New Generation Computing manuscript No.
(will be inserted by the editor)

Bidirectional Parsing and Reflective Printing for Fully Disambiguated Grammars

Zirun Zhu · Hsiang-Shang Ko · Yongzhe Zhang · Pedro Martins · João Saraiva · Zhenjiang Hu

Received: date / Accepted: date

Abstract Language designers usually need to implement parsers and printers. Despite being two closely 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 a 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 propagate 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 a fully disambiguated context-free grammar. BIYACC is based on the theory of bidirectional transformations, which helps to guarantee by construction that the generated pairs of parsers and reflective

Zirun Zhu
National Institute of Informatics, Japan
E-mail: zhu@nii.ac.jp

Hsiang-Shang Ko
National Institute of Informatics, Japan
E-mail: hsiang-shang@nii.ac.jp

Yongzhe Zhang
National Institute of Informatics, Japan
E-mail: zyz915@nii.ac.jp

Pedro Martins
University of California, Irvine, USA
E-mail: pribeiro@uci.edu

João Saraiva
University of Minho, Portugal
E-mail: jas@di.uminho.pt

Zhenjiang Hu
National Institute of Informatics, Japan
E-mail: hu@nii.ac.jp

printers are consistent. Handling grammatical ambiguity is particularly challenging: we propose an approach based on generalised parsing and disambiguation filters, which produce all the parse results and (try to) select the only correct one in the parsing direction; the filters are carefully bidirectionalised so that they also work in the printing direction and do not break the consistency between the parsers and reflective printers. We show that BIYACC is capable of facilitating many tasks such as Pombrio and Krishnamurthi's 'resugaring', simple refactoring, and language evolution.

Keywords Bidirectional Transformations · Disambiguation Filters · Domain-Specific Languages · Parsing · Reflective Printing

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 closely 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 [6, 14, 35, 41, 49] 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 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 back to the program text to make it easy for the user to check where the changes are. With a conventional printer, 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

Original program text:

```
-a /* a 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 /* a is the variable denoting... */
* (2 + (a))
```

Fig. 1 Comparison between conventional printing and reflective printing.

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 [12, 28],
- in program refactoring where the comments and layouts in the original program are completely lost after the AST is transformed by the system [23], and
- in language-based editors, as introduced by Reps [42, 43], where the user needs to interact with different printed representations of the same underlying AST.

To address the problem, we propose a domain-specific language BIYACC, which enables the user describe both a parser and a *reflective* printer for a fully disambiguated context-free grammar (CFG) 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.

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.[1]

$$parse\ (print\ s\ t) = t\ . \tag{1}$$

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.

$$print\ s\ (parse\ s) = s\ . \tag{2}$$

These two properties are inspired by the theory of *bidirectional transformations* [19], in particular *lenses* [18], and are guaranteed by construction for all BIYACC programs.

[1] Throughout this paper, we typeset general definitions and properties in *math style* and specific examples in `code style`.

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 the test cases used in the paper. The structure of the paper is as follows: We start with 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 [21, 26, 27], we first give the semantics of a basic version of BIYACC that deals with only unambiguous grammars by compiling it to BIGUL in Section 4, guaranteeing the properties (1) and (2) by construction. Then, inspired by the research on *generalised parsing* [47] and disambiguation filters [25], we revise the basic BIYACC architecture in Section 5 to allow the use of ambiguous grammars and disambiguation directives while still retaining the above-mentioned properties. We present a case study in Section 6, showing that BIYACC is capable of describing TIGER [4], which shares many similarities with fully fledged languages. We demonstrate that BIYACC can handle syntactic sugar, partially subsume Pombrio and Krishnamurthi's 'resugaring' [39, 40], and facilitate language evolution. In Section 7, we present detailed related work including comparison with other systems. Finally, contributions and future work are summarised in Section 8.

This is the extended version of our previous work *Parsing and Reflective Printing, Bidirectionally* presented at SLE'16 [51], and the differences in a nutshell are as follows: (i) We propose the notion of bidirectionalised filters and integrate them into BIYACC for handling grammatical ambiguity (Section 5); the related work section is also updated accordingly. (ii) We restructure the narration for introducing the basic BIYACC system and in particular elaborate on the isomorphism between program text and CSTs. (iii) We present the definitions and theorems in a more formal way, and complete their proofs. (iv) We make several other revisions such as renewing the figures for introducing the BIYACC system and the syntax of BIYACC programs.

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 the arithmetic expression example mentioned in Section 1. This program consists of definitions of the abstract syntax, concrete syntax, directives, and actions for reflectively printing ASTs to CSTs.

2.1 Syntax Definitions

Abstract Syntax. The abstract syntax part — which starts with the keyword `#Abstract` — is just one or more definitions of Haskell datatypes. In our example, the abstract syntax is defined in lines 2–7 by a single datatype `Arith` whose elements are constructed from constants and arithmetic operators.

```
1  #Abstract
2  data Arith = Num Int
3             | Var String
4             | Add Arith Arith
5             | Sub Arith Arith
6             | Mul Arith Arith
7             | Div Arith Arith
8
9  #Concrete
10 Expr   -> Expr '+' Term
11         | Expr '-' Term
12         | Term ;
13
14 Term   -> Term '*' Factor
15         | Term '/' Factor
16         | Factor ;
17
18 Factor -> '-' Factor
19         | Numeric
20         | Identifier
21         | '(' Expr ')' ;
```

```
28 #Directives
29 LineComment:  "//"       ;
30 BlockComment: "/*"  "*/" ;
31
32 #Actions
33 Arith +> Expr
34   Add x y  +>  [x +> Expr] '+' [y +> Term];
35   Sub x y  +>  [x +> Expr] '-' [y +> Term];
36   e        +>  [e +> Term];
37 ;;
38 Arith +> Term
39   Mul x y  +>  [x +> Term] '*' [y +> Factor];
40   Div x y  +>  [x +> Term] '/' [y +> Factor];
41   e        +>  [e +> Factor];
42 ;;
43 Arith +> Factor
44   Sub (Num 0) y  +>  '-' [y +> Factor];
45   Num i          +>  [i +> Numeric];
46   Var n          +>  [n +> Identifier];
47   e              +>  '(' [e +> Expr] ')';
48 ;;
```

Fig. 2 A BIYACC program for the expression example.

Concrete Syntax. On the other hand, the concrete syntax — which is defined in the second part beginning with the keyword `#Concrete` — is defined by a context-free grammar. For our expression example, in lines 10–21 we use a standard unambiguous grammatical structure to encode operator precedence and order of association, which involves three nonterminal symbols `Expr`, `Term`, and `Factor`: An `Expr` can produce a left-sided tree of `Term`s, each of which can in turn produce a left-sided tree of `Factor`s. To produce right-sided 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 program text. (There are also predefined nonterminals *Numeric* and *Identifier*, which produce numeric values and identifiers respectively.) BIYACC also supports ambiguous grammars with disambiguation directives (Section 5) — for example, an ambiguous version of the expression grammar is shown in Figure 6.

Directives. The `#Directives` part is for defining the syntax of comments and disambiguation directives. For example, line 29 shows that the syntax for a single line comments is '`//`', while line 30 states that '`/*`' and '`*/`' are respectively the beginning mark and ending mark for a block comment. Since the grammar for arithmetic expressions is unambiguous, there is no need to give any disambiguation directive for this example (whereas the ambiguous version of the grammar in Figure 6 needs to be augmented with a few disambiguation directives).

2.2 Printing Actions

The main part of a BIYACC program starts with the keyword `#Actions`, and describes how to update a CST with an AST. For our expression example, the actions are

defined in lines 34–48 in Figure 2. Before explaining the actions, we should first emphasise that program text is identified with CSTs when programming BIYACC actions: 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 4.3.1.

The `Actions` part consists of groups of actions, and each group begins with a 'type declaration' of the form *hsType* '+>' *nonterminal* stating that the actions in this group specify updates on CSTs generated from *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 nonterminals 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 34 of Figure 2:

```
Add x y +> [x +> Expr] '+' [y +> Term];
```

The AST pattern `Add x y` 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 `Numeric`. BIYACC'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 BIYACC 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.

Complex Patterns. By using more complex patterns, we can write actions that establish nontrivial relationships between CSTs and ASTs. For example, the action at line 44 of Figure 2 associates abstract subtraction expressions whose left operand is zero with concrete negated expressions; this action is the key to preserving negated expressions

in the CST. For an example of a more complex CST pattern: Suppose that we want to write a pattern that matches those CSTs 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 first nonterminal `Factor` (an additional pair of parentheses is required for the expanded nonterminal): `'-' (Factor -> '-' Factor)`. More examples involving this kind of deep patterns will be presented in Section 6.

Layout and Comment Preservation. The reflective printer generated by BIYACC 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 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.[2] 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.

Usefulness of the Concrete Syntax. Observant readers may have noticed that the production rules in the concrete syntax part resemble the right-hand sides of the actions. In fact these production rules in the concrete syntax part can be omitted, and BIYACC will automatically generate them by extracting the right-hand sides of all actions. However, we recommend the users still write these production rules, so that BIYACC can perform some simple checks: It will make sure that, in an action group, the users have covered all of the production rules of the nonterminal appearing in the 'type declaration', and will never use undefined production rules.

Parsing Semantics. So far we have been describing the reflective printing semantics of the BIYACC 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 BIYACC 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.

[2] One might argue that layouts and comments should in fact be handled differently, since comments are usually attached to some entities which they describe. For example, when a function declaration is moved to somewhere else (e.g. by a refactoring tool), we will want the comment describing that function to be moved there as well. We leave the proper treatment of comments as future work.

3 Foundation of BIYACC: Putback-based Bidirectional Programming

From a BIYACC program, in addition to generating a parser and a printer, 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), in particular *lenses* [18]. We will give a brief introduction to BXs below; for a comprehensive treatment, the readers are referred to the lecture notes for the 2016 Oxford Summer School on Bidirectional Transformations [19].

3.1 Parsing and Printing as Lenses

The *parse* and *print* semantics of BIYACC programs are potentially *partial* — for example, if the actions in a BIYACC 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 [31, 37] of asymmetric lenses [18].

Definition 1 (Lenses) A *lens* between a *source* type S and a *view* type V is a pair of functions

$$\begin{aligned} &\mathit{get} :: S \to \mathsf{Maybe}\, V \\ &\mathit{put} :: S \to V \to \mathsf{Maybe}\, S \end{aligned}$$

satisfying the *well-behavedness* laws:

$$\mathit{put}\ s\ v = \mathsf{Just}\ s' \quad \Rightarrow \quad \mathit{get}\ s' = \mathsf{Just}\ v \qquad \text{(PUTGET)}$$

$$\mathit{get}\ s = \mathsf{Just}\ v \quad \Rightarrow \quad \mathit{put}\ s\ v = \mathsf{Just}\ s \qquad \text{(GETPUT)}$$

Intuitively, a *get* function extracts a part of a source of interest to the user as a view, and a *put* function takes a source and a view and produces an updated source incorporating information from the view. Partiality is explicitly represented by making the functions return Maybe-values: a *get* or *put* function returns Just r where r is the result, or Nothing if the input is not in the domain. 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 lens, required by definition to satisfy the two well-behavedness laws, which are exactly the consistency properties (1) and (2) reformulated in a partial setting.

3.2 Putback-based Bidirectional Programming in BiGUL

Having rephrased parsing and printing in terms of lenses, we can now 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 lens. Specifically, BiYacc programs are compiled to the *putback-based* bidirectional programming language BiGUL [27]. It has been formally verified in Agda [36] that BiGUL programs always denote well-behaved lenses, and BiGUL has been ported to Haskell as an embedded DSL library [21], which will be introduced in more detail in Section 3.2. 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 fact that, given a *put* function, there is at most one *get* function that forms a (well-behaved) lens with this *put* function [17]. That is, once we describe a *put* function as a BiGUL program, the *get* semantics of the program is completely determined by its *put* semantics. We can therefore focus solely on the printing (*put*) behaviour, leaving the parsing (*get*) behaviour only implicitly (but unambiguously) specified. How the programmer can effectively work with this paradigm has been more formally explained in terms of a Hoare-style logic for BiGUL [26].

Compilation of BiYacc to BiGUL (Section 4) uses only three BiGUL operations, which we briefly introduce here; more details can be found in the lecture notes on BiGUL programming [21]. A BiGUL program has type BiGUL *s v*, where *s* and *v* are respectively the source and view types.

Replace. The simplest BiGUL operation we use is

$$\mathsf{Replace} :: \mathsf{BiGUL}\ s\ s$$

which discards the original source and returns the view — which has the same type as the source — as the updated source. That is, the *put* semantics of Replace is the function $\lambda\ s\ v \to \mathsf{Just}\ v$.

Update. The next operation `update` is more complex, and is implemented with the help of Template Haskell [46]. The general form of the operation is

$$\$(\mathit{update}\ [p|\ \mathit{spat}\ |]\ [p|\ \mathit{vpat}\ |]\ [d|\ \mathit{bs}|]) :: \mathsf{BiGUL}\ s\ v\ .$$

