it, leading to § 5 where a case study is presented with all types and mappings from Figure 1 implemented in Rascal [32], a metamodelling/metaprogramming language.

The main contribution of the paper is the structured megamodel of activities that can be viewed as either "parsing" or "unparsing"; the case study demonstrates its otherwise non-apparent aspects; the rest of the paper revisits and catalogues common software language engineering processes. The reader is expected to be familiar with modelling and megamodelling; background on parsing and bidirectionality is appreciated but not required: the necessary explanations and literature references are provided.

3 Bidirectionality

In this section we will recall some of the kinds of bidirectional mappings covered by previously existing research, in order to establish the background needed to appreciate and comprehend the main contribution of the paper (the megamodel on Figure 1). To simplify their comparison, we take the liberty of reformulating the definitions in a way that leaves them equivalent to their corresponding originals. We will also put a dot above a relation to account for undefined cases: e.g., the sign "$\dot{=}$" will denote "if all subformulae are defined, then equals".

The simplest form of a bidirectional mapping is a reversible function [34]:

Definition 1 (reversible function). *For a relation $\Psi \subseteq L \times R$, a* reversible function *f is a pair of functions for its forward execution $\overrightarrow{f} : L \to R$ and reverse execution $\overleftarrow{f} : R \to L$, such that*

$$\forall x \in L,\ \langle x, \overrightarrow{f}(x)\rangle \mathrel{\dot{\in}} \Psi \tag{1}$$

$$\forall y \in R,\ \langle \overleftarrow{f}(y), y\rangle \mathrel{\dot{\in}} \Psi \tag{2}$$

If it is bijective, then also

$$\forall x \in L,\ (\overleftarrow{f} \circ \overrightarrow{f})(x) \mathrel{\dot{=}} x \tag{3}$$

$$\forall y \in R,\ (\overrightarrow{f} \circ \overleftarrow{f})(y) \mathrel{\dot{=}} y \tag{4}$$

A lens [25] is a more complex bidirectional mapping defined asymmetrically such that one of its components can observe both the "old" value being updated and the changed one. We will call them "Foster BX" here to avoid confusion with many variations of lenses (for the purpose of this paper, we only need well-behaved lenses).

Definition 2 (Foster BX). *A* get *function $\nearrow : L \to R$ and a* putback *function $\searrow : R \times L \to L$ form a well-behaved lens, if*

$$\forall x \in L,\ \searrow(\nearrow(x), x) \mathrel{\dot{=}} x \tag{5}$$

$$\forall x \in L, \forall y \in R,\ \nearrow(\searrow(y, x)) \mathrel{\dot{=}} y \tag{6}$$

If a reversible function f exists, then constructing a lens is trivial: $\nearrow \equiv \overrightarrow{f}$ and $\forall x \in L, \searrow(y, x) \equiv \overleftarrow{f}(y)$. The inverse construction of a reversible function from a lens is only possible if both $\nearrow$ and $\searrow$ are bijective (Eq. 3 and Eq. 4 hold).

As an example of symmetric bidirectional mapping, we recall the notation by Meertens [35], with terms for properties inherited from Stevens [44]:

Definition 3 (Meertens BX). *A* bidirectional mapping *is a relation Ψ and its **maintainer**, which is a pair of functions $\triangleright : L \times R \to R$ and $\triangleleft : L \times R \to L$ that are* correct*:*

$$\forall x \in L, \forall y \in R, \langle x, x \triangleright y\rangle \mathrel{\dot{\in}} \Psi, \langle x \triangleleft y, y\rangle \mathrel{\dot{\in}} \Psi \tag{7}$$

and hippocratic*:*

$$\forall x \in L, \forall y \in R, \langle x, y\rangle \in \Psi \Rightarrow x \triangleright y = y, x \triangleleft y = x \tag{8}$$

Intuitively, correctness means that the result of either function is according to the relation Ψ. Hippocraticness means that no modification happens if the two values are already properly related (i.e., the transformation is guaranteed to "do no harm"). In other words, a maintainer $\triangleright$ and $\triangleleft$ can *maintain* the relation Ψ either by leaving their arguments unchanged if the relation is already respected or by massaging one of them with the data from the other one, until they do.

Constructing a trivial Meertens maintainer from a lens is straightforward:

$$x \triangleright y \equiv \nearrow (x)$$
$$x \triangleleft y \equiv \searrow (y, x)$$

Obviously, the inverse operation is only possible when the right semi-maintainer does not require y for computing its result.

While the symmetry of the definition allows us to research scenarios of two or more views of equal importance and comparable expressiveness, semi-maintainers always assume one side to be correct, which especially in the context of parsing only models straightforward precise parsing [1] or noncorrecting recovery strategies [40]. For more complicated scenarios, we introduce the following kind of bidirectional transformations:

Definition 4 (Final BX). *A* final bidirectional mapping *is a relation Ψ and its* ***sustainer****, which is a pair of functions $\blacktriangleright\colon L \times R \to \Psi$ and $\blacktriangleleft\colon L \times R \to \Psi$ that are hippocratic:*

$$\forall x \in L, \forall y \in R, \langle x, y\rangle \in \Psi \Rightarrow x \blacktriangleright y = \langle x, y\rangle, x \blacktriangleleft y = \langle x, y\rangle \tag{9}$$

Final BX is also *correct* in the sense of the codomain of $\blacktriangleright$ and $\blacktriangleleft$ being Ψ.

Constructing a sustainer from a Meertens BX or a Foster BX is trivial:

$$\begin{array}{rcccl} x \blacktriangleright y & \equiv & \langle x, x \triangleright y\rangle & \equiv & \langle x, \nearrow (x)\rangle \\ x \blacktriangleleft y & \equiv & \langle x \triangleleft y, y\rangle & \equiv & \langle \searrow (y, x), y\rangle \end{array}$$

Correctness of this construction is a direct consequence of the (rather strict) properties we have demanded in Def. 2 and Def. 3. For example, if maintainer functions were allowed to violate correctness, we would have needed to construct a sequence of updates until a fixed point would have been reached.

