The Journal of Systems & Software

BIT: A template-based approach to incremental and bidirectional model-to-text transformation

--Manuscript Draft--

Manuscript Number:	JSSOFTWARE-D-23-01182
Article Type:	VSI:MDE4SA
Keywords:	bidirectional transformation; model-to-text transformation; template language; model-driven development; round-trip engineering
Abstract:	Model-driven development is a model-centric software development paradigm that automates the development process by converting high-level models into low-level code and documents. To maintain synchronization between models and code/documents—which can evolve independently—this paper introduces BIT, a bidirectional language that can serve as a conventional template language for model-to-text transformations. However, a BIT program can function as both a printer, generating text by filling template holes with values from the input model, and a parser, putting parsed values back into the model. BIT comprises a surface language for better usability and a core language for formal definition. We define the semantics of the core language based on the theory of bidirectional transformation, and provide the translation from the surface to the core. We present the proof sketch of the well behavedness of BIT as a formal evidence of soundness. We also conduct two preliminary case studies to empirically demonstrate the expressiveness of BIT. Based on the proof and the case studies, BIT covers the major features of existing template languages, and offers sufficient expressiveness to define real-world model-to-text transformations that can be executed bidirectionally and incrementally.

BIT: A template-based approach to incremental and bidirectional model-to-text transformation*

Xiao He[a], Tao Zan[b,*]

[a]School of Computer and Communication Engineering, University of Science and Technology Beijing, No. 30, Xueyuan Road, Haidian district, 100083, Beijing, China
[b]Longyan University, No. 1, North Dongxiao Road, Xinluo district, 364012, Longyan, China

Abstract

Model-driven development is a model-centric software development paradigm that automates the development process by converting high-level models into low-level code and documents. To maintain synchronization between models and code/documents—which can evolve independently—this paper introduces `BIT`, a bidirectional language that can serve as a conventional template language for model-to-text transformations. However, a `BIT` program can function as both a *printer*, generating text by filling template holes with values from the input model, and a *parser*, putting parsed values back into the model. `BIT` comprises a surface language for better usability and a core language for formal definition. We define the semantics of the core language based on the theory of bidirectional transformation, and provide the translation from the surface to the core. We present the proof sketch of the well behavedness of `BIT` as a formal evidence of soundness. We also conduct two preliminary case studies to empirically demonstrate the expressiveness of `BIT`. Based on the proof and the case studies, `BIT` covers the major features of existing template languages, and offers sufficient expressiveness to define real-world model-to-text transformations that can be executed bidirectionally and incrementally.

Keywords: bidirectional transformation, model-to-text transformation, template language, model-driven development, round-trip engineering

1. Introduction

Model-driven development (MDD) is a model-centric software development paradigm [1, 2, 3] that has been intensively studied and applied in both academia

*This work is funded by National Key Research and Development Program of China (No. 2023YFB3002903), Natural Science Foundation of Fujian Province for Youths (No. 2021J05230), Beijing Natural Science Foundation (No. 4192036).
*Corresponding author
Email addresses: `hexiao@ustb.edu.cn` (Xiao He), `zan@lyun.edu.cn` (Tao Zan)

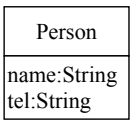
```
def generateJavaClass(UMLClass c)
'''
class «c.name» {
  «FOR p : c.ownedProperty»
    public «p.type.name» «p.name»;
  «ENDFOR»
  «FOR o : c.ownedOperation»
    public «o.returnType.name» «o.name»(
        «FOR p : o.parameters SEPARATOR ', '»«p.type.name» «p.name»«ENDFOR») {
      throw new UnsupportedOperationException();
    }
  «ENDFOR»
}
'''
```

hole

control directive

text literals

(a) A code template in Xtend



(b) Input CD

```
class Person {
  public String name;
  public String tel;
}
```

(c) Generated code

```
class Person {
  public String name;
  ~~public String tel;~~
  public int age;
}
```

(d) Changed code

Figure 1: A template example and its application

and industry over past decades [4, 5, 6]. In MDD, a software system is generally developed and maintained by (1) specifying the system models at the high abstraction level and then (2) transforming the models into some low-level artifacts, including low-level models, source code, and documents, using model-to-model transformation [7] and model-to-text (M2T) transformation [8].

M2T transformations are typically realized using *templates*. A template, which consists of text literals, holes, and control directives, is a unidirectional transformation from models to text (e.g., code and documents). For example, Figure 1 illustrates a simple template *generateJavaClass* written in Xtend [9]. This template generates a Java class from a UML class. Assuming that the input UML class (as shown in Figure 1b) is provided, a snippet of Java code (as shown in Figure 1c) will be generated.

In practice, it is inevitable for developers to manually modify and customize the generated code/documents [10]. For instance, developers may modify the code as shown in Figure 1d, where field `tel` is deleted and field `age` is added. Consequently, the UML class (Figure 1b) and the code become inconsistent.

How to synchronize high-level models and derived artifacts to maintain their consistency has become a fundamental challenge in model-driven community. Numerous research efforts have been made on the synchronization over models (i.e., graph-like data structures) [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. Nevertheless, there are very few solutions for synchronizing models and text. A practical way of model-code synchronization, as employed in many model-driven tools (e.g., Eclipse Modeling Framework (EMF) and Papyrus), is to develop a separate reverse engineering module, as a companion of the code generator, which can convert the text back to the original model. However, this practical solution has two significant limitations as follows.

1. It requires more development costs to implement both the code generator

and the reverse engineering module.

2. A code generator and the corresponding reverse engineering module are expected to have consistent behaviors: if code C is generated from model M, the reverse engineering module should derive a model M' from C, such that M and M' are identical; if model M is reverse-engineered from code C, the code generator should produce code C' from M, such that C and C' are identical. Because they are *independently developed* and *algorithmically different*, ensuring their behavioral consistency is challenging.

Bidirectional transformation (BX) [22, 23, 24, 25, 26, 27, 28] can serve as the foundation of data synchronization. A BX program is a *single* specification that can be *consistently* evaluated in both forward and backward directions. Following the principles of BX, Yu et al. [29] proposed a framework for model-code synchronization that facilitates bidirectional conversion between Java code and Ecore models. However, their approach is limited code generation from Ecore models and is not generally applicable.

In this paper, we aim to introduce a novel template-based approach, called `BIT`, for model-text synchronization. `BIT` enables developers to write a single template that can be interpreted as a M2T transformation (known as a *printer*), similar to existing template engines. It can also be used to automatically derive a reverse engineering module (known as a *parser*) to reduce development costs. Furthermore, `BIT` ensures in theory that the derived printer and parser exhibit behavioral compatibility (i.e., they satisfy the round-trip properties). Specifically, we first propose a general-purpose template language for developers to specify M2T transformation. For better usability, our template language, which serves as a surface language, largely inherits the grammar of Xtend, with a few syntactic extensions to enable backward evaluation. Second, we design a core language, to which our template language can be translated. The core language consists of 5 primitive BXs and 8 combinators. A primitive BX tells how to parse/print a specific value according to a certain format, and a combinator allows us to combine smaller BXs into a larger one.

This paper focuses on the following two challenges that hinder the application of existing approaches to the synchronization between models and text.

1. Existing BX approaches are defined upon structured data (e.g., trees [25], relational databases [28], graphs [26, 27], and models [20]). However, a string, which is a sequence of characters, is generally considered as unstructured. To address the unstructured nature of strings, we ask template developers to annotate each template hole with a lexical rule, so that the derived parser can determine the boundary of the string generated by the hole for the given input string. Furthermore, we adopt the mechanism of partial grammars (inspired by [30]) to analyze the structure of strings.
2. Existing BX approaches usually assume that BXs are pure functions. However, in our template-based bidirectional printing/parsing, some computations, such as local assignments and incremental parsing, require computational side effects. To handle these effects, we propose the concepts of

accumulative BXs and *effect-binding BXs* to manage incremental parsing and local assignments, respectively.

The rest of this paper is structured as follows. Section 2 introduces the background information and presents a demonstration of our approach. Section 3 presents the detailed definitions of the surface language and the core language of `BIT`. Section 4 discusses the proof sketch of the well behavedness of the `BIT` semantics. Section 5 presents two case studies. Section 6 discusses the related work. The last section concludes the paper and future work.

2. Background and demonstration

2.1. Bidirectional transformation

A bidirectional transformation (BX) is a program that bidirectionally converts between the source type S and the view type V. An asymmetric BX, written as $S \leftrightarrow V$, can be viewed as a pair (get, put) of functions. The forward transformation $get : S \to V$ generates a view value from the source, while the backward transformation $put : S \times V \to S$ updates the original source by taking the modified view into account. A pair (get, put) of functions form a *well-behaved* BX iff. they satisfy the corresponding round-trip properties. For asymmetric BXs, the following round-trip properties must hold:

$$put\ s\ (get\ s) = s \qquad \text{(GetPut)}$$

$$get\ (put\ s\ v) = v \qquad \text{(PutGet)}$$

GetPut law states that updating the source s with the unmodified view generated from s should not cause any changes to s, while PutGet law states that if we perform the forward transformation immediately after the backward transformation with the view v, we should obtain the same v.

Consider a concrete example. Assume that

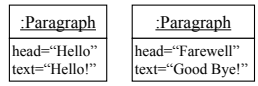
$$get_{head}\ [x_1, x_2, ..., x_n] = x_1$$
$$put_{head}\ [x_1, x_2, ..., x_n]\ x_1' = [x_1', x_2, ..., x_n]$$

The forward transformation extracts the head element x_1 of a source array $[x_1, x_2, ..., x_n]$ as the view value; the backward transformation simply replaces the head element of the original source array with the given view value x_1' to produce an updated array. It is easy to verify that both GetPut and PutGet laws hold, so the two functions form a well-behaved BX.

Bidirectional programming is a programming paradigm that enables developers to define a single specification from which a well-behaved BX program can be derived, thereby minimizing the development efforts. There are three basic approaches to bidirectional programming: the get-based, the putback-based, and the relational approach. The get-based approach [15, 26] derives the backward transformation *put* from the forward transformation *get*, while the putback-based approach [22, 20, 28] derives *get* from the backward transformation *put*; and the relational one [11] derives both *get* and *put* from a set of consistency relations over the source and the view.

```
1.  def someHTML(List<Paragraph> paragraphs)
2.  '''
3.    <html>
4.      <body>
5.        «var no = 1»
6.        «FOR p : paragraphs»
7.          «IF p.head != null»
8.          «no = no + 1»
9.            <h1>«no».«p.head.toUpperCase»</h1>
10.         «ENDIF»
11.         <p>
12.           «p.text»
13.         </p>
14.       «ENDFOR»
15.     </body>
16.   </html>
17. '''
```

(a) Xtend template



Printing

```
<html>
  <body>
    <h1>1.HELLO</h1>
    <p>
      Hello!
    </p>
    <h1>2.FAREWELL</h1>
    <p>
      Good Bye!
    </p>
  </body>
</html>
```

(b) Generated HTML

Figure 2: An Xtend template and its generated HTML document

2.2. Xtend templates

Xtend is a dialect of Java that improves on many aspects of Java, such as extension methods, operator overloading, and template expressions. Xtend has been used in mobile development, Web development, and model-driven domain-specific language engineering. In particular, the template expressions in Xtend allow for readable string concatenation and text generation, which are frequently used for code/document generation.

Figure 2a shows an Xtend template. In Xtend, templates are surrounded by triple single quotes ('''); template holes and control directives are placed within « and ». For example, «`p.head.toUpperCase`» in line 9 is a template hole, which is intended to replace the placeholder with the evaluation result of `p.head.toUpperCase` at runtime. As for control directives, Xtend templates support loops (e.g., lines 6–14), conditions (e.g., lines 7–10), and assignments (e.g., lines 5 and 8). Within an Xtend template, other templates may be invoked.

Xtend compiles the template in Figure 2a into a Java method. If we input a list of paragraphs (each paragraph consisting of a head and a text field), the method generates HTML code by filling in the field values in the holes. For example, supposing that the input paragraphs are [{head="Hello",text="Hello!"}, {head="Farewell",text="Good Bye!"}] in textual[1], the template will produce HTML code as shown in Figure 2b.

If we want to modify the generated text (e.g., we want to change "1.HELLO" in Figure 2b to "1.GREETING") and keep the text consistent with the input data, we must go back to the input and locate the fields that affect the text fragment to be changed. After modifying the input, we must re-run the text generation to see if the text is updated as expected.

[1]For simplicity, we may represent a model, e.g., the one in Figure 2b, in a JSON-like format, which can be supported by our tool.

```
1.  type Paragraph = {head:String, text:String}
2.  template someHTML(paragraphs:[Paragraph])
3.  '''
4.    <html>
5.      <body>
6.        «var no = 1»
7.        «FOR p:Paragraph IN paragraphs»
8.          «IF p.head != null»
9.          «no = no + 1»
10.           <h1>«no|INT».«p.head.toUpperCase»</h1>
11.         «ENDIF»
12.         <p>
13.           «p.text»
14.         </p>
15.       «ENDFOR»
16.     </body>
17.   </html>
18. '''
```

(a) BIT template example

```
<html>
 <body>
  <h1>1. GREETING</h1>
  <p>
   Hello!
  </p>
  <h1>2. FAREWELL</h1>
  <p>Good Bye!</p>
  <h1>3. APPRECIATION</h1>
  <p>
   Thanks!
  </p>
 </body>
</html>
```

Parsing

```
[{head="Greeting",text="Hello!"},
 {head="Farewell",text="Good Bye!"},
 {head="appreciation",text="Thanks!"}]
```

(b) Parsing modified text

Figure 3: Demonstration of BIT template (colored background shows the changed text layout)

2.3. A taste of our approach

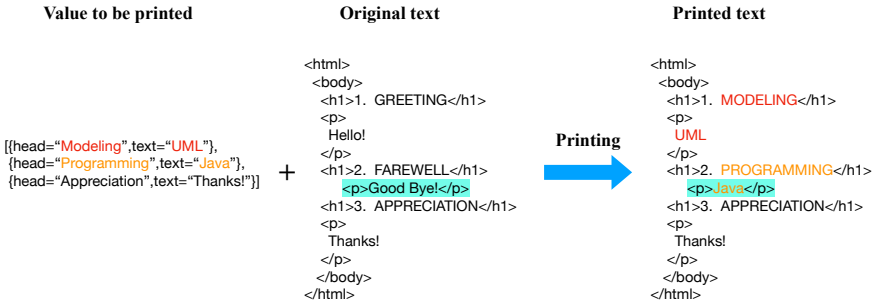
Figure 3a shows the template defined in our BIT approach, which corresponds to the Xtend template in Figure 2a. A BIT template shares a similar syntax to a Xtend template, with the key difference being that in the BIT template, each template hole is annotated with a lexical rule that guides our approach in the parsing mode. For example, «no|INT» in line 10 indicates that this hole will be filled with a string that is produced by the expression no and conforms to lexical rule INT, where the rule is defined by regular expression -?[0-9]+. If the lexical rule is missing (e.g., the hole in line 13), then our tool implementation will try to infer a lexical rule.