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

```
$(update [p| (x, _) |] [p| x |] [d| x = 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:

$$\mathsf{Case} :: [\mathsf{Branch}\ s\ v] \to \mathsf{BiGUL}\ s\ v\ .$$

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 $spat'$:

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

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

$$\$(adaptiveSV\ [p|\ spat\ |]\ [p|\ vpat\ |]) :: (s \to v \to s) \to \mathsf{BiGUL}\ s\ v\ .$$

Their semantics in the *put* direction 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 source should satisfy the exit condition $spat'$ (or otherwise execution fails); 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 The Basic BIYACC

In this section we expound on a basic version of BIYACC that handles only unambiguous grammars. (Section 5 will present extensions for dealing with ambiguous grammars with disambiguation.) The architecture is illustrated in Figure 3, where a BIYACC program

$$\text{`}\texttt{\#Abstract}\text{'}\ decls\ \text{`}\texttt{\#Concrete}\text{'}\ pgs\ \text{`}\texttt{\#Directives}\text{'}\ drtvs\ \text{`}\texttt{\#Actions}\text{'}\ ags\ , \quad (3)$$

consisting of abstract syntax, concrete syntax, directives, and printing actions, as formally defined in Figure 4, is compiled into a single executable file (by GHC) for converting between program text and ASTs. Specifically:

- The abstract syntax part (*decls*) is already valid Haskell code and is (almost) directly used as the definitions of AST datatypes.
- The concrete syntax part (*pgs*) is translated to definitions of CST datatypes (whose elements are representations of how a string is produced using the production rules), and also used to generate the pair of *concrete* parser (including a lexer) and printer for the conversion between program text and CSTs. This pair of concrete parser and printer can be shown to form a (partial) isomorphism.

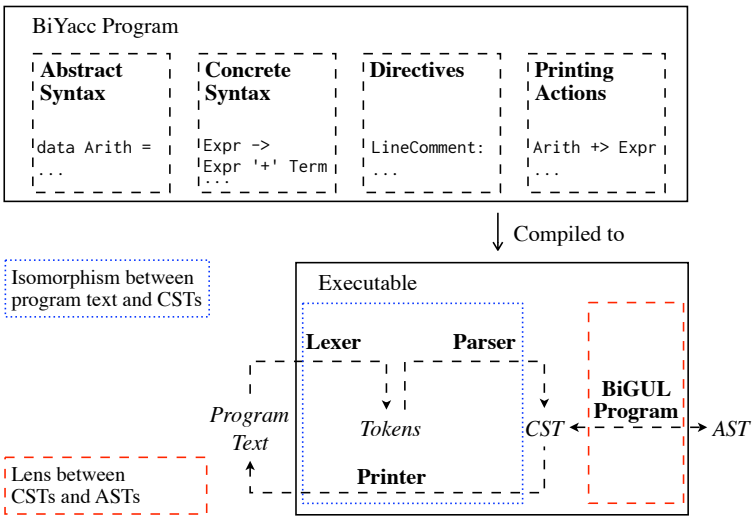

Fig. 3 Architecture of BIYACC.

- The directives part (*drtvs*) is used in the lexer for recognising single-line and multi-line comments.
- The printing actions part (*ags*) is translated to a BIGUL program (which is a lens) for handling (the semantic part of) parsing and reflective printing between CSTs and ASTs.

The whole executable is a lens because it is the composition of the concrete parsing and printing isomorphism and the lens arising from the BIGUL program. After giving the general definitions of isomorphism and composition in Section 4.1, we discuss the CST datatypes and concrete printers, the concrete parsing and printing isomorphism, and the BIGUL lens in the executable respectively in Sections 4.2, 4.3, and 4.4.

4.1 Composition of Isomorphisms and Lenses

We start from the definition of isomorphisms.

Definition 2 (Isomorphism) A (partial) *isomorphism* between two types A and B is a pair of functions

$$\begin{aligned} \mathit{to} &:: A \to \mathsf{Maybe}\ B \\ \mathit{from} &:: B \to \mathsf{Maybe}\ A \end{aligned}$$

such that

$$\mathit{to}\ a = \mathsf{Just}\ b \quad \Leftrightarrow \quad \mathit{from}\ b = \mathsf{Just}\ a\ .$$

Given an isomorphism (*to* and *from*) between A and B and a lens (*get* and *put*) between B and C, we can compose them to form a new lens between A and C, whose

$$
\begin{aligned}
\mathit{Program} &::= \text{`\texttt{\#Abstract}'}\ \mathit{HsDeclarations} \\
&\quad [\text{`\texttt{\#Concrete}'}\ \mathit{ProductionGroup}^+] \\
&\quad \text{`\texttt{\#Directives}'}\ \mathit{CommentSyntaxDecl}\ \mathit{Disambiguation} \\
&\quad \text{`\texttt{\#Actions}'}\ \mathit{ActionGroup}^+ \\
&\quad [\text{`\texttt{\#OtherFilters}'}\ \mathit{OtherFilters}] \\
\mathit{ProductionGroup} &::= \mathit{Nonterminal}\ \text{`\texttt{->}'}\ \mathit{ProductionBody}^+\{\text{`\texttt{|}'}\}\ \text{`\texttt{;}'} \\
\mathit{ProductionBody} &::= [\text{`\texttt{[}'}\mathit{Constructor}\text{`\texttt{]}'}]\ \mathit{Symbol}^+\ [\text{`\texttt{\{\#}'}\text{`\texttt{Bracket}'}\text{`\texttt{\#\}}'}] \\
\mathit{Symbol} &::= \mathit{Primitive} \mid \mathit{Terminal} \mid \mathit{Nonterminal} \\
\mathit{Constructor} &::= \mathit{Nonterminal} \\
\mathit{CommentSyntaxDecl} &::= \text{`\texttt{LineComment:}'}\ \mathit{String}\ \text{`\texttt{;}'}\ \text{`\texttt{BlockComment:}'}\ \mathit{String}\ \text{`\texttt{;}'} \\
\mathit{Disambiguation} &::= [\mathit{Priority}]\ [\mathit{Associativity}] \\
\mathit{ActionGroup} &::= \mathit{HsType}\ \text{`\texttt{+>}'}\ \mathit{Nonterminal} \\
&\quad \mathit{Action}^+\ \text{`\texttt{;;}'} \\
\mathit{Action} &::= \mathit{HsPattern}\ \text{`\texttt{+>}'}\ \mathit{Update}^+\ \text{`\texttt{;}'} \\
\mathit{Update} &::= \mathit{Symbol} \\
&\quad \mid \text{`\texttt{[}'}\ \mathit{HsVariable}\ \text{`\texttt{+>}'}\ \mathit{UpdateCondition}\ \text{`\texttt{]}'} \\
&\quad \mid \text{`\texttt{(}'}\ \mathit{Nonterminal}\ \text{`\texttt{->}'}\ \mathit{Update}^+\ \text{`\texttt{)}'} \\
\mathit{UpdateCondition} &::= \mathit{Symbol} \\
&\quad \mid \text{`\texttt{(}'}\ \mathit{Nonterminal}\ \text{`\texttt{->}'}\ \mathit{UpdateCondition}^+\ \text{`\texttt{)}'}
\end{aligned}
$$

Fig. 4 Syntax of BiYacc programs (Nonterminals with prefix *Hs* denote Haskell entities and follow the Haskell syntax; the notation $nt^+\{sep\}$ denotes a nonempty sequence of the same nonterminal *nt* separated by *sep*. Optional elements are enclosed in a pair of square brackets. The parts relating to disambiguation and filters will be explained in Section 5.)

components get' and put' are defined by

$$
\begin{aligned}
&get' \quad :: A \to \mathsf{Maybe}\ C \\
&get'\ a \quad = to\ a \gg\!= get \\
&put' \quad :: A \to C \to \mathsf{Maybe}\ A \\
&put'\ a\ c = to\ a \gg\!= \lambda\ b \to put\ b\ c \gg\!= from
\end{aligned}
$$

where

$$
\begin{aligned}
&(\gg\!=) \quad :: \mathsf{Maybe}\ a \to (a \to \mathsf{Maybe}\ b) \to \mathsf{Maybe}\ b \\
&\mathsf{Just}\ x \quad \gg\!= f = f\ x \\
&\mathsf{Nothing} \gg\!= f = \mathsf{Nothing}\ .
\end{aligned}
$$

This is specialised from the standard definition of lens composition [18] — an isomorphism can be lifted to a lens (whose *put* component passes its second argument to the *from* component of the isomorphism), which can then be composed with another lens to give rise to a new lens. We omit the (standard) proof that the new lens is indeed well-behaved.

4.2 Generating CST Datatypes and Concrete Printers

The production rules in a context-free grammar dictate how to produce strings from nonterminals, and a CST can be regarded as encoding one particular way of producing

a string using the production rules. In BIYACC, we represent CSTs starting from a nonterminal *nt* as an automatically generated Haskell datatype named *nt*, whose constructors represent the production rules for *nt*. For each of these datatypes, we also generate a printing function which takes a CST as input and produces a string as dictated by the production rules in the CST.

For instance, in Figure 2, the group of production rules from the nonterminal `Factor` (lines 18–21) is translated to the following Haskell datatype and concrete printing function:

```
data Factor = Factor1 String Factor
            | Factor2 (String, String)
            | Factor3 (String, String)
            | Factor4 String Expr String
            | FactorNull

cprintFactor :: Factor -> String
cprintFactor (Factor1 s1 factor1)       = "-" ++ s1 ++ cprintFactor factor1
cprintFactor (Factor2 (numeric, s1))    = numeric ++ s1
cprintFactor (Factor3 (identifier, s1)) = identifier ++ s1
cprintFactor (Factor4 s1 expr1 s2)      = "(" ++ s1 ++ cprintExpr expr1 ++
                                          ")" ++ s2
cprintFactor FactorNull                 = ""

-- cprintExpr :: Expr -> String
```

where `Factor1` ... `Factor4` are constructors corresponding to the four production rules, while `FactorNull` represents an empty CST of type *Factor* and is used as the default value whenever we want to create new program text depending on the view only. As an example, `Factor1` represents the production rule `Factor -> '-' Factor`, and its `String` field stores the whitespaces appearing after a negation sign in the program text.

Following this idea, we define the translation from production rule groups (*pgs* in (3)) to datatype definitions by source-to-source compilation

$$
\begin{aligned}
&[\![pgs]\!]_{ProductionGroup} = \left\langle [\![pg]\!]_{ProductionGroup} \;\middle|\; pg \in pgs \right\rangle \\
&[\![nt \text{ `->'} bodies]\!]_{ProductionGroup} = \\
&\quad \text{`data'}\ nt\ \text{`='} \left\langle \textsc{con}(nt, body)\ \left\langle \textsc{field}(s) \;\middle|\; s \in body \right\rangle\ \text{`|'} \;\middle|\; body \in bodies \right\rangle \\
&\qquad \textsc{nullCon}(nt)
\end{aligned}
$$

where compilation rules are defined with the semantic bracket ($[\![\cdot]\!]$), 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. 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*. In more detail: $\textsc{con}(nt, body)$ retrieves the constructor for a production rule. 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 FIELD — nonterminals that are not primitives are left unchanged (using their names for datatypes), primitives are converted to the `String` type [3], terminal symbols are dropped, and an additional `String` field is added for each terminal and

[3] The reason for converting primitives to the `String` type is because `String` is the most precise representation that will not cause the loss of any information. For instance, this is useful for retaining the leading zeros of an integer such as 073. Storing 073 as `Integer` will cause the loss of the leading zero.

primitive for storing layout information (whitespaces and comments) appearing after the terminal or primitive in the program text. The last step is to insert an additional empty constructor, whose name is denoted by NULLCON(nt).

4.3 The Concrete Parsing and Printing Isomorphism

Let the type CST be the set of all the CSTs defined by the grammar of a BIYACC program; by default it is the source type (nonterminal) of the first group of actions in the `#Action` part. We have seen in Section 4.2 how to generate its datatype definition and a concrete printing function

$$\mathit{cprint} :: \mathsf{CST} \to \mathsf{String} \ .$$

On the other hand, from the grammar we can straightforwardly use a parser generator to generate a concrete parsing function

$$\mathit{cparse} :: \mathsf{String} \to \mathsf{Maybe\ CST} \ ,$$

which is Maybe-valued since a piece of input text can be invalid. This *cparse* function is one direction of the isomorphism in the executable, while the other direction is

$$\mathsf{Just} \circ \mathit{cprint} :: \mathsf{CST} \to \mathsf{Maybe\ String} \ .$$

Below we show in Section 4.3.1 that the inverse properties amount to the requirements that the generated parser is 'correct' and the grammar is unambiguous, and introduce in Section 4.3.2 the implementation of *cparse*.

4.3.1 The Inverse Properties

Our concrete parsers are generated by some parser generator (e.g. HAPPY) and we need to assume that they satisfy some essential properties, for we cannot control the generation process and verify those properties.

Definition 3 (Parser Correctness) A parser *cparse* is *correct* with respect to a printer *cprint* exactly when

$$\mathit{cparse}\ \mathit{text} = \mathsf{Just}\ \mathit{cst} \quad \Rightarrow \quad \mathit{cprint}\ \mathit{cst} = \mathit{text} \tag{4}$$

$$\mathit{cprint}\ \mathit{cst} = \mathit{text} \quad \Rightarrow \quad \exists\, \mathit{cst}'.\ \mathit{cparse}\ \mathit{text} = \mathsf{Just}\ \mathit{cst}' \ . \tag{5}$$

To see what (4) means, recall that our CSTs, as described in Section 4.4, encode precisely the derivation trees, with the CST constructors representing the production rules used, and *cprint* traverses the CSTs and follows the encoded production rules to produce the derived program text. Now consider what *cparse* is supposed to do: It should take a piece of program text and find a derivation tree for it, i.e. a CST which *cprint*s to that piece of program text. This statement is exactly (4). In other words, (4) is the functional specification of parsing, which is satisfied if the parser generator we use behaves correctly. Also it is reasonable to expect that a parser will be able to successfully parse any valid program text and this is exactly (5).

We can now establish the isomorphism (which is rather straightforward).

Theorem 1 *If a parser cparse is correct with respect to an injective printer cprint, then cparse and* $\mathsf{Just} \circ cprint$ *form an isomorphism, that is,*

$$cparse\ text = \mathsf{Just}\ cst \quad \Leftrightarrow \quad (\mathsf{Just} \circ cprint)\ cst = \mathsf{Just}\ text\ .$$

Proof The left-to-right direction is immediate since the right-hand side is equivalent to $cprint\ cst = text$, and the whole implication is precisely (4). For the right-to-left direction, again the antecedent is equivalent to $cprint\ cst = text$, and we can invoke (5) to obtain $cparse\ (text) = \mathsf{Just}\ cst'$ for some cst'. This is already close to our goal — what remains to be shown is that cst' is exactly cst, which is indeed the case because

$$\begin{array}{ll} & cparse\ text = \mathsf{Just}\ cst' \\ \Rightarrow & \{\,\text{antecedent}\,\} \\ & cparse\ (cprint\ cst) = \mathsf{Just}\ cst' \\ \Rightarrow & \{\,(4)\,\} \\ & cprint\ cst' = cprint\ cst \\ \Rightarrow & \{\,cprint\ \text{is injective}\,\} \\ & cst' = cst\ . \end{array}$$

□

Note that the injectivity of *cprint* amounts to saying that the grammar is unambiguous, since for any piece of program text there is at most one CST that prints to it.

4.3.2 Concrete Lexer and Parser

In the current BIYACC, the implementation of the *cparse* 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.

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 integers, strings, and identifiers, respectively produced by the nonterminals `Numeric`, `String`, and `Identifier`. For now, the forms of these literals are pre-defined, but we take this as a step towards a lexerless grammar, in which strings produced by nonterminals can be specified in terms of regular expressions. Furthermore, whitespaces and comments are carefully handled 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.

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 HAPPY [32], which takes a BNF specification of a grammar with semantic actions and produces a Haskell module containing a parser function. The grammar we feed into HAPPY is still essentially the one specified in a BIYACC program, but in addition to parsing and constructing CSTs, the HAPPY actions also transfer the whitespaces wrapped in tokens to corresponding places in the CSTs. For example, the production rules for `Factor` in the expression example, as shown on the left below, are translated to the HAPPY specification on the right:

```
Factor                          Factor
  -> '-' Factor                   : token1 Factor
                                      { Factor1 $1 $2 }
   | Numeric                      | tokenNumeric
                            ⇝         { Factor2 $1 }
   | Identifier                   | tokenIdentifier
                                      { Factor3 $1 }
   | '(' Expr ')';                | token1 Expr token2
                                      { Factor4 $1 $2 $3 } .
```

We use the first expansion (`token1 Factor`) to explain how whitespaces are transferred: The generated HAPPY token `token1` 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 `Factor1` by the associated Haskell action.

4.4 Generating the BIGUL Program

Since we identify program text with CSTs, the semantics of a BIYACC program can be defined by the synchronisation between CSTs and ASTs — which is handled by the generated BIGUL program — instead of the (complete) synchronisation between program text and ASTs. This section will give the source-to-source compilation from a BIYACC program

'`#Abstract`' *decls* '`#Concrete`' *pgs* '`#Directives`' *drtvs* '`#Actions`' *ags*

to a BIGUL program, as shown in Figure 5. Additional arguments to the semantic bracket are typeset in superscript. We explain the compilation from the following aspects. (We already handled production groups (*pgs*) in Section 4.2.)

Action Groups. Each group of actions is translated into a small BIGUL program, whose name is determined by the view type *vt* and source type *st* and denoted by $\textsc{prog}(vt, st)$. The BIGUL program has one single `Case` statement, and each action is translated into two branches in this `Case` statement, one normal and the other adaptive. All the adaptive branches are gathered in the second half of the `Case` statement, so that

$$[\![\texttt{`Abstract'}\ decls\ \texttt{`Concrete'}\ pgs\ \texttt{`Action'}\ ags]\!]_{Program} = decls\ \langle [\![pg]\!]_{ProductionGroup} \mid pg \in pgs\rangle\ \langle [\![ag]\!]_{ActionGroup} \mid ag \in ags\rangle$$

$$[\![vt\ \texttt{`+>'}\ st\ acts]\!]_{ActionGroup} = \textsc{prog}(vt,st)\ \texttt{`::'}\ \texttt{`BiGUL'}\ st\ vt\ \ \textsc{prog}(vt,st)\ \texttt{`='}\ \texttt{`Case'}\ \ \texttt{`['}\ \langle [\![a]\!]^{\mathsf{N},vt,st}_{Action}\ \texttt{`,'} \mid a \in acts\rangle\ \langle [\![a]\!]^{\mathsf{A},st}_{Action} \mid a \in acts\rangle\{\texttt{`,'}\}\ \texttt{`]'}$$

$$[\![vpat\ \texttt{`+>'}\ updates]\!]^{\mathsf{N},vt,st}_{Action} = \texttt{`\$(normalSV'}\ \texttt{`[p|'}\ \textsc{srcCond}(\textsc{ersVars}(\texttt{`['}\ st\ \texttt{`->'}\ updates\ \texttt{`]'})_{Update})\ \texttt{`|]'}\ \texttt{`[p|'}\ vpat\ \texttt{`|]'}\ \texttt{`[p|'}\ \textsc{srcCond}(\textsc{ersVars}(\texttt{`('}\ st\ \texttt{`->'}\ updates\ \texttt{`)'})_{Update})\ \texttt{`|])'}\ \texttt{`\$(update'}\ \texttt{`[p|'}\ \textsc{removeAs}(vpat)\ \texttt{`|]'}\ \texttt{`[p|'}\ \textsc{srcPat}(\texttt{`('}\ st\ \texttt{`->'}\ updates\ \texttt{`)'})_{Update}\ \texttt{`|]'}\ \texttt{`[d|'}\ \langle [\![u]\!]^{vt,vpat}_{Update} \mid u \in updates\rangle\ \texttt{`|])'}$$

$$[\![\texttt{`['}\ var\ \texttt{`+>'}\ uc_{Primitive}\ \texttt{`]'}]\!]^{vt,vpat}_{Update} = var\ \texttt{`= Replace;'}$$

$$[\![\texttt{`['}\ var\ \texttt{`+>'}\ uc_{Nonterminal}\ \texttt{`]'}]\!]^{vt,vpat}_{Update} = var\ \texttt{`='}\ \textsc{prog}(\textsc{varType}(vt,vpat,var),uc)\ \texttt{`;'}$$

$$[\![\texttt{`['}\ var\ \texttt{`+>'}\ \texttt{`('}\ nt\ \texttt{`->'}\ \ldots\texttt{`)'}\ \texttt{`]'}]\!]^{vt,vpat}_{Update} = [\![\texttt{`['}\ var\ \texttt{`+>'}\ nt\ \texttt{`]'}]\!]^{vt,vpat}_{Update}$$

$$[\![\texttt{`('}\ \ldots\ \texttt{`->'}\ updates\ \texttt{`)'}]\!]^{vt,vpat}_{Update} = \langle [\![u]\!]^{vt,vpat}_{Update}\ \texttt{`;'} \mid u \in updates\rangle$$

$$[\![symbol]\!]^{vt,vpat}_{Update} = \texttt{``}$$

$$[\![vpat\ \texttt{`+>'}\ updates]\!]^{\mathsf{A},st}_{Action} = \texttt{`\$(adaptiveSV'}\ \texttt{`[p| _ |]'}\ \texttt{`[p|'}\ vpat\ \texttt{`|])'}\ \texttt{`(\_ _ ->'}\ \textsc{defaultExpr}(\textsc{ersVars}(\texttt{`('}\ st\ \texttt{`->'}\ updates\ \texttt{`)'}))$$

$$\textsc{field}(nt)_{Nonterminal} = nt$$
$$\textsc{field}(t)_{Terminal} = \texttt{`String'}$$
$$\textsc{field}(p)_{Primitive} = \texttt{`(}\ p\ \texttt{`, String)'}$$

$$\textsc{ersVars}(\texttt{`['}\ var\ \texttt{`+>'}\ uc\ \texttt{`]'})_{Update} = uc$$
$$\textsc{ersVars}(\texttt{`('}\ nt\ \texttt{`->'}\ updates\ \texttt{`)'})_{Update} = \texttt{`('}\ nt\ \texttt{`->'}\ \langle \textsc{ersVars}(u) \mid u \in updates\rangle\ \texttt{`)'}$$
$$\textsc{ersVars}(symbol)_{Update} = symbol$$

$$\textsc{srcCond}(\texttt{`('}\ nt\ \texttt{`->'}\ uconds\ \texttt{`)'})_{UpdateCondition} = \texttt{`('}\ \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in uconds\rangle)\ \langle \textsc{srcCond}(uc) \mid uc \in uconds\rangle\ \texttt{`)'}$$
$$\textsc{srcCond}(symbol)_{UpdateCondition} = \texttt{`\_'}$$

$$\textsc{condHead}(\texttt{`('}\ nt\ \texttt{`->'}\ \ldots\ \texttt{`)'})_{UpdateCondition} = nt$$
$$\textsc{condHead}(symbol)_{UpdateCondition} = symbol$$

$$\textsc{srcPat}(\texttt{`['}\ var\ \texttt{`+>'}\ uc_{Primitive}\ \texttt{`]'})_{Update} = \texttt{`('}\ var\ \texttt{`, \_)'}$$
$$\textsc{srcPat}(\texttt{`['}\ var\ \texttt{`+>'}\ uc_{Nonterminal}\ \texttt{`]'})_{Update} = var$$
$$\textsc{srcPat}(\texttt{`('}\ nt\ \texttt{`->'}\ updates\ \texttt{`)'})_{Update} = \texttt{`('}\ \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in \textsc{ersVars}(updates)\rangle)\ \langle \textsc{srcPat}(u) \mid u \in updates\rangle\ \texttt{`)'}$$
$$\textsc{srcPat}(symbol)_{Symbol} = \texttt{`\_'}$$

$$\textsc{defaultExpr}(symbol)_{Primitive} = \texttt{`(undefined, "")'}$$
$$\textsc{defaultExpr}(symbol)_{Nonterminal} = \textsc{nullCon}(symbol)$$
$$\textsc{defaultExpr}(symbol)_{Terminal} = \texttt{`""'}$$
$$\textsc{defaultExpr}(\texttt{`('}\ nt\ \texttt{`->'}\ uconds\ \texttt{`)'})_{UpdateCondition} = \textsc{con}(nt, \langle \textsc{condHead}(uc) \mid uc \in uconds\rangle)\ \langle \textsc{defaultExpr}(uc) \mid uc \in uconds\rangle$$

Fig. 5 Semantics of BiYacc programs (as BiGUL programs).

normal branches will be tried first. For example, the third group of type `Arith +> Factor` is compiled to

```
bigulArithFactor :: BiGUL Factor Arith
bigulArithFactor = Case [...] .
```

Normal Branches. We said in Section 2 that the semantics of an action is to perform pattern matching on both the source and view, and then update parts of the source with parts of the view. This semantics is implemented with a normal branch: The source and view patterns are compiled to the entry condition, and, together with the updates overlaid on the source pattern, also to an `update` operation. For example, the first action in the `Arith—Factor` group

```
Sub (Num 0) y +> '-' (y +> Factor)
```

is compiled to

```
$(normalSV [p| (Factor1 _ _) |] [p| Sub (Num 0) y |]
           [p| (Factor1 _ _) |])
  $(update [p| Sub (Num 0) y |] [p| (Factor1 _ y) |]
           [d| y = bigulArithFactor; |]) .
```

When the CST is a `Factor1` and the AST matches `Sub (Num 0) y`, we enter this branch, decompose the source and view by pattern matching, and use the view's right subtree `y` to update the second field of the source while skipping the first field (which stores whitespaces); the name of the BiGUL program for performing the update is determined by the type of the smaller source `y` (deduced by VARTYPE) and that of the smaller view.

Adaptive Branches. When all actions in a group fail to match, we should adapt the source into a proper shape to correspond to the view. This is done by generating adaptive branches from the actions during compilation. For example, the first action in the `Arith—Factor` group is compiled to

```
$(adaptiveSV [p| _ |] [p| Sub (Num 0) _ |])
  (\ _ _ ->  Factor1 " " FactorNull) .
```

The body of the adaptation function is generated by the auxiliary function DEFAULTEXPR, which creates a skeletal value that matches the source pattern.

Entry Point. The entry point of the program is chosen to be the BiGUL program compiled from the first group of actions. This corresponds to our assumption that the initial input concrete and abstract syntax trees are of the types specified for the first action group. It is rather simple so the rules are not shown in the figure. For the expression example, we generate a definition

```
entrance = bigulArithExpr
```

which is invoked in the `main` program.

```
#Concrete                                    #Directives
Expr -> [Plus]       Expr '+' Expr           Priority:
      | [Minus]      Expr '-' Expr           Times > Plus     ;
      | [Times]      Expr '*' Expr           Times > Minus    ;
      | [Division]   Expr '/' Expr           Division > Plus  ;
      | [Paren]      '(' Expr ')'            Division > Minus ;
      | [Lit]        Numeric
      ;                                      Associativity:
                                             Left: Plus, Minus, Times, Division ;

#Actions
Arith +> Expr
  Add x y  +>  [x +> Expr] '+' [y +> Expr] ;
  Sub x y  +>  [x +> Expr] '-' [y +> Expr] ;
  Mul x y  +>  [x +> Expr] '*' [y +> Expr] ;
  Div x y  +>  [x +> Expr] '/' [y +> Expr] ;
  Num i    +>  [i +> Numeric]              ;
  e        +>  '(' [e +> Expr] ')'         ;
;;
```

Fig. 6 Arithmetic expressions defined by an ambiguous grammar and the corresponding printing actions. (For simplicity, the variable and negation productions are omitted.)

5 Handling Grammatical Ambiguity

In Section 4, we have explained the basic version of BIYACC, which have an important assumption (stated in Section 4.3.1) that grammars have to be unambiguous. Having this assumption can be rather inconvenient in practice, however, as ambiguous grammars (with disambiguation directives) are often preferred since they are considered more natural and human-friendly than their unambiguous versions [2, 25]. Therefore the purpose of this section is to revise the architecture of basic BIYACC to allow the use of ambiguous grammars and disambiguation directives. This is in fact a long-standing problem: tools designed for building parser and printer pairs usually do not support such functionality (Section 7.1).

For example, consider the ambiguous grammar (with disambiguation directives) and printing actions in Figure 6, which we will refer to throughout this section. Note that the parenthesis structure is dropped when converting a CST to its AST (as stated by the last printing action of `Arith +> Expr`). The grammar is converted to CST datatypes and constructors as in Section 4.2, but here we explicitly give names such as `Plus` and `Times` to production rules, and these names (instead of automatically generated ones) are used for constructors in CSTs. Compared with this grammar, the unambiguous one shown in Figure 2 is less intuitive as it uses different nonterminals to resolve the ambiguity regarding operator precedence and associativity.

In this section, we explain the problem brought by ambiguous grammars (Section 5.1) and address it (Section 5.2) using *generalised parsing* and *bidirectionalised filters* (bi-filters for short). Then we extend BIYACC by incorporating bi-filters (Section 5.3) while still retaining the well-behavedness. To program with bi-filters easily, we provide compositional bi-filter directives (Section 5.4) which compile to priority and associativity bi-filters. Power users can also define their own bi-filters (Section 5.5),

and we illustrate this by writing a bi-filter that solves the (in)famous dangling-else problem.

5.1 The Problem with Ambiguous Grammars

Consider the original architecture of BIYACC in Figure 3, which we want to (and basically will) retain while adapting it to support ambiguous grammars. The first component (of the executable) we should adapt is *cparse* :: String → Maybe CST, the (concrete) parsing direction of the isomorphism: since there can be multiple CSTs corresponding to the same program text, *cparse* needs to choose one of them as the result. Disambiguation directives [22] were invented to describe how to make this choice. For example, with respect to the grammar in Figure 6, text `1 + 2 * 3` will have either of the two CSTs[4]:

$$cst_1 = \sharp\texttt{Plus 1 (Times 2 3)}$$
$$cst_2 = \sharp\texttt{Times (Plus 1 2) 3}$$

depending on the precedence of addition and multiplication. Conventionally, we can use the YACC-style disambiguation directives `%left '+'; %left '*';` to specify that multiplication has higher precedence over addition, and instruct the parser to choose cst_1.

However, merely adapting *cparse* with disambiguation behaviour is not enough, since the isomorphism, in particular its right to left direction (Theorem 1, which is simplified as *cparse* (*cprint cst*) = Just *cst*), is broken when an ambiguous grammar is used — in the example above, *cparse* (*cprint* cst_2) = Just $cst_1 \neq$ Just cst_2. This is because the range of *cparse* is strictly smaller than the domain of *cprint*: if we start from any CST not in the range of *cparse*, we will never be able to get back to the same CST through *cprint* and then *cparse*. This tells us that, to retain the isomorphism, the domain of *cprint* should not be the whole CST but only the range of *cparse*, i.e. the set of *valid* CSTs (as defined by the disambiguation directives), which we denote by CST_F (for reasons that will be made clear in Section 5.3).

Now that the right-hand side domain of the isomorphism is restricted to CST_F, the source of the lens should be restricted to this set as well. For *get* :: CST → Maybe AST we need to restrict its domain, which is easy; for *put* :: CST → AST → Maybe CST we should revise its type to $\mathsf{CST}_F \to \mathsf{AST} \to \mathsf{Maybe}\ \mathsf{CST}_F$, meaning that *put* should now guarantee that the CSTs it produces are valid, which is nontrivial. For example, consider the result of *put cst ast* where *ast* = `Mul (Add (Num 1) (Num 2)) (Num 3)` and *cst* is some arbitrary tree. A natural choice is cst_2, which, however, is excluded from CST_F by disambiguation. One way to solve the problem could be making *put* refuse to produce a result from *ast*, but this is unsatisfactory since *ast* is perfectly valid and should not be ignored by *put*. A more satisfactory way is creating a CST with *proper parentheses*, like $cst_3 = \sharp$`Times (Paren (Plus 1 2)) 3`. But it is not clear in what cases parentheses need to be added, in what cases they need not, and in what cases they cannot.

[4] For simplicity, we use $\sharp$ to annotate type-incorrect CSTs in which fields for layouts (and comments) and unimportant constructors such as `Lit` are omitted.

We are now led to a fundamental problem: generally, *put* strategies for producing valid CSTs should be inferred from the disambiguation directives, but the semantics of YACC disambiguation directives are defined over the implementation of YACC's underlying LR parsing algorithm with a stack [3, 22], and therefore it is nontrivial to invent a dual semantics in the *put* direction. To have a simple and clear semantics of the disambiguation process, we turn away from YACC's traditional approach and opt for an alternative approach based on *generalised parsing* with *disambiguation filters* [9, 25], whose semantics can be specified implementation-independently. Based on this simple and clear semantics, we will be able to devise ways to amend *put* to produce only valid CSTs, and formally state the conditions under which the executable generated by the revised BIYACC is well-behaved.

5.2 Generalised Parsing and Bidirectionalised Filters

The idea of generalised parsing is for a parser to produce all possible CSTs corresponding to its input program text instead of choosing only one CST (possibly prematurely) [15, 44, 47, 50], and works naturally with ambiguous grammars. In practice, a generalised parser can be generated using, e.g., HAPPY's GLR mode [32], and we will assume that given a grammar we can obtain a generalised parser:

$$\mathit{cgparse} :: \mathsf{String} \to [\mathsf{CST}] \,.$$

The result of *cgparse* is a list of CSTs. We do not need to wrap the result type in Maybe — if *cgparse* fails, an empty list is returned. And we should note that, while the result is a list, what we really mean is a set (commonly represented as a list in Haskell) since we do not care about the order of the output CSTs and do not allow duplicates.

With generalised parsing, program text is first parsed to all the possible CSTs; disambiguation then becomes an extremely simple concept: removing CSTs that the user does not want. One possible semantics of disambiguation may be a function *judge* :: Tree $\to$ Bool; during disambiguation, this function is applied to all candidate CSTs, and a candidate *cst* is removed if *judge cst* returns False, and kept otherwise. We call these functions *disambiguation filters* ('filters' for short).[5] For example, to state that top-level addition is left-associative, we can use the following filter[6] to reject right-sided trees:

