Abstract
Language designers usually need to implement parsers and printers. Despite being two intimately related programs, in practice they are often designed separately, and then need to be revised and kept consistent as the language evolves. It will be more convenient if the parser and printer can be unified and developed in one single program, with their consistency guaranteed automatically.

Furthermore, in certain scenarios (like showing compiler optimisation results to the programmer), it is desirable to have a more powerful reflective printer that, when an abstract syntax tree corresponding to a piece of program text is modified, can reflect the modification to the program text while preserving layouts, comments, and syntactic sugar.

To address these needs, we propose a domain-specific language BiYACC, whose programs denote both a parser and a reflective printer for an unambiguous context-free grammar. BiYACC is based on the theory of bidirectional transformations, which helps to guarantee by construction that the pairs of parsers and reflective printers generated by BiYACC are consistent. We show that BiYACC is capable of facilitating many tasks such as Pombrio and Krishnamurthi’s “resugaring”, simple refactoring, and language evolution.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Specialized Application Languages

Keywords parsing, reflective printing, bidirectional transformations, domain-specific languages

1. Introduction
Whenever we come up with a new programming language, as the front-end part of the system we need to design and implement a parser and a printer to convert between program text and an 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 extracts the tree structure from such a string to a concrete syntax tree (CST), and converts it to an abstract syntax tree (AST), which is a more structured and simplified representation and is easier for the back-end to manipulate. On the other hand, a printer converts an AST back to a piece of program text, which can be understood by the user of the system; this is useful for debugging the system, or reporting internal information to the user.

Parsers and printers do conversions in opposite directions and are intimately related — for example, the program text printed from an AST should be parsed to the same tree. It is certainly far from being economical to write parsers and printers separately: The parser and printer need to be revised from time to time as the language evolves, and each time we must revise the parser and printer and also keep them consistent with each other, which is a time-consuming and error-prone task. In response to this problem, many domain-specific languages (Boulton 1966; Visser 1997; Rendel and Ostermann 2010; Duregård and Jansson 2011; Matsuda and Wang 2013) have been proposed, in which the user can describe both a parser and a printer in a single program.

Despite their advantages, these domain-specific languages cannot deal with synchronisation between program text and ASTs. Let us look at a concrete example in Figure 1: The
Original program text:

\[-a \text{ /* } a \text{ is the variable denoting... } */ (1 + 1 + (a))\]

Abstract syntax tree:

```
Mul (Sub (Num 0) (Var "a"))
(Add (Add (Num 1) (Num 1)) (Var "a"))
```

Optimised abstract syntax tree:

```
Mul (Sub (Num 0) (Var "a"))
(Add (Num 2) (Var "a"))
```

Printed result from a conventional printer:

\[(0 - a) * (2 + a)\]

Printed result from our reflective printer:

\[-a \text{ /* } a \text{ is the variable denoting... } */ (2 + (a))\]

**Figure 1.** Comparing conventional and reflective printing

original program text is an arithmetic expression, containing negation, a comment, and (redundant) parentheses. It is first parsed to an AST (supposing that addition is left-associative) where the negation is desugared to a subtraction, parentheses are implicitly represented by the tree structure, and the comment is thrown away. Suppose the AST is optimised by replacing `Add (Num 1) (Num 1)` with a constant `Num 2`. The user may want to observe the optimisation made by the compiler, but the AST is an internal representation not exposed to the user, so a natural idea is to reflect the change on the AST printing. However, the printed result will likely mislead the programmer into thinking that the negation is replaced by a subtraction by the compiler; also, since the comment is not preserved, it will be harder for the programmer to compare the updated and original versions of the text. The problem illustrated here has also been investigated in many other practical scenarios where the parser and printer are used as a bridge between the system and the user, for example,

- in program debugging where part of the original program in an error message is displayed much differently from its original form (De Jonge 2002; Kort and Lämmel 2003), and

- in program refactoring where the comments and layouts in the original program are completely lost after the AST is transformed by the system (De Jonge and Visser 2012).

To address the problem, we propose a domain-specific language BiYACC, which lets the user describe both a parser and a reflective printer for an unambiguous context-free grammar in a single program. Different from a conventional printer, a reflective printer takes a piece of program text and an AST, which is usually slightly modified from the AST corresponding to the original program text, and reflects the modification back to the program text. Meanwhile the comments (and layouts) in the unmodified parts of the program text are all preserved. This can be seen clearly from the result of using our reflective printer on the above arithmetic expression example as shown in Figure 1. It is worth noting that reflective printing is a generalisation of the conventional notion of printing, because a reflective printer can accept an AST and an empty piece of program text, in which case it will behave just like a conventional printer, producing a new piece of program text depending on the AST only. (BiYACC currently does not support pretty printing, which however can be achieved in the future by letting the user define their own “adaptive functions” (see Section 4) to specify the layouts in the generated program text.)

From a BiYACC program we can generate a parser and a reflective printer; in addition, we want to guarantee that the two generated programs are consistent with each other. Specifically, we want to ensure two inverse-like properties:

- Firstly, a piece of program text `s` printed from an abstract syntax tree `t` should be parsed to the same tree `t`, i.e.,

  \[
  \text{parse } (\text{print } s) = t
  \]

- Secondly, updating a piece of program text `s` with an AST parsed from `s` should leave `s` unmodified (including formatting details like parentheses and spaces), i.e.,

  \[
  \text{print } (\text{parse } s) = s
  \]

These two properties are inspired by the theory of bidirectional transformations (Czarnecki et al. 2009; Hu et al. 2011), and are guaranteed by construction for all BiYACC programs.

An online tool that implements the approach described in the paper can be accessed at http://www.prg.nii.ac.jp/project/biyacc.html. The webpage also contains input examples with various test cases used in the paper. The structure of the paper is as follows:

- We first give an overview of BiYACC in Section 2, explaining how to describe in a single program both a parser and a reflective printer for synchronising program text and its abstract syntax representation.