A BIT template can be used as a conventional template. Nevertheless, our approach is unique from existing template languages in two significant aspects.

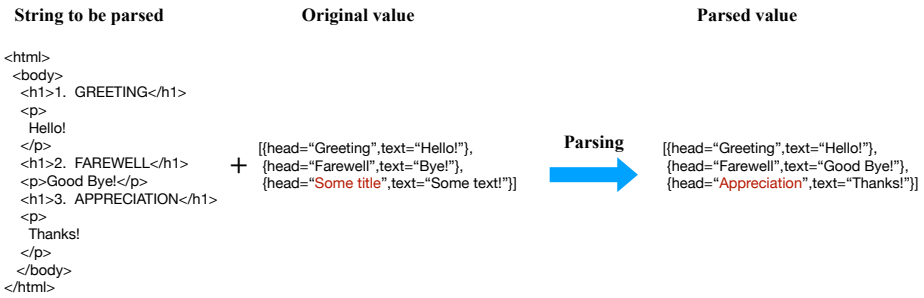
First, our approach allows for the direct modification to the generated text. By bidirectonalizing the template, it can propagate text changes back to the input. For example, we change the text in Figure 2b into Figure 3b by alerting the content of the first h1, appending a new fragment "APPRECIATION", and adjusting the spaces. The derived parser of this template reads the changed text and updates the input to [{head="Greeting",text="Hello!"}, {head="Farewell", text="Good Bye!"}, {head="appreciation",text="Thanks!"}].

Second, our approach allows for incremental printing of a value on an existing string. As illustrated in Figure 4a, supposing that we print [{head="Modeling", text="UML"}, {head="Programming", text="Java"}, {head="Appreciation", text = "Thanks!"}] based on the original text, our approach rewrites the original text with the new value while attempting to retain as much of the original string as possible. Therefore, the printed text maintains the white-space layout of the original text. Note that our approach keeps not only white-spaces but also non-whitespaces during incremental printing if specific directives (e.g., DEFAULT construct, see Section 3.1) are provided in the template definition.

Third, our approach also supports incremental parsing, as illustrated in Fig-



(a) Incremental printing (colored background implies the changed text layout)



(b) Incremental parsing

Figure 4: Incremental printing and parsing in our approach

ure 4b. The string to be parsed is identical to the one in Figure 3b. However, if we provide an original value that contains an extra paragraph with a head of "Some title", then the parsed value will be different from the one in Figure 3b—the new head of the third paragraph will be "Appreciation", rather than "appreciation", because the first character in the old head is capitalized as "S".

3. The BIT approach

This section presents the technical details of our approach. Section 3.1 introduces the surface language of BIT, which is designed for the template designers; Section 3.2 discusses the formal foundation, explaining the concepts of accumulative BXs and effect-binding BXs and defining the generic structure of a BIT primitive; Section 3.3 defines the core language of BIT that contains 5 primitives and 8 combinators, into which the surface language can be translated.

3.1. The surface language

BIT is a template-based approach for synchronizing models and text. BIT allows developers to define text templates, just like the ones in classical MDD. Subsequently, BIT derives a BX program, consisting both a printer and a parser, from these text templates. To facilitate the adoption of BIT, we have defined a surface language for developers, whose essential grammar is shown in Figure 5.

TemplateUnit := TemplateList
Template := template Name (ParameterListComma) ''' TempFragment '''
Type := record types, tuple types, list types, and primitive types
Parameter := VariableDecl
VariableDecl := Name : Type
TempFragment := TempLiteral ‖ TempLiteral HoleOrControl TempFragment
HoleOrControl := Hole ‖ Control
Hole := «Expr | LexRule»
Control := «IF Expr» TempFragment ElseBranch «ENDIF»
‖ «FOR VariableDecl IN Expr ForLits» TempFragment «ENDFOR»
‖ «DEFAULT | LexRule» TempFragment «ENDDEFAULT»
‖ «UNORD» TempFragment UnordFragList «ENDUNORD»
‖ «FINAL» TempFragment «ENDFINAL»
‖ «VAR VariableDecl = Expr» ‖ «Name = Expr»
ElseBranch := «ELSEIF Expr» TempFragment ElseBranch
‖ «ELSE» TempFragment ‖ ε
ForLits := ForLit ForLits ‖ ε
ForLit := SEPARATOR STRING ‖ BEFORE STRING ‖ AFTER STRING
UnordFragList := || TempFragment ‖ || TempFragment UnordFragList
Expr := basic arithmetic, relational, and boolean expressions
‖ instanceof expressions
‖ Expr PathCall ‖ CallExpr ‖ ValueLiteral
PathCall := . Name PathCall ‖ . CallExpr PathCall ‖ ε
CallExpr := Name (ExprListComma)
TempLiteral := any char sequence that does not contain « and '''
LexRule := regular expressions or predefined rule names
Name := identifiers

Figure 5: Essential grammar of the surface language

Notation: SmallCap denotes non-terminals; SansSeri denotes terminal constants; ε means nothing; if unspecified, a non-terminal X-List (e.g., TemplateList) is expanded into ε or X X-List ‖ X, while X-ListComma is expanded into ε or X, X-ListComma ‖ X.

For simplicity, the Type t in BIT can be a record type {f_1:t_1, f_2:t_2, ...} (e.g., {name:String, age:int}), a tuple type (t_1, t_2,...) (e.g., (String, int)), a list type [t'] (e.g., [int]), or a primitive type (i.e., int, String, and boolean). It is possible to name a type in BIT, so that the type can be referred by this name. For example, in Figure 3a, Paragraph is defined as {head:String, text:String}.

The ValueLiteral v is a value of a specific type. It can be a record (e.g., {head="Greeting", text="Hello!"}), a tuple (e.g., ("Hello", 1)), a list (e.g., [1,2,3,4,5]), or a primitive value (e.g., "Hello", 1, true, false, and null).

We assume that the input value (i.e., the model to be synchronized) of a template be encoded as a record. For example, the input of the template in Figure 3a is a record containing {paragraphs=[{head=...,text=...},{head=...,text=...},...].

Just like conventional programming languages, expressions (Expr) in BIT include basic arithmetic expressions (e.g., +, -), relational expressions (e.g.,

==, !=, >, <), boolean expressions (e.g., &&, ||, !), instanceof expressions, path calls, and template calls (i.e., a CallExpr not occurring in a path call). We assume that every expression, except for the template call, is equipped with bidirectional semantics, as defined in existing BX languages [15, 31, 32]; while the semantics of template calls is specified in Section 3.3.

At first glance, the surface language appears to have a similar syntax to the Xtend template expressions. A `BIT` template (Template) starts with keyword template, followed by a template name and a parameter list. The body of a template is a TempFragment surrounded by '''. In brief, a TempFragment is a string containing template holes (Hole) and control directives (Control), e.g.,

<h1>«p.head|ID»</h1>

A hole/control directive is a construct marked by « and ». Like existing template languages, a hole specifies the dynamic content that will be filled during text generation, i.e., the result of the hole expression (e.g., p.head). `BIT` requires a hole to be annotated with a lexical rule (LexRule) to facilitate deriving a parser for that hole. A LexRule is a regular expression or a rule name bound to a regular expression, e.g., ID refers to [_a-zA-Z][_a-zA-Z0-9]*.

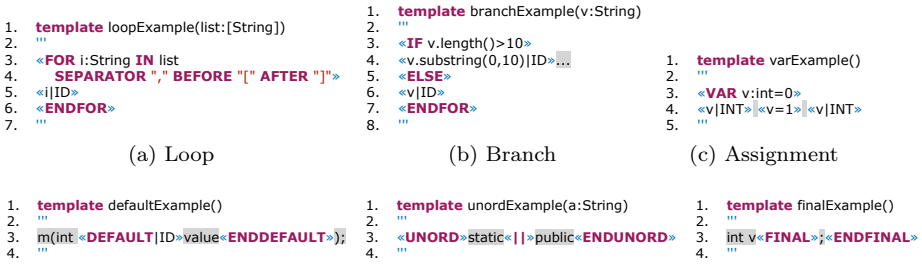
`BIT` supports common control directives, including loops and conditionals.

A loop (i.e., the FOR construct) is used to print values in a list. Figure 6a shows a concrete loop example that aims to print a list of Strings. Line 3 defines an iterator variable i:String to enumerate the strings in list. For each string bound to i, line 5 prints it with the hole «i|ID». Similar to Xtend, we can specify a separator string, and starting/ending strings (see line 4). The separator string will be inserted automatically between two consecutive iterations; the starting and ending strings will be appended before and after the loop that has at least one iteration, respectively. For the template in Figure 6a, if list=["a","b"], then it yields string "[a,b]"; if list=["a"], then it prints out "[a]"; however, if list is empty, then it produces an empty string. In parsing mode, a loop attempts to parse the string according to the loop body and returns a list.

A conditional (i.e., the IF construct) prints different branches according to the branch condition. As shown in Figure 6b, this template aims to print a string v: if the string length is greater than 10 (line 3), then it prints the first 10 characters and appends "..." (line 4); otherwise, it prints the entire string (line 6). For instance, if v="abcdefghight", then the template selects the first branch and prints out "abcdefghig...". In theory, the ELSE branch is required. If the ELSE branch is missing in practice, then `BIT` will append a pseudo else-branch that prints nothing. In parsing mode, a conditional tries to parse the input string with different branches. It will select the branch that consumes more characters, and will update the model to enforce the corresponding branch condition.

`BIT` also supports local variable definitions and assignments, which is an important feature in many template languages. For example, as shown in Figure 6c, line 3 defines a local variable v:int and initializes it with 0; line 4 assigns 1 to v so that in the rest of the template, v refers to 1, rather than 0. If we run this template, then we shall get a string "0 1". In parsing mode, `BIT` carefully keeps track of these assignments so as to correctly update the model.

In addition, `BIT` provides three extra control directives, namely, DEFAULT,



(a) Loop (b) Branch (c) Assignment

(d) Default (e) Unord (f) Final

Figure 6: Examples of control directives

UNORD, and FINAL, to enrich the bidirectional behavior of BIT templates.

The DEFAULT construct is used to print some default text. During parsing, it accepts a string that conforms to a lexical rule, even if the string is different from the default one. Consider the case of generating a method declaration in a Java interface. Figure 6d shows a tiny example where the template generates a method m(int value) with an integer parameter whose default name is value. The DEFAULT construct in the template is responsible for printing "value" initially. Developers may change the parameter name arbitrarily (e.g., changing the method signature to m(int arg)). The template accepts the changed signature in parsing mode, and will not overwrite arg during incremental printing.

The UNORD construct generates a list of strings whose order may vary. For instance, Java modifiers (e.g., static and public) may occur in any order. UNORD is designed to handle this case, as shown in Figure 6e: when printing, if the original string is empty, it prints "static" and "public" sequentially; if the original string is not empty (e.g., "public static"), it keeps the original one; when parsing, it accepts both "static public" and "public static".

The FINAL construct outputs a cached string (obtained during parsing) before printing its body fragment. Consider the template of a variable declaration, as shown in Figure 6f. Initially, the template prints out "int v;". Because a variable may have an initializer, developers may change the declaration to "int v = 0;". The FINAL construct in line 3 tells the derived parser to consume any character after "int v" until it meets ";". In incremental printing mode, the construct preserves these characters before appending the final ";".

3.2. Formalization of bidirectional templates

To specify the semantics of BIT, we should formally define the function signatures of the printer and the parser that are derived from a BIT template, as well as the round-trip properties they must follow. Let us start from the trivial case that a parser and a printer can be defined as the following functions

$$parse : \mathbb{S} \to V \qquad \text{(TrivialParse)}$$

$$print : V \to \mathbb{S} \qquad \text{(TrivialPrint)}$$

where $\mathbb{S}$ denotes the string type and V is a certain value type, function *parse* consumes an input string and yields a value, and function *print* serializes a value to a string. We assume that there is a special string $\perp$ that denotes the *initially empty string*. In string calculation, $\perp$ is treated as `""`.

Such a simple signature does not support the major features of `BIT`, including incremental printing, incremental parsing, and local assignments. Our goal is to find a an appropriate definition that enables the embedding of these features.

Incremental Printing. As illustrated in Figure 4a, incremental printing allows for printing a value by rewriting an existing string. To achieve this, the *print* function must accept the original string as an additional input, so that it can determine which parts of the original string should be overwritten and which should be preserved. Hence, *print* must be declared as IncPrint:

$$print : \mathbb{S} \times V \to \mathbb{S} \tag{IncPrint}$$

Obviously, TrivialParse and IncPrint can be viewed as a BX $\mathbb{S} \leftrightarrow V$.

