BiYacc: Roll Your Parser and Reflective Printer into One

Zirun Zhu[1,2] Hsiang-Shang Ko[2] Pedro Martins[3] João Saraiva[3] Zhenjiang Hu[1,2]

[1] SOKENDAI (The Graduate University for Advanced Studies), Japan
{zhu,hu}@nii.ac.jp
[2] National Institute of Informatics, Japan
hsiang-shang@nii.ac.jp
[3] HASLab/INESC TEC & University of Minho, Portugal
{prmartins,jas}@di.uminho.pt

Abstract

Language designers usually need to implement parsers and printers. Despite being two related programs, in practice they are designed and implemented separately. This approach has an obvious disadvantage: as a language evolves, both its parser and printer need to be separately revised and kept synchronised. Such tasks are routine but complicated and error-prone. To facilitate these tasks, we propose a language called BiYacc, whose programs denote both a parser and a printer. In essence, BiYacc is a domain-specific language for writing *putback-based* bidirectional transformations — the printer is a putback transformation, and the parser is the corresponding get transformation. The pairs of parsers and printers generated by BiYacc are thus always guaranteed to satisfy the usual round-trip properties. The highlight that distinguishes this *reflective* printer from others is that the printer — being a putback transformation — accepts not only an abstract syntax tree but also a string, and produces an updated string consistent with the given abstract syntax tree. We can thus make use of the additional input string, with mechanisms such as simultaneous pattern matching on the view and the source, to provide users with full control over the printing-strategies.

1 Introduction

Whenever we come up with a new programming language, as part of its compiler we need to design and implement a parser and a printer to convert between program text and its internal representation. A piece of program text, while conforming to a *concrete syntax* specification, is a flat string that can be easily edited by the programmer. The parser recovers the tree structure of such a string and converts it to an *abstract syntax* tree, which is a structured and simplified representation that is easier for the compiler backend to manipulate. On the other hand, a printer flattens an abstract syntax tree to a string, which is typically in a human readable format. This is useful for debugging the compiler or reporting internal information to the user, for example.

Parsers and printers do conversions in opposite directions, however, they are closely related — for example, we normally expect that a string printed from an abstract syntax tree can be parsed to the same tree. This is also clearly shown on language-based editors, as introduced by Reps [20, 21], where the user interacts with a pretty printed representation of the underlying abstract syntax tree. Thus, each user text update is performed as an

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

abstract syntax tree transformation that has to be automatically synchronised with its concrete representation. Despite this relationship, current practice is to write parsers and printers separately. This approach has an obvious disadvantage: the parser and the printer need to be revised from time to time as the language evolves. Each time, we must revise the parser and the printer and also keep them consistent with each other, which is a time-consuming and error-prone task. To address the problem, we propose a prototype domain-specific language BiYacc, in which the user can describe both a parser and a printer in a single program, contrary to designing and writing them separately as is traditional. By unifying these two pieces of software and deriving them from single, unambiguous and centralised code, we are creating a unified environment, which is easier to maintain and update, therefore respecting the "Don't Repeat Yourself" principle of software development [11].

Distinct from traditional kinds of printers, the printer generated from a BiYacc program is reflective: it takes not only an abstract syntax tree but also a piece of program text, and produces an updated piece of program text into which information from the abstract syntax tree is properly embedded. We illustrate reflective printing with the following small but non-trivial example (which is also used as the running example in subsequent sections) about a simple language of arithmetic expressions. The concrete syntax has negation, parentheses, and the four elementary arithmetic operations, while the abstract syntax has only the four elementary arithmetic operations — negated expressions are represented in the abstract syntax as subtraction expressions whose left operand is zero, and parenthesised expressions are translated into tree structures. Now suppose we write an arithmetic expression as a plain string, and parse it to an abstract syntax tree. Later, the abstract syntax tree is somehow modified (say, after some optimisation done by the compiler), and we want to print it back to a string for the user to compare with what was written originally (say, to understand what the compiler's optimisation does). To make it easier for the user to compare these two strings, we should try to maintain the syntactic characteristics of the original string when producing the updated string. Here we may choose to

- preserve all brackets — even redundant ones — in the original string, and
- preserve negation expressions in the original string instead of changing them to subtraction expressions.