- After reviewing some background on bidirectional transformations in Section 3, in particular the bidirectional programming language BiGUL (Ko et al. 2016), we give the semantics of BiYACC by compiling it to BiGUL in Section 4, guaranteeing the properties (1) and (2) by construction.

- We present a case study in Section 5, showing that although BiYACC is currently restricted to handling unambiguous grammars, it is capable of describing TIGER (Appel 1998), which shares many similarities with full grown, widely used languages. We demonstrate that BiYACC can handle syntactic sugar, partially subsume Pombrio and Krishnamurthi (2014)’s “resugaring”, and facilitate language evolution.

- Related work including detailed comparison with other systems and conclusion are presented in Section 6 and Section 7 respectively.

## 2. A First Look at BiYACC

We first give an overview of BiYACC by going through the BiYACC program shown in Figure 2, which deals with
defined by a context-free grammar. For our expression exam-
ple, in lines 16–27 we use a standard grammatical structure to encode operator precedence and order of association, which involves three nonterminal symbols \texttt{Expr}, \texttt{Term}, \texttt{Factor}: An \texttt{Expr} can produce a left-leaning tree of \texttt{Terms}, each of which can in turn produce a left-leaning tree of \texttt{Factors}. 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 \texttt{Factor} \rightarrow (\texttt{Expr}) \texttt{'}{}, resulting in parentheses in the produced program text. (There is also a nonterminal \texttt{Name}, which produces identifiers.)

\begin{figure}
\begin{verbatim}
Program ::= ‘Abstract’ HsDeclarations
  ‘Concrete’ ProductionGroup+
  ‘Actions’ ActionGroup

ProductionGroup ::= Nonterminal ‘->’ ProductionBody+ {‘|’} ‘;’

ProductionBody ::= Symbol+
  Symbol ::= Primitive | Terminal | Nonterminal

ActionGroup ::= HsType ‘->’ Nonterminal
  Action+
  Action ::= HsPattern ‘->’ Update+ ‘;’
  Update ::= Symbol
    | (‘ HsVariable ‘->’ UpdateCondition ’)
    | (‘ Nonterminal ‘->’ Update ’)

UpdateCondition ::= Symbol
  | (‘ Nonterminal ‘->’ UpdateCondition ’)
\end{verbatim}
\caption{Syntax of \texttt{BiYACC} programs (Nonterminals with prefix \texttt{Hs} denote Haskell entities and follow the Haskell syntax; the notation \texttt{nt}+{\texttt{sep}} denotes a nonempty sequence of the same non-terminal \texttt{nt} separated by \texttt{sep}).}
\end{figure}

\subsection{Definitions of the Abstract and Concrete Syntax}
A \texttt{BiYACC} program consists of definitions of the abstract and concrete syntax and \texttt{actions} for reflectively printing ASTs to CSTs. The abstract syntax part — which starts with the keyword \texttt{Abstract} — is just one or more definitions of Haskell datatypes. In our example, the abstract syntax is defined in lines 1–9 by a single datatype \texttt{Arith} whose elements are constructed from constants and arithmetic operators.

On the other hand, the concrete syntax — which is defined in the second part beginning with the keyword \texttt{Concrete} — is defined by a context-free grammar. For our expression example, we give the definition of \texttt{BiYACC} syntax in Figure 3, which will also be referred to in Section 4.

\subsection{Actions}
The last and main part of a \texttt{BiYACC} program starts with the keyword \texttt{actions}, and describes how to update a CST with an AST. For our expression example, the actions are defined in lines 29–45 in Figure 2. Before explaining the actions, we should first emphasise that we are identifying program text with CSTs: Conceptually, whenever we write a piece of program text, we are actually describing a CST rather than just a sequence of characters. We will expound on this identification of program text with CSTs in Section 4.2.1.

The \texttt{actions} part consists of groups of actions, and each group begins with a “type declaration” of the form \texttt{hsType ‘->’ nonterminal} stating that the actions in this group specify updates on CSTs generated from \texttt{nonterminal} using ASTs.
of type `hsType`. Informally, given an AST and a CST, the semantics of an action is to perform pattern matching simultaneously on both trees, and then use components of the AST to update corresponding parts of the CST, possibly recursively. (The syntax `+>` suggests that information from the left-hand side is embedded into the right-hand side.) Usually the non-terminals in a right-hand side pattern are overlaid with update instructions, which are also denoted by `+>`. Let us look at a specific action — the first one for the expression example, at line 32 of Figure 2:

\[
\text{Add } x \ y +> (x +> \text{Expr}) \ '++' \ (y +> \text{Term});
\]

The AST pattern is just a Haskell pattern; as for the CST pattern, the main intention is to refer to the production rule `Expr -> Expr '++' Term` and use it to match those CSTs produced by this rule. Since the action belongs to the group `Arith -> Expr`, the part `Expr ->` of the production rule can be inferred, and thus is not included in the CST 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 AST are respectively used to update the left and right subtrees of the CST.

Having explained what an action means, we can now explain the semantics of the entire program. Given an AST and a CST 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, from top to bottom, by performing simultaneous pattern matching on both trees. If pattern matching for an action succeeds, the updating operations specified by the action is executed; otherwise the next action is tried. Execution of the program ends when the matched action specifies either no updating operations or only updates to primitive datatypes such as `int`. `ByYacc`'s most interesting behaviour shows up when all actions in the chosen group fail to match — in this case a suitable CST will be created. The specific approach adopted by `ByYacc` is to perform pattern matching on the AST only and choose the first matched action. A suitable CST conforming to the CST pattern is then created, and after that the whole group of actions is tried again. This time the pattern matching should succeed at the action used to create the CST, and the program will be able to make further progress.