Incremental Parsing. When considering the feature of incremental parsing, which not only returns a value parsed from the string but also (incrementally) updates a model $\mathbb{M}$, the *parse* function must read a model and return an updated one. In other words, *parse* must be refined upon TrivialParse as IncParse:

$$parse : (\mathbb{S}, \mathbb{M}) \to (V, \mathbb{M}) \tag{IncParse}$$

Unfortunately, IncParse and IncPrint cannot be combined into a BX because IncPrint does not use a model. As the model is read-only during printing, $\mathbb{M}$ can be considered as an additional view type that contributes to the transformation. Subsequently, IncPrint is redefined as IncPrintM:

$$print : \mathbb{S} \times (V, \mathbb{M}) \to \mathbb{S} \tag{IncPrintM}$$

To combine IncParse and IncPrintM together, we propose a new kind of bidirectional transformations, namely, *accumulative BX* (αBX for short).

Definition 1 (Accumulative BX)**.** *Given two functions* $parse : (\mathbb{S}, \mathbb{M}) \to (V, \mathbb{M})$ *and* $print : \mathbb{S} \times (V, \mathbb{M}) \to \mathbb{S}$*, they can be combined into an accumulative BX, written* $\mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} V$*, iff. they satisfy the following round-trip properties:*

$$s \neq \perp \wedge parse(s, m) = (v, m') \implies print(s, (v, m')) = s \tag{1}$$

$$print(s, (v, m)) = s' \implies s' \neq \perp \wedge parse(s', m) = (v, m) \tag{2}$$

Just like GetPut and PutGet laws, properties (1) and (2) state that *parse* and *print* are mutually reversed. If $s = \perp$, *parse* effectively does nothing, and *print* generates a fresh string. Therefore, 1 does not have to hold in this case.

αBX reflects the following bidirectional behavior: during parsing, a string s is consumed to produce the view v and update the model m into m'; during printing, the original string s is updated by considering the view v and the current model m. Obviously, IncParse and IncPrintM fit this specific behavior.

Special Case. Considering the case when the updated model m' is computed by *putting* the view value v into the original model m, we can define a model-value BX between $\mathbb{M}$ and V, i.e., $val : \mathbb{M} \leftrightarrow V$, and interpret $\alpha BX\ l : \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} V$ as follows:

- to compute $parse_l(s, m)$, l firsts converts s into v, and then updates m into m' by performing $m' = put_{val}(m, v)$;
- to compute $print_l(s, (v, m))$, l updates s into s' when $v = get_{val}(m)$.

This case actually requires that $\mathbb{M}$ and V be consistent in terms of the model-value BX. V is derivable from $\mathbb{M}$ and thus redundant. Formally, we propose a constructor $(*) : (\mathbb{M} \leftrightarrow V) \to (\mathbb{S} \leftrightarrow V) \to (\mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathit{Unit})$ to construct $(val * sl) : \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathit{Unit}$ from a model-value BX $val : \mathbb{M} \leftrightarrow V$ and a string-value BX $sl : \mathbb{S} \leftrightarrow V$, as follows:

$$
\begin{array}{l||l}
parse_{val*sl}(s, m) \equiv & print_{val*sl}(s, (\mathsf{unit}, m)) \equiv \\
\quad \textbf{do} & \quad \textbf{do} \\
\quad\quad v \leftarrow get_{sl}(s) & \quad\quad v \leftarrow get_{val}(m) \\
\quad\quad m' \leftarrow put_{val}(m, v) & \quad\quad s' \leftarrow put_{sl}(s, v) \\
\quad\quad \textbf{return}\ (\mathsf{unit}, m') & \quad\quad \textbf{return}\ s'
\end{array}
$$

where *Unit* is the bottom type of all data types, which has one concrete value unit. Because *Unit*-typed arguments and return values can be ignored, a special αBX can also be regarded as $\alpha BX = \{parse : (\mathbb{S}, \mathbb{M}) \to \mathbb{M}, print : \mathbb{S} \times \mathbb{M} \to \mathbb{S}\}$.

Theorem 1. *For well-behaved $val : \mathbb{M} \leftrightarrow V$ and $sl : \mathbb{S} \leftrightarrow V$, $(*)$ ensures the well-behavedness of $val * sl$.*

For example, assume that $\mathbb{M}$ is a record type, $sl : \mathbb{S} \leftrightarrow Int$ converts a string and an integer bidirectionally, and val retrieves/stores a value from/to the field k of a record. When $s = \texttt{"123"}$ and $m = \{\mathsf{k} = 5, \mathsf{j} = 6\}$, (1) $get_{sl}(s) = 123$ and $put_{val}(m, 123) = m' \equiv \{\mathsf{k} = 123, \mathsf{j} = 6\}$, resulting in $parse_{val*sl}(s, m) = m'$l; (2) $get_{val}(m') = 123$ and $put_{sl}(s, 123) = \texttt{"123"}$, so $print_{val*sl}(s, m') = \texttt{"123"}$.

Composability. So far, we assume that a *parse* function shall consume the entire input string. Nevertheless, a single parser may only parse a prefix of the input in practice, leaving the rest to the subsequent parsers. This fashion can be declared as a *prefixParse* function $\mathbb{S} \to (V, \mathbb{S})$. For better composability, we should integrate *prefixParse* into our formalization.

First, we borrow the idea of many modern compilers that *a parser converts a string (i.e., code) into a pair of an internal representation $\mathbb{T}$ (e.g., concrete/abstract syntax trees) and the remaining string*, as outlined below:

$$parse : \mathbb{S} \to (\mathbb{T}, \mathbb{S}) \qquad \text{(SynParse)}$$

SynParse (called *syntactic parser*) is similar to *prefixParse*, except that SynParse produces an internal representation $\mathbb{T}$, rather than a concrete value V. By straightforwardly inverting *prefixParse*, we obtain a *syntactic printer*

$$print : (\mathbb{T}, \mathbb{S}) \to \mathbb{S} \qquad \text{(SynPrint)}$$

which prints the internal representation $\mathbb{T}$ to a string and joins it to the remaining string. We call SynParse and SynPrint a well-behaved $synBX\ \mathbb{S} \leftrightsquigarrow \mathbb{T}$ iff. they make the following round-trip properties hold:

$$s \neq \bot \wedge parse(s) = (t, s_T) \implies print(t, s_T) = s \tag{3}$$
$$t \neq \bot \wedge print(t, s_T) = s \implies parse(s) = (t, s_T) \tag{4}$$

Property (4) means $parse$ must *exactly* consume the prefix printed by $print(t, s_T)$.

Second, we redefine the special case of $\alpha BX : \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} Unit$ into the *semantic BX* $semBX : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$, which comprises SemParse and SemPrint as follows:

$$parse : (\mathbb{T}, \mathbb{M}) \to \mathbb{M} \tag{SemParse}$$
$$print : \mathbb{T} \times \mathbb{M} \to \mathbb{T} \tag{SemPrint}$$

In short, $semBX$ consumes/produces $\mathbb{T}$, rather than a string.

Finally, we propose a constructor $\circledast$ between $synBX$ and $semBX$

$$\circledast : (\mathbb{S} \leftrightsquigarrow \mathbb{T}) \to (\mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit) \to (\mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S})$$

which constructs a *composable* $\alpha BX\ l_1 \circledast l_2 : \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$ from a $synBX\ l_1 : \mathbb{S} \leftrightsquigarrow \mathbb{T}$ and a $semBX\ l_2 : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$, as follows:

$parse_{l_1 \circledast l_2}(s, m) \equiv$
  do
    $(t, s_T) \leftarrow parse_{l_1}(s)$
    $m' \leftarrow parse_{l_2}(t, m)$
    return (s_T, m')

$print_{l_1 \circledast l_2}(s, (s_T, m)) \equiv$
  do
    $(t, s'_{tail}) \leftarrow parse_{l_1}(s)$
    $t' \leftarrow print_{l_2}(t, m)$
    $s' \leftarrow print_{l_1}(t', s_T)$
    return s'

Theorem 2. *If $l_1 : \mathbb{S} \leftrightsquigarrow \mathbb{T}$ and $l_2 : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$ are well behaved, then $l_1 \circledast l_2$ is also a well behaved αBX.*

Local Assignments. In a template language, a local assignment `v=e` can be viewed as a computational effect—it changes the binding of variable `v` to expression `e`. Assume $\mathbb{B}$ is the type of variable binding set and any $\beta : \mathbb{B}$ represents a set of variable bindings. Each variable binding has a form of $v \mapsto e$.

We can view a `BIT` template as a composition of local assignments and a special composable αBX. We define such a composition as an *effect-binding BX* (βBX for short). For any assignment `v=e` in a template, `e` is determined by constants, current variable bindings, and the model to be printed.

Definition 2 (Effect-binding BX)**.** *An effect-binding BX, written $\mathbb{B} \mapsto \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$, is a pair of parse and print functions*

$$parse : (\mathbb{S}, \mathbb{M}, \mathbb{B}) \to (\mathbb{S}, \mathbb{M}) \times \mathbb{B}$$
$$print : (\mathbb{S}, \mathbb{B}) \times (\mathbb{S}, \mathbb{M}) \to (\mathbb{S}, \mathbb{B})$$

It is well-behaved if it satisfies the following round-trip properties

$$s \neq \bot \wedge parse(s, m, \beta) = ((s_T, m'), \beta') \Rightarrow print((s, \beta), (s_T, m)) = (s, \beta') \quad \text{(PARSEPRINT)}$$

$$print((s, \beta), (s_T, m)) = (s', \beta') \Rightarrow parse(s', m, \beta) = ((s_T, m), \beta') \quad \text{(PRINTPARSE)}$$

To construct a βBX and explain its behavior, we propose the constructor

$$\odot : (\mathbb{B} \times \mathbb{M} \to \mathbb{B}) \to (\mathbb{B} \to \mathbb{S} \xleftrightarrow{\mathbb{M}} \mathbb{S}) \to (\mathbb{B} \mapsto \mathbb{S} \xleftrightarrow{\mathbb{M}} \mathbb{S})$$

for βBXs, such that for $r : \mathbb{B} \times \mathbb{M} \to \mathbb{B}$ and $pl : \mathbb{B} \to \mathbb{S} \xleftrightarrow{\mathbb{M}} \mathbb{S}$,

$parse_{r \odot pl}(s, m, \beta) \equiv$
do
$l \leftarrow pl(\beta)$
$(s_T, m') \leftarrow parse_l(s, m)$
$\beta' \leftarrow r(\beta, m')$
return $((s_T, m'), \beta')$

$print_{r \odot pl}((s, \beta), (s_T, m)) \equiv$
do
$l \leftarrow pl(\beta)$
$s \leftarrow print_l(s, (s_T, m))$
$\beta' \leftarrow r(\beta, m)$
return (s, β')

where

- r is a binding update function that updates the variable bindings based on existing bindings and a model; for example, an assignment `v=v+u` can update a binding set $\{\mathsf{v} \mapsto a, \mathsf{u} \mapsto b\}$ to $\{\mathsf{v} \mapsto a + b, \mathsf{u} \mapsto b\}$;
- pl a binding-aware generator for αBX that generates an αBX based on given variable bindings, .e.g, it generates «`a+b+1|INT`» from an initial hole specification «`v+u|INT`» if the current bindings are $\{\mathsf{v} \mapsto \mathsf{a} + \mathsf{b}, \mathsf{u} \mapsto 1\}$.

Theorem 3. *If r and $pl(\beta)$ are well behaved for any β, then $r \odot pl$ is also a well-behaved βBX.*

Before going on, we define two auxiliary functions

- $resolve(\beta, expr)$: return a new expression by substituting the free variables occurring in the expression $expr$ with their bindings in β. For example, $resolve(\beta, v_1 \times v_2) = (a + b) \times v_2$ if $\beta = \{v_1 \mapsto a + b\}$.
- $update(\beta, v \mapsto u)$: return $(\beta - \{v \mapsto x | v \mapsto x \in \beta\}) \cup \{v \mapsto resolve(\beta, u)\}$.

3.3. Definition of the core language

The core language is designed to formally and precisely specify the semantics of `BIT`, into which the surface language can be translated. As shown in Figure 7, the core language contains 5 `BIT` primitives and 8 `BIT` combinators. t stands for *template* (and *template fragment*); e and ρ denote a general *expression* (e.g., arithmetic, relational, boolean, and path call) and a *regular pattern*, respectively; v refers to a *variable*, which is also an expression; s denotes *string literals*; τ is the internal representation of parsed text, whose type is $\mathbb{T}$.

t	::=	p i.e., primitives	c	::=	$\mathsf{seq}(t_1, t_2)$
	$\parallel$	c i.e., combinators		$\parallel$	$\mathsf{ite}(e, t_1, t_2)$
e	::=	expressions		$\parallel$	$\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \to t)$
ρ	::=	regular patterns		$\parallel$	$\mathsf{scope}(s_b, s_a, t)$
v	::=	variables		$\parallel$	$\mathsf{default}(\rho, t)$
s	::=	string literals and $\perp$		$\parallel$	$\mathsf{unord}(t_1, t_2)$
p	::=	$\mathsf{const}(s_c)$		$\parallel$	$\mathsf{final}(t)$
	$\parallel$	$\mathsf{lex}(\rho, e)$		$\parallel$	$\mathsf{call}(t, v_1 = e_1, v_2 = e_2, ...)$
	$\parallel$	$\mathsf{space}(s_w)$	τ	::=	$s \parallel [\tau_1, \tau_2, ...] \parallel \mathsf{L}\ \tau \parallel \mathsf{R}\ \tau$
	$\parallel$	$\mathsf{assign}(v, e)$		$\parallel$	$(\tau_b, [\tau_{r_1}, \tau_{r_2}, ...], \tau_a) \parallel \mathsf{C}\ \tau$
	$\parallel$	nop		$\parallel$	$\mathsf{D}\ \tau \parallel \mathsf{U}_i\ \tau \parallel\ < s_b, \tau, s_a >$

Figure 7: Syntax of the core language

We assume every expression e is equipped with a bidirectional semantics. That is, we can interpret e as a BX. However, the bidirectional semantics of e is out of the scope of this paper, and please refer to [15, 31, 32] for more details.