The inverse operation is only possible for noncorrecting sustainers:

Definition 5 (noncorrection property). *A sustainer is* noncorrecting*, if*

$$\forall x \in L, \forall y \in R \;\; x \blacktriangleright y = \langle x, y'\rangle \tag{10}$$
$$\forall x \in L, \forall y \in R \;\; x \blacktriangleleft y = \langle x', y\rangle \tag{11}$$

Noncorrecting sustainers are equivalent to maintainers.

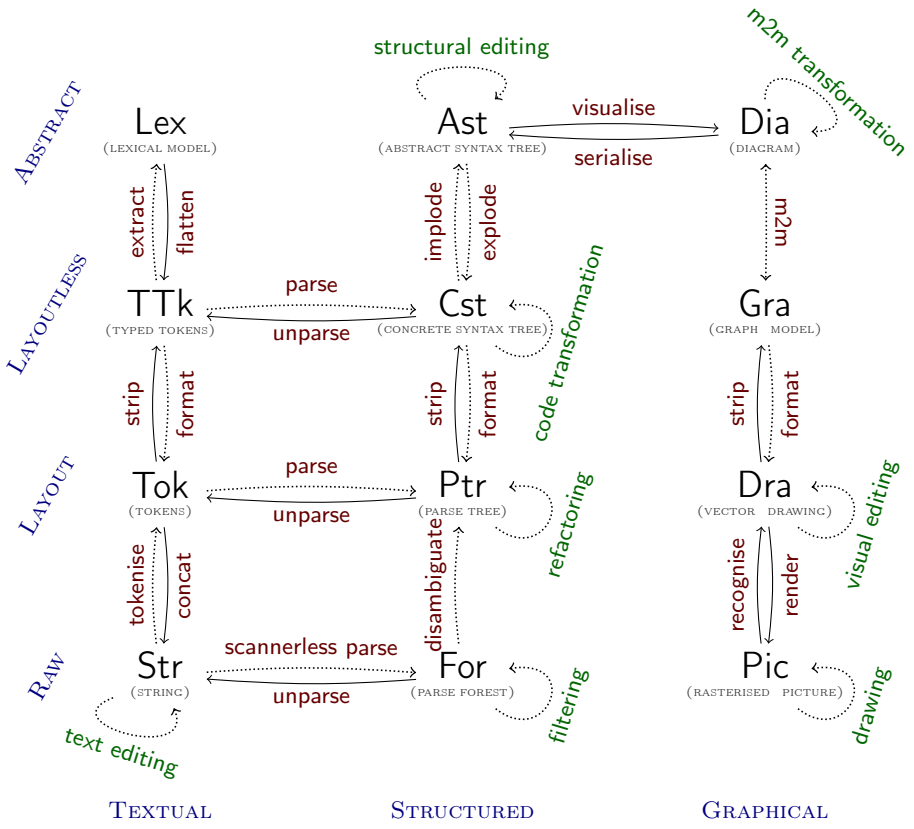

Fig. 1. Bidirectional megamodel of parsing. Dotted lines denote mappings that rely on either lexical or syntactic definitions; solid lines denote universally defined mappings. The loops are examples of transformations.

4 Artefacts and Mappings

Let us first introduce the kinds of artefacts we will use for the remainder of the paper:

- Str — a string.
- Tok — a finite sequence of strings (called *tokens*) which, when concatenated, yields Str. Includes spaces, line breaks, comments, etc — collectively, *layout*.
- TTk — a finite sequence of typed tokens, with layout removed, some classified as numbers of strings, etc.
- Lex — a lexical source model [16,36] that addes grouping to typing; in fact a possibly incomplete tree connecting most tokens together in one structure.
- For — a forest of parse trees, a parse graph or an ambiguous parse tree with sharing; a tree-like structure that models Str according to a syntactic definition.

- Ptr — an unambiguous parse tree where the leaves can be concatenated to form Str.
- Cst — a parse tree with concrete syntax information. Structurally similar to Ptr, but without layout.
- Ast — a tree which contains only abstract syntax information.
- Pic — a picture, which can be an ad hoc model, a natural model [50] or a rendering of a formal model.
- Dra — a graphical representation of a model (not necessarily a tree), a drawing in the sense of GraphML or SVG, or a metamodel-indepenent syntax but metametamodel-specific syntax like OMG HUTN.
- Gra — an entity-relationship graph or any other primitive "boxes and arrows" level model.
- Dia — a figure, a graphical model in the sense of EMF or UML, a model with an explicit advanced metamodel.

Figure 1 shows a megamodel of all the different artefacts and the mappings between them. The artefacts in the left column of the megamodel are *textual* (examples of these can be seen in Figure 2), the ones in the middle are *structured* (examples of these can be seen in Figure 3), and the ones on the right are *graphical* (Figure 4). Going "up" the megamodel increases the level of details in annotations: in Str we have one monolithic chunk, in Tok we know the boundaries between tokens, in TTk some tokens have types, in Lex some are grouped together (and similarly for other columns).

For example, classic parsing (e.g., using `yacc` [28]) is TTk $\to$ Cst or Tok $\to$ Ptr; layout-sensitive generalised scannerless parsing is Str $\to$ For (and possibly Str $\to$ For $\to$ Ptr). Going from TTk or Cst to Tok or Ptr is code formatting.

An interesting and important detail of those mappings for us is whether they are defined generally or parametric with a language specification of some kind (usually a grammar, a metamodel, a lexical definition, a regexp). For example, Ptr $\to$ Tok unparsing can be done by traversing a tree and collecting all its leaves from left to right. However, in order to construct a meaningful Cst tree from a TTk sequence, we need to rely on the hierarchy of linguistic categories (i.e., a grammar). Especially for the case of Ast $\rightleftharpoons$ Cst there are fairly complicated mapping inference strategies from (annotated) language specifications [45,49].