**Layout and comment preservation.** The reflective printer generated by `ByYacc` is capable of preserving layouts and comments, but, perhaps mysteriously, in Figure 2 there is no clue as to how layouts and comments are preserved. This is because we decide to hide layout preservation from the programmer, so that the more important logic of abstract and concrete syntax synchronisation is not cluttered with layout preserving instructions. Our current approach is fairly simplistic: We store layout information following each terminal in an additional field in the CST implicitly, and treat comments in the same way as layouts.\(^1\) During the printing stage, if the pattern matching on an action succeeds, the layouts and comments after the terminals shown in the right-hand side of that action are preserved; on the other hand, layouts and comments are dropped when a CST is created in the situation where pattern matching fails for all actions in a group. The layouts and comments before the first terminal are always kept during the printing.

**Parsing semantics.** So far we have been describing the reflective printing semantics of the `ByYacc` program, but we may also work out its parsing semantics intuitively by interpreting the actions from right to left, converting the production rules to the corresponding constructors. (This might remind the reader of the usual `Yacc` actions.) In fact, this paper will not define the parsing semantics formally, because the parsing semantics is completely determined by the reflective printing semantics: If the actions are written with the intention of establishing some relation between the CSTs and ASTs, then `ByYacc` will be able to derive the only well-behaved parser, which respects that relation. We will explain how this is achieved in the next section.

### 3. Foundation for `ByYacc`: Putback-Based Bidirectional Transformations

The behaviour of `ByYacc` is totally nontrivial: Not only do we need to generate two different programs from one, but we also need to guarantee that the two generated programs are consistent with each other, i.e., satisfy the properties (1) and (2) stated in Section 1. It is possible to separately implement the `print` and `parse` semantics in an ad hoc way, but verifying the two consistency properties takes extra effort. The implementation we present, however, is systematic and guarantees consistency by construction, thanks to the well-developed theory of bidirectional transformations (BXs for short). (See Czarnecki et al. (2009) and Hu et al. (2011) for a comprehensive introduction.)

#### 3.1 Parsing and Printing as Bidirectional Transformations

The `parse` and `print` semantics of `ByYacc` programs are potentially partial — for example, if the actions in a `ByYacc` program do not cover all possible forms of program text and abstract syntax trees, `parse` and `print` will fail for those uncovered inputs. Thus we should take partiality into account when choosing a BX framework in which to model `parse` and `print`. The framework we use in this paper is an explicitly partial version of asymmetric lenses (Foster et al. 2007): A (well-behaved) lens between a `source` type `S` and a `view` type `V` is a pair of functions `get` and `put`, where

- the function `get :: S -> Maybe V` extracts a part of a source of interest to the user as a view, and
- the function `put :: S -> V -> S` updates the source

\(^1\) One might argue that layouts and comments should in fact be handled differently, since comments are usually attached to some entities which they
• the function \( \text{put} :: S \to V \to \text{Maybe } S \) takes a source and a view and produces an updated source incorporating information from the view.

Partiality is explicitly represented by wrapping the result types in the Maybe monad. The pair of functions should satisfy the well-behavedness laws:

\[
\text{put } s \, v = \text{Just } s' \implies \text{get } s' = \text{Just } v \quad (\text{PUTGET})
\]

\[
\text{get } s = \text{Just } v \implies \text{put } s \, v = \text{Just } s \quad (\text{GETPUT})
\]

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 parse and print semantics of a BiYACC program will be the pair of functions get and put in a BX, satisfying the PUTGET and GETPUT laws by definition. The well-behavedness laws are then exactly the consistency properties (1) and (2) reformulated for a partial setting.

### 3.2 Putback-Based Bidirectional Programming

Having rephrased parsing and printing in terms of BXs, we can now easily construct consistent pairs of parsers and printers using bidirectional programming techniques, in which the programmer writes a single program to denote the two directions of a well-behaved BX. Specifically, BiYACC programs are compiled to the putback-based bidirectional programming language BiGUL (Ko et al. 2016). It has been formally verified in AGDA (Norell 2007) that BiGUL programs always denote well-behaved BXs, and BiGUL has been ported to HASSELS as an embedded DSL library, which will be introduced in more detail in Section 3.3. BiGUL is putback-based, meaning that a BiGUL program describes a put function, but — since BiGUL is bidirectional — can also be executed as the corresponding get function. The advantage of putback-based bidirectional programming lies in the following theorem (Foster 2009):

**Theorem.** Given a put function, there is at most one get function that forms a (well-behaved) BX with this put function.

That is, once we describe a put function in BiGUL, not only can we immediately obtain a get function satisfying PUTGET and GEPUT with the put function, but also guarantee that the get function is unique, i.e., completely determined by the put function. Thanks to this theorem, in this paper we can focus solely on the printing (put) behaviour, leaving the parsing (get) behaviour only implicitly (but unambiguously) specified.

### 3.3 Bidirectional Programming in BiGUL

Compilation of BiYACC to BiGUL (Section 4) only uses three BiGUL operations, which we explain here. A BiGUL program has type BiGUL \( s \, v \), where \( s \) and \( v \) are respectively the source and view types; its put interpreter can then be given the type

\[
\text{put} :: \text{BiGUL } s \, v \to s \to v \to \text{Maybe } s
\]

**Replace.** The simplest BiGUL operation we use is

\[
\text{Replace} :: \text{BiGUL } s \, v
\]

which discards the original source and returns the view — which has the same type as the source — as the updated source. That is,

\[
\text{put } \text{Replace } _v v = v
\]

**Update.** The next operation update is more complex, and is implemented with the help of Template Haskell (Sheard and Jones 2002). The general form of the operation is

\[
\$(\text{update } [p| \text{spat } |] [p| \text{vpat } |] [d| \text{bs } |]) :: \text{BiGUL } s \, v
\]

This operation decomposes the source and view by pattern matching with the patterns spat and vpat respectively, pairs the source and view components as specified by the patterns (see below), and performs further BiGUL operations listed in bs on the source–view pairs; the way to determine which source and view components are paired and which operation is performed on a pair is by looking for the same names in the three arguments.