To define the semantics, the following helper functions are needed:

- $+\!\!+ : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ denotes string concatenation, e.g., "ab"++"12" = "ab12".
- $\mathsf{lookAt} : \mathbb{R} \times \mathbb{S} \to \mathbb{S} \times \mathbb{S}$ returns a prefix of the input string that matches the given regular pattern or $\perp$ if failed, where $\mathbb{R}$ denotes the type of regular patterns. For example, lookAt([0-9]+, "12ab") = ("12", "ab") and lookAt([0-9]+, "x12a") = ($\perp$, "x12a"). We also use lookAt to match a string constant because we can convert a string constant into a regular pattern.

Internal structure. $\tau : \mathbb{T}$ is the internal structure of the parsed text, produced by *syn* and consumed by *sem*. Different τ corresponds to different primitives and combinators. τ can be a string s, a sequence $[\tau_{r_1}, \tau_{r_2}, ...]$, a L/R-labeled structure $\mathsf{L}\ \tau/\mathsf{R}\ \tau$ for branches, a D-labeled structure for default fragments, a loop structure $(\tau_b, [\tau_{r_1}, \tau_{r_2}, ...], \tau_a)$, a unordered fragment structure $\mathsf{U}_i\ \tau$ (where i denotes the index of the body fragment of the unord primitive, which prints/parses τ), a template-call structure $\mathsf{C}\ \tau$, and a scope structure $< s_b, \tau, s_a >$.

Formal structure. A BIT template is a $\beta BX\ \mathbb{B} \mapsto \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$, which can be defined as the record type BIT:

$$\textbf{data}\ \mathsf{BIT} = \mathsf{BIT}\ \{syn : \mathbb{S} \leftrightsquigarrow \mathbb{T}, gSem : \mathbb{B} \to \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit, \mathit{eff} : (\mathbb{B}, \mathbb{M}) \to \mathbb{B}\}$$

where $gSem$ is a $semBX$ generator and *eff* is the binding update function. Given $r : \mathsf{BIT}$, a βBX is built by $r.\mathit{eff} \odot \big(\lambda\beta \to r.syn \circledast r.gSem(\beta)\big)$.

The structure of a BIT primitive can further be refined as a record type BIT′:

$$\textbf{data}\ \mathsf{BIT}' = \mathsf{BIT}'\ \{syn : \mathbb{S} \leftrightsquigarrow \mathbb{T}, sem : \mathbb{T} \leftrightarrow V, val : \mathbb{M} \leftrightarrow V, \mathit{eff} : (\mathbb{B}, \mathbb{M}) \to \mathbb{B}\}$$

Then, given record r' of BIT′, it can be converted into a record of BIT as follows $\mathsf{BIT}\ \{syn = r'.syn, gSem = \lambda\beta \to \big(resolve(\beta, r'.val) * r'.sem\big), \mathit{eff} = r'.\mathit{eff}\}$.

Primitives. This paper proposes 5 primitives, i.e., const, lex, space, assign, nop.

The primitive $\mathsf{const}(s_c)$ prints/parses a constant string s_c. It can be defined as the following structure:

$$\mathsf{const}(s_c) \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \left\{ \begin{array}{l} parse(s_I) \triangleq \textbf{if } s_I = \bot \textbf{ then } (\bot, \bot) \\ \qquad\qquad \textbf{elif } s_c{+\!+}s_T = s_I \textbf{ then } (s_c, s_T) \textbf{ else error} \\ print(\tau, s_T) \triangleq \textbf{if } \tau = s_c \textbf{ then } s_c{+\!+}s_T \textbf{ else error} \end{array} \right. \\ sem = \left\{ \begin{array}{l} parse(\tau) \triangleq \textbf{if } \tau = s_c \textbf{ then unit else error} \\ print(\tau, \mathsf{unit}) \triangleq \textbf{if } \tau = s_c \vee \tau = \bot \textbf{ then } s_c \textbf{ else error} \end{array} \right. \\ val = UnitBX, \mathit{eff} = \mathit{IdEff} \end{array} \right\}$$

where $UnitBX \equiv \{get(m) = \mathsf{unit}, put(m, \mathsf{unit}) = m\}$, $\mathit{IdEff}(\beta, m) \equiv \beta$, and error denotes a runtime exception which will abort execution if left uncaught.

The primitive $\mathsf{lex}(\rho, e)$ aims to print/parse the value of e according to a regular pattern ρ, considering e as $\mathbb{M} \leftrightarrow \mathbb{S}$. It is defined as follows.

$$\mathsf{lex}(\rho, e) \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \left\{ \begin{array}{l} parse(s_I) \triangleq \textbf{if } s_I = \bot \textbf{ then } (\bot, \bot) \\ \qquad\qquad \textbf{elif } \mathsf{lookAt}(\rho, s_I) = (s, s_T) \textbf{ then } (s, s_T) \\ \qquad\qquad \textbf{else error} \\ print(\tau, s_T) \triangleq \textbf{if } \tau = s \wedge \mathsf{lookAt}(\rho, s{+\!+}s_T) = (s, s_T) \\ \qquad\qquad\qquad \textbf{then } s{+\!+}s_T \textbf{ else error} \end{array} \right. \\ sem = \left\{ \begin{array}{l} parse(\tau) \triangleq \textbf{if } \tau = s \textbf{ then } s \textbf{ else error} \\ print(\tau, s) \triangleq s \end{array} \right. \\ val = e, \mathit{eff} = \mathit{IdEff} \end{array} \right\}$$

The primitive $\mathsf{space}(s_w)$ handles white spaces. In parsing, it consumes the prefix white spaces; in printing, it tries to preserve the existing spaces or prints s_w (s_w must be white spaces) if the original string is empty. Supposing that ρ_w is the regular pattern that matches white spaces, $\mathsf{space}(s_w)$ is defined as follows:

$$\mathsf{space}(s_w) \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \left\{ \begin{array}{l} parse(s_I) \triangleq \textbf{if } s_I = \bot \textbf{ then } (\bot, \bot) \\ \qquad\qquad \textbf{elif } \mathsf{lookAt}(\rho_w, s_I) = (s, s_T) \textbf{ then } (s, s_T) \textbf{ else error} \\ print(\tau, s_T) \triangleq \textbf{if } \tau = s \neq \bot \wedge \mathsf{lookAt}(\rho_w, s{+\!+}s_T) = (s, s_T) \\ \qquad\qquad \textbf{then } s{+\!+}s_T \textbf{ else error} \end{array} \right. \\ sem = \left\{ \begin{array}{l} parse(\tau) \triangleq \textbf{if } \tau = s \neq \bot \textbf{ then unit else error} \\ print(\tau, \mathsf{unit}) \triangleq \textbf{if } \tau = \bot \textbf{ then } s_w \\ \qquad\qquad\qquad \textbf{elif } \tau = s \textbf{ then } s \textbf{ else error} \end{array} \right. \\ val = UnitBX, \mathit{eff} = \mathit{IdEff} \end{array} \right\}$$

The primitive $\mathsf{assign}(v, e)$ changes the binding of v, rather than directly printing/parsing strings. It is defined as follows:

$$\mathsf{assign}(v, e) \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \{parse(s_I) \triangleq (\bot, s_I), print(\bot, s_T) \triangleq s_T\}, \\ sem = \{parse(\bot) \triangleq \mathsf{unit}, print(\bot, \mathsf{unit}) \triangleq \bot\}, \\ val = UnitBX, \mathit{eff} = \lambda(\beta, m) \rightarrow update(\beta, v \mapsto e) \end{array} \right\}$$

Primitive nop does nothing in both printing and parsing:

$$\mathsf{nop} \triangleq \mathsf{BIT}' \left\{ \begin{array}{l} syn = \{parse(s_I) \triangleq (\bot, s_I), print(\bot, s_T) \triangleq s_T\}, \\ sem = \{parse(\bot) \triangleq \mathsf{unit}, print(\bot, \mathsf{unit}) \triangleq \bot\}, \\ val = UnitBX, \mathit{eff} = \mathit{IdEff} \end{array} \right\}$$

Printing examples	Parsing examples
1 $print_{\mathsf{const}("a")}((\bot, \beta), ("1", m)) = ("a1", \beta)$	i $parse_{\mathsf{const}("a")}("ab", m, \beta) = (("b", m), \beta)$
2 $print_{\mathsf{const}("a")}(("a2", \beta), ("1", m)) = ("a1", \beta)$	ii $parse_{\mathsf{const}("a")}("bb", m, \beta) = \mathsf{error}$
3 $print_{\mathsf{const}("a")}(("b2", \beta), ("1", m)) = \mathsf{error}$	
4 $print_{\mathsf{lex}("a+",v)}((\bot, \{v \mapsto u\}), ("b", \{u = "aa"\})) = ("aab", \{v \mapsto u\})$	iii $parse_{\mathsf{lex}("a+",v)}(("aab", \{\}, \{\})) = (("b", \{v = "aa"\}), \{\})$
5 $print_{\mathsf{lex}("a+",v)}((\_, \{v \mapsto u\}), (\_, \{u = "bb"\})) = \mathsf{error}$	iv $parse_{\mathsf{lex}("a+",v)}(("aab", \{v \mapsto u\}, \{\})) = (("b", \{u = "aa"\}), \{\})$
6 $print_{\mathsf{lex}("a+",v)}(("ab", \{\}), ("b", \{v = "aa", u = "aaa"\})) = ("aab", \{\})$	v $parse_{\mathsf{lex}("a+",v)}(("cb", \_, \_)) = \mathsf{error}$
7 $print_{\mathsf{space}(" ")}((\bot, \beta), (" b", m)) = (" b", \beta)$	vi $parse_{\mathsf{space}(" ")}(" b", m, \beta) = (("b", m), \beta)$
8 $print_{\mathsf{space}(" ")}((" ", \beta), ("b", m)) = (" b", \beta)$	vii $parse_{\mathsf{space}(" ")}("b", m, \beta) = (("b", m), \beta)$
9 $print_{\mathsf{space}(" ")}(("a", \beta), ("b", m)) = ("b", \beta)$	
10 $print_{\mathsf{assign}(v,a+b)}((\_, \{\}), ("b", \_)) = ("b", \{v \mapsto a + b\})$	xi $parse_{\mathsf{assign}(v,a+b)}("b", m, \{\}) = (("b", m), \{v \mapsto a + b\})$
11 $print_{\mathsf{assign}(v,a+b)}((\_, \{a \mapsto u\}), ("b", \_)) = ("b", \{a \mapsto u, v \mapsto u + b\})$	x $parse_{\mathsf{assign}(v,a+b)}("b", m, \{a \mapsto u\}) = (("b", m), \{a \mapsto u, v \mapsto u + b\})$

Figure 8: Examples of primitives

Figure 8 shows some examples about BIT primitives. 1, 4, and 7 demonstrate how to print from an empty string. 2 and 3 print constant string "a" onto the original strings, but 3 fails because the original string does not start with "a". 5 and 6 print the values of v, but 5 fails because v refers to "bb" that does not satisfy the lexical rule "a+". 8 and 9 demonstrate how space tries to preserve the original white-spaces as much as possible. i to x demonstrate the parsing behaviors. ii and v fail because the strings to be parsed do not match the lexical rules of the primitives.

Combinators. The core language has 8 combinators for complex behaviors.

The combinator $\mathsf{seq}(t_1, t_2)$ combines two template fragments t_1 and t_2 sequentially. Supposing that t_1 and t_2 are BIT records, $\mathsf{seq}(t_1, t_2)$ is defined by

$$
\begin{array}{l}
\mathsf{seq}(t_1, t_2) \triangleq \\
\quad \mathsf{BIT} \left\{ \begin{array}{l}
\quad syn = \left\{ \begin{array}{l}
parse(s_0) \triangleq \textbf{if } s_0 = \bot \textbf{ then } (\bot, \bot) \\
\qquad\qquad \textbf{elif } parse_{t_i.syn}(s_{i-1}) = (\tau_i, s_i) \textbf{ then } ([\tau_1, \tau_2], s_2) \\
\qquad\qquad \textbf{else } \mathsf{error} \\
print([\tau_1, \tau_2], s_2) \triangleq \textbf{if } s_{i-1} = print_{t_i.syn}(\tau_i, s_i) \\
\qquad\qquad\qquad\qquad \textbf{then } s_0 \textbf{ else } \mathsf{error}
\end{array} \right. \\
\quad gSem = \lambda\beta \rightarrow \\
\qquad \left\{ \begin{array}{l}
parse([\tau_1, \tau_2], m) \triangleq \textbf{do} \\
\quad m_1 \leftarrow parse_{t_1.gSem(\beta)}(\tau_1, m_0), \beta_1 \leftarrow t_1.\, \mathit{eff}(\beta, m_1) \\
\quad m_2 \leftarrow parse_{t_2.gSem(\beta_1)}(\tau_2, m_1) \\
\quad \textbf{assert } \beta_1 = t_1.\, \mathit{eff}(\beta, m_2) \wedge m_2 = parse_{t_1.gSem(\beta)}(\tau_1, m_2) \\
\quad \textbf{return } m_2 \\
print(\tau, m) \triangleq \textbf{do} \\
\quad [\tau_1, \tau_2] \leftarrow \textbf{if } \tau = \bot \textbf{ then } [\bot, \bot] \textbf{ elif } \tau = [\tau_1', \tau_2'] \textbf{ then } [\tau_1', \tau_2'] \\
\quad \tau_1' \leftarrow print_{t_1.gSem(\beta)}(\tau_1, m), \beta_1 \leftarrow t_1.\, \mathit{eff}(\beta, m) \\
\quad \tau_2' \leftarrow print_{t_2.gSem(\beta_1)}(\tau_2, m) \\
\quad \textbf{return } [\tau_1', \tau_2']
\end{array} \right. \\
\quad \mathit{eff} = \lambda(\beta, m) \rightarrow t_2.\, \mathit{eff}(t_1.\, \mathit{eff}(\beta, m), m)
\end{array} \right\}
\end{array}
$$

where **assert** throws error when the assertion predicate fails. The **assert** statement requires that t_2 does not break the consistency established by t_1.