For example, the string "$((-1))$" is parsed to an abstract syntax tree "`SUB (NUM 0) (NUM 1)`" (a subtraction node whose left subtree is a numeral node `0` and right subtree is another numeral node `1`); if we change the abstract syntax tree to "`SUB (NUM 0) (NUM 2)`" and update the string with our reflective printer, we get "$((-2))$" instead of "$0 - 2$". (Section 2 will present a BiYacc program that describes exactly this printing strategy.) Reflective printing is a generalisation of traditional printing because our reflective printer can accept an abstract syntax tree and an *empty* string, in which case it will behave just like a traditional printer, producing a new string depending on the abstract syntax tree only.

Under the bonnet, BiYacc is based on the theory of *bidirectional transformations* (BXs for short) [2, 7, 9]. BXs are used for synchronising two sets of data, one called the *source* and the other the *view*. Denoting the source set by S and the view set by V, a (well-behaved) BX is a pair of functions called *get* and *put*:

- the function $get : S \to V$ extracts a part of a source of interest to the user as a *view*, while
- the function $put : S \times V \to S$ takes a source and a view and produces an updated source incorporating information from the view.

The pair of functions should satisfy the following laws:

$$get(put(s, v)) = v \qquad \forall s \in S,\ v \in V \tag{PutGet}$$

$$put(s, get(s)) = s \qquad \forall s \in S \tag{GetPut}$$

Informally, the PutGet law enforces that *put* must embed all information of the view into the updated source, so the view can be recovered from the source by *get*, while the GetPut law prohibits *put* from performing unnecessary updates by requiring that putting back a view directly extracted from a source by *get* must produce the same, unmodified source. The parser and reflective printer generated from a BiYacc program are exactly the functions *get* and *put* in a BX, and are thus guaranteed to satisfy the PutGet and GetPut laws. In the context of parsing and printing, PutGet ensures that a string printed from an abstract syntax tree is parsed to the same tree, and GetPut ensures that updating a string with an abstract syntax tree parsed from the string leaves the string unmodified (including formatting details like parentheses). A BiYacc program thus not only conveniently expresses a parser and a reflective printer simultaneously, but also ensures that the parser and the reflective printer are consistent with each other in a precise sense.

```
1  Abstract
2
3  Arith = ADD Arith Arith
4        | SUB Arith Arith
5        | MUL Arith Arith
6        | DIV Arith Arith
7        | NUM String
8
9  Concrete
10
11 Expr   -> Expr '+' Term
12         | Expr '-' Term
13         | Term
14
15 Term   -> Term '*' Factor
16         | Term '/' Factor
17         | Factor
18
19 Factor -> '-' Factor
20         | String
21         | '(' Expr ')'
22
23 Actions
24
25 Arith +> Expr
26 ADD x y  ->  (x => Expr) '+' (y => Term)
27 SUB x y  ->  (x => Expr) '-' (y => Term)
28 arith    ->  (arith => Term)
29
30 Arith +> Term
31 MUL x y  ->  (x => Term) '*' (y => Factor)
32 DIV x y  ->  (x => Term) '/' (y => Factor)
33 arith    ->  (arith => Factor)
34
35 Arith +> Factor
36 SUB (NUM "0") y  ->  '-' (y => Factor)
37 NUM n            ->  (n => String)
38 arith            ->  '(' (arith => Expr) ')'
```

Figure 1: A BiYacc program for the expression example.

We have implemented BiYacc in Haskell and tested the example about arithmetic expressions mentioned above, which we will go through in Section 2. There is an interactive demo website:

`http://www.prg.nii.ac.jp/project/biyacc.html`

from which the source code of BiYacc can also be downloaded. The reader is invited to vary the input source string and abstract syntax tree, run the forward and backward transformations, and even modify the BiYacc programs to see how the behaviour changes. A sketch of the implementation is presented in Section 3, and we conclude the paper with some discussions of related work in Section 4.