For example, executing the update operation

\[
\$(\text{update } [p| (x, \_, _) |] [p| x |] [d| x = \text{Replace } |])
\]

on the source \( (1,2) \) and the view \( 3 \) will produce \( (3,2) \) as the updated source. (In the source pattern, the part marked by underscore (_) simply means that it will be skipped during the update.) In general, any (type-correct) BiGUL program can be used in the list of further updates, not just the primitive Replace.

**Case.** The most complex operation we use is Case for doing case analysis on the source and view:

\[
\text{Case} :: [\text{Branch } s \, v] \to \text{BiGUL } s \, v
\]

Case takes a list of branches, of which there are two kinds: normal branches and adaptive branches. For a normal branch, we should specify a main condition using a source pattern spat and a view pattern vpat, and an exit condition using a source pattern spat'.

\[
\$(\text{normalSV } [p| \text{spat } |] [p| \text{vpat } |] [p| \text{spat' } |]) :: \text{BiGUL } s \, v \to \text{Branch } s \, v
\]

An adaptive branch, on the other hand, only needs a main condition:

\[
\$(\text{adaptiveSV } [p| \text{spat } |] [p| \text{vpat } |]) :: (s \to v \to s) \to \text{Branch } s \, v
\]

Their put semantics are as follows: A branch is applicable when the source and view respectively match spat and vpat in its main condition. Execution of a Case chooses the first applicable branch from the list of branches, and continues with that branch. When the chosen branch is normal, the associated BiGUL operation is performed, and the updated

---

2 This representation of lists of named BiGUL operations is admittedly an abuse of syntax, but simplifies prototyping this system with Template Haskell.
source should satisfy the exit condition $spat'$ (otherwise it is a runtime error)$^3$; when the chosen branch is adaptive, the associated function is applied to the source and view to compute an adapted source, and the whole Case is rerun on the adapted source and the view, and should go into a normal branch this time. Think of an adaptive branch as bringing a source that is too mismatched with the view to a suitable shape so that a normal branch — which deals with sources and views in some sort of correspondence — can take over. This adaptation mechanism is used by BiYacc to print an AST when the source program text is too different from the AST or even nonexistent at all.

4. Implementation of BiYacc

The architecture of BiYacc is illustrated in Figure 4. The programmer supplies a three-part BiYacc program as described in Section 2, which is compiled into an executable for converting between program text and ASTs. The conversion is essentially the composition of two bidirectional transformations, one between program text and CSTs and the other between CSTs and ASTs. The second BX is constructed as a BiGUL program, and is well-behaved by construction; we will explain in Section 4.1 how this BiGUL program is derived. The first BX, on the other hand, is in fact an isomorphism, which we construct in a more ad hoc manner. More specifically, we derive a lexer and a parser (using the parser generator HAppy) for converting program text to CSTs, and a printer for flattening CSTs to program text; this isomorphism, which we call concrete parsing and printing, will be explained in Section 4.2. As is well known, isomorphisms are special cases of BXs, and the composition of two BXs is again a BX, so the composite transformations are still well-behaved.

Note that the grammar accepted by BiYacc is restricted to pure LALR (1) without disambiguation rules in the current implementation. And the well-behavedness of a BiYacc program is guaranteed by the compiled BiGUL program, which means that sometimes BiGUL will raise a runtime error for the bad transformations not satisfying the properties (1) and (2). In the future, we plan to extend BiYacc by making more static checks, and supporting disambiguation rules or employing other parser generators so that it can handle a wider class of grammars.

4.1 Generating the BiGUL Program

The semantics of BiYacc, shown in Figure 5, is defined by source-to-source compilation to BiGUL. Compilation rules are defined with the semantic bracket ($[[ ]])$, and refer to some auxiliary functions, whose names are in SMALL CAPS. A nonterminal in subscript gives the “type” of the argument or metavariable before it, and additional arguments to the semantic bracket are typeset in superscript.

Top-level structure. A BiYacc Program has the form

`Abstract decls 'Concrete' pgs 'Action' ags`

and is compiled to a three-part Haskell program by copying `decls` (which is already valid Haskell code), converting each group of production rules in `pgs` to a datatype, and each action group in `ags` to a small BiGUL program. The angle bracket notation $\langle f e \mid e \in es \rangle$ denotes the generation of a list of entities of the form $f e$ for each element $e$ in the list $es$, in the order of their appearance in $es$. Comment syntax declarations `xcommentLine` and `xcommentBlock` are only relevant to concrete parsing and printing, and are ignored in Figure 5.

CST datatypes. The production rules in a context-free grammar dictate how to generate strings from nonterminals, and a CST can be regarded as encoding one particular way of generating a string using the production rules. In Haskell, we represent CSTs starting from a nonterminal $nt$ as a datatype named $nt$, whose constructors represent the production rules for $nt$. For each constructor we generate a unique name, which is denoted by $\text{con}(nt, \text{sym})$. The fields of a constructor are generated from the right-hand side of the corresponding production rule in the way described by the auxiliary function $\text{field}$: Nonterminals are left unchanged (using their names for datatypes), terminal symbols are dropped, and an additional $\text{String}$ field is added for terminals and primitives for storing layout information (whitespaces and comments) appearing after them in the program text. The last step is to insert an additional empty constructor, the unique name generated for which is denoted by $\text{nullCon}(nt)$; this empty constructor is used as a default value to a BiYacc printer whenever we want to create a new piece of program text depending on the view only.