Note that seq can be extended to combine more than two template fragments: $\mathsf{seq}(t_1, t_2, ..., t_n)$ is equivalent to $\mathsf{seq}(t_1, \mathsf{seq}(t_2, \mathsf{seq}(t_3, ...)))$.

$\mathsf{ite}(e, t_1, t_2)$ is the conditional combinator, corresponding to the IF construct in the surface language. In printing, it selects from t_1 and t_2 based on the branch condition e; in parsing, it chooses t_i to parse the input string if t_i consumes more characters than the other branch. ite is defined as follows:

$$
\begin{array}{l}
\mathsf{ite}(e, t_1, t_2) \triangleq \\
\quad \left\{ \begin{array}{l}
\quad syn = \left\{ \begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad (\tau_i, s_i) \leftarrow parse_{t_i.syn}(s_I) \quad s.t. \quad i = 1, 2 \\
\quad \textbf{return if } len(s_1) \leq len(s_2) \textbf{ then } (\mathsf{L}\ \tau_1, s_1) \textbf{ else } (\mathsf{R}\ \tau_2, s_2) \\
print(\mathsf{L}\ \tau_1, s_T) \triangleq \textbf{do} \\
\quad s' \leftarrow print_{t_1.syn}(\tau_1, s_T), (\tau_2, s_2) \leftarrow parse_{t_2.syn}(s') \\
\quad \textbf{assert } len(s_T) \leq len(s_2) \\
\quad \textbf{return } s' \\
print(\mathsf{R}\ \tau_2, s_T) \triangleq \textbf{do} \\
\quad s' \leftarrow print_{t_2.syn}(\tau_2, s_T), (\tau_1, s_1) \leftarrow parse_{t_1.syn}(s') \\
\quad \textbf{assert } len(s_T) < len(s_1) \\
\quad \textbf{return } s'
\end{array} \right. \\
gSem = \lambda\beta \rightarrow \\
\quad \left\{ \begin{array}{l}
parse(\mathsf{L}\ \tau, m) \triangleq \textbf{do} \\
\quad l \leftarrow t_1.gSem(\beta), m_1 \leftarrow parse_l(\tau, m), m' \leftarrow put_e(m, true) \\
\quad \textbf{assert } m' = parse_l(\tau, m') \\
\quad \textbf{return } m' \\
parse(\mathsf{R}\ \tau, m) \triangleq \textbf{do} \\
\quad l \leftarrow t_2.gSem(\beta), m_2 \leftarrow parse_l(\tau, m), m' \leftarrow put_e(m, false) \\
\quad \textbf{assert } m' = parse_l(\tau, m') \\
\quad \textbf{return } m' \\
print(\tau, m) \triangleq \textbf{do} \\
\quad \textbf{if } get_e(m) = true \textbf{ then} \\
\quad\quad l \leftarrow t_1.gSem(\beta), \tau^* \leftarrow \textbf{if } \tau = \mathsf{L}\ \tau_1 \textbf{ then } \tau_1 \textbf{ else } \bot \\
\quad\quad \textbf{return } \mathsf{L}\ print_l(\tau^*, m) \\
\quad \textbf{else} \\
\quad\quad l \leftarrow t_2.gSem(\beta), \tau^* \leftarrow \textbf{if } \tau = \mathsf{R}\ \tau_2 \textbf{ then } \tau_2 \textbf{ else } \bot \\
\quad\quad \textbf{return } \mathsf{R}\ print_l(\tau^*, m)
\end{array} \right. \\
eff = \lambda(\beta, m) \rightarrow \textbf{if } get_e(m) = true \textbf{ then } t_1.eff(\beta, m) \textbf{ else } t_2.eff(\beta, m)
\end{array} \right. \\
\mathtt{BIT}
\end{array}
$$

The combinator $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \rightarrow t)$, corresponding to the FOR construct, prints each element using a list e_{arr} by loop body t with an iteration variable v, where s_s is the separator inserted between two consecutive iterations, while s_b and s_a are the strings inserted before and after all iterations.

To specify the semantics of $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \rightarrow t)$, we first convert s_s, s_b, and s_a to BIT primitives t_s, t_b, and t_a. If s_x $(x = s, b, a)$ is a non-empty string, then $t_x = \mathsf{const}(s_x)$; otherwise, $t_x = \mathsf{nop}$. Assume that t_i denotes the ith-iteration of the loop body t whose iteration variable is renamed to v_i. For simplicity, we assume that there is no collision among variable names. For example, supposing that the loop body is $\mathsf{lex}(\rho, v)$ (where v is the iteration variable), then t_i is $\mathsf{lex}(\rho, v_i)$. Let $ls_0 \equiv \mathsf{nop}$ and $ls_n \equiv \mathsf{seq}(t_b, t_1, t_s, t_2, t_s, ..., t_s, t_n, t_a)$

$(n > 0)$. The behavior of $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \to t)$ can be interpreted as a certain sequence ls_n. Formally, loop is defined as follows:

$$
\begin{array}{l}
\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \to t) \triangleq \\
\left\{\begin{array}{ll}
syn = & \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \text{let } n = \text{the largest integer s.t. } parse_{ls_n.syn}(s_I) \text{ succeeds} \\
\quad \textbf{if } n = 0 \textbf{ then return } (\bot, s_I) \\
\quad [\tau_b, \tau_1, \tau_{s_1}, ..., \tau_n, \tau_a] \leftarrow parse_{ls_n.syn}(s_I) \\
\quad \textbf{return } (\tau_b, [\tau_1, \tau_{s_1}, ..., \tau_n], \tau_a) \\
print(\tau, s_T) \triangleq \textbf{do} \\
\quad (n_l, \tau_l) \leftarrow \textbf{if } \tau = (\tau_b, [\tau_1, \tau_{s_1}, ..., \tau_n], \tau_a) \\
\qquad\qquad \textbf{then } (n, [\tau_b, \tau_1, \tau_{s_1}, ..., \tau_n, \tau_a]) \textbf{ else } (0, \bot) \\
\quad s' \leftarrow print_{ls_{n_l}.syn}(\tau_l, s_T) \\
\quad \textbf{if } parse_{ls_{n+1}.syn}(s') \text{ throws error } \textbf{then return } s'
\end{array}\right. \\
\mathsf{BIT} & \left\{\begin{array}{ll}
gSem = \lambda\beta \to & \left\{\begin{array}{l}
parse(\tau, m) \triangleq \textbf{do} \\
\quad e' \leftarrow resolve(\beta, e_{arr}) \\
\quad \textbf{if } \tau = (\tau_b, [\tau_1, \tau_{s_1}, ..., \tau_n], \tau_a) \textbf{ then} \\
\quad\quad ls \leftarrow ls_n, m' \leftarrow parse_{ls_n.gSem(\beta)}(\tau, m) \\
\quad\quad arr \leftarrow [get_{v_1}(m'), get_{v_2}(m'), ..., get_{v_n}(m')] \\
\quad \textbf{else } ls \leftarrow ls_0, m' \leftarrow m, arr \leftarrow [] \\
\quad m'' \leftarrow put_{e'}(m', arr) \\
\quad \textbf{assert } m'' = parse_{ls.gSem(\beta)}(\tau, m'') \\
\quad \textbf{return } m'' \\
print(\tau, m) \triangleq \textbf{do} \\
\quad e' \leftarrow resolve(\beta, e_{arr}), arr \leftarrow get_{e'}(m), n_a \leftarrow len(arr) \\
\quad \textbf{if } n_a = 0 \textbf{ then return } \bot \\
\quad \textbf{elif } \tau = (\tau_b, [\tau_1, \tau_{s_1}, ..., \tau_n], \tau_a) \textbf{ then} \\
\quad\quad \textbf{if } n \geq n_a \textbf{ then } \tau' \leftarrow [\tau_b, \tau_1, \tau_{s_1}, ..., \tau_{n_a}, \tau_a] \\
\quad\quad \textbf{else } \tau' \leftarrow [\tau_b, \tau_1, \tau_{s_1}, ..., \tau_n, \underbrace{s_s, \bot, ..., s_s, \bot}_{s_s \text{ occurs } n_a - n \text{ times}}, \tau_a] \\
\quad \textbf{else } \tau' \leftarrow [s_b, \underbrace{\bot, s_s, ..., s_s, \bot}_{s_s \text{ occurs } n_a - 1 \text{ times}}, s_a] \\
\quad m' \leftarrow m \cup \{v_1 = arr[0], ..., v_{n_a} = arr[n_a - 1]\} \\
\quad [\tau'_b, \tau'_1, \tau'_{s_1}, ..., \tau'_{n_a}, \tau'_a] \leftarrow print_{ls_a.gSem(\beta)}(\tau', m') \\
\quad \textbf{return } (\tau'_b, [\tau'_1, \tau'_{s_1}, ..., \tau'_{n_a}], \tau'_a)
\end{array}\right. \\
eff = \lambda(\beta, m) \to ls_n.\ eff(\beta, m) \\
\quad \text{where } e' = resolve(\beta, e_{arr}) \wedge arr = get_{e'}(m) \wedge n = len(arr)
\end{array}\right.
\end{array}
$$

The combinator $\mathsf{scope}(s_b, s_a, t)$ is used to handle cases of balanced brackets (and comment delimiters). In source code, brackets must be correctly paired to recognize the code structure. For example, "(f())" must be parsed to "(", "f()", ")", i.e., the first "(" should be paired with the second ")", rather than the first one. Although a template language does not know the grammar of the printed text, our approach can be configured to handle these cases by specifying the opening and the closing tags (i.e., s_b, s_a). Then, $\mathsf{scope}(s_b, s_a, t)$ can recognize a text scope s_s, in which s_b and s_a are *balanced*, written $isBal(s_s, s_b, s_a)$. For example, $isBal(\text{"f[a+b[]]"}, \text{"["}, \text{"]"}) = true$, but $isBal(\text{"]+b[]["}, \text{"["}, \text{"]"}) = false$. The behavior of scope is defined as follows:

$$
\begin{array}{l}
\mathsf{scope}(s_b, s_a, t) \triangleq \\
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad \textbf{elif } s_I = s_b{+\!+}s_s{+\!+}s_a{+\!+}s_T \wedge isBal(s_s, s_b, s_a) \textbf{ then} \\
\quad\quad (\tau', s_{s,T}) \leftarrow parse_{t.syn}(s_s) \\
\quad\quad \textbf{if } s_{s,T} = \texttt{""} \textbf{ then return } (<s_b, \tau', s_a>, s_T) \\
print(<s_b, \tau', s_a>, s_T) \triangleq \textbf{do} \\
\quad s_s \leftarrow print_{t.syn}(\tau', \texttt{""}) \\
\quad \textbf{assert } isBal(s_s, s_b, s_a) = true \\
\quad \textbf{return } s_b{+\!+}s_s{+\!+}s_a{+\!+}s_T
\end{array}\right. \\
gSem = \lambda\beta \rightarrow \left\{\begin{array}{l}
parse(<s_b, \tau', s_a>, m) \triangleq parse_{t.gSem(\beta)}(\tau', m) \\
print(\tau, m) \triangleq \textbf{do} \\
\quad \tau' \leftarrow \textbf{if } \tau = <s_b, \tau'', s_a> \textbf{ then } \tau'' \\
\quad\quad\quad \textbf{elif } \tau = \bot \textbf{ then } \bot \textbf{ else } \mathsf{error} \\
\quad \textbf{return } <s_b, print_{t.gSem(\beta)}(\tau', m), s_a>
\end{array}\right. \\
\mathit{eff} = t.\,\mathit{eff}
\end{array}\right.
\end{array}
$$

The combinator $\mathsf{default}(\rho, t)$, corresponding to the DEFAULT construct, generates a default string conforming to pattern ρ if the original string is empty, or preserves the original string if it is non-empty. It is defined as follows. Note that $\mathsf{default}(\rho, t)$ requires that t is not a default construct and do not contain local assignments (i.e., its binding update function must be *IdEff*).

$$
\begin{array}{l}
\mathsf{default}(\rho, t) \triangleq \\
\mathsf{BIT}\left\{\begin{array}{l}
syn = \left\{\begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad (s_p, s_T) \leftarrow \mathsf{lookAt}(\rho, s_I) \\
\quad \textbf{assert } s_p \neq \bot \\
\quad \textbf{if } parse_{t.syn}(s_p) = (\tau', \bot) \textbf{ then return } (\mathsf{D}\ \tau', s_T) \\
\quad \textbf{else return } (s_p, s_T) \\
print(\tau, s_T) \triangleq \textbf{do} \\
\quad s' \leftarrow \textbf{if } \tau = \mathsf{D}\ \tau' \textbf{ then } print_{t.syn}(\tau', s_T) \\
\quad\quad\quad \textbf{elif } \tau = s_p \textbf{ then } s_p{+\!+}s_T \textbf{ else } \mathsf{error} \\
\quad \textbf{assert } \mathsf{lookAt}(\rho, s') = (s_p, s_T) \wedge s_p \neq \bot \\
\quad \textbf{return } s'
\end{array}\right. \\
gSem = \lambda\beta \rightarrow \\
\quad \left\{\begin{array}{l}
parse(\tau, m) \triangleq \textbf{if } \tau = \bot \textbf{ then } \mathsf{error} \\
\quad\quad\quad\quad \textbf{elif } \tau = \mathsf{D}\ \tau' \textbf{ then } parse_{t.gSem(\beta)}(\tau', m) \textbf{ else } m \\
print(\tau, m) \triangleq \textbf{if } \tau = \mathsf{D}\ \tau' \textbf{ then } \mathsf{D}\ print_{t.gSem(\beta)}(\tau', m) \\
\quad\quad\quad\quad \textbf{elif } \tau = \bot \textbf{ then } \mathsf{D}\ print_{tgSem(\beta)}(\bot, m) \textbf{ else } \tau
\end{array}\right. \\
\mathit{eff} = t.\,\mathit{eff} = \mathit{IdEff}
\end{array}\right.
\end{array}
$$