4.1 Fundamental Operations

Tokenisation. A tokeniser $\mathsf{tokenise}_L : \mathsf{Str} \to \mathsf{Tok}$ for some lexical grammar L, maps a character sequence $c_1, \ldots, c_n$ to a token sequence $w_1, \ldots, w_k$ in such a way that their concatenations are equal (i.e., $c_1 + \cdots + c_n = w_1 \cdots + w_k$). We call the reverse operation concat:

$$\forall x \in \mathsf{Str}, \mathsf{concat}(\mathsf{tokenise}_L(x)) \doteq x \tag{12}$$

$$\forall y \in \mathsf{Tok}, \mathsf{tokenise}_L(\mathsf{concat}(y)) \doteq y \tag{13}$$

Note that concat can be defined independently of the lexical grammar.

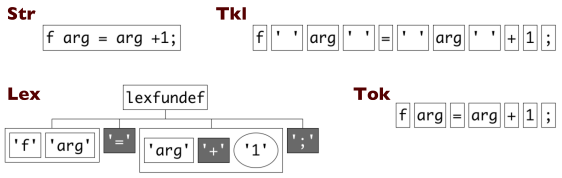

Fig. 2. Textual representations of a simple program. Clockwise from top left, Str (initial string), Tok (including layout), TTk (tokenised), Lex (lexical model).

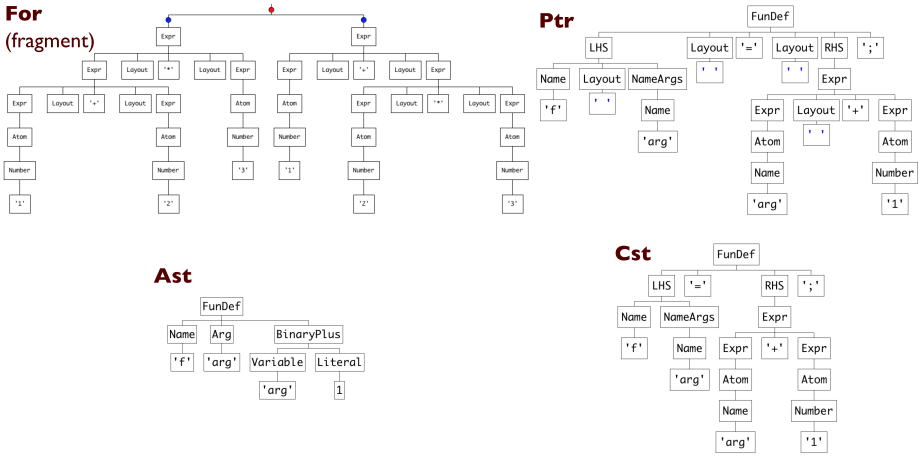

Fig. 3. Structured representations of a simple program. Clockwise from top left, For (forest of ambiguous interpretations), Ptr (parse tree including layout), Cst (concrete syntax), Ast (abstract syntax).

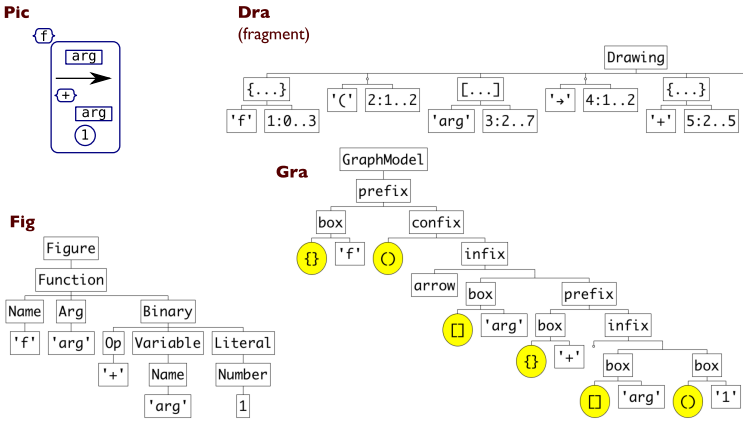

Fig. 4. Graphical representations of a simple program. Clockwise from top left, Pic (rasterised picture), Dra (vector picture), Gra (specific graphical model), Dia (abstract model).

Adding/Removing Layout. The strip operation removes layout; format introduces it. While stripping of layout is grammar independent, format is not. We can apply stripping and formatting to sequences ($\mathsf{strip} : \mathsf{Tok} \to \mathsf{TTk}$, $\mathsf{format}_{\mathrm{L}} : \mathsf{TTk} \to \mathsf{Tok}$), trees ($\mathsf{strip} : \mathsf{Ptr} \to \mathsf{Cst}$, $\mathsf{format}_{\mathrm{G}} : \mathsf{Cst} \to \mathsf{Ptr}$) and models ($\mathsf{strip} : \mathsf{Dra} \to \mathsf{Gra}$, $\mathsf{format}_{\mathrm{M}} : \mathsf{Gra} \to \mathsf{Dra}$).

Stripping has the following property (shown for $\mathsf{format}_{\mathrm{M}}$, defined similarly for the other two variants):

$$\forall x \in \mathsf{Gra}, \mathsf{strip}(\mathsf{format}(x)) \doteq x \tag{14}$$

Bidirectional stripping/formatting is at least a Foster BX: without knowing what the original input looked like prior to stripping, it is generally impossible to format it in the same way. With a Final BX we can model error correcting formatting as well. For example, if an arc is added to a Gra model, the visual formatter can suggest to add the target node to the model because it knows that drawing an edge always ends in a vertex. However, it is often desirable that such a sustainer is deterministic in the sense of Eq. 14 or close to it — i.e., reformatting a model will not result in a totally alien graph.

Layout preservation and propagation through transformations remains a challenging research topic even without considering error correction [12,31].