For instance, the third group of the concrete syntax defined in Figure 2 is translated to the following Haskell declarations:

```haskell
data Factor = Factor0 String Factor | Factor1 (Int, String)
```
Figure 5. Semantics of BiYACC programs (as BiGUL programs)
shown in the figure. For the expression example, we generate a
definition

def = bigulArithExpr
which is invoked in the main program.

4.2 Generating the Concrete Parser and Printer

Having explained the bidirectional transformation between
CSTs and ASTs, the remaining task is to establish an isomor-
phism between program text and CSTs.

4.2.1 The Inverse Properties

Assuming that the grammar is unambiguous and the parser
generator is “correct”, we will show that there exists an
isomorphism between program text and CSTs:

\[
\begin{align*}
\text{parse text} &= \text{Just cst} \implies \text{print cst} = \text{text} \quad (3) \\
\text{parse (print cst)} &= \text{Just cst} \quad (4)
\end{align*}
\]

One direction is a partial (Maybe-valued) function \text{parse} that
converts program text into CSTs according to the grammar,
and the other is a total function \text{print} from CSTs to program
text.

To see what (3) means, we should first clarify what \text{print}
does: Our CSTs, as described in Section 4.1, encode precisely
the derivation trees, with the CST constructors representing
the production rules used; what \text{print} does is simply traversing
the CSTs and applying the encoded production rules to
produce the derived program text. Now consider what \text{parse}
is supposed to do: It should take a piece of program text and
find a derivation tree for it, i.e., a CST which \text{prints} to that
piece of program text. This statement is exactly (3). In other
words, (3) is the functional specification of parsing, which is
satisfied if the parser generator we use behaves correctly.

For (4), since the grammar is unambiguous, for any piece
of program text there is at most one CST that prints to it,
which is equivalent to saying that \text{print} is injective. In
addition, it is reasonable to expect that a generated parser will be
able to successfully parse any valid program text; that is,
for any \text{cst} we have

\[
\text{parse (print cst)} = \text{Just cst}
\]

for some \text{cst}'. This is already close to (4); It remains to show
that \text{cst}' is exactly \text{cst}, which is indeed the case because

\[
\text{parse (print cst)} = \text{Just cst}' \\
\implies \{ (3) \} \\
\text{print cst} = \text{print cst}' \\
\implies \{ \text{print is injective} \} \\
\text{cst}' = \text{cst}
\]

4.2.2 Concrete Lexer and Parser

In current BiYACC, the implementation of the \text{parse} function
is further separated into two phases: tokenising and parsing.
In both phases, the layout information (whitespaces and
comments) is automatically preserved, which makes the CSTs
isomorphic to the program text.

\textbf{Lexer}. Apart from handling the terminal symbols appearing
in a grammar, the lexer automatically derived by BiYACC
can also recognise several kinds of literals, including inte-
gers, strings, and identifiers, respectively produced by the
nonterminals \text{Int}, \text{String}, and \text{Name}. For now, the forms of
these literals are pre-defined, but we take this as a step to-
wards a lexerless grammar, in which strings produced by
nonterminals can be specified in terms of regular expressions.
Furthermore, whitespaces and comments are carefully hand-
held in the derived lexer, so they can be completely stored in
CSTs and correctly recovered to the program text in printing.
This feature of BiYACC, which we explain below, makes
layout preservation transparent to the programmer.

An assumption of BiYACC is that whitespaces are only
considered as separators between other tokens. (Although
there exist some languages such as Haskell and Python
where indentation does affect the meaning of a program,
there are workarounds, e.g., writing a preprocessing program
to insert explicit separators.) Usually, token separators are
thrown away in the lexing phase, but since we want to keep
layout information in CSTs, which are built by the parser,
the lexer should leave the separators intact and pass them
to the parser. The specific approach taken by BiYACC is
wrapping a lexeme and the whitespaces following it into a
single token. Beginning whitespaces are treated separately
from lexing and parsing, and are always preserved. And in
this prototype implementation, comments are also considered
as whitespaces.

\textbf{Parser}. The concrete parser is used to generate a CST
from a list of tokens according to the production rules in the
grammar. Our parser is built using the parser generator
\text{HAPPY}, which takes a BNF specification of a grammar and
produces a \text{HASKELL} module containing a parser function.
The grammar we feed into \text{HAPPY} is still essentially the one
specified in a BiYACC program, but in addition to parsing
and constructing CSTs, the \text{HAPPY} actions also transfer the
whitespaces wrapped in tokens to corresponding places in
the CSTs. For example, the production rules for \text{Factor} in the
expression example, as shown on the left below, are translated
to the \text{HAPPY} specification on the right:

\[
\text{Factor} \rightarrow \text{'}\text{'} \text{Factor} : \text{token0 Factor} \\
| \text{Int} \rightarrow \text{tokenInt} \\
| \text{Name} \rightarrow \text{tokenName} \\
| \text{'}(\text{ Expr } \text{'})'; \rightarrow \text{token1 Expr token2} \\
\]

We use the first expansion (\text{token0 Factor}) to explain how
whitespaces are transferred: The generated \text{HAPPY} token
\text{token0} matches a ‘-’ token produced by the lexer, and extracts
the whitespaces wrapped in the ‘-’ token; these whitespaces
are bound to $1$, which is placed into the first field of \text{Factor0}
by the associated \text{HASKELL} action.
5. Case Study

The design of BiYACC may look simplistic and make the reader wonder how much it can describe. However, in this section we demonstrate with a larger case study that, without any extension, BiYACC can already handle real-world language features. For this case study, we choose the TIGER language, which is a statically typed imperative language first introduced in Appel’s textbook on compiler construction (Appel 1998). Since TIGER’s purpose of design is pedagogical, it is not too complex and yet covers many important language features including conditionals, loops, variable declarations and assignments, and function definitions and calls. TIGER is therefore a good case study with which we can test the potential of our BX-based approach to constructing parsers and reflective printers. Some of these features can be seen in this TIGER program:

```
function foo() =
  (for i := 0 to 10
   do (print(if i < 5 then "smaller" else "bigger"));
       print("\n")
  )
```

To give a sense of TIGER’s complexity, it takes a grammar with 81 production rules to fully TIGER’s syntax, while for C89 and C99 it takes respectively 183 and 237 rules without any disambiguation declarations (based on Kernighan et al. (1988) and the draft version of 1999 ISO C standard, excluding the preprocessing part). The difference is basically due to the fact that C has more primitive types and various kinds of assignment statements.