The combinator $\mathsf{unord}(t_1, t_2)$, corresponding to the UNORD construct, prints a string with t_1 and t_2. However, during parsing, it attempts to parse the string with $\mathsf{seq}(t_1, t_2)$ and $\mathsf{seq}(t_2, t_1)$. $\mathsf{unord}(t_1, t_2)$ requires that no matter in what order the binding update functions of t_1 and t_2 can be combined, the composite functions are equivalent. Supposing $ls_{i,j} = \mathsf{seq}(t_i, t_j)$, unord is defined as follows:

$$
\begin{array}{l}
\mathsf{unord}(t_1, t_2) \triangleq \\
\quad \mathsf{BIT} \left\{ \begin{array}{l}
\quad syn = \left\{ \begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad ([\tau_{12,1}, \tau_{12,2}], s_{T,12}) \leftarrow parse_{ls_{1,2}.syn}(s_I) \\
\quad ([\tau_{21,2}, \tau_{21,1}], s_{T,21}) \leftarrow parse_{ls_{2,1}.syn}(s_I) \\
\quad \textbf{if } len(s_{T,12}) \leq len(s_{T,21}) \textbf{ then return } [\mathsf{U}_1\ \tau_{12,1}, \mathsf{U}_2\ \tau_{12,2}] \\
\quad \textbf{else return } [\mathsf{U}_2\ \tau_{21,2}, \mathsf{U}_1\ \tau_{21,1}] \\
print([\mathsf{U}_i\ \tau_i, \mathsf{U}_j\ \tau_j], s_T) \triangleq \textbf{do} \\
\quad s' \leftarrow print_{ls_{i,j}}([\tau_i, \tau_j], s_T) \\
\quad \textbf{assert } (\_, s'_T) = parse_{ls_{j,i}}(s') \implies len(s_T) \leq len(s'_T) \\
\quad \textbf{return } s'
\end{array} \right. \\
gSem = \lambda\beta \rightarrow \\
\quad \left\{ \begin{array}{l}
parse([\mathsf{U}_i\ \tau_i, \mathsf{U}_j\ \tau_j], m) \triangleq parse_{ls_{ij}}([\tau_i, \tau_j], m) \\
print(\tau, m) \triangleq \textbf{do} \\
\quad [\tau'_i, \tau'_j] \leftarrow \textbf{if } \tau = \bot \textbf{ then } print_{ls_{12}.gSem(\beta)}(\bot, m) \\
\qquad\qquad \textbf{elif } \tau = [\mathsf{U}_i\ \tau_i, \mathsf{U}_j\ \tau_j] \textbf{ then } print_{ls_{ij}}([\tau_i, \tau_j], m) \\
\quad \textbf{return } [\mathsf{U}_i\ \tau'_i, \mathsf{U}_j\ \tau'_j]
\end{array} \right. \\
eff = \mathsf{seq}(t_1, t_2).\ eff = \mathsf{seq}(t_2, t_1).\ eff
\end{array} \right.
\end{array}
$$

Note that $\mathsf{unord}(t_1, t_2)$ can be generalized to $\mathsf{unord}(t_1, t_2, t_3, ..., t_n)$.

The combinator $\mathsf{final}(t)$, corresponding to the FINAL construct, preserves arbitrary characters in the original string before t can be applied; in parsing, it eats up the input characters until t can be applied to parse the remaining string. We define a helper function $until(parse, s)$ to find the first suffix of s that is parseable by a syntactic parser $parse$ (written as $parse(s) \uparrow$) as follows

$$
until(parse, s) = (s_P, s_T) \quad \text{if } parse(s_T) \uparrow \wedge \forall s'_T \big(parse(S'_T) \uparrow \Rightarrow len(s'_T) < len(s_T)\big)
$$

$\mathsf{final}(t)$ is defined as follows.

$$
\begin{array}{l}
\mathsf{final}(t) \triangleq \\
\quad \mathsf{BIT} \left\{ \begin{array}{l}
\quad syn = \left\{ \begin{array}{l}
parse(s_I) \triangleq \textbf{do} \\
\quad \textbf{if } s_I = \bot \textbf{ then return } (\bot, \bot) \\
\quad (s_p, s_T) \leftarrow until(parse_{t.syn}, s_I), (\tau, s'_T) \leftarrow parse_{t.syn}(s_T) \\
\quad \textbf{return } ([s_p, \tau], s'_T) \\
print(\tau, s_T) \triangleq \textbf{do} \\
\quad \textbf{if } \tau = \bot \textbf{ then } \mathsf{error} \\
\quad \textbf{else if } \tau = [s_p, \tau'] \textbf{ then} \\
\qquad s'_T \leftarrow print_{t.syn}(\tau', s_T), s' \leftarrow s_p +\!\!+ s'_T \\
\qquad \textbf{assert } until(parse_{t.syn}, s') = (s_p, s'_T) \\
\qquad \textbf{return } s'
\end{array} \right. \\
gSem = \lambda\beta \rightarrow \\
\quad \left\{ \begin{array}{l}
parse(\tau, m) \triangleq \textbf{do} \\
\quad \textbf{if } \tau = \bot \textbf{ then return } parse_{t.gSem(\beta)}(\bot, m) \\
\quad \textbf{else if } \tau = [s_p, \tau'] \textbf{ then return } parse_{t.gSem(\beta)}(\tau', m) \\
print(\tau, m) \triangleq \textbf{do} \\
\quad \textbf{if } \tau = \bot \textbf{ then return } [\texttt{""}, print_{t.gSem(\beta)}(\bot, m)] \\
\quad \textbf{else if } \tau = [s_p, \tau'] \textbf{ then return } [s_p, parse_{t.gSem(\beta)}(\tau', m)]
\end{array} \right. \\
eff = t.\ eff
\end{array} \right.
\end{array}
$$

Combinator $\mathsf{call}(t, v_1 = e_1, ..., v_n = e_n)$ denotes a template call whose target is t with actual argument e_i passed to formal parameter v_i. In brief, it prepares a new model by $\{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), ..., v_n = get_{e_n}(m)\}$ first, and then delegates the conversion to t. The behavior of call is defined as follows.

$$
\begin{array}{l}
\mathsf{call}(t, v_1 = e_1, ..., v_n = e_n) \triangleq \\
\mathsf{BIT} \left\{ \begin{array}{l}
syn = \left\{ \begin{array}{l} parse(s_I) \triangleq \textbf{if } s_I = \bot \textbf{ then } (\bot, \bot) \\ \qquad\qquad \textbf{else if } (\tau, s_T) = parse_{t.syn}(s_I) \textbf{ then } (\mathsf{C}\ \tau, s_T) \\ print(\tau, s_T) \triangleq \textbf{if } \tau = \bot \textbf{ then } print_{t.syn}(\bot, s_T) \\ \qquad\qquad \textbf{else if } \tau = \mathsf{C}\ \tau' \textbf{ then } print_{t.syn}(\tau', s_T) \end{array} \right. , \\
gSem = \lambda\beta \rightarrow \\
\quad \left\{ \begin{array}{l} parse(\tau, m) \triangleq \textbf{do} \\ \quad m_t \leftarrow \{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), ..., v_n = get_{e_n}(m)\} \\ \quad \textbf{if } \tau = \mathsf{C}\ \tau' \textbf{ then} \\ \quad\quad m'_t \leftarrow parse_{t.gSem(\beta)}(\tau', m_t) \\ \quad\quad m_i \leftarrow put_{e_i}(m_{i-1}, get_{v_i}(m'_0)) \quad \text{where } i = 1..n, m_0 = m \\ \quad\quad \textbf{assert } get_{e_i}(m_n) = get_{v_i}(m'_t) \quad (i = 1..n) \\ \quad\quad \textbf{return } m_n \\ print(\tau, m) \triangleq \textbf{do} \\ \quad m_t \leftarrow \{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), ..., v_n = get_{e_n}(m)\} \\ \quad \textbf{if } \tau = \bot \textbf{ then return } \mathsf{C}\ print_{t.gSen(\beta)}(\bot, m_t) \\ \quad \textbf{else if } \tau = \mathsf{C}\ \tau' \textbf{ then return } \mathsf{C}\ print_{t.gSen(\beta)}(\tau', m_t) \end{array} \right. \\
eff = \lambda(\beta, m) \rightarrow \textbf{do} \\
\quad m_t \leftarrow \{v_1 = get_{e_1}(m), v_2 = get_{e_2}(m), ..., v_n = get_{e_n}(m)\} \\
\quad \textbf{return } t.\ eff(\beta, m_t)
\end{array} \right.
\end{array}
$$

Note that when propagating the updates back to the original model, it requires no conflict in the merged result (see the assertion in *parse* of *gSem*).

3.4. Translation from surface to core

This subsection describes the translation from the surface language, designed for better usability, to the core language, whose semantics is formally defined. The basic idea of the translation is to map template fragments in the surface language onto primitives/combinators in the core language. It is not difficult to find a strong correspondence between the surface language and the core language if we compare their grammars in Fig. 5 and Fig. 7. For example, the FOR construct and the IF construct correspond to loop and ite, respectively; a hole corresponds to lex; a constant template literal corresponds to const.

For example, a template '''int «a|ID»()''' can straightforwardly be translated into a BIT BX t— $\mathsf{seq}(\mathsf{const}(\texttt{"int "}), \mathsf{lex}(\mathsf{ID}, \mathsf{a}), \mathsf{const}(\texttt{"()"}))$. Note that there is a tailing space in "int ". This simple translation strategy has a limitation that makes the derived parser inflexible. For example, the derived parser of t accepts "int f()" but rejects "int f ( )".

The root cause is that the simple translation does not take the grammar of the generated text into account. If we know that the template generates a signature of a Java method, then we can translate the template into t'

$$\mathsf{seq}\big(\mathsf{const}(\texttt{"int"}), \mathsf{space}(\texttt{" "}), \mathsf{lex}(\mathsf{ID}, \mathsf{a}), \mathsf{space}(\texttt{""}), \mathsf{scope}\big(\texttt{"("}, \texttt{")"}, \mathsf{space}(\texttt{""})\big)\big)$$

by considering the white-space rules and bracket rules.

For simplicity, our approach does not require the full grammar of the generated text. Instead, we can define a *partial grammar* to guide the surface-to-core translation. The partial grammar is not intended to specify the syntactic constraints of the generated text, but is only used to enhance the derived parser. The *partial grammar* includes the following rules:

- **White-space rule** tells whether the white-spaces in the generated text can be *relaxed*, e.g., white-spaces can be added, changed, or removed whenever they may occur. If the rule is set, our approach uses space to handle white-spaces; otherwise, our approach will treat white-spaces as constants.
- **Operator rule** specifies the tokens (e.g., +, *) in template literals that must be considered as operators. If the rule and white-space rule are both set, then our approach will insert zero-width spaces around operators. In this way, template '''a+b''' accepts "a+b", "a +b", and "a + b".
- **Balanced bracket rule** tells what brackets should be balanced. If the rule is set, then our approach will scan template literals to match the scopes that start and end with balanced brackets (called *balanced scopes*).

We explain the surface-to-core translation by an example template

$$\texttt{'''«a|ID»()\{return 1+1;\}'''}$$

First, we adopt the straightforward translation strategy and obtain an initial core program $t_0 = \mathsf{seq}(\mathsf{lex}(\mathsf{ID}, \mathsf{a}), \mathsf{const}(\texttt{"()\{return 1+1\}"}))$.

Second, supposing the balanced bracket rule tells (,), and {, } must be balanced, we scan const primitives in t_0 and detect balanced scopes. We rewrite t_0 to t_1 by extracting each balanced scope to a scope, as follows:

$$\mathsf{seq}(\mathsf{lex}(\mathsf{ID}, \mathsf{a}), \mathsf{scope}(\texttt{"("}, \texttt{")"}, \mathsf{nop}), \mathsf{scope}(\texttt{"\{"}, \texttt{"\}"}, \mathsf{const}(\texttt{"return 1+1"})))$$

Third, supposing the operator rule tells + is an operator, we rewrite t_1 into t_2 by tokenizing const primitives with operator +, as follows:

$$\begin{array}{l}\mathsf{seq}(\mathsf{lex}(\mathsf{ID}, \mathsf{a}), \mathsf{scope}(\texttt{"("}, \texttt{")"}, \mathsf{nop}), \mathsf{scope}(\texttt{"\{"}, \texttt{"\}"},\\ \quad \mathsf{seq}(\mathsf{const}(\texttt{"return 1"}), \mathsf{const}(\texttt{"+"}), \mathsf{const}(\texttt{"1"}))))\end{array}$$

Finally, if the white space rule is set, we extract white-spaces from const primitives and insert zero-width spaces (let $z = \mathsf{space}(\texttt{""})$) when necessary:

$$\begin{array}{l}\mathsf{seq}\big(\mathsf{lex}(\mathsf{ID}, \mathsf{a}), z, \mathsf{scope}(\texttt{"("}, \texttt{")"}, z), \mathsf{space}(\texttt{""}), \mathsf{scope}(\texttt{"\{"}, \texttt{"\}"},\\ \quad \mathsf{seq}(\mathsf{const}(\texttt{"return"}), \mathsf{space}(\texttt{" "}), \mathsf{const}(\texttt{"1"}), z, \mathsf{const}(\texttt{"+"}), z, \mathsf{const}(\texttt{"1"})))\big)\end{array}$$

4. Well behavedness

In this paper, a BX is well behaved if it satisfies corresponding round-trip properties. This section discusses the round-trip properties of BIT BXs, which reflect the compatible behaviors of printers and parsers, as the *formal evidence of the soundness* of our approach.