2 An overview of BiYacc

This section gives an overview to the structure, syntax, and semantics of BiYacc by going through a program dealing with the example about arithmetic expressions. The program, shown in Figure 1, consists of three parts:

- abstract syntax definition,
- concrete syntax definition, and
- actions describing how to update a concrete syntax tree with an abstract syntax tree.

2.1 Defining the abstract syntax

The abstract syntax part of a BiYacc program starts with the keyword `Abstract`, and can be seen as definitions of *algebraic datatypes* commonly found in functional programming languages (see, e.g., [10, Section 5]). For the expression example, we define a datatype `Arith` of arithmetic expressions, where an arithmetic expression can be either an addition, a subtraction, a multiplication, a division, or a numeric literal. For simplicity, we represent a literal as a string. Different constructors — namely `ADD`, `SUB`, `MUL`, `DIV`, and `NUM` — are used to construct different kinds of expressions, and in the definition each constructor is followed by the types of arguments it takes. Hence the constructors `ADD`, `SUB`, `MUL`, and `DIV` take two subexpressions (of type `Arith`) as arguments, while the last constructor `NUM` takes a `String` as argument. (`String` is a built-in datatype of strings.) For instance, the expression "$1 - 2 \times 3 + 4$" can be represented as this abstract syntax tree of type `Arith`:

```
ADD (SUB (NUM "1")
         (MUL (NUM "2")
              (NUM "3")))
    (NUM "4")
```

2.2 Defining the concrete syntax

Compared with an abstract syntax, the structure of a concrete syntax is more refined such that we can conveniently yet unambiguously represent a concrete syntax tree as a string. For instance, we should be able to interpret the string "$1 - 2 \times 3 + 4$" unambiguously as a tree of the same shape as the abstract syntax tree at the end of the previous subsection. This means that the concrete syntax for expressions should, in particular, somehow encode the conventions that multiplicative operators have higher precedence than additive operators and that operators of the same precedence associate to the left.

In a BiYacc program, the concrete syntax is defined in the second part starting with the keyword `Concrete`. The definition is in the form of a context-free grammar, which is a set of production rules specifying how nonterminal symbols can be expanded to sequences of terminal or nonterminal symbols. For the expression example, we use a standard syntactic structure to encode operator precedence and order of association, which involves three nonterminal symbols `Expr`, `Term`, and `Factor`: an `Expr` can produce a left-leaning tree of `Term`s, each of which can in turn produce a left-leaning tree of `Factor`s. To produce right-leaning trees or operators of lower precedence under those with higher precedence, the only way is to reach for the last production rule `Factor -> '(' Expr ')'`, resulting in parentheses in the produced string.

Note that there is one more difference between the concrete syntax and the abstract syntax in this example: the concrete syntax has a production rule `Factor -> '-' Factor` for producing negated expressions, whereas in the abstract syntax we can only write subtractions. This means that negative numbers will have to be converted to subtractions and these subtractions will have to be converted back to negative numbers in the opposite direction. As we will see, this mismatch can be easily handled in BiYacc.

2.3 Defining the actions

The last and main part of a BiYacc program starts with the keyword `Actions`, and describes how to update a concrete syntax tree — i.e., a well-formed string — with an abstract syntax tree. Note that we are identifying strings representing program text with concrete syntax trees: Conceptually, whenever we write an expression as a string, we are actually describing a concrete syntax tree with the string (instead of just describing a sequence of characters). Technically, it is almost effortless to convert a (well-formed) string to a concrete syntax tree with existing parser technologies; the reverse direction is even easier, requiring only a traversal of the concrete syntax tree. By integrating with existing parser technologies, BiYacc actions can focus on describing conversions between concrete and abstract syntax trees — the more interesting part in the tasks of parsing and pretty-printing.