```
plusJudge :: Expr -> Bool
plusJudge (♯Plus _ (Plus _ _)) = False
plusJudge _ = True .
```

This simple and clean semantics of disambiguation is then amenable to 'bidirectionalisation', which we do next.

[5] The general type for disambiguation filters is *filter* :: $[t] \to [t]$, which allows comparison among a list of CSTs. However, since in this paper we only consider *property filters* defined in terms of predicates (on a single tree), it is sufficient to use the simplified type *filter* :: $t \to$ Bool. See Section 7.2.

[6] This is not a very realistic filter, although it sufficiently demonstrates the use of filters and removes ambiguity in simplest cases like `1 + 2 * 3`. In general, the filter should be *complete* (Definition 7) so that ambiguity is fully removed from the grammar.

Note that, unlike YACC's disambiguation directives, which assign precedence and associativity to individual tokens and *implicitly* exclude 'some' CSTs, in `plusJudge` above we *explicitly* ban incorrect CSTs through pattern matching. Having described which CSTs are incorrect, we can further specify what to do with incorrect CSTs in the printing direction. Whenever a CST 'in a bad shape', i.e. rejected by a filter like `plusJudge`, is produced, we can repair it so that it becomes 'in a good shape':

```
plusRepair :: Expr -> Expr
plusRepair (♯Plus t1 (Plus t2 t3)) = ♯Plus t1 (Paren (Plus t2 t3))
plusRepair t = t .
```

The above function states that whenever a `Plus` is another `Plus`'s right child, there must be a parenthesis structure `Paren` in between. Observant readers might have found that the trees processed by `plusJudge` and `plusRepair` have the same pattern. So we can pair the two functions and make a *bidirectionalised filter* ('bi-filters' for short):

```
plusLAssoc :: Expr -> (Expr, Bool)
plusLAssoc (♯Plus t1 (Plus t2 t3)) = (♯Plus t1 (Paren (Plus t2 t3)), False)
plusLAssoc t = (t, True) .
```

But there is still some redundancy in the definition of `plusLAssoc`, for when the input tree is correct we always return the same input tree; this can be further optimised:

```
plusLAssoc' :: Expr -> Maybe Expr
plusLAssoc' (♯Plus t1 (Plus t2 t3)) = Just (♯Plus t1 (Paren (Plus t2 t3)))
plusLAssoc' _ = Nothing .
```

Generalising the example above, we arrive at the definition of bi-filters.

Definition 4 (Bidirectionalised Filters) A *bidirectional filter* F working on trees of type t is a function of type $\mathsf{BiFilter}\ t$ defined by

$$\textbf{type}\ \mathsf{BiFilter}\ t\ =\ t \to \mathsf{Maybe}\ t$$

satisfying

$$\mathit{repair}\ F\ t = t' \quad \Rightarrow \quad \mathit{judge}\ F\ t' = \mathsf{True} \tag{RepairJudge}$$

where the two directions *repair* and *judge* are defined by

$$\begin{aligned}
&\mathit{repair} :: \mathsf{BiFilter}\ t \to (t \to t)\\
&\mathit{repair}\ F\ t = \textbf{case}\ F\ t\ \textbf{of}\\
&\qquad\qquad \mathsf{Nothing} \to t\\
&\qquad\qquad \mathsf{Just}\ t' \quad \to t'\\
&\mathit{judge} :: \mathsf{BiFilter}\ t \to (t \to \mathsf{Bool})\\
&\mathit{judge}\ F\ t = \textbf{case}\ F\ t\ \textbf{of}\\
&\qquad\qquad \mathsf{Nothing} \to \mathsf{True}\\
&\qquad\qquad \mathsf{Just}\ _ \quad \to \mathsf{False}\ .
\end{aligned}$$

Functions *repair* and *judge* accept a bi-filter and return respectively the specialised *repair* and *judge* functions for that bi-filter. For clarity, we let repair_F denote *repair* F and let judge_F denote *judge* F. The bi-filter law RepairJudge dictates that repair_F should bring its input tree into a correct state with respect to judge_F. The reader may

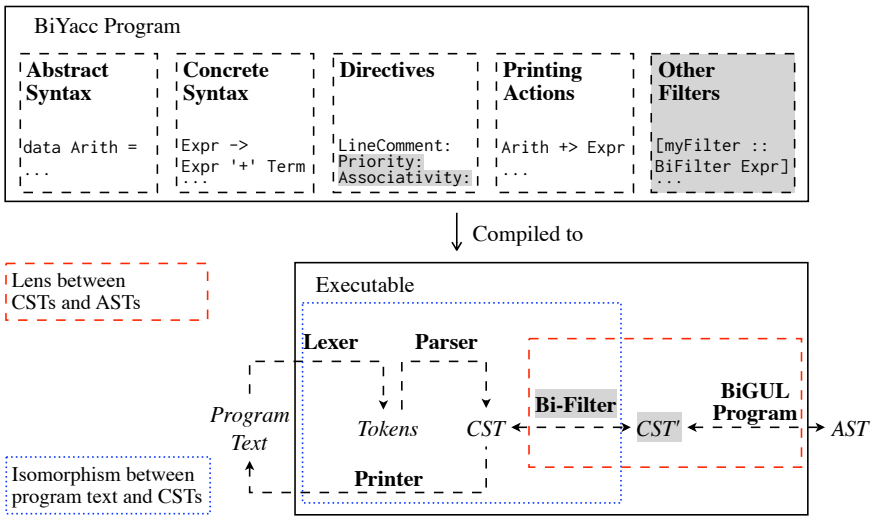

Fig. 7 New architecture of BIYACC. (New components are in light grey.)

wonder why there is not a dual JudgeRepair law saying that if a tree is already in a good shape justified by $judge_F$, then $repair_F$ should leave it unchanged. In fact this is always satisfied according to the definitions of *judge* and *repair*, so we formulate it as a lemma.

Lemma 1 (JudgeRepair) *Any bi-filter F satisfies the* JudgeRepair *property:*

$$judge_F\ t = \mathsf{True} \quad \Rightarrow \quad repair_F\ t = t\ .$$

Proof From $judge_F\ t = \mathsf{True}$ we deduce $F\ t = \mathsf{Nothing}$, which implies $repair_F\ t = t$. □

In the next section, we will describe how to fit generalised parsers and bi-filters into the architecture of BIYACC. To let bi-filters work with the lens part (of the whole system), we require a further property characterising the interaction between the repairing direction of a bi-filter and the get direction of a lens.

Definition 5 (PassThrough) A bi-filter F satisfies the *PassThrough* property with respect to a function *get* exactly when

$$get \circ repair_F = get\ .$$

If we think of a *get* function as mapping CSTs to their semantics (in our case ASTs), then the PassThrough property is a reasonable requirement since it guarantees that the repaired CST will have the same semantics as before (as it is converted to the same AST). This property will be essential for establishing the well-behavedness of the executable generated by the revised BIYACC.

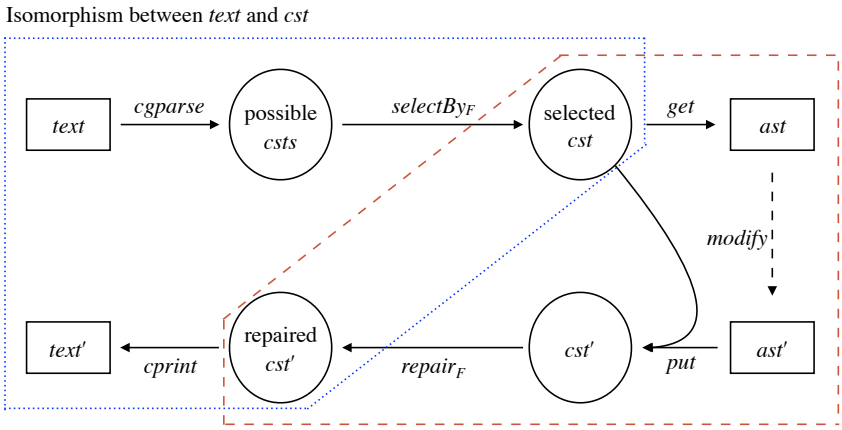

Fig. 8 A schematic diagram showing how parsing and printing work with a bi-filter.

5.3 The New BIYACC System

As depicted in Figure 7, the executable generated by the new BIYACC system is still the composition of an isomorphism and a lens, which is the structure we have tried to retain. To precisely notice the changes in several generated components (in the executable file) and demonstrate how parsing and printing work with a bi-filter, we present Figure 8 and will use this one instead. In the new system, we will still use the *get* and *put* transformations generated from printing actions and the concrete printer *cprint* from grammars, while the concrete parser *cparse* is replaced with a generalised parser *cgparse*. Additionally, the `#Directives` and `#OtherFilters` parts will be used to generate a bi-filter F, whose $judge_F$ (used in the $selectBy_F$ function in Figure 8) and $repair_F$ components are integrated into the isomorphism and lens parts respectively, so that the right-hand side domain of the isomorphism and the source of the lens become CST_F, the set of valid CSTs:

$$\mathsf{CST}_F = \{\, cst \in \mathsf{CST} \mid judge_F\ cst = \mathsf{True} \,\} .$$

Next, we introduce the (new) isomorphism and lens parts, and prove the inverse properties and well-behavedness respectively.

5.3.1 The Revised Isomorphism

Let us first consider the isomorphism part between String and CST_F, enclosed within the blue dotted lines in Figure 8, and consisting of *cprint*, *cgparse*, and $selectBy_F$:

$$\begin{aligned}
&cprint :: \mathsf{CST} \to \mathsf{String}\\
&cgparse :: \mathsf{String} \to [\mathsf{CST}]\\
&selectBy_F :: [\mathsf{CST}] \to \mathsf{Maybe}\ \mathsf{CST}_F\\
&selectBy_F\ csts = \mathbf{case}\ selectBy\ judge_F\ csts\ \mathbf{of}\\
&\qquad\qquad\qquad [cst] \ \to\ \mathsf{Just}\ cst
\end{aligned}$$

$$\begin{array}{l} \_ \quad \to \mathsf{Nothing} \\ \mathit{selectBy} :: (a \to \mathsf{Bool}) \to [a] \to [a] \\ \mathit{selectBy}\ p\ [\,] = [\,] \\ \mathit{selectBy}\ p\ (x : xs) \mid p\ x = x : \mathit{selectBy}\ p\ xs \\ \mathit{selectBy}\ p\ (x : xs) \mid \mathit{otherwise} = \mathit{selectBy}\ p\ xs\ . \end{array}$$

In the parsing direction, first *cgparse* produces all the CSTs; then $\mathit{selectBy}_F$ utilises a function *selectBy* and a predicate judge_F to (try to) select the only correct *cst*; if there is no correct CST or more than one correct CST, Nothing is returned. The function *selectBy*, which selects from the input list exactly the elements satisfying the given predicate, is named *filter* in Haskell's standard libraries but renamed here to avoid confusion. In the printing direction, we still use *cprint* to flatten a (correct) CST back to program text. Formally, constructed from *cgparse* and *cprint*, the two directions of the isomorphism are

$$\begin{array}{l} \mathit{cparse}_F\ :: \mathsf{String} \to \mathsf{Maybe}\ \mathsf{CST}_F \\ \mathit{cparse}_F\ = \mathit{selectBy}_F \circ \mathit{cgparse} \\ \\ \mathit{cprint}_F\ :: \mathsf{CST}_F \to \mathsf{Maybe}\ \mathsf{String} \\ \mathit{cprint}_F\ = \mathsf{Just} \circ \mathit{cprint}\ . \end{array}$$

We are eager to give the (revised version of) inverse-like properties and their proofs, which, however, depend on two assumptions about generalised parsers and bi-filters. So let us present them in order.

Definition 6 (Generalised Parser Correctness) A generalised parser *cgparse* is *correct* exactly when

$$\mathit{cgparse}\ \mathit{text} = \{\, \mathit{cst} \in \mathsf{CST} \mid \mathit{cprint}\ \mathit{cst} = \mathit{text} \,\}\ .$$

This is exactly Definition 3.7 of Klint and Visser's paper [25]. We remind the reader again that we use sets and lists interchangeably for the parsing results.

Definition 7 (Bi-Filter Completeness) A bi-filter F is *complete* with respect to a printer *cprint* exactly when

$$\mathit{text} \in \mathsf{Img}\ \mathit{cprint} \quad \Rightarrow \quad \big|\{\, \mathit{cst} \in \mathsf{CST}_F \mid \mathit{cprint}\ \mathit{cst} = \mathit{text} \,\}\big| = 1\ .$$

($\mathsf{Img}\ f = \{\, y \mid \exists x.\ fx = y \,\}$ is the image of the function f.)

This is revised from Definition 4.3 of Klint and Visser's paper [25], where they require that filters select exactly one CST and reject all the others. Since it is undecidable to judge whether a given context-free grammar is ambiguous [10], we cannot tell whether a (bi-)filter (for the full CFG) is complete, either. But still, some checks can be performed for simple cases, as stated in Section 7.

The following two lemmas connect our two assumptions, Definitions 6 and 7, with the definitions of cparse_F and cprint_F.

Lemma 2 *In the definitions of* cparse_F *and* cprint_F*, if cgparse is correct and F is complete with respect to cprint, then*

$$\mathit{text} \in \mathsf{Img}\ \mathit{cprint} \quad \Rightarrow \quad \exists \mathit{cst} \in \mathsf{CST}_F.\ \mathit{cparse}_F\ \mathit{text} = \mathsf{Just}\ \mathit{cst}\ \wedge\ \mathit{cprint}\ \mathit{cst} = \mathit{text}\ .$$

Proof We reason:

$$
\begin{aligned}
& \mathit{selectBy}_F\ (\mathit{cgparse}\ \mathit{text}) \\
= \quad & \{\ \text{Definition of } \mathit{SelectBy}_F\ \} \\
& \textbf{case}\ \mathit{selectBy}\ \mathit{judge}_F\ (\mathit{cgparse}\ \mathit{text})\ \textbf{of}\ \{\ [\mathit{cst}] \to \mathsf{Just}\ \mathit{cst}; \_ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ \text{Generalised Parser Correctness}\ \} \\
& \textbf{case}\ \mathit{selectBy}\ \mathit{judge}_F\ \{\ \mathit{cst} \in \mathsf{CST} \mid \mathit{cprint}\ \mathit{cst} = \mathit{text}\ \}\ \textbf{of} \\
& \qquad \{\ [\mathit{cst}] \to \mathsf{Just}\ \mathit{cst};\ \_ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ \mathit{selectBy}\ \mathit{judge}_F \text{ only selects correct CSTs regarding } F\ \} \\
& \textbf{case}\ \{\ \mathit{cst} \in \mathsf{CST}_F \mid \mathit{cprint}\ \mathit{cst} = \mathit{text}\ \}\ \textbf{of}\ \{\ [\mathit{cst}] \to \mathsf{Just}\ \mathit{cst};\ \_ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ \text{Bi-Filter Completeness, } \exists \mathit{cst}' \text{ s.t. } \{\ \mathit{cst} \in \mathsf{CST}_F \mid \mathit{cprint}\ \mathit{cst} = \mathit{text}\ \} = [\mathit{cst}']\ \} \\
& \textbf{case}\ [\mathit{cst}']\ \textbf{of}\ \{\ [\mathit{cst}] \to \mathsf{Just}\ \mathit{cst};\ \_ \to \mathsf{Nothing}\ \} \\
= \quad & \{\ \text{Definition of } \textbf{case}\ \} \\
& \mathsf{Just}\ \mathit{cst}\ .
\end{aligned}
$$

Moreover, *cst* satisfies $\mathit{cprint}\ \mathit{cst} = \mathit{text}$, since the latter is the comprehension condition of the set from which *cst* is chosen, and therefore $\mathit{cprint}_F\ \mathit{cst} = \mathsf{Just}\ \mathit{text}$. □

Lemma 3 (Printer Injectivity) *If F is a complete bi-filter, then cprint_F is injective.*

Proof Assume that $\mathit{cst}, \mathit{cst}' \in \mathsf{CST}_F$ and $\mathit{cprint}\ \mathit{cst} = \mathit{cprint}\ \mathit{cst}' = \mathit{text}$ for some *text*; that is, both *cst* and cst' are in the set $P = \{\ \mathit{cst} \in \mathsf{CST}_F \mid \mathit{cprint}\ \mathit{cst} = \mathit{text}\ \}$. Since $\mathit{text} \in \mathsf{Img}\ \mathit{cprint}$, by the completeness of F we have $|P| = 1$, and hence $\mathit{cst} = \mathit{cst}'$. □

Theorem 2 (Inverse Properties with Bi-Filters) *In the definitions of cparse_F and cprint_F, if cgparse is correct and F is complete, then*

$$\mathit{cparse}_F\ \mathit{text} = \mathsf{Just}\ \mathit{cst} \quad \Rightarrow \quad \mathit{cprint}_F\ \mathit{cst} = \mathsf{Just}\ \mathit{text} \tag{6}$$

$$\mathit{cprint}_F\ \mathit{cst} = \mathsf{Just}\ \mathit{text} \quad \Rightarrow \quad \mathit{cparse}_F\ \mathit{text} = \mathsf{Just}\ \mathit{cst}\ . \tag{7}$$

Proof For (6): Let $\mathsf{Just}\ \mathit{cst} = \mathit{selectBy}_F\ (\mathit{cgparse}\ \mathit{text})$. According to the definition of $\mathit{selectBy}_F$, we have $\mathit{cst} \in \mathit{cgparse}\ \mathit{text}$. By Generalised Parser Correctness $\mathit{cprint}\ \mathit{cst} = \mathit{text}$, and therefore $\mathit{cprint}_F\ \mathit{cst} = \mathsf{Just}\ \mathit{text}$.

For (7): The antecedent implies $\mathit{cprint}\ \mathit{cst} = \mathit{text}$. By Lemma 2, we have $\mathit{cparse}_F\ \mathit{text} = \mathsf{Just}\ \mathit{cst}'$ for some $\mathit{cst}' \in \mathsf{CST}_F$ such that $\mathit{cprint}_F\ \mathit{cst}' = \mathsf{Just}\ \mathit{text} = \mathit{cprint}_F\ \mathit{cst}$. By Lemma 3 we know $\mathit{cst}' = \mathit{cst}$, and thus $\mathit{cparse}_F\ \mathit{text} = \mathsf{Just}\ \mathit{cst}$. □

5.3.2 The Lens

Recall that the `#Action` part of a BIYACC program produces a lens (BIGUL program) consisting of a pair of well-behaved *get* and *put* functions:

$$
\begin{aligned}
& \mathit{get} :: \mathsf{CST} \to \mathsf{Maybe}\ \mathsf{AST} \\
& \mathit{put} :: \mathsf{CST} \to \mathsf{AST} \to \mathsf{Maybe}\ \mathsf{CST}\ .
\end{aligned}
$$

To work with a bi-filter F, in particular its $repair_F$ component, they need to be adapted to get_F and put_F, which accept only valid CSTs:

$$\begin{array}{ll} get_F & :: \mathsf{CST}_F \to \mathsf{Maybe\ AST} \\ get_F & = get \\ put_F & :: \mathsf{CST}_F \to \mathsf{AST} \to \mathsf{Maybe\ CST}_F \\ put_F\ cst\ ast & = fmap\ repair_F\ (put\ cst\ ast) \end{array}$$

where *fmap* is a standard Haskell library function defined (for Maybe) by

$$\begin{array}{ll} fmap & :: (a \to b) \to \mathsf{Maybe}\ a \to \mathsf{Maybe}\ b \\ fmap\ f\ \mathsf{Nothing} & = \mathsf{Nothing} \\ fmap\ f\ (\mathsf{Just}\ x) & = \mathsf{Just}\ (f\ x)\ . \end{array}$$

We will need a lemma about *fmap*, which can be straightforwardly proved by a case analysis.

Lemma 4 *If fmap f mx* $=$ Just *y, then there exists x such that mx* $=$ Just *x and f x* $= y$.

Now we prove that get_F and put_F are well-behaved.

Theorem 3 (Well-behavedness with Bi-Filters) *Given a complete bi-filter F and a well-behaved lens consisting of get and put, if get and F additionally satisfy PassThrough, then the* get_F *and* put_F *functions with respect to F is also well-behaved:*

$$put_F\ cst\ ast = \mathsf{Just}\ cst' \quad \Rightarrow \quad get_F\ cst' = \mathsf{Just}\ ast \tag{8}$$

$$get_F\ cst = \mathsf{Just}\ ast \quad \Rightarrow \quad put_F\ cst\ ast = \mathsf{Just}\ cst\ . \tag{9}$$

Proof For (8): The antecedent expands to $fmap\ repair_F\ (put\ cst\ ast) = \mathsf{Just}\ cst'$, which, by Lemma 4, implies $put\ cst\ ast = \mathsf{Just}\ cst''$ for some cst'' such that $repair_F\ cst'' = cst'$. Now we reason:

$$\begin{array}{ll} & get_F\ cst' \\ = & \{\ \text{Definition of } get_F \text{ and } cst \in \mathsf{CST}_F\ \} \\ & get\ cst' \\ = & \{\ \text{Definition of } cst'\ \} \\ & get\ (repair_F\ cst'') \\ = & \{\ \text{PassThrough}\ \} \\ & get\ cst'' \\ = & \{\ \text{PutGet}\ \} \\ & \mathsf{Just}\ ast\ . \end{array}$$

For (9):

$$
\begin{aligned}
& put_F\ cst\ ast \\
= \quad & \{ \text{ Definition of } put_F \ \} \\
& fmap\ repair_F\ (put\ cst\ ast) \\
= \quad & \{ \text{ GetPut } \} \\
& fmap\ repair_F\ (\mathsf{Just}\ cst) \\
= \quad & \{ \text{ Definition of } fmap \ \} \\
& \mathsf{Just}\ (repair_F\ cst) \\
= \quad & \{ \text{ Since } cst \in \mathsf{CST}_F, judge_F\ cst = \mathsf{True}. \text{ By JudgeRepair } \} \\
& \mathsf{Just}\ cst\ .
\end{aligned}
$$

□

5.4 Bi-Filter Directives

Until now, we have only considered working with a single bi-filter, but this is without loss of generality because we can provide a bi-filter composition operator (Section 5.4.1) so that we can build large bi-filters from small ones. This is a suitable semantic foundation for introducing YACC-like directives for specifying priority and associativity into BIYACC (Section 5.4.2), since we can give these directives a bi-filter semantics and interpret a collection of directives as the composition of their corresponding bi-filters. We will also discuss some properties related to this composition (Section 5.4.3).

5.4.1 Bi-Filter Composition

We start by defining bi-filter composition, with the intention of making the net effect of applying a sequence of bi-filters one by one the same as applying their composition. Although the intention is better captured by Lemma 5, which describes the *repair* and *judge* behaviour of a composite bi-filter in terms of the component bi-filters, we give the definition of bi-filter composition first.

Definition 8 (Bi-Filter Composition) The composition of two bi-filters is defined by

$$
\begin{aligned}
& (\lhd) :: (t \to \mathsf{Maybe}\ t) \ \to \ (t \to \mathsf{Maybe}\ t) \ \to \ (t \to \mathsf{Maybe}\ t) \\
& (j \lhd i)\ t = \mathbf{case}\ i\ t\ \mathbf{of} \\
& \qquad\qquad \mathsf{Nothing} \to j\ t \\
& \qquad\qquad \mathsf{Just}\ t' \quad \to \mathbf{case}\ j\ t'\ \mathbf{of} \\
& \qquad\qquad\quad \mathsf{Nothing} \to \mathsf{Just}\ t' \\
& \qquad\qquad\quad \mathsf{Just}\ t'' \quad \to \mathsf{Just}\ t''\ .
\end{aligned}
$$

When applying a composite bi-filter $j \lhd i$ to a tree t, if t is correct with respect to i (i.e. $i\ t = \mathsf{Nothing}$), we directly pass the original tree t to j; otherwise t is repaired by i, yielding t', and we continue to use j to repair t'. Note that if $j\ t' = \mathsf{Nothing}$, we return the tree t' instead of $\mathsf{Nothing}$.

Lemma 5 *For a composite bi-filter $j \lhd i$, the following two equations hold:*

$$\begin{aligned} \mathit{repair}\ (j \lhd i)\ t &= (\mathit{repair}_j \circ \mathit{repair}_i)\ t \\ \mathit{judge}\ (j \lhd i)\ t &= \mathit{judge}_j\ t \wedge \mathit{judge}_i\ t\ . \end{aligned}$$

Proof By the definition of bi-filter composition. □

Composition of bi-filters should still be a bi-filter and satisfy RepairJudge and PassThrough. This is not always the case though — to achieve this, we need some additional constraint on the component bi-filters, as formulated below.

Definition 9 Let i and j be bi-filters. We say that j *respects* i exactly when

$$\mathit{judge}_i\ t = \mathsf{True} \quad \Rightarrow \quad \mathit{judge}_i\ (\mathit{repair}_j\ t) = \mathsf{True}\ .$$

If j respects i, then a later applied repair_j will never break what may already be repaired by a previous repair_i. We can thus safely compose a bi-filter after another if the former respects the latter. This is proved as the following theorem.

Theorem 4 *Let i and j be bi-filters (satisfying RepairJudge and PassThrough). If j respects i, then $j \lhd i$ also satisfy RepairJudge and PassThrough.*

Proof For RepairJudge, we reason:

$$\begin{aligned} & \mathit{judge}\ (j \lhd i)\ (\mathit{repair}\ (j \lhd i)\ t) \\ = \quad & \{\ \text{Lemma 5}\ \} \\ & \mathit{judge}\ (j \lhd i)\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \\ = \quad & \{\ \text{Lemma 5}\ \} \\ & \mathit{judge}_j\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \wedge \mathit{judge}_i\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \\ = \quad & \{\ \text{RepairJudge of } j\ \} \\ & \mathsf{True} \wedge \mathit{judge}_i\ (\mathit{repair}_j\ (\mathit{repair}_i\ t)) \\ = \quad & \{\ \mathit{judge}_i\ (\mathit{repair}_i\ t') = \mathsf{True};\ j \text{ respects } i\} \\ & \mathsf{True} \wedge \mathsf{True} \\ = \quad & \mathsf{True}\ . \end{aligned}$$

And for PassThrough:

$$\begin{aligned} & \mathit{get}\ (\mathit{repair}\ (j \lhd i)\ t) \\ = \quad & \{\ \text{Lemma 5}\ \} \\ & \mathit{get}\ (\mathit{repair}_j(\mathit{repair}_i\ t)) \\ = \quad & \{\ \text{PassThrough of } j\ \} \\ & \mathit{get}\ (\mathit{repair}_i\ t) \\ = \quad & \{\ \text{PassThrough of } i\ \} \\ & \mathit{get}\ t\ . \end{aligned}$$

□

5.4.2 Priority and Associativity Directives

To relieve the burden of writing bi-filters manually and guaranteeing respect among bi-filters being composed, we provide some directives for constructing bi-filters dealing with priority[7] and associativity, which are generally comparable to YACC's conventional disambiguation directives. The bi-filter directives in a BIYACC program can be thought of as specifying 'production priority tables', analogous to the operator precedence tables of, for example, the C programming language [24] (chapter *Expressions*) and Haskell [33] (page 51). The main differences (in terms of the parsing direction) are:

- For bi-filters, priority can be assigned independently of associativity and vice versa, while the YACC-style approach does not permit so — by design, when the YACC directives (`%left`, `%right`, and `%nonassoc`) are used on multiple tokens, they necessarily specify both the precedence and associativity of those tokens.
- For bi-filters, priority and associativity directives may be used to specify more than one production priority tables, making it possible to put unrelated operators in different tables and avoid (unnecessarily) specifying the relationship between them. It is impossible to do so with the YACC-style approach, for its concise syntax only allows a single operator precedence table.

(The bi-filter semantics of) our bi-filter directives repair CSTs violating priority and associativity constraints by adding parentheses — for example, if the production of addition expressions in Figure 6 is left-associative, then we can repair $\sharp$`Plus 1 (Plus 2 3)` by adding parentheses around the right subtree, yielding $\sharp$`Plus 1 (Paren (Plus 2 3))`, provided that the grammar has a production of parentheses annotated with the *bracket attribute* [8, 49]:

```
Expr -> ...
      | [Paren]  '(' Expr ')' {# Bracket #} .
```

It instructs our bi-filter directives to use this production when parentheses need to be added. Internally, from the production and bracket attribute annotation, a type class AddParen and corresponding instances for each datatype generated from concrete syntax (Expr for this example) are automatically created:

```
class AddParen t where
  canAddPar :: t -> Bool
  addPar    :: t -> t
```

where `canAddPar` tells whether a CST can be wrapped in a parenthesis structure, while `addPar` adds that structure if it is possible, or behaves as an identity function otherwise. This makes it possible to automatically generate bi-filters to repair incorrect CSTs (and help the user to define their own bi-filters more easily — see Section 5.5).

In order for bi-filter directives to work correctly, the user should notice the following requirements: (i) Directives shall not mention the parenthesis production annotated with bracket attribute so that they *respect* each other and work properly (as introduced later in Definition 9). (ii) Suppose that the parenthesis production is $NT \to \alpha NT_R \beta$

[7] The YACC-style approach adopts the word *precedence* [22] while the filter-based approaches tend to use the word *priority* [9, 25]. We follow the traditions and use either word depending on the context.

where α and β denote a sequence of terminals, there shall be exactly one printing action defined for the parenthesis production in the form of $v \Rightarrow \alpha[v \Rightarrow NT_R]\beta$ for the PassThrough property to hold: for any CST, the (added) parenthesis structure will all be dropped through the conversion to its AST.

Next we introduce our priority and associativity directives and their bi-filter semantics. From a directive, we first generate a bi-filter that checks and repairs only the top of a tree; this bi-filter is then lifted to check and repair all the subtrees in a tree. In the following we will give the semantics of the directives in terms of the generation of the top-level bi-filters, and then discuss the lifted bi-filters and other important properties they satisfy in Section 5.4.3.

Priority Directives

A priority directive defines relative priority between two productions; it removes (in the parsing direction) or repairs (in the printing direction) CSTs in which a node of (relatively) lower priority is a direct child of the node of (relatively) higher priority. For instance, we can define that (the production of) multiplication has higher priority than (the production of) addition for the grammar in Figure 6 by writing