4.1. Well behavedness of constructors

The formalization of our approach is built upon the constructors $*, \circledast, \odot$ defined in Section 3.2. Theorems 1, 2, and 3 state that these constructors preserve well-behavedness. The proof sketches are listed as follows.

Proof sketch of Theorem 1. Given well-behaved $val : \mathbb{M} \leftrightarrow V$ and $l : \mathbb{T} \leftrightarrow V$, we must prove $val * l$ satisfies properties (1) and (2) of $\alpha BX : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$. Because unit can be ignored from the definition of $*$, to prove 1 is equivalent to prove

$$v = get_l(t) \wedge m' = put_{val}(m, v) \implies v = get_{val}(m') \wedge s = put_l(t, v)$$

and to prove 2 is equivalent to prove

$$v = get_{val}(m) \wedge t' = put_l(t, v) \implies v = get_l(t') \wedge m = put_{val}(m, v)$$

Since val and l are well behaved (they satisfy GetPut and PutGet), the above two formulas hold. Thus, Theorem 1 holds.

Proof sketch of Theorem 2. Given well-behaved $l_1 : \mathbb{S} \leftrightsquigarrow V$ and $l_2 : \mathbb{T} \overset{\mathbb{M}}{\longleftrightarrow} Unit$, we must prove $l_1 \circledast l_2$ satisfies properties (1) and (2) of $\alpha BX : \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$. To prove 1 is equivalent to prove

$$\begin{aligned}(t, s_T) = parse_{l_1}(s) \wedge m' = parse_{l_2}(t, m)\\ \implies (t, s_T) = parse_{l_1}(s) \wedge t = print_{l_2}(t, m') \wedge s = print_{l_1}(t, s_T)\end{aligned}$$

and to prove 2 is equivalent to prove

$$\begin{aligned}(t, s_T) = parse_{l_1}(s) \wedge t' = print_{l_2}(t, m) \wedge s' = print_{l_1}(t', s_T)\\ \implies (t', s_T) = parse_{l_1}(s') \wedge m = parse_{l_2}(t', m)\end{aligned}$$

Particularly, because l_1 and l_2 satisfy ParsePrint and PrintParse of $synBX$ and αBX, respectively, $m' = parse_{l_2}(t, m) \Rightarrow t = print_{l_2}(t, m')$ and $t' = print_{l_2}(t, m) \Rightarrow m = parse_{l_2}(t', m)$. Thus, Theorem 2 holds.

Proof sketch of Theorem 3. Given a binding update function $r : \mathbb{B} \times \mathbb{M} \to \mathbb{B}$ and a generator of αBX $pl : \mathbb{B} \to \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$, we must prove $r \odot pl$ satisfies ParsePrint and PrintParse of $\beta BX : \mathbb{B} \mapsto \mathbb{S} \overset{\mathbb{M}}{\longleftrightarrow} \mathbb{S}$. Because the *parse* and *print* functions of a βBX always start from the same binding β, pl generates the same αBX from the same β. Thus, Theorem 3 obviously holds.

4.2. Well behavedness of `BIT` primitives

`BIT` primitives are defined as BIT' records. To prove a `BIT` primitive t is well behaved, we only need to prove $t.syn$, $t.sem$, and $t.val$ are well behaved. Afterwards, the constructors $*, \circledast, \odot$ shall ensure the well behavedness of t. We present the proof sketch of the well behavedness of `BIT` primitives as follows.

- Case $\mathsf{const}(s_c)$—Since *syn* prints/parses the same constant s_c and *sem* maps s_c onto unit, it is trivial to prove that they are well behaved. Besides, *val* is $UnitBX$. Accordingly, $\mathsf{const}(s_c)$ is well behaved.

- Case $\mathsf{lex}(\rho, e)$—*syn* uses a regular pattern ρ to parse a string and to verify the string it prints out so it is well behaved. *sem* is equal to the BX, namely, REPLACE, defined in BiGUL [22]. *val*, which is e, is assumed to be a well-behaved expression BX. Accordingly, $\mathsf{const}(s_c)$ is well behaved.

- Case $\mathsf{space}(s_w)$—Its *syn* is very similar to that of lex, where space uses a regular pattern ρ_w for white-spaces. Its *sem* maps a non-$\perp$ string of white-spaces onto unit bidirectionally. Accordingly, $\mathsf{space}(s_w)$ is well behaved.

- Cases $\mathsf{assign}(v, e)$ and nop—It is trivial to prove the two primitives are well behaved because their *syn* and *sem* functions actually do nothing.

4.3. Well behavedness of `BIT` combinators

`BIT` combinators are defined as BIT records. To prove a `BIT` combinator t is well behaved, we only need to prove $t.syn$ and $t.gSem(\beta)$ are well behaved. Afterwards, the constructors $\circledast, \odot$ shall ensure the well behavedness of t. We present the proof sketch of the well behavedness of `BIT` combinators as follows.

- Case $\mathsf{seq}(t_1, t_2)$—*syn* applies $t_1.syn, t_2.syn$ in the forward order in parsing and in the reverse order in printing. Hence, *syn* should be well behaved if $t_1.syn, t_2.syn$ are well behaved. Similar to *syn*, *gSem* applies $t_1.gSem, t_2.gSem$ in the forward order in parsing and in the reverse order in printing. Particularly, the assertion in $gSem(\beta)$ requires that t_1 is not affected by t_2. Hence, $gSem(\beta)$ is also well behaved.

- Case $\mathsf{ite}(e, t_1, t_2)$—*syn* chooses $t_i.syn$ to parse the input string if $t_i.syn$ consumes longer prefix; while in printing, it chooses a branch according to the label (i.e., L/R) of the internal structure and asserts that the printed string cannot be consumed by the other branch. Hence, *syn* is well behaved. $gSem(\beta)$ chooses $t_i.gSem(\beta)$ according to the label of the internal structure and then enforces the branch condition e (which is also a well-behaved expression BX); while in printing, it chooses a branch according to the branch condition. Hence, $gSem(\beta)$ is well behaved.

- Case $\mathsf{loop}(e_{arr}, s_s, s_b, s_a, \lambda v \to t)$—Since we interpret the behavior of loop as seq, loop is well behaved if seq is so.

- Case $\mathsf{scope}(s_b, s_a, t)$—In parsing, *syn* matches a scope in which s_b and s_a are balanced, and asks $t.syn$ to parse the inner content; while in printing, *syn* prepends s_b and appends s_a before and after the string printed by $t.syn$, in which s_b and s_a must be balanced. Hence, *syn* is well behaved. Since $gSem(\beta)$ delegates the conversion to $t.gSem(\beta)$, it is also well behaved. Accordingly, scope is well behaved.

- Case default(ρ, t)—If the input string can be parsed/printed by $t.syn$, then syn applies $t.syn$ to handle this string; otherwise, syn uses the regular pattern ρ to parse the string and to verify the string to be printed out. Hence, syn is well behaved. $gSem(\beta)$ delegates the conversion to $t.gSem(\beta)$ when the input is parsed/printed by $t.syn$; otherwise, $gSem(\beta)$ basically skips the conversion. It is easy to prove that $gSem(\beta)$ is also well behaved. As a result, default is well behaved.

- Case unord(t_1, t_2)—Similar to ite, syn of unord chooses from two pseudo branches seq(t_1, t_2) and seq(t_2, t_1). Particularly, syn assures that the string printed by one branch cannot be parsed by the other. Hence, syn is well behaved (see the proof of ite). Regarding $gSem(\beta)$, it chooses from the pseudo branches according to the labels of the internal structure, and then delegates the conversion to the branch it selects (and there is no branch switching that may happen in ite). It is easy to show that $gSem(\beta)$ is well behaved. Hence, unord is also well behaved.

- Case fina(t)—ParsePrint of syn holds by definition; PrintParse of syn also holds because the assertion in $print$ ensures that the string printed can still be correctly parsed. Hence, syn is well behaved. $gSem(\beta)$ actually delegates the conversion to $t.gSem(\beta)$, so it is well behaved. Accordingly, fina is well behaved.

- Case call$(t, v_1 = e_1, ..., v_n = e_n)$—Due to the fact that call actually uses t to realize the conversion, it shall be well behaved.

5. Case studies

To evaluate our approach empirically, we conducted two case studies. In the first case study (see Section 5.1), we used `BIT` to implement a code template originating from a real-world UML tool, namely UMLLab, which generates a Java class from a UML *Class* element. In the second case study (see Section 5.2), we used `BIT` to define 42 templates which were collected from the tutorials of 7 template languages. The supplemental package [33] includes a demo and the code of `BIT`, and the details of the two case studies.

5.1. Case 1: UML class code template

UMLLab is a visual UML tool that supports both code generation from UML models and reverse engineering from Java code using code template. In code generation, UMLLab applies code templates to generate Java source code; in reverse engineering, it uses the same set of code templates to extract a model from Java code.

In this case study, we used `BIT` to realize the core template in UMLLab, which generates UML *Classes* to Java classes (also including code generation from *Classes*, *Properties*, *Operations*, and *Associations*). We fully realized the code template using `BIT`. The `BIT` implementation has 273 LOC and 29 sub-templates. Figure 9 presents the excerpt of our `BIT` implementation. The tem-

```
1. template classifierForClass(c : Class)
2. '''
3. «comment(c.ownedComment)»
4. «visibility(c.visibility)»«IF c.isAbstract»abstract «ENDIF»«IF c.keywords.includes("static")»static «ENDIF»«IF c.isLeaf»final
   «ENDIF»class «c.name|ID»
5.  «IF c.superType != null»extends «c.superType.typeString|QUALIFIEDNAME»«ENDIF»
6.  «IF !c.superInterfaces.isEmpty»implements «FOR i : TypeRef IN c.superInterfaces SEPARATOR ","»«i.typeString|
   QUALIFIEDNAME»«ENDFOR»«ENDIF»
7. {
8. «FOR p : Property IN c.ownedProperty.filter((p:Property)->p.association==null)»
9. «attributeDeclarationForClass(p)»
10.«attributeSetterForClass(p, c)»
11.«attributeGetterForClass(p, c)»
12.«ENDFOR»
13.«FOR p : Property IN c.ownedProperty.filter((p:Property)->p.association != null && p.otherEnd != null)»
14.«roleDeclarationForClass(p, c)»
15.«roleSetterForClass(p, c)»
16.«roleGetterForClass(p, c)»
17.«ENDFOR»
18.«FOR p : Operation IN c.ownedOperation»
19.«operationOfClass(p, c)»
20.«ENDFOR»
21.}
22.'''

24.template comment(comments : [Comment])
25.'''
26.«IF !comments.isEmpty()»
27./**«FOR comment : Comment IN comments SEPARATOR " * ----------------------------------------------"»«FOR cl : String IN
   comment.body.split("\n")»
28. * «cl|COMMENTLINE»«ENDFOR»
29.«ENDFOR» */
30.«ENDIF»
31.'''
```

Figure 9: Excerpt of `BIT` templates for code generation from UML Classes

plate *classifierForClass* generates a Java class from a UML *Class* element. On line 3, it calls the template *comment* to generate a Javadoc based on the comments owned by the input class. Line 4 generates the modifiers and the class declaration. Lines 5 and 6 generate the super class and super interface list. Lines 7–21 generate the class body: lines 8–12 serialize owned UML *Properties*; lines 13–17 serialize owned UML *Associations*; lines 18–20 print UML *Operations*. The template *comment* generates a Javadoc from a list of Comments. If the Comment list is non-empty, then it generates an array of comment fragments, separated by a long horizontal line. Each comment fragment, which is printed by the inner FOR construct, is a list of comment lines: first, the inner FOR construct splits a *Comment* body into a string array using the line terminator; second, it prints each string to a comment line, beginning with an asterisk.

We tested the `BIT` implementation and Figure 10 shows an example test. The UML model we used to perform the code generation contained three UML *Classes*: *Person*, *Student*, and *Course*. *Person* owned a *Property* called *name*; *Student* owned an *Operation*; there was an *Association* called *students_ to_ courses* between *Student* and *Course*. Then, we applied our `BIT` templates to generate three Java classes from the model. The Java code was consistent with the UML model. After that, we modified the generated Java code and performed the parsing procedure to update the model. As shown in the bottom half of Figure 10, we added a new field *favorites* in the Java class *Student* (the green lines). Finally, after parsing the modified code, we obtained an updated model which contains a new association *favorite_ courses* between *Student* and *Course*.