Excerpts of the abstract and concrete syntax of TIGER are shown in Figure 6. The abstract syntax is substantially the same as the original one defined in Appel’s textbook (page 98); as for the concrete syntax, Appel does not specify the whole grammar in detail, so we use a version slightly adapted from Hirzel and Rose (2013)’s lecture notes. Some changes are made to handle features that are not supported by current BiYACC. For example, to make the grammar become unambiguous without disambiguation rules, the operators are divided into several groups, with the highest-precedence terms (like literals) placed in the last group, just like what we did in the arithmetic expression example (Figure 2); the AST constructors TFunctionDec or TTypeDec take a single function or type declaration instead of a list of adjacent declarations (for representing mutual recursion) as in Appel (1998), since we cannot handle the synchronisation between a list of lists (in ASTs) and a list (in CSTs) with BiYACC’s current syntax; finally, to circumvent the “dangling else” problem, a terminal “end” is added to mark the end of an if-then expression.

Even though the underlying BiGUL programs have runtime checks as mentioned in the beginning of Section 4, we have successfully tested our BiYACC program for TIGER on all the sample programs provided on the homepage of Appel’s book\(^4\), including a merge sort implementation and an eight-queen solver, and there is no problem parsing and printing them. In the following subsections, we will present some printing strategies used in that TIGER program to demonstrate what BiYACC, in particular reflective printing, can achieve.

5.1 Syntactic Sugar

We start with a simple example about syntactic sugar, which is pervasive in programming languages and lets the programmer use some features in an alternative (perhaps conceptually higher-level) syntax. For instance, TIGER represents boolean values false and true respectively as zero and nonzero integers, and the logical operators \(\&\) ("and") and \(\mid\) ("or") are converted to if expressions in the abstract syntax: \(e1 \& e2\) is desugar and parsed to \(\text{IIf} \ e1 \ e2 \ (\text{TInt} \ 0)\) and \(e1 \mid e2\) to

---

\(^4\)https://www.cs.princeton.edu/~appel/modern/testcases/
The printing actions for them in BiYACC are:

```
TExp -> PrimitiveOpt
TIf (TInt 1) (Just e2) ->
  (e1 -> PrimitiveOpt) '!' (e2 -> PrimitiveOpt1);
```

```
TExp -> PrimitiveOpt1
TIf e1 e2 (Just (TInt 0)) ->
  (e1 -> PrimitiveOpt1) '$' (e2 -> PrimitiveOpt2);
```

The `parse` function for these syntactic sugar is not injective, since the alternative syntax and the features being desugared into are both mapped to the latter. A conventional printer — which takes only the AST as input — cannot reliably determine whether an abstract expression should be resugared or not, whereas a reflective printer can make the decision by inspecting the CST.

### 5.2 Resugaring

We have seen that BiYACC can handle syntactic sugar. In this subsection we will present another application based on that.

The idea of resugaring (Pombrio and Krishnamurthi 2014) is to print evaluation sequences in a core language in terms of a concrete syntax. Here we show that, without any extension, BiYACC is already capable of reflecting some of the AST changes resulting from evaluation back to the concrete syntax, subsuming a part of Pombrio and Krishnamurthi’s work.

We borrow their example of resugaring evaluation sequences for the logical operators “or” and “not”, but recast the example in TIGER. The “or” operator has been defined as syntactic sugar in Section 5.1. For the “not” operator, which TIGER lacks, we introduce “~”, represented by `TNot` in the abstract syntax. Now consider the source expression

```
~1 | ~0
```

which is parsed to

```
TIf (TNot (TInt 1)) (TInt 1) (J (TNot (TInt 0)))
```

where `J` is a shorthand for `Just`. A typical call-by-value evaluator will produce the following evaluation sequence given the above AST:

```
TIf (TNot (TInt 1)) (TInt 1) (J (TNot (TInt 0)))
  -> TIf (TInt 0) (TInt 1) (J (TNot (TInt 0)))
  -> TNot (TInt 0)
  -> TInt 1
```

If we perform reflective printing after every evaluation step using BiYACC, we will get the following evaluation sequence on the source:

```
1 | 0 -> 0 | 0 -> 0 -> 1
```

Due to the `PUTGET` property, parsing these concrete terms will yield the corresponding abstract terms in the first evaluation sequence, and this is exactly Pombrio and Krishnamurthi’s “emulation” property, which they have to prove for their system; for BiYACC, however, the emulation property holds by construction, since BiYACC’s semantics is defined in terms of BiGUL, whose programs are always well-behaved. Also different from Pombrio and Krishnamurthi’s approach is that we do not need to insert any additional information into the ASTs for remembering the form of the original sources.

The advantage of our approach is that we can keep the abstract syntax pure, so that other tools — the evaluator in particular — can process the abstract syntax without being modified, whereas in Pombrio and Krishnamurthi’s approach, the evaluator has to be adapted to work on the enriched abstract syntax.