2.3.1 Individual action groups

The `Actions` part consists of groups of actions, and each group of actions begins with a type declaration:

abstract syntax datatype `+>` *concrete nonterminal symbol*

The symbol '`+>`' indicates that this group of actions describe how to put information from an abstract syntax tree of the specified datatype into a concrete syntax tree produced from the specified nonterminal symbol. Each action takes the form

abstract syntax pattern `->` *concrete syntax update pattern*

The left-hand side pattern describes a particular shape for abstract syntax trees and the right-hand side one for concrete syntax trees; also the right-hand side pattern is overlaid with updating instructions denoted by '`=>`'. For brevity, we call the left-hand side patterns *view patterns* and the right-hand side ones *source patterns* (in this case, representing an abstract and a concrete representation, respectively), hinting at their roles in terms of the underlying theory of bidirectional transformations. Given an abstract syntax tree and a concrete syntax tree, the semantics of an action is to simultaneously perform *pattern matching* on both trees (like in functional programming languages), and then use components of the abstract syntax tree to update parts of the concrete syntax tree, possibly recursively.

2.3.2 Individual actions

Let us look at a specific action — the first one for the expression example, at line 26 of Figure 1:

```
ADD x y -> (x => Expr) '+' (y => Term)
```

For the view pattern `ADD x y`, an abstract syntax tree (of type `Arith`) is said to match the pattern when it starts with the constructor `ADD`; if the match succeeds, the two arguments of the constructor (i.e., the two subexpressions of the addition expression) are then respectively bound to the variables `x` and `y`. (BiYacc adopts the naming convention in which variable names start with a lowercase letter and names of datatypes and nonterminal symbols start with an uppercase letter.) For the source pattern of the action, the main intention is to refer to the production rule

```
Expr -> Expr '+' Term
```

and use this to match those concrete syntax trees produced by first using this rule. Since the action belongs to the group `Arith +> Expr`, the part '`Expr ->`' of the production rule can be inferred and hence is not included in the source pattern. Finally we overlay '`x =>`' and '`y =>`' on the nonterminal symbols `Expr` and `Term` to indicate that, after the simultaneous pattern matching succeeds, the subtrees `x` and `y` of the abstract syntax tree are respectively used to update the left and right subtrees of the concrete syntax tree.

It is interesting to note that more complex view and source patterns are also supported, which can greatly enhance the flexibility of actions. For example, the view pattern

```
SUB (NUM "0") y
```

of the action at line 36 of Figure 1 accepts those subtraction expressions whose left subexpression is zero. This action is the key to preserving negation expressions in the concrete syntax tree. For an example of a more complex source pattern: Suppose that in the `Arith +> Factor` group we want to write a pattern that matches those concrete syntax trees produced by the rule `Factor -> '-' Factor`, where the inner nonterminal `Factor` produces a further `'-' Factor` using the same rule. This pattern is written by overlaying the production rule on the nonterminal `Factor` in the top-level appearance of the rule:

```
'-' (Factor -> '-' Factor)
```

2.3.3 Semantics of the entire program

Now we can explain the semantics of the entire program. Given an abstract syntax tree and a concrete syntax tree as input, first a group of actions is chosen according to the types of the trees. Then the actions in the group are tried in order, by performing simultaneous pattern matching on both trees. If pattern matching for an action succeeds, the update specified by the action is executed (recursively); otherwise the next action is tried. (Execution of the program stops when the matched action specifies either no updating operations or only updates to `String`.) BiYacc's most interesting behaviour shows up when pattern matching for all of the actions in the chosen group fail: in this case a suitable source will be created. The specific approach here is to do pattern matching just on the abstract syntax tree and choose the first matching action. A suitable concrete syntax tree matching the source pattern is then created, whose subtrees are recursively created according to the abstract syntax tree. We justify this approach as follows: if none of the source patterns match, it means that the input

concrete syntax tree differs too much from the abstract syntax tree, so we should throw the concrete syntax tree away and print a new one according to the abstract syntax tree.