Parsing/Unparsing. Parsing recovers the implicit structure in the input sequence, representing it explicitly in a tree. The forward operations $\mathsf{parse}_{\mathrm{G}} : \mathsf{Tok} \to \mathsf{Ptr}$ and $\mathsf{parse}_{\mathrm{G}} : \mathsf{TTk} \to \mathsf{Cst}$ uncover the grammatical structure (defined in a grammar G) of a sequence (with or without layout). Parsing is readily reversible, with universally defined reverse operations $\mathsf{unparse} : \mathsf{Ptr} \to \mathsf{Tok}$ and $\mathsf{unparse} : \mathsf{Cst} \to \mathsf{TTk}$:

$$\forall x \in \mathsf{Tok}, \mathsf{unparse}(\mathsf{parse}_{\mathrm{G}}(x)) \doteq x \tag{15}$$

$$\forall x \in \mathsf{TTk}, \mathsf{unparse}(\mathsf{parse}_{\mathrm{G}}(x)) \doteq x \tag{16}$$

$$\forall y \in \mathsf{Ptr}, \mathsf{parse}_{\mathrm{G}}(\mathsf{unparse}(y)) \doteq y \tag{17}$$

$$\forall y \in \mathsf{Cst}, \mathsf{parse}_{\mathrm{G}}(\mathsf{unparse}(y)) \doteq y \tag{18}$$

Unparsing may be implemented by, for instance, collecting the leaves of the tree into a sequence.

Implosion/Explosion. For conversion to and from abstract syntax trees, we have $\mathsf{implode}_{\mathrm{G}} : \mathsf{Cst} \to \mathsf{Ast}$ and $\mathsf{explode}_{\mathrm{G}} : \mathsf{Ast} \to \mathsf{Cst}$. The explode mapping is non-trivial and requires knowledge of the intended syntax; implode can be defined in multiple ways, including the straightforward uniform mapping of Stratego/XT's `implode-asfix` [14] or Rascal's `implode` [32]. The explosion is harder to implement, and therefore it is rarely found in standard libraries.

Tree disambiguation. Earlier approaches to parsing always tried to burden the grammar with additional information to guide the parsing (and sometimes also the unparsing) process [1]. The state of the art in practical source code

manipulation usually relies on a relatively uniform parsing algorithm (SGLR, GLL or Packrat) that yields a For structure which is then filtered according to extra information [6,32]. In such a decoupled scenario this additional knowledge can come from the grammar, from a separate specification, from a classifying oracle, etc, which gives more flexibility to the language engineer. Thus, we must add the disambiguate operation both as a (metamodel-specific) mapping from For to Ptr, and as a For $\to$ For refinement. There is no currently available research results on bidirectionalising this mapping, even though many recommenders can possibly be implemented as Ptr $\to$ For transformations.

Rendering/Recognising. Image recognition techniques can be applied to extract structured graph information from a picture, identifying (with some degree of certainty) model elements, their positioning and relationships. Of course, some natural models are never meant to be recognised this way [50]. The reverse mapping is much more trivial and present in most visual editors.

4.2 Familiar Operations Decomposed

We can now decompose common operations into the fundamental components of the previous section; either single-step mappings $L \to R$ from one representation to another, or transformations $L \to L$ within the same representation.

For example, *code reindentation* is a transformation at the layout level, $\mathsf{indent} : \mathsf{Tok} \to \mathsf{Tok}$ (or $\mathsf{Ptr} \to \mathsf{Ptr}$), modifying the layout of the input while preserving the property:

$$\forall x \in \mathsf{Tok}, \mathsf{strip}(\mathsf{indent}(x)) \doteq \mathsf{strip}(x) \tag{19}$$

That is, changes in indentation make no difference at the layoutless level.

A compiler can be similarly decomposed, with the crucial transformation being, for example, $\mathsf{Ast}_{\mathrm{C}} \to \mathsf{Ast}_{\mathrm{ASM}}$. A full C compiler might be a pipeline $\mathsf{Str}_{\mathrm{C}} \to \mathsf{Tok}_{\mathrm{C}} \to \mathsf{Ptr}_{\mathrm{C}} \to \mathsf{Cst}_{\mathrm{C}} \to \mathsf{Ast}_{\mathrm{C}} \to \mathsf{Ast}_{\mathrm{IR}} \to \mathsf{Ast}_{\mathrm{ASM}} \to \mathsf{Cst}_{\mathrm{ASM}} \to \mathsf{TTk}_{\mathrm{ASM}} \to \mathsf{Tok}_{\mathrm{ASM}} \to \mathsf{Str}_{\mathrm{ASM}}$.

Examples

- A traditional lexer does Str $\to$ Tok $\to$ TTk in a single integrated step.
- Classic compiler textbook parsing is TTk $\to$ Cst [1]; though the resulting tree is often implicit (e.g., syntax-directed compilation).
- Layout-sensitive parsing is Tok $\to$ Ptr [24].
- Scannerless parsing is, in effect, parsing with Tok = Str or TTk = Str [43].
- The crucial steps of the PGF code formatting framework [4] are Ptr $\to$ Tok $\to^*$ Tok $\to$ Str, with tokens being annotated with extra information, and additional information from the parse tree appearing as control tokens.
- Code refactoring is, for instance, Ptr $\to$ Ptr [31], lowered to Str $\to$ Str.
- A structural editor does user-directed Ast $\to$ Ast transformation internally, while maintaining a Str representation for the user's benefit [46]. An IDE editor does user-directed Str $\to$ Str transformation, while maintaining an internal Ast representation.

- Wadler's prettier printer [47] does $\mathsf{Cst} \to \mathsf{Ptr} \to \mathsf{Tok}$ in single integrated step.