Also note that the above resugaring for TIGER is achieved for free — the programmer does not need to write additional, special actions to achieve that. In general, BiYACC can easily and reliably reflect AST changes that involve only “simplification”, i.e., replacing part of an AST with a simpler tree, so it should not be surprising that BiYACC can also reflect simplification-like optimisations such as constant propagation and dead code elimination, and some refactoring transformations such as variable renaming and adding or removing parameters. All these can be achieved by one “general-purpose” BiYACC program, which does not need to be tailored for each application.

### 5.3 Language Evolution

We conclude this section by looking at a practical scenario in language evolution, incorporating all the features we introduced before in this section. When a language evolves, some new features of the language (e.g., `for` loops in Java 5) can be implemented by desugaring to some existing features (e.g., ordinary `for` loops), so that the compiler does not need to be extended to handle the new features. As a consequence, all the engineering work about refactoring or optimising transformations that has been developed for the abstract syntax remains valid.

Consider a kind of “generalised-if” expression allowing more than two cases, resembling the alternative construct in Dijkstra (1975)’s guarded command language. We extend TIGER’s concrete syntax with the following production rules:

```
Exp  -> ... | Guard | ... ;
Guard -> 'guard' CaseBs 'end';
CaseBs -> CaseB CaseBs | CaseB ;
CaseB -> LValue '=' Int '->' Exp ;
```

For simplicity, we restrict the predicate produced by `CaseB` to the form `LValue '=' Int`, but in general this can be any expression computing an integer. The reflective printing actions for this new construct can still be written within BiYACC, but require much deeper pattern matching:

```
TExp -> Guard
TIf (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing
  -> 'guard' (CaseBs -> (CaseB ->
    (lv <- LValue) '=' (i <- Int) '->' (e1 <- Exp))
    'end';
TIf (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@(TIf _ _ _))
  -> 'guard' (CaseBs -> (CaseB ->
    (lv <- LValue) '=' (i <- Int) '->' (e1 <- Exp))
    (if2 <- CaseBs)
    'end';
TExp -> CaseBs
TIf (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing
  -> (CaseB ->
    (lv <- LValue) '=' (i <- Int) '->' (e1 <- Exp));
```
And the reflected evaluation sequence on the concrete expression will be:

\[
\text{TIf (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@TIf _ _) \rightarrow (CaseB -> (lv \rightarrow LVvalue) = ' (i -> Int) '->' (e1 \rightarrow Exp)) (if2 \rightarrow CaseBs));}
\]

Though complex, these printing actions are in fact fairly straightforward: The first group of type Tiger \& Guard handles the enclosing guard-end pairs, distinguishes between single- and multi-branch cases, and delegates the latter case to the second group, which prints a list of branches recursively.

This is all we have to do — the corresponding parser is automatically derived and guaranteed to be consistent. Now guard expressions are desugared to nested if expressions in parsing and preserved in printing, and we can also resugar evaluation sequences on the ASTs to program text. For instance, the following guard expression

\[
\text{guard choice = 1 \rightarrow 4} \\
\text{choice = 2 \rightarrow 8} \\
\text{choice = 3 \rightarrow 16 end}
\]

is parsed to

\[
\text{TIf (TTop (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (J TIf (TTop (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (J TIf (TTop (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) Nothing)))}
\]

where TSimpleVar is shortened to TSV, and choice is shortened to c. Suppose that the value of the variable choice is 2. The evaluation sequence on the AST will then be:

\[
\text{TIf (TTop (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (J TIf (TTop (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (J TIf (TTop (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) Nothing)))}
\]

\[
\rightarrow \text{TIf (TInt 0) (TInt 4) (J TIf (TTop (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (J TIf (TTop (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) Nothing))})
\]

\[
\rightarrow \text{TIf (TInt 1) (TInt 8) (J TIf (TTop (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) Nothing}})
\]

\[
\rightarrow \text{TInt 8}
\]

And the reflected evaluation sequence on the concrete expression will be:

\[
\text{guard choice = 1 \rightarrow 4} \\
\text{choice = 2 \rightarrow 8} \\
\text{choice = 3 \rightarrow 16 end} \\
\rightarrow \text{guard choice = 2 \rightarrow 8} \\
\rightarrow \text{choice = 3 \rightarrow 16 end} \\
\rightarrow 8
\]

Reflective printing fails for the first and third steps (the program text becomes an if-then-else expression if we do printing at these steps), but this behaviour in fact conforms to Pombrio and Krishnamurthi’s “abstraction” property, which demands that core evaluation steps that make sense only in the core language must not be reflected to the surface. In our example, the first and third steps in the TIf-sequence evaluate the condition to a constant, but conditions in guard expressions are restricted to a specific form and cannot be a constant; evaluation of guard expressions thus has to proceed in bigger steps, throwing away or going into a branch in each step, which corresponds to two steps for TIf.

The reader may have noticed that, after the guard expression is reduced to two branches, the layout of the second branch is disrupted; this is because the second branch is in fact printed from scratch. In current BiYACC, the printing from an AST to a CST is accomplished by recursively performing pattern matching on both tree structures. This approach naturally comes with the disadvantage that the matching is mainly decided by the position of nodes in the AST and CST. Consequently, a minor structural change on the AST may completely disrupt the matching between the AST and the CST. We intend to handle this problem in the future.

6. Related Work

Comparison with our earlier prototype. This paper is primarily based on and significantly extends our previous tool demonstration paper (Zhu et al. 2015), which conducts an early experiment with the idea of casting parsing and reflective printing in the framework of bidirectional transformations. Compared to our previous prototype, the current system has the following improvements:

• The well-behavedness of the underlying bidirectional language BiGUL (Section 3) we use in this version is formally verified (Ko et al. 2016), so we can guarantee the well-behavedness of BiYACC programs much more confidently;

• concrete parsers and printers between program text and CSTs are automatically derived (Section 4.2), so BiYACC synchronises program text and ASTs, not just CSTs and ASTs as with the previous version;

• we present many nontrivial applications such as resugaring and language evolution (Section 5);

• our current system tries to preserve layouts and comments in a transparent manner.