2.3.4 An example of program execution

To illustrate, let us go through the execution of the program in Figure 1 on the abstract syntax tree

```
ADD (SUB (NUM 0) (NUM 4)) (NUM 5)
```

and the concrete syntax tree denoted by the string

$(-1 + 2 * 3)$

The abstract syntax tree is obtained from the concrete syntax tree by ignoring the pair of parentheses, desugaring the negation to a subtraction, and replacing the number 1 with 4 and the multiplication subexpression with 5. Executing the program will leave the pair of parentheses intact, update the number 1 in the concrete syntax tree with 4, preserving the negation, and update the multiplication subexpression to 5. In detail:

1. Initially the types of the two trees are assumed to match those declared for the first group, and hence we try the first action in the group, at line 26. The view-side pattern matching succeeds but the source-side one fails, because the first production rule used for the source is not `Expr -> Expr '+' Term` but `Expr -> Term` (followed by `Term -> Factor` and then `Factor -> '(' Expr ')'`, in order to produce the pair of parentheses).
2. So, instead, the action at line 28 is matched. The update specified by this action is to proceed with updating the subtree produced from `Term`, so we move on to the second group.
3. Similarly, the actions at lines 33 and 38 match, and we are now updating the subtree $-1 + 2 * 3$ produced from `Expr` inside the parentheses. Note that, at this point, the parentheses have been preserved.
4. For this subtree, we should again try the first group of actions, and this time the first action (at line 26) matches, meaning that we should update the subtrees -1 and $2 * 3$ with `SUB (NUM 0) (NUM 4)` and `NUM 5` respectively.
5. For the update of -1, we go through the actions at lines 28, 33, 36, and 37, eventually updating the number 1 with 4, preserving the negation.
6. As for the update of $2 * 3$, all the actions in the group `Arith +> Term` fail, so we create a new concrete syntax tree from `NUM 5` by going through the actions at lines 33 and 37.

2.3.5 Parsing

So far we have been describing the putback semantics of the BiYacc program, but we may also work out its get semantics by intuitively reading the actions in Figure 1 from right to left (which might remind the reader of the usual Yacc actions from this opposite angle): The production rules for addition, subtraction, multiplication, and division expressions are converted to the corresponding constructors, and the production rule for negation expressions is converted to a subtraction whose left operand is zero. The other production rules are ignored and do not appear in the resulting abstract syntax tree.

3 Implementation

Although what a BiYacc program describes is an update, i.e., reflective printing, it can also be interpreted in the opposite direction, generating an abstract syntax tree from a concrete syntax tree, which is the more interesting part of the task of parsing. We realise the bidirectional interpretation by compiling BiYacc programs to BiFluX [16], a putback-based bidirectional programming language for XML document synchronisation. The actual semantics of parsing and reflective printing are not separately implemented, but derived from the underlying BiFluX program compiled from a BiYacc program. Although the actual semantics of parsing in terms of BiFluX is more complicated compared with the intuitive reading given in Section 2.3.5, it is derived for free, thanks to the fact that BiFluX is a bidirectional language. Inside the implementation, both well-formed strings and abstract syntax trees must be in XML format before they can be synchronised by BiFluX, so BiYacc is bundled with parsers and printers converting between well-formed strings/abstract syntax trees and XML, and the user can use BiYacc without knowing that synchronisation is done on XML documents internally.

4 Related work

Many domain-specific languages have been proposed for describing both a parser and a printer as a single program to remove redundancy and potential inconsistency between two separate descriptions of the parser and the pretty-printer. There are basically three approaches:

- The first approach is to extend the grammar description for parsers with information for printers and abstract syntax tree types [1, 5, 17, 18, 22]. For instance, Syn [1] is a language for writing extended BNF grammars with explicit annotations of layout for printing and implicit precedence (binding strength) by textual ordering; then, from a description in Syn, a parser and a printer can be automatically generated.

- The second approach is to extend the printer description with additional grammar information [12, 14]. For instance, FliPpr [14] is a program transformation system that uses program inversion to produce a context-free grammar parser from an enriched printer.