4.3 Discussion

Source-to-Source Transformations and Lowering. From the user's perspective, a transformation such as reindentation is an operation $\mathsf{indent}_{\mathsf{Str}} : \mathsf{Str} \to \mathsf{Str}$ on text, rather than on tokens. It has the property:

$$\forall x \in \mathsf{Str}, \mathsf{tokenise}(\mathsf{indent}_{\mathsf{Str}}(x)) \doteq \mathsf{indent}_{\mathsf{Tok}}(\mathsf{tokenise}(x)) \tag{20}$$

An implementation of $\mathsf{indent}_{\mathsf{Str}}$ may be obtained by applying concat to both sides and reducing:

$$\begin{aligned} \mathsf{concat}(\mathsf{tokenise}(\mathsf{indent}_{\mathsf{Str}}(x))) \doteq \mathsf{concat}(\mathsf{indent}_{\mathsf{Tok}}(\mathsf{tokenise}(x))) \iff \\ \mathsf{indent}_{\mathsf{Str}}(x) \doteq \mathsf{concat}(\mathsf{indent}_{\mathsf{Tok}}(\mathsf{tokenise}(x))) \qquad \text{by Eq. 12} \end{aligned}$$

This is called the *lowering* (to an operation on more concrete representations) of indent. A transformation lowered to $\mathsf{Str} \to \mathsf{Str}$ is usually called a *source-to-source* transformation. Given the operations in § 4.1 we may lower any transformation, as we please.

Transformation tools often allow the use of concrete syntax when specifying transformations – in effect, specifying transformation rules at the Str or Pic level. The rules are then *lifted* to the representation where the transformation actually takes place. For example; Stratego/XT [14] allows concrete syntax in transformation rules. Such rules are then parsed, disambiguated, stripped and imploded (and optionally desugared) into transformations on Ast.

In general, given a series of converting transformations from artifact A to B and back, we may implement any transformation on A by a transformation on B. Further explorations are needed to determine the required properties of such conversions.

Model-to-Model Transformation. There are models that have an advanced metamodel and many relations like "inherits from" or "conforms to" (i.e., Dia), which are apparently distinct from a simple view on the same models (i.e., Gra). Such a simplified view, a "concrete visual syntax" so to speak, can concern itself with the fact that we have nodes of various kinds (boxes, tables, pictures) which are connected by edges (lines, arrows, colours), all of them possibly labelled. If such distinction was to be made, we could see the difference between model-to-model transformations that refine and evolve the baseline model, and model-to-model transformations that "downgrade" it to one of the possible forms suitable for rendering. Then, $\mathsf{Gra} \to \mathsf{Dia}$ is a model-to-model transformation that can also be seen as "parsing" of the visual lexems to a model in a chosen language (a diagram).

$\mathsf{Ast} \to \mathsf{Dia}$ mappings are often viewed as *visualisations* and $\mathsf{Dia} \to \mathsf{Ast}$ ones as *serialisations*, even though in general they are glorified tree-to-graph transformations and graph-to-tree ones. So far we could not spot any research attempts to investigate mappings between lower levels of the central and the right column of Figure 1, except for idiosyncratic throwaway visualisations that are not meant to be edited and not meant to be transformed.

Layout Preservation. Layout preservation is an important requirement in certain grammarware applications, such as for instance automated refactoring. A *layout-preserving transformation* is one where all layout is intact, except for layouts in the parts that have been changed by the transformation. In essence, a transformation on an abstract representation is reflected in the concrete representation.

A layout-preserving transformation on a Cst is a transformation $t : \mathsf{Cst} \to \mathsf{Cst}$, lowered to $u : \mathsf{Ptr} \to \mathsf{Ptr}$, using a Foster BX with $\nearrow = \mathsf{strip}$:

$$u(x) = \searrow (t(\nearrow (x)), x), \quad x \in \mathsf{Ptr} \tag{21}$$

Lowering all the way to Str is simple, and gives us $r : \mathsf{Str} \to \mathsf{Str}$:

$$r(x) = \mathsf{concat}(\mathsf{unparse}(u(\mathsf{parse}(\mathsf{tokenise}(x))))), \quad x \in \mathsf{Str} \tag{22}$$

Incrementality. Suppose that we have a concrete and an abstract representation (e.g., a Str and a Cst), and a change in the concrete one should be reflected in the abstract one (this could be seen as the dual of layout preservation).

Again, we can use Foster BX. For example, for incremental parsing, we define editing $\mathsf{edit}_{\mathsf{Ptr}}$ of parse trees as follows, where $\mathsf{edit}_{\mathsf{Str}}$ is the user's editing transformation:

$$\mathsf{edit}_{\mathsf{Ptr}}(x) = \mathsf{parse}_{\mathsf{inc}}(\mathsf{edit}_{\mathsf{Tok}}(\mathsf{unparse}(x)), x) \tag{23}$$

$$\mathsf{edit}_{\mathsf{Tok}}(y) = \mathsf{tokenise}_{\mathsf{inc}}(\mathsf{edit}_{\mathsf{Str}}(\mathsf{concat}(y)), y) \tag{24}$$

The $\mathsf{unparse}$ and concat operation corresponds to the $\nearrow$ of Foster BX, while $\mathsf{parse}_{\mathsf{inc}} : \mathsf{Tok} \times \mathsf{Ptr} \to \mathsf{Ptr}$ and $\mathsf{tokenise}_{\mathsf{inc}} : \mathsf{Str} \times \mathsf{Tok} \to \mathsf{Tok}$ corresponds to $\searrow$. While the former two operations are trivial (defined in § 4.1), the latter two would be somewhat more challenging to implement.

We consider incremental parsing as a function taking the changed source and the original parse tree as input. In practice incremental parsing could hardly be constructed in this way as we still need to at least scan the source, which takes $O(n)$ time. It is more often that the incremental transformation takes the *change* directly as input [48], which is better formalised as *delta lenses* [21,22] or *edit lenses* [26].

Multiple Equitable Views. Foster BX inherently prefers one view over the other(s), which is acceptable for any framework with a clear baseline artefact and ones derived from it. However, there are cases when we want to continuously maintain relations among several views of the same "importance", and that is where we switch to Meertens BX. Imagine an IDE-enabled textual DSL with a built-in pretty-printing functionality. In that case, we have Str which is being edited by the user, whose typing actions can be mapped to incremental transformations on Ptr, since the grammar is known to the tool. However, an automated pretty-printing feature is a mapping from Cst to Ptr. In both cases we would like to retain the information present the the "pre-transformation" state of both Cst

and Ptr entities, for the sake of optimisation (incremental parsing makes IDE much more responsive) and usability (a user should be able to pretty-print a fragment of code without destroying the rest of the program).