```
Expr -> Expr '*' Expr > Expr -> Expr '+' Expr ;
```
or just
```
Times > Plus ; .
```

The directive first produces the following top-level bi-filter:[8]

```
fTimesPlusPrio  (Times t1 t2 t3) =
  case or [match t1 p, match t2 p, match t3 p, False] of
    False -> Nothing
    True  -> Just (Times (if match t1 p then addPar t1 else t1)
                         (if match t2 p then addPar t2 else t2)
                         (if match t3 p then addPar t3 else t3))
  where p = Plus undefined undefined undefined .
```

We first check whether any of the subtrees `t1`, `t2`, and `t3` violates the priority constraint, i.e. having `Plus` as its top-level constructor — this is checked by the `match` function, which compares the top-level constructors of its two arguments. The resulting boolean values are aggregated using the list version of logical disjunction $or :: [\mathsf{Bool}] \to \mathsf{Bool}$. If there is any incorrect part, we repair it by inserting a parenthesis structure using `addPar`.

In general, the syntax of priority directives is

$$
\begin{aligned}
\mathit{Priority} &::= \text{`}\texttt{Priority:}\text{'}\ \mathit{PDirective}^{+} \\
\mathit{PDirective} &::= \mathit{ProdOrCons}\ \text{`}\texttt{>}\text{'}\ \mathit{ProdOrCons}\ \text{`}\texttt{;}\text{'} \\
&\quad \mid \mathit{ProdOrCons}\ \text{`}\texttt{<}\text{'}\ \mathit{ProdOrCons}\ \text{`}\texttt{;}\text{'} \\
\mathit{ProdOrCons} &::= \mathit{Prod} \mid \mathit{Constructor} \\
\mathit{Prod} &::= \mathit{Nonterminal}\ \text{`}\texttt{->}\text{'}\ \mathit{Symbol}^{+}
\end{aligned}
$$

where *Constructor* and *Symbol* is already defined in Figure 4; for each priority declaration, we can use either *production*s or the *constructor*s of the productions.

[8] Although terminals such as `'*'` and `'+'` are uniquely determined by constructors and not explicitly included in the CSTs, there are fields in CSTs for holding whitespaces after them. Thus `Times` still has three subtrees.

If the user declares that production $NT_1 \to RHS_1$ has higher priority than production $NT_2 \to RHS_2$, the following priority bi-filter will be generated:

$$
\begin{aligned}
&\textsc{toPrioFilter}[\![(RHS_1, NT_1, RHS_2, NT_2)]\!] = \\
&\quad \text{`f'} \mathit{conRHS}_1\, \mathit{conRHS}_2 \text{`Prio' `('} \mathit{conRHS}_1\ \textsc{fillVars}(RHS_1) \text{`) ='} \\
&\qquad \text{`case or [' } \left\langle \text{`match'}\ t\ \text{`p,'} \;\middle|\; t \in \textsc{fillVars}(RHS_1) \right\rangle \text{`False'`] of'} \\
&\qquad\quad \text{`False -> Nothing'} \\
&\qquad\quad \text{`True  -> Just ('}\ \mathit{conRHS}_1\ \left\langle \textsc{repair}(t) \;\middle|\; t \in \textsc{fillVars}(RHS_1) \right\rangle \text{`)'} \\
&\qquad \text{`where p = '}\ \textsc{con}(NT_2, RHS_2)\ \textsc{fillUndefined}(RHS_2) \\
&\quad \text{`f'} \mathit{conRHS}_1\, \mathit{conRHS}_2 \text{`Prio' `\_' `=' `Nothing'} \\
&\textsc{repair}(t) = \text{`(if match'}\ t\ \text{`p' `then addPar'}\ t\ \text{`else'}\ t\text{`)'} \\
&\mathit{conRHS}_1 = \textsc{con}(NT_1, RHS_1) \\
&\mathit{conRHS}_2 = \textsc{con}(NT_2, RHS_2)\ .
\end{aligned}
$$

CON looks up constructor names for input productions (divided into nonterminals and right-hand sides); FILLVARS generates variable names for each terminal and nonterminal in its argument (here RHS_1); FILLUNDEFINED is similar to FILLVARS but it produces `undefined` values instead. If productions are referred to using their constructors, we can simply look up the nonterminals and right-hand sides and use the same code generation strategy.

Transitive Closures. In the same way as conventional YACC-style approaches, the priority directives are considered transitive. For instance,

```
Expr -> Expr '*' Expr  >  Expr -> Expr '+' Expr ;
Expr -> Expr '+' Expr  >  Expr -> Expr '&' Expr ;
```

implies that `Expr -> Expr '*' Expr  >  Expr -> Expr '&' Expr ;`. The feature is important in practice since it reduces the amount of routine code the user needs to write much (for large grammars).

Associativity Directives

Associativity directives assign (left- or right-) associativity to productions. A left-associativity directive bans (or repairs, in the printing direction) CSTs having the pattern in which a parent and its right-most subtree are both left-associative, if the (relative) priority between the parent and the subtree is not defined; a right-associativity directive works symmetrically.

As an example, we can declare that both addition and subtraction are left-associative (regarding the grammar in Figure 6) by writing

```
Left: Expr -> Expr '+' Expr, Expr -> Expr '-' Expr;
```

or just `Left: Plus, Minus;`. Since the relative priority between `Plus` and `Minus` is not defined, we generate top-level bi-filters for all the four possible pairs formed out of `Plus` and `Minus`:

```
fPlusPlusLAssoc (Plus t1 t2 t3) =
  if match t3 p then Just (Plus t1 t2 (addPar t3)) else Nothing
  where p = Plus undefined undefined undefined
fPlusPlusLAssoc _               = Nothing

fMinusMinusLAssoc (Minus t1 t2 t3) =
  if match t3 p then Just (Minus t1 t2 (addPar t3)) else Nothing
  where p = Minus undefined undefined undefined
fMinusMinusLAssoc _                = Nothing

fPlusMinusLAssoc (Plus t1 t2 t3) =
  if match t3 p then Just (Plus t1 t2 (addPar t3)) else Nothing
  where p = Minus undefined undefined undefined
fPlusMinusLAssoc _               = Nothing

fMinusPlusLAssoc (Minus t1 t2 t3) =
  if match t3 p then Just (Minus t1 t2 (addPar t3)) else Nothing
  where p = Plus undefined undefined undefined
fMinusPlusLAssoc _                = Nothing .
```

For instance, `fPlusPlusLAssoc` accepts ♯`Plus (Plus 1 2) 3` but not ♯`Plus 1 (Plus 2 3)`, which is repaired to ♯`Plus 1 (Paren (Plus 2 3))`.

Generally, the syntax of associativity directives is

$$\begin{aligned} \textit{Associativity} &::= \text{`}\texttt{Associativity:}\text{'}\ \textit{LeftAssoc}\ \textit{RightAssoc} \\ \textit{LeftAssoc} &::= \text{`}\texttt{Left:}\text{'}\ \textit{ProdOrCons}^{+}\{\text{`}\texttt{,}\text{'}\}\ \text{`}\texttt{;}\text{'} \\ \textit{RightAssoc} &::= \text{`}\texttt{Right:}\text{'}\ \textit{ProdOrCons}^{+}\{\text{`}\texttt{,}\text{'}\}\ \text{`}\texttt{;}\text{'}\ . \end{aligned}$$

Now we explain the generation of (top-level) bi-filters from associativity directives. We will consider only left-associativity directives, as right-associativity directives are symmetric. For every pair of left-associative productions whose relative priority is not defined — including cases where the two productions are the same — we generate a bi-filter to repair CSTs whose top uses the first production and whose right-most child uses the second production. Let $NT_1 \to \alpha_1 NT_{1R}$ and $NT_2 \to \alpha_2 NT_{2R}$ be two such productions, where α_1 (α_2) matches a sequence of arbitrary symbols of any length and NT_{1R} (NT_{2R}) is the right-most symbol and must be a nonterminal. (If it is not a nonterminal, it is meaningless to discuss associativity.) The generated bi-filter is

$$\begin{aligned} &\textsc{toLAssocFilter}[\![\alpha_1 NT_{1R}, NT_1, \alpha_2 NT_{2R}, NT_2]\!] = \\ &\quad \text{`}\texttt{f}\text{'}\ \textit{conRHS}_1\ \textit{conRHS}_2\ \text{`}\texttt{LAssoc}\text{'}\ \text{`}\texttt{(}\text{'}\textit{conRHS}_1\ \textsc{fillVars}(\alpha_1 NT_{1R})\ \text{`}\texttt{)}\text{'}\ \text{`}\texttt{=}\text{'} \\ &\qquad \text{`}\texttt{if match }\text{'}\ \textit{ntrVar}\ \text{`}\texttt{ p}\text{'} \\ &\qquad\quad \text{`}\texttt{then Just (}\text{'}\ \textit{conRHS}_1\ \textsc{fillVars}(\alpha_1)\ \text{`}\texttt{(addPar}\text{'}\ \textit{ntrVar}\ \text{`}\texttt{))}\text{'} \\ &\qquad\quad \text{`}\texttt{else Nothing}\text{'} \\ &\qquad \text{`}\texttt{where p = }\text{'}\textit{conRHS}_2\ \textsc{fillUndefined}(\alpha_2 NT_{2R}) \\ &\quad \text{`}\texttt{f}\text{'}\ \textit{conRHS}_1\ \text{`}\texttt{LAssoc}\text{'}\ \text{`}\texttt{\_}\text{'}\ \text{`}\texttt{=}\text{'}\ \text{`}\texttt{Nothing}\text{'} \\ &\textit{conRHS}_1 = \textsc{con}(NT_1, \alpha_1 NT_{1R}) \\ &\textit{conRHS}_2 = \textsc{con}(NT_2, \alpha_2 NT_{2R}) \\ &\textit{ntrVar} = \textsc{fillVarsFrom}(\textsc{length}(\alpha_1), NT_{1R})\ . \end{aligned}$$

Functions CON, FILLUNDEFINED, and FILLVAR have the same behaviour as before; FILLVARSFROM is a variation of FILLVARS, which generates variable names for each terminal and non-

terminal in its argument with suffix integers counting from a given number to avoid name clashing.

5.4.3 Properties of the Generated Bi-Filters

We discuss some properties of the bi-filters generated from our priority and associativity directives, to justify that it is safe to use these bi-filters without disrupting the well-behavedness of the whole system. Specifically:

- The generated top-level bi-filters satisfy RepairJudge, and it is easy to write actions to make them satisfy PassThrough.
- The bi-filters lifted from the top-level bi-filters still satisfy RepairJudge and PassThrough.
- The lifted bi-filters are *commutative*, which not only implies that all such bi-filters respect each other and can be composed in any order, but also guarantees that we do not have to worry about the order of composition since it does not affect the behaviour.

We will give only high-level, even informal, arguments for these properties, since, due to the generic nature of the definitions of these bi-filters (in terms of *Scrap Your Boilerplate* [30]), to give formal proofs we would have to introduce rather complex machinery (e.g., datatype-generic induction), which would be tedious and distracting.

Top-level bi-filters. The fact that the generated top-level bi-filters satisfy RepairJudge can be derived from the requirement that the directives do not mention the parenthesis production. Because of the requirement, in the generated bi-filters, repairing is always triggered by `match`ing a non-parenthesis production, and after that repairing will not be triggered again because a parenthesis production will have been added. For example, in the bi-filter `fTimesPlusPrio` (in Section 5.4.2), with `match t1 p`, `match t2 p`, and `match t1 p` we check whether `t1`, `t2`, and `t3` has `Plus` as the top-level production, which is different from the parenthesis production `Paren`; if any of the `match`ing succeeds, say `t1`, then `addPar t1` will add `Paren` at the top of `t1`, and `match (addPar t1) p` is guaranteed to be `False`, so the subsequent invocation of `judge fTimesPlusPrio` will return `True`. For PassThrough, since all the top-level bi-filters do is add parenthesis productions, we can simply make sure that appearances of the parenthesis production are ignored by *get*, i.e. $get\ (addPar\ s) = get\ s$ for all s; this, by well-behavedness, is the same as making *put* (printing actions) skip over parentheses. For example, for the grammar in Figure 6, we should write `t +> '(' [t +> Expr] ')'` as the only printing action mentioning parentheses, which means that $put\ (\mathsf{Paren}\ s)\ t = fmap\ \mathsf{Paren}\ (put\ s\ t)$ for all s and t. Then

the following reasoning implies that $get\ (\mathsf{Paren}\ s) = get\ s$ for all s:

$$
\begin{array}{ll}
 & get\ (\mathsf{Paren}\ s) = \mathsf{Just}\ t \\
\Leftrightarrow & \{\ \Rightarrow \text{by GetPut and} \Leftarrow \text{by PutGet}\ \} \\
 & put\ (\mathsf{Paren}\ s)\ t = \mathsf{Just}\ (\mathsf{Paren}\ s) \\
\Leftrightarrow & \{\ \text{By the above statement: } put\ (\mathsf{Paren}\ s)\ t = fmap\ \mathsf{Paren}\ (put\ s\ t)\ \} \\
 & fmap\ \mathsf{Paren}\ (put\ s\ t) = \mathsf{Just}\ (\mathsf{Paren}\ s) \\
\Leftrightarrow & \{\ \text{Lemma 4 and the definition of } fmap\ \} \\
 & put\ s\ t = \mathsf{Just}\ s \\
\Leftrightarrow & \{\ \Rightarrow \text{by PutGet and} \Leftarrow \text{by GetPut}\ \} \\
 & get\ s = \mathsf{Just}\ t
\end{array}
$$

for all s and t.

Lifted bi-filters. The lifted bi-filters apply the top-level bi-filters to all the subtrees in a CST in a bottom-up order. Formally, we can define, datatype-generically, a lifted bi-filter as a composition of top-level bi-filters, and use datatype-generic induction to prove that there is suitable respect among the top-level bi-filters being composed, and that the lifted bi-filter satisfies RepairJudge and PassThrough if the top-level ones do. But here we provide only an intuitive argument. What the lifted bi-filters do is find all prohibited pairs of adjoining productions and separate all the pairs by adding parenthesis productions. For RepairJudge, since all prohibited pairs are eliminated after repairing, there will be nothing left to be repaired in the resulting CST, which will therefore be deemed valid. For PassThrough, the intuition is the same as that for the top-level bi-filters.

Commutativity. Composite bi-filters $i \triangleleft j$ and $j \triangleleft i$ may have different behaviour, so in general we need to know the order of composition to figure out the exact behaviour of a composite bi-filter. This can be difficult when using our bi-filter directives, since a lot of bi-filters are implicitly generated from the directives, and it is not straightforward to specify the order in which all the explicitly and implicitly generated bi-filters are composed. Fortunately we do not need to do so, for all the bi-filters generated from the directives are *commutative*, meaning that the order of composition does not affect the behaviour.

Definition 10 (Bi-Filter Commutativity) Two bi-filters i and j are *commutative* exactly when

$$repair_i \circ repair_j = repair_j \circ repair_i\ .$$

By Lemma 5, this implies $repair\ (i \triangleleft j) = repair\ (j \triangleleft i)$. Note that $judge\ (i \triangleleft j) = judge\ (j \triangleleft i)$ by definition, so we do not need to require this in the definition of commutativity.

An important fact is that commutativity is stronger than respect, so it is always safe to compose commutative bi-filters.

Lemma 6 *Commutative bi-filters respect each other.*

Proof Given commutative bi-filters i and j, we show that j respects i. Suppose that $judge_i\ t = \mathsf{True}$ for a given tree t. Then

$$
\begin{aligned}
& judge_i\ (repair_j\ t) \\
= \quad & \{\ repair_i\ t = t, \text{ since } judge_i\ t = \mathsf{True}\ \} \\
& judge_i\ (repair_j\ (repair_i\ t)) \\
= \quad & \{\ i \text{ and } j \text{ are commutative}\ \} \\
& judge_i\ (repair_i\ (repair_j\ t)) \\
= \quad & \{\ \text{RepairJudge}\ \} \\
& \mathsf{True}\ .
\end{aligned}
$$

It follows by symmetry that i respects j as well. □

Now let us consider why any two different lifted bi-filters are commutative. (Commutativity is immediate if the two bi-filters are the same.) There are two key facts that lead to commutativity: (i) repairing does not introduce more prohibited pairs of productions, and (ii) the prohibited pairs of adjoining productions checked and repaired by the two bi-filters are necessarily different. Therefore the two bi-filters always repair different parts of a tree, and can repair the tree in any order without changing the final result. Fact (i) is, again, due to the requirement that the directives do not mention the parenthesis production, which is the only thing we add to a tree when repairing it. Fact (ii) can be verified by a careful case analysis. For example, we might be worried about the situation where a left-associative directive looks for production Q used at the right-most position under production P, while a priority directive also similarly looks for Q used under P, but the two directives cannot coexist in the first place since the first directive implies P and Q have no relative priority whereas the second one implies Q has lower priority than P.

5.5 Manually Written Bi-Filters

There are some other ambiguities that our directives cannot eliminate. In these cases, the user can define their own bi-filters and put them in the `#OtherFilters` part in a BIYACC program as shown in Figure 4. The syntax is

OtherFilters ::= '`[`' *HsFunDecl*$^+${'`,`'} '`]`' *HsCode*
HsFunDecl ::= *HsFunName* ' `:: BiFilter` ' *Nonterminal* .

That is, this part of the program begins with a list of declarations of the names and types of the user-defined bi-filters, whose Haskell definitions are then given below.

Now we demonstrate how to manually write a bi-filter by resolving the ambiguity brought by the dangling else problem. But before that, let us briefly review the problem, which arises, for example, in the following grammar:

```
Exp -> [ITE] 'if' Exp 'then' Exp 'else' Exp
     | [IT]  'if' Exp 'then' Exp .
```

With respect to this grammar, the program text `if a then if x then y else z` can be recognised as either `if a then (if x then y else z)` or `if a then (if x then y) else z`. To resolve the ambiguity, usually we prefer the 'nearest match' strategy (which is adopted by Pascal, C, and Java): `else` should match its nearest `then`, so that `if a then ( if x then y else z)` is the only correct interpretation.

The user may think that the problem can be solved by a priority (bi-)filter `ITE > IT;`, in the hope that the production 'if-then-else' binds tighter than the production 'if-then'. Unfortunately, this is incorrect as pointed out by Klint and Visser [25], because the corresponding (bi-)filter incorrectly rules out the pattern $\sharp$`ITE _ _ (IT _ _)`, which prints to unambiguous text, e.g., `if a then b else if x then y`. In fact, the (dangling else) problem is tougher than one might think and cannot be solved by any (bi-)filter performing pattern matching with a fixed depth [25].

Klint and Visser [25] proposed an idea to disambiguate the dangling-else grammar: Let Greek letters $\alpha, \beta, \ldots$ match a sequence of symbols of any length. Then the program text `if` α `then` β `else` γ should be banned if the right spine of β contains any `if` ψ `then` ω, as shown in the paper [25]. With the full power of (bi-)filters, which are fully-fledged Haskell functions, we can implement this solution in the following bi-filter:

```
fCond (ITE c1 e1 e2) = case checkRightSpine e1 of
  True  -> Nothing
  False -> Just (ITE c1 (addPar e1) e2)

-- collect the names of the constructors in the right spine and
-- check if the collected constructors contain "IT"
checkRightSpine t = ... .
```

This bi-filter is commutative with the bi-filters generated from our directives, for it (i) only searches for non-parenthesis productions that are not declared in any other directives, and (ii) inserts only a parenthesis production when repairing incorrect CSTs. The reader may find the code of `checkRightSpine` in more detail in Figure 10.

6 Case Studies

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 case study that 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 [4]. 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 specify TIGER's syntax, while for C89 and C99 it takes respectively 183 and 237 rules without any disambiguation declarations (based on Kernighan et al. [24] 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 9. 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 [20]'s lecture notes. To be precise, we add a parenthesis production to the grammar (and discard it when converting CSTs to ASTs, so that the PassThrough property could be satisfied), for TIGER's original grammar has no parenthesis production and an expression within round parentheses is regarded as a singleton expression sequence. This modification also makes it necessary to change the enclosing brackets for expression sequences from round brackets `()` to curly brackets `{}`, which helps (LALR(1) parsers) to distinguish a singleton expression sequence from an expression within parentheses. There is also another slight change in the definition of ASTs for handling a feature not supported by current BIYACC: 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 [4], since we cannot handle the synchronisation between a list of lists (in ASTs) and a list (in CSTs) with BIYACC's current syntax.

Following Hirzel and Rose [20]'s specification, the disambiguation directives for TIGER are shown in Figure 10; for instance, we define multiplication to be left-associative. The directives also include a concrete treatment of the dangling else problem, which is usually 'not solved' when using a YACC-like (LA)LR parser generator to implement parsers: rather than resolving the grammatical ambiguity, we often rely on the default behaviour of the parser generator — preferring shift.

We have successfully tested our BIYACC program for TIGER on all the sample programs provided on the homepage of Appel's book[9], including a merge sort implementation and an eight-queen solver, and there is no problem parsing and printing them with well-behavedness guaranteed. In the following sub-sections, we will present some printing strategies described in the BIYACC program to demonstrate what BIYACC, in particular reflective printing, can achieve.

6.1 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 `|` ('or') are converted to `if` expressions in the abstract syntax: `e1 & e2` is desugared and parsed to `TCond e1 e2 (TInt 0)` and `e1 | e2` to `TCond e1 (TInt 1) e2`. The printing actions for them in BIYACC are:

[9] `https://www.cs.princeton.edu/~appel/modern/testcases/`

```
#Abstract
type TSymbol = String                  data BBool = TT | FF
                                       data MMaybe a = NN | JJ a
data Tuple a b = Tuple a b             data List a = Nil | Cons a (List a)

data TExp = TString String | TInt Int  | TNilExp | TCond TExp TExp (MMaybe TExp)
  | TLet (List TDec) TExp  | TOp TExp TOper TExp | TExpSeq (List TExp) | ...

data TOper = TPlusOp | TMinusOp | ... | TEqOp | TNeqOp | ...

data TDec = TVarDec TSymbol BBool (MMaybe TSymbol) TExp
          | TTypeDec (Tuple TSymbol TTy) | TFunctionDec TFundec

data TFundec = TFundec TSymbol (List TFieldDec) (MMaybe TSymbol) TExp
...

#Concrete
Exp ->  LetExp | ArrExp | IfThen   | IfThenElse | Prmtv
     |  ForExp | RecExp | WhileExp | Assignment | 'break' ;

VarDec -> 'var' Identifier               ':=' Exp
        | 'var' Identifier ':' Identifier ':=' Exp ;

LValue      -> Identifier  |  OtherLValue ;
OtherLValue -> LValue '.' Identifier
  |  Identifier '[' Exp ']' | OtherLValue '[' Exp ']'  ;

SeqExp  -> '{'        '}' | '{' ExpSeq '}' ;
ExpSeq  -> Exp ';' ExpSeq | Exp ;

Prmtv  -> [Paren] '(' Exp ')' {# Bracket #} | CallExp | SeqExp | ...
        | [Or]   Prmtv '|' Prmtv | [And]   Prmtv '&' Prmtv
        | [Plus] Prmtv '+' Prmtv | [Times] Prmtv '*' Prmtv | ...
        | [Neg] '-' Prmtv | Numeric | String | LValue | 'nil' ;

IfThenElse -> [ITE] 'if' Exp 'then' Exp 'else' Exp ;
IfThen     -> [IT]  'if' Exp 'then' Exp ;
...
```

Fig. 9 An excerpt of TIGER's abstract and concrete syntax. (Here we define our own BBool type and MMaybe type for avoiding name clashing with Haskell's built-in ones.)

```
TExp +> Prmtv
  TCond e1 (TInt 1) (JJ e2)  +>  [e1 +> Prmtv] '|' [e2 +> Prmtv];
  TCond e1 e2 (JJ (TInt 0))  +>  [e1 +> Prmtv] '&' [e2 +> Prmtv]; .
```

The *parse* function for these kinds of 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.

```
#Directives                          Associativity:
Priority:                            Left:  Times, Plus, And  ... ;
Times >  Plus ;                      Right: Assign,  ... ;
And   >  Or   ; ...

#OtherFilters
[ fDanglingElse :: BiFilter IfThenElse ]

fDanglingElse (ITE t1 exp1 t2 exp2 t3 exp3) =
  case checkRightSpine exp2 of
    True  -> Nothing
    False -> Just (ITE t1 exp1 t2 (addPar exp2) t3 exp3)

checkRightSpine t = let spineStrs = getRSpineCons t
                    in  and $ map (\str -> str /= "IT") spineStrs

class GetRSpineCons t where
  getRSpineCons :: t -> [String]

instance GetRSpineCons IfThenElse where
  getRSpineCons (ITE _ _ _ _ _ r) = ["ITE"] ++ getRSpineCons r

instance GetRSpineCons IfThen where
  getRSpineCons (IT _ _ _ r) = ["IT"] ++ getRSpineCons r

instance GetRSpineCons LetExp where
  getRSpineCons (LetExp1 _ _ _ _ _) = ["LetExp1"]
...
```

Fig. 10 An excerpt of the disambiguation directives for TIGER. (A type class `GetRSpineCons` is defined and implemented for collecting the constructors on the right spine of a given tree. Function `getRSpineCons` is recursively invoked for CSTs whose right-most subtree is (parsed from) a nonterminal.)

6.2 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* [39] is to print evaluation sequences in a core language in terms of a surface 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 6.1. 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

```
TCond (TNot (TInt 1)) (TInt 1) (JJ (TNot (TInt 0))) .
```

A typical call-by-value evaluator will produce the following evaluation sequence given the above AST:

```
   TCond (TNot (TInt 1)) (TInt 1) (JJ (TNot (TInt 0)))
→ TCond (TInt 0) (TInt 1) (JJ (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.

6.3 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. `foreach` 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 [13]'s guarded command language. We extend TIGER's concrete syntax with the following production rules:

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

For simplicity, we restrict the predicate produced by `CaseB` to the form `LValue '=' Numeric`, but in general this can be any expression computing integer. The reflective printing

actions for this new construct can still be written within BIYACC, but require much deeper pattern matching:

```
TExp +> Guard
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing  +>
    'guard' (CaseBs -> (CaseB -> [lv +> LValue] '=' [i +> Numeric] '->' [e1 +> Exp])
            ) 'end';
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@(TCond _ _ _))  +>
    'guard' (CaseBs -> (CaseB -> [lv +> LValue] '=' [i +> Numeric] '->' [e1 +> Exp])
                       [if2 +> CaseBs]
            ) 'end';
;;
TExp +> CaseBs
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 Nothing  +>
    (CaseB -> [lv +> LValue] '=' [i +> Numeric] '->' [e1 +> Exp]);
  TCond (TOp (TVar lv) TEqOp (TInt i)) e1 (J if2@(TCond _ _ _))  +>
    (CaseB -> [lv +> LValue] '=' [i +> Numeric] '->' [e1 +> Exp])
    [if2 +> CaseBs];
;; .
```

Although being a little 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

```
guard  choice = 1  ->  4
       choice = 2  ->  8
       choice = 3  ->  16  end
```

is parsed to

```
TCond (TOp (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (JJ
  (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))))
```

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:

```
   TCond (TOp (TVar (TSV "c")) TEqOp (TInt 1)) (TInt 4) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
     (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))))
→ TCond (TInt 0) (TInt 4) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
     (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))))
→ TCond (TOp (TVar (TSV "c")) TEqOp (TInt 2)) (TInt 8) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))
→ TCond (TInt 1) (TInt 8) (JJ
    (TCond (TOp (TVar (TSV "c")) TEqOp (TInt 3)) (TInt 16) NN))
→ TInt 8 .
```

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

```
   guard  choice = 1  ->  4
          choice = 2  ->  8
          choice = 3  ->  16  end
↛
→ guard  choice = 2  ->  8
          choice = 3 -> 16 end
↛
→ 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 `TCond`-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 `TCond`.

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.

7 Related Work

7.1 Unifying Parsing and Printing

Much research has been devoted to describing parsers and printers in a single program. For example, both Rendel and Ostermann [41] and Matsuda and Wang [35] adopt a combinator-based approach, where small components are glued together to yield more sophisticated behaviour, and can guarantee inverse properties similar to the ones of our concrete parsing and printing isomorphism (with CST replaced by AST in the equations). In Rendel and Ostermann [41]'s system (called 'invertible syntax descriptions', which we shorten to ISDs henceforth), both parsing and printing semantics are pre-defined in the combinators and the consistency is guaranteed by their partial isomorphisms, whereas in Matsuda and Wang [35]'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. [7] 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.

Although the above-mentioned systems are tailored for unifying parsing and printing, there are design differences. 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:

```
ppr' i (Minus x y) =
 parensIf (i >= 6) $ group $
   ppr 5 x <> nest 2
     (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.

Just like the conference version of BiYacc [51], all of the systems described above handle unambiguous grammars only. When the user-defined grammar (or the derived grammar) is ambiguous, the behaviour of these systems is as follows: Neither ISDs nor 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. [7] give a detailed discussion about ambiguity detection, and XSugar statically checks if 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 detection of the input grammar is performed by the employed parser generator (currently Happy), and the result is reported at compile time; if no warning is reported, the well-behavedness is always guaranteed. Note that the ambiguity detection can produce false negatives: warnings only mean that the grammar is not LALR(1) but does not necessarily mean that the grammar is ambiguous — ambiguity detection is undecidable for the full CFG [10]. Finally, we briefly discuss ambiguity detection for the filter approaches: Priority and associativity (bi-)filters can be applied to (LA)LR parse tables to resolve

(shift/reduce) conflicts [9, 25, 48, 49], and thus the completeness for simple (bi-)filters (see Definition 7) on LALR(1) grammars can be statically checked. However, our implementation does not support it, for bi-filter directives are more general, as stated in the beginning of Section 5.4, and therefore cannot be transformed to the underlying parser generator's YACC-style directives. Finding a way to directly apply priority and associativity bi-filters to parse tables (generated by HAPPY) is left as future work.