- The third approach is to provide a framework for describing both a parser and a printer hand in hand in a constructive way [19]; primitive parsers and their corresponding printers are first specified, and more complex ones are built up from simpler ones using a set of predefined constructors.

Different from our system, these approaches cannot (do not intend to) deal with synchronisation between the concrete and abstract syntax trees, in the sense that a printer will print the concrete syntax tree from scratch without taking into account the original concrete syntax tree if it already exists. In contrast, our method can not only do parsing and printing, but also enable flexible updates over the old concrete syntax tree when doing printing (i.e., update-based pretty-printing).

Various attempts have been made to build up update-based pretty-printers from (extended) parsers that can preserve comments, layouts, and structures in the original source text. One natural idea is to store all information that does not take part in the abstract syntax tree, which includes whitespace, comments and (redundant) parentheses, and the modified source code is reconstructed from the transformed abstract syntax tree by layout-aware pretty-printing [3, 13]. It becomes more efficient to take origin tracking as a mechanism to relate abstract terms with their corresponding concrete representation [4]. Origin tracking makes it possible to locate moved subtrees in the original text. All these systems are hard-coded in the sense that the user can neither control information to be preserved nor describe specific updating strategies to be used in printing.

Our work was greatly inspired by the recent progress on bidirectional transformations [2, 7, 9]. In particular, it is a nontrivial application of the new putback-based framework [6, 8, 15, 16] for bidirectional programming, where a single putback function has been proved to be powerful enough to fully control synchronisation behaviour.

Acknowledgements

This work was partially supported by the Project NORTE-07-0124-FEDER-000062, co-financed by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). It was also partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (A) No. 25240009, and the MOU grant of the National Institute of Informatics (NII), Japan. The authors would like to thank Jeremy Gibbons for discussions about the design of BiYacc, Tao Zan for helping with setting up the demo website, and the anonymous reviewers for their valuable comments and suggestions.

References

[1] R. Boulton. Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report Number 390, Computer Laboratory, University of Cambridge, 1966.

[2] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *International Conference on Model Transformation*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer-Verlag, 2009.

[3] M. de Jonge. Pretty-printing for software reengineering. In *International Conference on Software Maintenance*, pages 550–559. IEEE, 2002.

[4] M. de Jonge and E. Visser. An algorithm for layout preservation in refactoring transformations. In *International Conference on Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 40–59. Springer-Verlag, 2012.

[5] J. Duregård and P. Jansson. Embedded parser generators. In *Haskell Symposium*, pages 107–117. ACM, 2011.

[6] S. Fischer, Z. Hu, and H. Pacheco. The essence of bidirectional programming. *SCIENCE CHINA Information Sciences*, 58(5):1–21, 2015.

[7] 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 Transactions on Programming Languages and Systems*, 29(3):17, 2007.

[8] Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming. In *International Symposium on Formal Methods*, pages 1–15. Springer-Verlag, 2014.

[9] Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Dagstuhl Seminar on Bidirectional Transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011.

[10] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *History of Programming Languages*, pages 1–55. ACM, 2007.

[11] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[12] P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming Languages and Systems*, pages 273–287. Springer-Verlag, 1999.

[13] J. Kort and R. Lammel. Parse-tree annotations meet re-engineering concerns. In *International Workshop on Source Code Analysis and Manipulation*. IEEE, 2003.

[14] K. Matsuda and M. Wang. FliPpr: A prettier invertible printing system. In *European Conference on Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 101–120. Springer-Verlag, 2013.

[15] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *Partial Evaluation and Program Manipulation*, pages 39–50. ACM, 2014.

[16] H. Pacheco, T. Zan, and Z. Hu. BiFluX: A bidirectional functional update language for XML. In *Principles and Practice of Declarative Programming*, pages 147–158. ACM, 2014.

[17] A. Ranta. Grammatical framework. *Journal of Functional Programming*, 14(2):145–189, 2004.

[18] A. Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. Center for the Study of Language and Information/SRI, 2011.

[19] T. Rendel and K. Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. In *Haskell Symposium*, pages 1–12. ACM, 2010.

[20] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, 1989.

[21] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

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