Correcting Updates. By using Final BX, we can perform correcting updates, where a change in one of the views can trigger a negotiation sequence ultimately resulting in multiple updates possibly at all other views, including the initially changed one. For example, a parse error (e.g., "`';'` expected") during incremental parsing may result in a corrective measure being taken (e.g., "insert `';'`") in order for the parsing to continue [30]. Final BX pushes this change back to the input, so that the error can be corrected at the source, most likely by some kind of user interaction, such as Eclipse's *Quick Fix* feature.

5 Case study

In order to demonstrate the main ideas of this paper, we have prepared an open source prototype available at `http://github.com/grammarware/bx-parsing`. This section will introduce it briefly, all interested readers are invited to investigate the code, illustrations and documentation at the repository. The language for the implementation is Rascal [32], which is a one-stop-shop language workbench suitable for rapid prototyping of grammarware, defining data types, specifying program analyses and visualising results. Most figures on pages of this paper were automatically generated by it. All type definitions include a validation and a visualisation function. All types and mappings contain test cases.

The following twelve algebraic data types are defined in our prototype:

- Str — a string;
- Tok — a list of strings;
- TTk — a list of strings non-empty modulo whitespace stripping;
- Lex — a lexical model: left hand side tokens, right hand side tokens; tokens are typed (alphabetic, numeric, special);
- For — a parse forest defined by an ambiguous grammar;
- Ptr — a parse tree with explicit layout;
- Cst — a parse tree with layout discarded;
- Ast — an abstract data type;
- Pic — a multiline string representing a textual drawing;
- Dra — a list of visual elements such as symbols and labels, with coordinates;
- Gra — a hierarchical structure of prefix/infix/confix operators with implicit positioning;
- Dia — an abstract graphical model.

The mappings within the left ("textual") column of Figure 1 are mostly string manipulations on regular language level: since Rascal's default parsing algorithm is GLL, we implemented an explicit DFA tokeniser for Str → Tok; a library function `trim()` is used for Tok → TTk; pattern matching with regular expression-based for TTk → Lex. Going down on the diagram is even easier: Lex → TTk

does not rely on the structure of Lex, it just grabs all the tokens from it sequentially; TTk → Tok intercalates tokens with one space between any adjacent ones; Tok → Str uses a standard concatenation function.

In the default setup of Rascal, For → Ptr is called disambiguation and/or filtering [6], and Ptr ⇌ Cst is provided automatically. In order to separate bijective mapping between instances of one equivalent type to another, from the actual adjustments, we defined For, Ptr and Cst with three different grammars. Traditional concrete syntax matching does not work on ambiguous grammars (since the query of the match is also ambiguous), so For → Ptr is the longest and the ugliest of the mappings since it relies on low level constructions. Ptr → Cst is a top-down traversal that matches all layout and reconstructs a tree without it. Cst → Ast is very similar, it traverses a tree and constructs an ADT instance.

In general, Pic → Dra involves some kind of image recognition, and in our prototype indeed we scan the lines of the textual picture to identify elements, and convert them to Dra elements with appropriate coordinates. (Avoiding true image recognition algorithms outweighing illustrative code was one of the reasons we opted for drawing with text instead of pixels). In Dra → Gra we make some assumptions about the structure of the drawing: for example, we expect parenthesis to match.(Parentheses in our textual picture correspond to box containers in pixel visualisations, and the parenthesis matching thus corresponds to checking whether other elements fit inside the box or are placed outside it). Gra ⇌ Dia are m2m transformations between domains of graph models (boxes and arrows) and of function definitions (arguments and expressions).

Horizontal mappings are easier, since one of the main design concerns behind Rascal is EASY [32] parsing and fact extraction. We provide both Ast/Dia bridges (which are not uncommon in modern practice) and Lex/Ast (which are non-existent) bridges. There are several bonus mappings illustrating technological shortcuts, which we will not describe here due to space constraints. For example, there is a Gra → Pic mapping using Rascal string comprehensions to avoid low level coordinates arithmetic.

Now let us consider Ptr ⇌ Dra: a bidirectional mapping between a parse tree and a vector drawing. As we know, a parse tree contains structured information about the instance, including textual indentation; a drawing is similar to that, but contains information about graphical elements comprising the picture. We have prepared several implementations of Ptr ⇌ Dra:

- Reversible.rsc (Def. 1): $\overrightarrow{f}$ is `ptr2dra`, $\overleftarrow{f}$ is `dra2ptr`, and a problem with obtaining a valid final or intermediate instance is modelled by throwing an exception. From the tests we can see that $\overleftarrow{f} \circ \overrightarrow{f}$ is not always an identity function, which breaks Eq. 3 — hence, this mapping is reversible, but not bijective.
- Foster.rsc (Def. 2): ↗ is still `ptr2dra`, but ↘ is a superposition of `dra2ptr` on an updated Dra instance and a balancing function that traverses two Ptr instances (the old one and the updated one) and in its result saves all the element information from the new instance with as much as possible from the indentation of the old one. This ensures the GetPut law (Eq. 5).

However, this mapping disregards repositioning of graphical elements, which breaks the PutGet law (Eq. 6). Hence, the well-behavedness of the mapping is only preserved if the elements of the vector drawing do not move frome their default locations.