Unifying parsing and printing. Much research has been devoted to describing parsers and printers in a single program. For example, both Rendel and Ostermann (2010) and Matsuda and Wang (2013) adopt a combinator-based approach, where small components are glued together to yield more sophisticated behaviour, and can guarantee inverse properties similar to (3) and (4) (with CST replaced by AST in the equations). In Rendel and Ostermann (2010)’s system (called “invertible syntax descriptions”, which we shorten to ISDs henceforth), both parsing and printing semantics are predefined in the combinators and the consistency is guaranteed by their partial isomorphisms, whereas in Matsuda and Wang (2013)’s system (called FlIppr), the combinators describing pretty printing are lately removed by a semantic-preserving transformation to a syntax, which is further processed by their grammar-based inversion system to achieve the parsing semantics. By specialising one of the syntax to be XML...
syntax, Brabrand et al. (2008) present a tool XSugar that handles bijection between the XML syntax and any other syntax for a grammar, guaranteeing that the transformation is reversible. However, the essential factor that distinguishes our system from others is that BiYACC is designed to handle synchronisation while others are all targeted at dealing with transformations.

Just like BiYACC, all of the systems described above handle unambiguous grammars without disambiguation declarations only. When the user-defined grammar (or the derived grammar) is ambiguous, the behaviour of these systems is as follows: Neither of ISDs and FlIPpr will notify the user that the (derived) grammar is ambiguous. For ISDs, property (4) will fail, while for FlIPpr both properties (3) and (4) will fail. (Since the discussion on ambiguous grammars has not been presented in their papers, we test the examples provided by their libraries.) In contrast, Brabrand et al. (2008) give a detailed discussion about ambiguity detection, and XSugar can statically check that the transformations are reversible. If any ambiguity in the program is detected, XSugar will notify the user of the precise location where ambiguity arises. In BiYACC, the ambiguity analysis is performed by the parser generator employed in the system, and the result is reported at compile time. If no warning is reported, the well-behavedness is always guaranteed, as explained in Section 4.2.1.

Even though all these systems handle unambiguous grammars only, there are design differences between them. An ISD is more like a parser, while FlIPpr lets the user describe a printer: To handle operator priorities, for example, the user of ISDs will assign priorities to different operators, consume parentheses, and use combinators such as chainl to handle left recursion in parsing, while the user of FlIPpr will produce necessary parentheses according to the operator priorities. In BiYACC, the user defines the concrete syntax that has a hierarchical structure (Expr, Term, and Factor) to express operator priority, and write printing strategies to produce (preserve) necessary parentheses. The user of XSugar will also likely need to use such a hierarchical structure.

It is interesting to note that, the part producing parentheses in FlIPpr essentially corresponds to the hierarchical structure of grammars. For example, to handle arithmetic expressions in FlIPpr, we can write:

\[
\text{ppr'} \ i \ (\text{Minus} \times \ y) = \\
\text{parenif} \ i \ (\text{>}> \ 6) \ & \ \text{group} \\
\text{ppr} \ 5 \ x \ <\ nst \ 2 \\
\text{(line' <> text '=-' <> space' <> ppr 6 y)};
\]

FlIPpr will automatically expand the definition and derive a group of ppr.i functions indexed by the priority integer i, corresponding to the hierarchical grammar structure. In other words, there is no need to specify the concrete grammar, which is already implicitly embedded in the printer program. This makes FlIPpr programs neat and concise. Following this idea, BiYACC programs can also be made more concise: In a BiYACC program, the user is allowed to omit the production rules in the concrete syntax part (or omit the whole concrete syntax part), and they will be automatically generated by extracting the terminals and nonterminals in the right-hand sides of all actions. However, if these production rules are supplied, BiYACC will perform some sanity checks: It will make sure that, in an action group, the user has covered all of the production rules of the nonterminal appearing in the “type declaration”, and never uses undefined production rules.

**Bidirectional transformations (BXs).** Our work is theoretically based on bidirectional transformations (Foster et al. 2007; Czarnecki et al. 2009; Hu et al. 2011), particularly taking inspiration from the recent progress on putback-based bidirectional programming (Pacheco et al. 2014a,b; Hu et al. 2014; Fischer et al. 2015; Ko et al. 2016). Also inspired by BXs, Martins et al. (2014) introduces an attribute grammar-based system for defining transformations between two representations of languages (two grammars). The utilisation is similar to BiYACC: The programmer defines both grammars and a set of rules specifying a get transformation, with a put transformation being automatically generated. One difference between Martins et al.’s system and ours is that their system performs synchronisation between two trees rather than program text and AST, and thus the preservation of layouts and comments is not considered. Another difference lies in the fact that, with their get-based system, certain decisions on the backward transformation are, by design, permanently encoded in the bidirectionalisation system and cannot be controlled by the user, whereas a putback-based system such as BiYACC can give the user fuller control.

7. **Conclusion**

We have presented the design and implementation of BiYACC, with which the programmer can describe both a parser and a reflective printer for an unambiguous context-free grammar in a single program. Our solution is enriched by its background of a putback-based bidirectional transformation framework which allows the guarantee of consistency properties by construction. This system can support various tasks of language engineering, from traditional constructions of basic machinery such as printers and parsers to more complex tasks such as resugar, simple refactor, and language evolution.

We plan to extend BiYACC to support disambiguation declarations. In many scenarios, it is more convenient for users to use ambiguous grammar with additional disambiguation declarations commonly found in parser generators such as YACC and HAPPY. We are also constructing techniques to provide better usability to the tool, namely through the detection of possible inconsistencies on the user provided inputs which can now only be detected at runtime.

**Acknowledgments**

We would like to thank reviewers for their valuable comments. This work was partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (A) No. 25240009.
References