7.2 Generalised Parsing, Disambiguation, and Filters

Programming languages are usually designed to be unambiguous and thus require a single CST; various parser-dependent disambiguation methods such as grammar transformation [29] and parse table conflicts elimination [22] have been developed to guide the parser to produce a single correct CST [25]. On the other hand, natural languages that are inherently ambiguous usually require their parsing algorithms to produce all the possible CSTs; this requirement gives rise to algorithms such as Earley [15] and generalised LR [47] (GLR for short). Although these parsing algorithms produce all the possible CSTs, both their time complexity and space complexity are reasonable. For instance, GLR runs in cubic time in the worst situation and in linear time if the grammar is 'almost unambiguous' [45].

The idea to relate generalised parsing with parser-independent disambiguation for programming languages is proposed by Klint and Visser [25]. They proposed two classes of filters, *property filters* (defined in terms of predicates on a single tree) and *comparison filters* (defined in terms of relations among trees), but we only adapt and bidirectionalise predicate filters in this paper. One difficulty lies in the fact that it is unclear how to define *repair* for comparison filters, as they generally select better trees rather than absolutely correct ones — in the printing direction, since *put* only produces a single CST, we do not know whether this CST needs repairing or not (for there is no other CST to compare). This is also one of the most important problems for our future work.

Parser-independent disambiguation (for handling priority and associativity conflicts) can also be found in LaLonde and des Rivieres's [29] and Aasa's [1] work. At first glance, our *repair* function is quite similar to LaLonde and des Rivieres's post-parse *tree transformations* that bring a CST into an *expression tree*, on whose nodes additional restrictions of priority and associativity are imposed. To be simple (but not completely precise), a CST's corresponding expression tree is obtained by first dropping all the nodes constructed from injective productions[10] (note that parentheses nodes are still kept) and then use *precedence-introducing tree transformation* to reshape the result. The precedence-introducing tree transformation will 'repair' by rotating all the adjacent nodes of the tree where priority or associativity constraint is violated. On the contrary, our *repair* function is simpler and only introduces parentheses to where the *judge* function returns `False`. In short, their tree transformations are a kind of parser-independent disambiguation which does not require generalised parsing; however, those tree transformations are (almost) not applicable in the printing

[10] An injective production, or a chain, is a production whose right-hand side is a single nonterminal; for instance, `E -> N`.

direction if well-behavedness is taken into consideration (due to the rotation of CSTs). Furthermore, it is not clear whether their approach can be generalised to handle other types of conflicts rather than the ones caused by priority and associativity.

There is much research on how to handle ambiguity in the parsing direction as discussed above; conversely, little research is conducted for 'handling ambiguity in the printing direction' and we find only one paper [8] that describes how to produce correct program text regarding priority and associativity, which is also one of the bases of our work. We extend their work [8] by allowing bracket attribute to work with injective productions such as `E -> T; T -> F; F -> '(' E ')' {# Bracket #};`. (The previous work seems to only support bracket attribute in the form of `E -> '(' E ')' {# Bracket #};`; whether the nonterminal `E` on the left-hand side and right-hand side can be different is not presented clearly.)

Finally, we compare our approach with the conventional ones in general. In history, a printer is believed to be much simpler than a parser and is usually developed independently (of its corresponding parser). While a few printers choose to produce parentheses at every occasion naively, most of them take disambiguation information (for example, from the language's operator precedence table) into account and try to produce necessary parentheses only. However, as the YACC-style conventional disambiguation [22] is parser-dependent, this parentheses-adding technique is also printer-dependent. As the post-parse disambiguation increases the modularity of the (front end of the) compiler [29], we believe that our post-print parentheses-adding increases the modularity once again. Additionally, the unification of disambiguation for both parsing and printing makes it possible for us to impose bi-filter laws, which further makes it possible to guarantee the well-behavedness of the whole system.

7.3 Get-based vs Putback-based Parsing and Reflective Printing

Our work is theoretically based on asymmetric lenses [18] of bidirectional transformations [11, 19], particularly taking inspiration from the recent progress on putback-based bidirectional programming [16, 26, 27, 37, 38]. As explained in Section 3, the purpose of bidirectional programming is to relieve the burden of thinking bidirectionally — the programmer writes a program in only one direction, and a program in the other direction is derived automatically. We call a language *get-based* when programs written in the language denote *get* functions, and call a language *putback-based* when its programs denote *put* functions. In the context of parsing and reflecting printing, the get-based approach lets the programmer describe a parser, whereas the putback-based approach lets the programmer describe a printer. Below we discuss in more depth how the putback-based methodology affects BIYACC's design by comparing BIYACC with a closely related, get-based system.

Martins et al. [34] introduces an attribute grammar–based BX 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 *forward* transformation (i.e. *get*), with a backward transformation (i.e. *put*) being automatically generated. For example, the BIYACC actions in lines

34–36 of Figure 2 can be expressed in Martins et al.'s system as

$$\begin{aligned} get_A^E(plus \quad (x, \text{`+'}, y)) &\to add(get_A^E(x), get_A^T(y)) \\ get_A^E(minus(x, \text{`-'}, y)) &\to sub\,(get_A^E(x), get_A^T(y)) \\ get_A^E(et(e)) &\to get_A^T(e) \end{aligned}$$

which describes how to convert certain forms of CSTs to corresponding ASTs. The similarity is evident, and raises the question as to how get-based and putback-based approaches differ in the context of parsing and reflective printing.

The difference lies in the fact that, with a 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 can give the user fuller control. For example, when no source is given and more than one rules can be applied, Martins et al.'s system chooses, by design, the one that creates the most specialised version. This might or might not be ideal for the user of the system. For example: Suppose that we port to Martins et al.'s system the BIYACC action that relates TIGER's concrete '`&`' operator with a specialised abstract `if` expression in Section 6.1, coexisting with a more general rule that maps a concrete `if` expression to an abstract `if` expression. Then printing the AST `TCond (TSV "a") (TSV "b") 0` from scratch will and can only produce `a & b`, as dictated by the system's hard-wired printing logic. By contrast, the user of BIYACC can easily choose to print the AST from scratch as `a & b` or `if a then b else 0` by suitably ordering the printing actions.

This difference is somewhat subtle, and one might argue that Martins et al.'s design simply went one step too far — if their system had been designed to respect the rule ordering as specified by the user, as opposed to always choosing the most specialised rule, the system would have given its user the same flexibility as BIYACC. Interestingly, whether to let user-specified rule/action ordering affect the system's behaviour is, in this case, exactly the line between get-based and putback-based design. The user of Martins et al.'s system writes rules to specify a forward transformation, whose semantics is the same regardless of how the rules are ordered, and thus it would be unpleasantly surprising if the rule ordering turned out to affect the system's behaviour. We can view this from another angle: If the user is required to specify a forward transformation while customising the backward behaviour by carefully ordering the rules, then the purpose of a bidirectionalisation system — which is to reduce the problem of writing bidirectional transformations to unidirectional programming — is largely defeated. By contrast, the user of BIYACC only needs to think in one direction about the printing behaviour, for which it is natural to consider how the actions should be ordered when an AST has many corresponding CSTs; the parsing behaviour will then be automatically and uniquely determined. In short, relevance of action ordering is incompatible with get-based design, but is a natural consequence of putback-based thinking.

8 Conclusion

We conclude the paper by summarising our contributions:

- We have presented the design and implementation of BIYACC, with which the programmer can describe both a parser and a reflective printer for a fully disambiguated context-free grammar in a single program. Our solution guarantees the consistency properties (1) and (2) by construction.
- We proposed the notion of bi-filters, which enables BIYACC to disambiguate ambiguous grammars while still respect the consistency properties. This is the main new contribution compared to the previous conference version [51].
- We have demonstrated that BIYACC can support various tasks of language engineering, from traditional constructions of basic machinery such as printers and parsers to more complex tasks such as resugaring, simple refactoring, and language evolution.

References

1. Aasa A (1995) Precedences in specifications and implementations of programming languages. Theoretical Computer Science 142(1):3–26
2. Afroozeh A, Izmaylova A (2015) Faster, practical gll parsing. In: Franke B (ed) Compiler Construction, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 89–108
3. Aho AV, Johnson SC, Ullman JD (1975) Deterministic parsing of ambiguous grammars. Commun ACM 18(8):441–452
4. Appel AW (1998) Modern Compiler Implementation in ML. Cambridge University Press
5. Bille P (2005) A survey on tree edit distance and related problems. Theor Comput Sci 337(1-3):217–239
6. Boulton R (1966) Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Tech. Rep. Number 390, Computer Laboratory, University of Cambridge
7. Brabrand C, Møller A, Schwartzbach MI (2008) Dual syntax for XML languages. Information Systems 33(4):385–406
8. van den Brand M, Visser E (1996) Generation of formatters for context-free languages. ACM Trans Softw Eng Methodol 5(1):1–41
9. van den Brand MGJ, Scheerder J, Vinju JJ, Visser E (2002) Disambiguation Filters for Scannerless Generalized LR Parsers, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 143–158
10. Cantor DG (1962) On the ambiguity problem of backus systems. J ACM 9(4):477–479
11. Czarnecki K, Foster JN, Hu Z, Lämmel R, Schürr A, Terwilliger J (2009) Bidirectional transformations: A cross-discipline perspective. In: International Conference on Model Transformation, Springer, Lecture Notes in Computer Science, vol 5563, pp 260–283
12. De Jonge M (2002) Pretty-printing for software reengineering. In: International Conference on Software Maintenance, IEEE, pp 550–559
13. Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8):453–457
14. Duregård J, Jansson P (2011) Embedded parser generators. In: Haskell Symposium, ACM, pp 107–117
15. Earley J (1970) An efficient context-free parsing algorithm. Commun ACM 13(2):94–102
16. Fischer S, Hu Z, Pacheco H (2015) The essence of bidirectional programming. Science China Information Sciences 58(5):1–21
17. Foster JN (2009) Bidirectional programming languages. PhD thesis, University of Pennsylvania
18. Foster JN, Greenwald MB, Moore JT, Pierce BC, Schmitt A (2007) Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems 29(3):17
19. Gibbons J, Stevens P (eds) (2018) International Summer School on Bidirectional Transformations (Oxford, UK, 25–29 July 2016), Lecture Notes in Computer

Science, vol 9715. Springer
20. Hirzel M, Rose KH (2013) Tiger language specification. https://cs.nyu.edu/courses/fall13/CSCI-GA.2130-001/tiger-spec.pdf
21. Hu Z, Ko HS (2018) Principles and practice of bidirectional programming in BiGUL. In: International Summer School on Bidirectional Transformations (Oxford, UK, 25–29 July 2016), Lecture Notes in Computer Science, vol 9715, Springer, chap 4, pp 100–150
22. Johnson SC (1975) Yacc: Yet another compiler-compiler. AT&T Bell Laboratories Technical Reports (AT&T Bell Laboratories Murray Hill, New Jersey 07974) (32)
23. de Jonge M, Visser E (2012) An algorithm for layout preservation in refactoring transformations. In: International Conference on Software Language Engineering, Springer, Lecture Notes in Computer Science, vol 6940, pp 40–59
24. Kernighan BW, Ritchie DM, Ejeklint P (1988) The C programming language, vol 2. prentice-Hall Englewood Cliffs
25. Klint P, Visser E (1994) Using filters for the disambiguation of context-free grammars. In: Proc. ASMICS Workshop on Parsing Theory, Citeseer, pp 1–20
26. Ko HS, Hu Z (2018) An axiomatic basis for bidirectional programming. Proceedings of the ACM on Programming Languages 2(POPL):41
27. Ko HS, Zan T, Hu Z (2016) BiGUL: A formally verified core language for putback-based bidirectional programming. In: Partial Evaluation and Program Manipulation, ACM, pp 61–72
28. Kort J, Lämmel R (2003) Parse-tree annotations meet re-engineering concerns. In: International Workshop on Source Code Analysis and Manipulation, IEEE, pp 161–170
29. LaLonde WR, des Rivieres J (1981) Handling operator precedence in arithmetic expressions with tree transformations. ACM Trans Program Lang Syst 3(1):83–103
30. Lämmel R, Jones SP (2003) Scrap your boilerplate: A practical design pattern for generic programming. SIGPLAN Not 38(3):26–37
31. Macedo N, Pacheco H, Cunha A, Oliveira JN (2013) Composing least-change lenses. In: International Workshop on Bidirectional Transformations, EASST, Electronic Communications of the EASST
32. Marlow S, Gill A (2001) The parser generator for haskell. https://www.haskell.org/happy/
33. Marlow S, et al. (2010) Haskell 2010 language report. `https://www.haskell.org/onlinereport/haskell2010/`
34. Martins P, Saraiva J, Fernandes JP, Van Wyk E (2014) Generating attribute grammar-based bidirectional transformations from rewrite rules. In: Partial Evaluation and Program Manipulation, ACM, pp 63–70
35. Matsuda K, Wang M (2018) FliPpr: A system for deriving parsers from pretty-printers. New Generation Computing 36(3):173–202
36. Norell U (2007) Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology
37. Pacheco H, Hu Z, Fischer S (2014) Monadic combinators for putback style bidirectional programming. In: Partial Evaluation and Program Manipulation, ACM, pp 39–50

38. Pacheco H, Zan T, Hu Z (2014) BiFluX: A bidirectional functional update language for XML. In: Principles and Practice of Declarative Programming, ACM, pp 147–158
39. Pombrio J, Krishnamurthi S (2014) Resugaring: Lifting evaluation sequences through syntactic sugar. Programming Language Design and Implementation pp 361–371
40. Pombrio J, Krishnamurthi S (2015) Hygienic resugaring of compositional desugaring. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA, no. 13 in ICFP 2015, pp 75–87
41. Rendel T, Ostermann K (2010) Invertible syntax descriptions: Unifying parsing and pretty printing. In: Haskell Symposium, ACM, pp 1–12
42. Reps T, Teitelbaum T (1984) The synthesizer generator. In: ACM Sigplan Notices, ACM, vol 19, pp 42–48
43. Reps T, Teitelbaum T, Demers A (1983) Incremental context-dependent analysis for language-based editors. ACM Transactions on Programming Languages and Systems (TOPLAS) 5(3):449–477
44. Scott E, Johnstone A (2010) GLL parsing. Electronic Notes in Theoretical Computer Science 253(7):177 – 189
45. Scott E, Johnstone A, Economopoulos R (2007) BRNGLR: a cubic tomita-style glr parsing algorithm. Acta Informatica 44(6):427–461
46. Sheard T, Jones SP (2002) Template meta-programming for Haskell. In: Haskell Workshop, ACM, pp 1–16
47. Tomita M (1985) An efficient context-free parsing algorithm for natural languages. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, IJCAI '85, pp 756–764
48. Visser E (1995) A case study in optimizing parsing schemata by disambiguation filters. In: Proceedings Accolade '95, Amsterdam: Dutch Graduate School in Logic (OZSL), pp 153–167
49. Visser E (1997) Syntax definition for language prototyping. PhD thesis, University of Amsterdam
50. Younger DH (1967) Recognition and parsing of context-free languages in time n3. Information and Control 10(2):189 – 208
51. Zhu Z, Zhang Y, Ko HS, Martins P, Saraiva Ja, Hu Z (2016) Parsing and reflective printing, bidirectionally. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, ACM, New York, NY, USA, SLE 2016, pp 2–14