- Meertens.rsc (Def. 3): both $\triangleright$ and $\triangleleft$ implemented in the same way $\searrow$ was explained above — $\triangleright$ traverses two Dra instances and $\triangleleft$ traverses two Ptr instances. There are two lessons to learn here: first, since we have achieved correctness and hippocraticness in all desired scenarios, this is probably the BX that we want to have for Ptr $\rightleftharpoons$ Dra; second, both Dra and Ptr traversals needed to be programmed separately, which leads to duplicated effort and error-proneness.
- Final.rsc (Def. 4): $\blacktriangleright$ and $\blacktriangleleft$ behave similarly to $\triangleright$ and $\triangleleft$ resp., with two major differences: they fix some mistakes in Ptr (referencing undeclared variables) and in Dra (unbalanced brackets). This error recovery is motivated by the fact that the main purpose of bidirectional model transformation is consistency restoration [44]. However, Final BX can also be used to detect certain properties of instances and consistently enforce them at all ends.

The entire prototype is around 3000 lines, well tested and documented.

6 Related Work

Danvy [19] was the first one to propose a type-safe approach to unparsing by using embedded function composition and continuations. More recent research resulted in both embedding of this approach, referred to as string comprehensions, in modern metaprogramming languages [32] and development of the counterpart solution for pattern-driven selective parsing, referred to as "un-unparsing" [3].

Matsuda and Wang [33] propose a way to derive a (simplified) grammar from an unparser specification, which allowed them to focus on unparsing and infer a parser automatically.

Rendel and Ostermann [39] and Duregård and Jansson [23] independently define collections of Haskell ADTs to represent invertible syntax definitions, which can be used together with compatible combinator libraries to infer both parser (with Alex and Happy) and unparsers.

Brabrand et al [10] propose XSugar language that is used to provide a readable non-XML syntactic alternative to XML documents while retaining editability of both instances and preserving schema- conformance on the XML side.

For quasi-oblivious lenses [8], where the put function ignores differences between equivalent concrete arguments, we can say that $\forall x \in L, \forall y \in R, \langle x, y\rangle \in \Psi \Rightarrow x \blacktriangleleft y = \langle x', y\rangle$, where $x \sim x', \langle x', y\rangle \in \Psi$. In general, the vision expressed in this paper, can be conceptually viewed as establishing several equivalence relations specific for the domain of parsing/unparsing. Moreover, our research is not limited to *dictionary lenses* of Bohannon et al [8], since it concerns Final BX (Def. 4) and allows to continue expanding the bidirectional view on *semi-parsing* methods [53], especially Lex $\rightleftharpoons$ Ast mappings that are entirely avoided in the current state of the art.

The distinction we draw between textual, structured and graphical representations on Figure 1 and Figures 2–4, relates to the concept of a technical space [7]. We admit not having enough knowledge to add another column related to ontologies [37] and perhaps other spaces.

Obviously, the landscape of bidirectional model transformation [44] is much broader than just introducing or losing structure. The topics our work is the most close to, are model driven reverse engineering [15,38,42] for the parsing part and an even bigger one of model driven generation for the unparsing part.

7 Concluding Remarks

In this paper, we have considered parsing, unparsing, formatting, pretty-printing, tokenising, syntactic and lexical analyses and other techniques related to mapping between textual, structured data and visual models, from the bidirectional transformation perspective. We have proposed a uniform megamodel (Figure 1) for twelve classes of software artefacts (Figures 2–4) involved in these mappings, and given a number of examples from existing software language processing literature. We were able to find a place for all the mappings that we have considered, even though some explicitly or implicitly "skip" a step. The framework that was introduced, can be used to study such mappings in detail and assess actual contributions, weaknesses and compatibility. For example, with such an approach, we can take a seemingly monolithic $\mathsf{Ast} \rightleftharpoons \mathsf{Str}$ mapping of Ensō [45] and decompose it in easily comprehensible stages of $\mathsf{Str} \rightleftharpoons \mathsf{Tok}$ which is bijective because of fixed lexical syntax; $\mathsf{Tok} \rightleftharpoons \mathsf{Ptr} \rightleftharpoons \mathsf{Cst}$ which relies on the Wadler algorithm [47]; and $\mathsf{Cst} \rightleftharpoons \mathsf{Ast}$ inferred by the authors' own interpreter relying on annotations in Ast specifications ("schemas"). Another example is clear positioning of techniques such as rewriting with layout [12] which provide data structures that work like Cst in some cases and like Ptr in others.

Detailed investigation of lowering/lifting operations deserves much more attention than we could spare within this paper, because these concepts can help us seek, understand and address cases of lost information due to its propagation through the artefacts of Figure 1. We have also not touched upon the very related topic of model synchronisation [20] as a scenario when both bidirectionally linked artefacts change simultaneously, and the system needs to evolve by incorporating both changes on both sides — it would be very interesting to see how the existing methods work on Final BX, especially on their compositionality, which requires termination proofs.

After introducing a megamodel for (un)parsing mappings in § 4, we have explained the difference between general mappings (the ones defined universally, like concatenation) and language-parametric (roughly speaking, the ones requiring a grammar), and presented a case study in § 5. This work will serve as a foundation for us to answer research questions not only like "how to map X to Y, given specifications for all involved syntaxes?", but also like "how to map X to some Y?" and "how to find the best Y to map from X?".

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. B. Arnoldus, M. van den Brand, and A. Serebrenik. Less is More: Unparser-Completeness of Metalanguages for Template Engines. In *GPCE*, pages 137–146, 2011.
3. K. Asai, O. Kiselyov, and C.-c. Shan. Functional un|unparsing. *Higher-Order and Symbolic Computation*, 24(4):311–340, 2011.
4. A. H. Bagge and T. Hasu. A Pretty Good Formatting Pipeline. In *SLE'11, LNCS 8225*, pages 177–196, 2013.
5. A. H. Bagge and V. Zaytsev. Workshop on Open and Original Problems in Software Language Engineering. In *WCRE'13*, pages 493–494. IEEE, 2013.
6. H. J. S. Basten and J. J. Vinju. Faster Ambiguity Detection by Grammar Filtering. In *LDTA*, 2010.
7. J. Bézivin and I. Kurtev. Model-based Technology Integration with the Technical Space Concept. In *MIS*. Springer, 2005.
8. A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful Lenses for String Data. In *POPL'08*, pages 407–419. ACM, 2008.
9. R. Boulton. *Syn: A Single Language for Specifying Abstract Syntax Trees, Lexical Analysis, Parsing and Pretty-printing*. University of Cambridge, 1996.
10. C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual Syntax for XML Languages. In *Database Programming Languages, LNCS 3774*, pages 27–41. Springer, 2005.
11. M. G. J. van den Brand, A. T. Kooiker, N. P. Veerman, and J. J. Vinju. An Architecture for Context-sensitive Formatting. In *ICSM'05*, 2005.
12. M. G. J. van den Brand and J. J. Vinju. Rewriting with Layout. In *RULE*, 2000.
13. M. G. J. van den Brand and E. Visser. Generation of Formatters for Context-Free Languages. *ACM TOSEM*, 5(1):1–41, 1996.
14. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *SCP*, 72(1-2):52–70, 2008.
15. H. Bruneliére, J. Cabot, F. Jouault, and F. Madiot. MoDisco: a Generic and Extensible Framework for Model Driven Reverse Engineering. In *ASE*, 2010.
16. A. Cox and C. Clarke. Syntactic Approximation Using Iterative Lexical Analysis. In *IWPC'03*, pages 154–163, 2003.
17. K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.
18. N. A. Danielsson. Correct-by-construction Pretty-printing. In *DTP*. ACM, 2013.
19. O. Danvy. Functional unparsing. *JFP*, 8(6):621–625, 1998.
20. Z. Diskin. Algebraic Models for Bidirectional Model Synchronization. In *MoDELS, LNCS 5301*, pages 21–36. Springer, 2008.
21. Z. Diskin, Y. Xiong, and K. Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *JOT*, 10:1–25, 2011.
22. Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In *MoDELS, LNCS 6981*, pages 304–318. Springer, 2011.
23. J. Duregård and P. Jansson. Embedded Parser Generators. In *Haskell*. ACM, 2011.
24. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-Sensitive Generalized Parsing. In *SLE'12, LNCS 7745*, pages 244–263. Springer, 2013.
25. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM TOPLAS*, 29, May 2007.

26. M. Hofmann, B. Pierce, and D. Wagner. Edit Lenses. In *POPL*. ACM, 2012.
27. J. Hughes. The Design of a Pretty-printing Library. In *AFP*, pages 53–96, 1995.
28. S. C. Johnson. *YACC—Yet Another Compiler Compiler*. Computer Science Technical Report 32, AT&T Bell Laboratories, 1975.
29. M. de Jonge. Pretty-Printing for Software Reengineering. In *ICSM*. IEEE, 2002.
30. M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg. Natural and Flexible Error Recovery for Generated Modular Language Environments. *ACM TOPLAS*, 34(4):15:1–15:50, Dec. 2012.
31. M. de Jonge and E. Visser. An Algorithm for Layout Preservation in Refactoring Transformations. In *SLE'11, LNCS 6940*, pages 40–59. Springer, 2012.
32. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *GTTSE'09, LNCS 6491*, pages 222–289. Springer, Jan. 2011.
33. K. Matsuda and M. Wang. FliPpr: A Prettier Invertible Printing System. In *ESOP'13*, pages 101–120. Springer, 2013.
34. J. McCarthy. The Inversion of Functions Defined by Turing Machines. In *Automata Studies*, pages 177–181. 1956.
35. L. Meertens. Designing Constraint Maintainers for User Interaction. June 1998.
36. G. C. Murphy and D. Notkin. Lightweight Lexical Source Model Extraction. *ACM TOSEM*, 5(3):262–292, July 1996.
37. F. S. Parreiras, S. Staab, and A. Winter. On Marrying Ontological and Metamodeling Technical Spaces. In *ESEC-FSE*, pages 439–448. ACM, 2007.
38. Ó. S. Ramón, J. S. Cuadrado, and J. G. Molina. Model-driven Reverse Engineering of Legacy Graphical User Interfaces. In *ASE*, pages 147–150. ACM, 2010.
39. T. Rendel and K. Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *Haskell'10*, pages 1–12. ACM, 2010.
40. H. Richter. Noncorrecting Syntax Error Recovery. *ACM TOPLAS*, 7(3):478–489, July 1985.
41. M. Ruckert. Conservative Pretty-Printing. *SIGPLAN Notices*, 23(2):39–44, 1996.
42. S. Rugaber and K. Stirewalt. Model-Driven Reverse Engineering. *IEEE Software*, 21(4):45–53, 2004.
43. D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. In *PLDI'89*, pages 170–178. ACM, 1989.
44. P. Stevens. A Landscape of Bidirectional Model Transformations. In *GTTSE'07, LNCS 5235*, pages 408–424. Springer, 2008.
45. T. van der Storm, W. R. Cook, and A. Loh. The Design and Implementation of Object Grammars. *SCP*, 2014.
46. M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering*. dslbook.org, 2013.
47. P. Wadler. A Prettier Printer, 1997. http://homepages.inf.ed.ac.uk/wadler/papers/prettier/prettier.pdf.
48. M. Wang, J. Gibbons, and N. Wu. Incremental Updates for Efficient Bidirectional Transformations. In *ICFP'11*, pages 392–403. ACM, 2011.
49. D. S. Wile. Abstract Syntax from Concrete Syntax. In *ICSE*. ACM, 1997.
50. Z. Zarwin, J.-S. Sottet, and J.-M. Favre. Natural Modeling: Retrospective and Perspectives an Anthropological Point of View. In *XM'12*, pages 3–8. ACM, 2012.
51. V. Zaytsev. The Grammar Hammer of 2012. *ACM CoRR*, 1212.4446:1–32, 2012.
52. V. Zaytsev. Case Studies in Bidirectionalisation. In *TFP 2014*, pages 51–58, 2014.
53. V. Zaytsev. Formal Foundations for Semi-parsing. In *CSMR-WCRE*, 2014.