We also tested other code modifications, including (1) adding a new Java

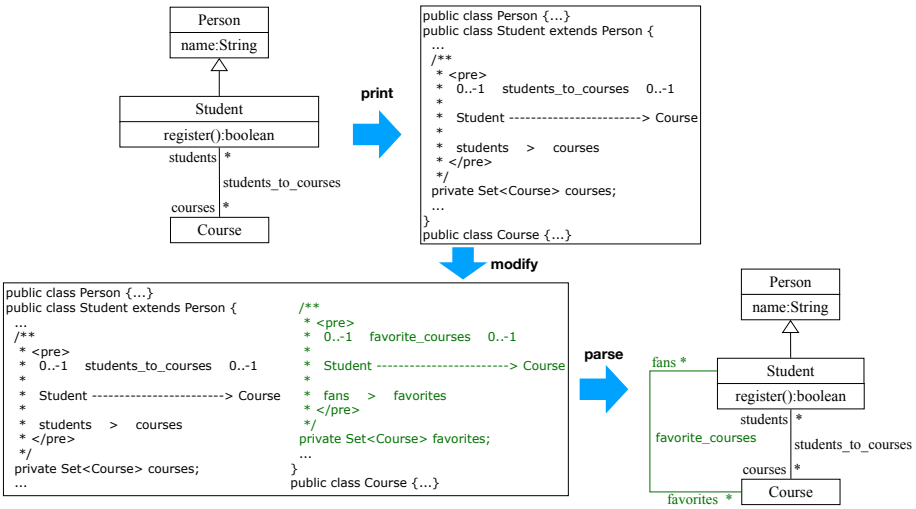


Figure 10: Example execution of Case 1 (some code details are omitted)

field corresponding to a UML *Property*; (2) deleting an existing Java field; (3) adding a new Java method; (4) deleting an exiting Java method; (5) modifying a method body; (6) adding a new Java class; (7) deleting an exiting Java class. `BIT` always can correctly synchronize the Java code with the UML model.

Although we only realized the template for UML *Classes*, we believe that `BIT` can also realize other templates for *Interfaces* (which can be viewed as a class without attributes and method bodies) and *Enumerations* (which can be viewed as a class without methods) in UMLLab because the code generation of classes is more complicated than that of interfaces and enumerations.

Failure Cases of UMLLab. Because UMLLab adopts code templates to guide the reverse engineering, we explored whether there were some cases which cannot be correctly bidirectionalized in UMLLab but can be handled in `BIT`. We found several failure cases of UMLLab, and Figure 11 presents two of them.

```
template failureCase1(expr:String)
'''
(«expr»).toString()
'''
```

(a) Failure case 1

```
template failureCase2(k:String)
'''
«IF k == "key"»key«ELSEIF k == null»«ENDIF»
'''
```

(b) Failure case 2

Figure 11: Failure cases of UMLLab

- Case 1, as shown in Figure 11a, is a simple template that prints a sub-expression literal *expr* within an outer *toString* expression. UMLLab failed in parsing an input string "((a+b)+c).toString()", because UMLLab is unaware of balanced brackets and expected to parse ".toString()" after it

Languages	Domains	#Examples	Tutorial URLs
Velocity	Web, Code generation	12	https://velocity.apache.org/engine/devel/user-guide.html#what-is-velocity
Freemaker	Web, Code generation	15	https://freemarker.apache.org/docs/index.html
Xtend	General purpose	4	https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates
Acceleo	MDE	2	https://wiki.eclipse.org/Acceleo/Getting_Started
Django	Web	4	https://docs.djangoproject.com/en/4.1/intro/tutorial03/ and https://docs.djangoproject.com/en/4.1/ref/templates/language/
Mustache	Web, Code generation	7	http://mustache.github.io/mustache.5.html
Nunjucks	Web	6	https://mozilla.github.io/nunjucks/templating.html

Table 1: General information of the benchmark examples

consumed the first ")". With the help of scope, BIT parsed the string correctly, yielding expr="(a+b)+c".

- Case 2, as shown in Figure 11b, is a conditional construct, which prints an empty string when k is null. UMLLab failed in parsing an empty string when k is originally "key". It is because that UMLLab's approach to round-trip engineering is not established upon bidirectional transformation and cannot handle conditionals correctly. Consequently, it cannot assure correct synchronization between models and code.

5.2. Case 2: Benchmark templates

Due to the lack of a public benchmark set, we collected our benchmark set as follows. We chose 7 popular template languages as the reference languages, including Velocity, Freemaker, Xtend, Acceleo, Django, Mustache, and Nunjucks, which covers different application domains. Then, we scanned their official tutorial documents and extracted some examples as our benchmarks. We excluded an example if (1) it was incomplete (e.g., trivial single-line code), (2) it was not intended to show the features of a template language (e.g., the usage of library functions), or (3) it was a simplified/equivalent version of other examples. Finally, we collected 50 examples. We collected the examples from the official tutorials because we think these examples illustrates the language features well, though they are not very complicated. The information on the 7 template languages, the numbers of extracted examples, and the URLs of the tutorials is listed in Table 1.

```
template django_case1(latest_question_list : [Question])
'''
«IF !latest_question_list.isEmpty()»
<ul>
«FOR question : Question IN latest_question_list»
  <li><a href="/polls/«question.id»/">«question.text»</a></li>
«ENDFOR»
</ul>
«ELSE»
<p>No polls are available.</p>
«ENDIF»
'''
```

(a) Example from Django

```
template freemaker_case8(animal : Animal)
'''
«IF animal.size == "small"»
 This will be processed if it is small
«ELSEIF animal.size == "medium"»
 This will be processed if it is medium
«ELSEIF animal.size == "large"»
 This will be processed if it is large
«ELSE»
 This will be processed if it is neither
«ENDIF»
'''
```

(b) Example from Freemaker

```
template mustache_case1(name : String, value : int, in_ca : boolean, taxed_value : int)
'''
Hello «name | ID»
You have just won «value | INT» dollars!
«IF in_ca»
Well, «taxed_value | INT» dollars, after taxes.
«ELSE»
«ENDIF»
'''
```

(c) Example from Mustache

Figure 12: Examples of benchmark templates implemented in BIT

We implemented 42 out of 50 templates using BIT. Figure 3a is one of our benchmark examples from Xtend (the original template is presented in Figure 2a). Figure 12 presents more examples:

- Figure 12a is a template originated from Django, which generates a HTML fragment of an unordered question list.
- Figure 12b is a template extracted from Freemaker, which generates a constant sentence according to the value of the animal's size.
- Figure 12c shows a template from Mustache, which a piece of text describing the dollars won by a person.

This implies that BIT has sufficient expressiveness to cover the major features (e.g., conditionals, loops, assignments, and template calls) which are supported by most modern template languages.

We tested each benchmark template with several test cases to check whether it can correctly print/parse strings, bidirectionally and incrementally. All the BIT templates passed the tests, implying that our approach (and our prototype tool) is functionally correct.

5.3. Limitations of BIT

Language Features. Although BIT covers the major features of existing template languages, there are 8 benchmark templates (1 from Velocity, 4 from Freemaker, 1 from Acceleo, 1 from Django, and 1 from Nunjucks) that cannot be handled by our approach. We investigated these templates, and found that they contained some extra features (e.g., verbose text and template extensions) which are unsupported in BIT.

For example, Figure 13a shows a Velocity template which contains verbose text (i.e., the content between #[[and #]]). Currently, BIT does not support

```
#[[
#foreach ($woogie in $boogie)
  nothing will happen to $woogie
#end
]]#
```

(a) An unsupported Velocity template

```
<#macro border>
  <p><#nested></p>
</#macro>
<@border>
content to be filled in the nested part
</@border>
```

(b) An unsupported Freemaker template

Figure 13: Examples of unsupported benchmark templates

verbose text so we did not realize this template. However, we think that the verbose text can be turned into string constants which are supported by BIT. Figure 13b shows a Freemaker template which contains *nested templates*. The macro definition specifies a template *border*, while the use of the template (i.e., <@border>...</@border>) fills the inner content to the <#nested> part. We think that these nested templates are similar (but not equal) to template calls.

It will be our future work to investigate how to extend BIT with more language features.

Left Recursions. Parsers derived from BIT templates cannot handle left recursions. Figure 14 shows a concrete example template containing left recursion. Currently, the tool support of BIT cannot check left recursion statically. It will be our future work to investigate how to detect left recursions in templates by adopting existing techniques in the field of compilers.

```
template leftRec(a:int)
'''
«IF a>0»«leftRec(a-1)»+«a»«ELSE»0«ENDIF»
'''
```

Figure 14: A template with left recursion

Efficiency. A potential limitation of BIT is the runtime efficiency. To ensure well behavedness, BIT applies many runtime checks (e.g., the **assert** statements in the definition of the core language). We need those runtime checks for two reasons. First, during parsing, the model is updated incrementally, so we must ensure that subsequent updates will not undo the effect of earlier updates. Second, during printing, we must ensure that the chain of two printers will not disturb the behaviors of the corresponding parsers. It will be our future work to investigate how to optimize the efficiency of BIT.

6. Related Work

6.1. Template languages and M2T transformation

Template languages, such as Velocity [34], Freemaker [35], Django templates [36], Nunjucks [37], and Mustache [38], are widely used in modern Web engineering. In model-driven architecture, template languages, such as Acceleo [39],

Xtend templates [9], and Java Emitter Template [40], are also used to achieve model-to-text transformations [41]. Specifically, Object Management Group proposed a standard template language, namely MOF Model to Text Transformation Language [8], for code generation and document generation. State of the art template languages are unidirectional, which convert models/data into text/code. In this paper, we proposed `BIT` as an extension to Xtend templates to support the reverse conversion from text to models.

6.2. Model-model and model-code synchronization

In model-driven architecture, how maintain the consistency between high-level models and low-level models/code is a crucial problem [42, 43].

Most research efforts on this topic focused on the synchronization between models. Triple Graph Grammars (TGGs) were widely used to achieve model-to-model synchronization. Hermann et al. [11] proposed a formal foundation for bidirectional transformation using TGGs. Concrete implementations and optimizations of TGG-based model synchronization [12, 13, 14] were also proposed. Xiong et al. [15] proposed an ATL-based approach to model synchronization by defining a reverse semantics for ATL. Macedo et al. [16] proposed a solver-based approach to bidirectional model transformation. Their basic idea is to convert ATL rules and QVT relations rules into Alloy constraints and then ask the Alloy solver to compute a synchronized model. EVL-Strace [17] is a model synchronization approach based on Epsilon Validation Language (EVL). It employs a trace model and EVL to achieve change propagation. Buchmann et al. [18] proposed a layered framework for bidirectional model transformations combining declarative and imperative programming. Their approach achieved a high expressiveness and scalability. Boronat [19] proposed EMF-Syncer which allowed for the synchronization between models and structured data. EMF-Syncer was highly scalable but limited to one-to-one mapping. Our previous work [20] on bidirectional model transformation proposed a putback-based language which enabled us to define a backward transformation from which a well-behaved BX can be derived. We also explored the synchronization between models and running systems [21].

There were a few research efforts on model-code synchronization. Yu et al. [29] proposed an approach to maintaining consistency between Ecore models and Java code. They employed bidirectional graph transformation [26] to achieve the synchronization between models and abstract syntax trees. Because their synchronization strategy was only applicable to Ecore models, their approach was not general purpose.

Chivers et al. [44] proposed XRound, a reversible template language for the synchronization between models and XML-based documents. However, XRound cannot be used to define templates for non-XML generation, such as source code.

Lemerre [45] proposed an approach RTL to reverse general-purpose templates. First, this approach derived a parser from a template to parse a string. Then, it used the concept of abstract interpretation and a constraint solver to determine a parsing tree and the value extracted out from the string. RTL was not built on the theory of bidirectional transformation so it did not have a

well behaved bidirectional semantics. Neither, RTL did not support incremental printing/parsing.

6.3. Bidirectional transformation and bidirectional parsers

Bidirectional transformations (aka lenses) [24, 46, 47] have been intensively studied in programming language community from the perspectives of theory, languages, and applications. Most existing BX approaches focused on the transformations over structured data [22], such as lists [23], trees [24, 25], graphs [26, 27], and relational databases [28]. Zhang et al. [31, 32] proposed bidirectional live programming approaches for incomplete and object-oriented UI programs.

Bohannon et al. [48] proposed Boomerang, the resourceful lenses for string data. Boomerang used regular patterns to tokenize strings into chunks and then synchronized these chunks with lenses for dictionaries and bidirectional regular transducers (aka string lens combinators). However, Boomerang is not template-based and cannot be used to reverse templates.

There were also many research efforts on bidirectional parsers [49, 50, 51]. Zhu et al. [49] proposed BiYacc, a domain-specific language for the specification of mapping rules from abstract syntax to concrete syntax of a programming language. Afterward, BiYacc rules can be compiled into BiGUL [22] programs to achieve the synchronization between abstract syntax trees (ASTs) and concrete syntax trees (CSTs). Matsuda et al. [50] proposed FliPpr, a system for deriving parsers from pretty-printers. FliPpr provided a surface language for describing a pretty printer of a language, which was further translated into an ugly printer by forgetting smart layouting mechanism. The ugly printer was then processed by their grammar-based inversion system [52] to realize the parsing semantics. Similar to BiYacc and FliPpr, `BIT` also addresses the issue of consistent printing and parsing. The major difference is that `BIT` is a template-based approach while BiYacc and FliPpr are grammar-based. The core language of `BIT` is designed to handle template fragments, rather than grammar productions.

Xia et al. [51] proposed the concept of *monadic profunctors* to define the behavior of bi-parsers (i.e., a pair of parse and print functions). In this way, bi-parsers can be combined in a monadic way. The concept of αBX can be regarded as an extension to the monadic profunctor—αBX adopts a value BX, rather than a *comap* function, to achieve a more flexible bidirectional behavior.

Comby [30] was a template language for code matching and rewriting. In matching process, Comby interprets a code template and finds code fragments which match the structure of the template. In rewriting process, Comby generates code by filling the holes in the template. `BIT` is inspired by Comby to use partial grammars to guide the interpretation of templates. However, Comby was not designed for bidirectional transformation so it does not assure any round-trip properties.

7. Conclusions and future work

In this paper, we proposed `BIT`, a bidirectional template-based approach to incremental printing and parsing. We defined the surface language of `BIT` by extending conventional template language for better usability. We formally specified the semantics of `BIT` by means of the definition of the core language of `BIT`. The proof sketch of the well behavedness and two case studies were also presented to show the soundness and the expressiveness of our approach.

Regarding the future work, we plan to improve our approach in the following aspects. First, we will enhance `BIT` by adding more language features, such as verbose text and template extension. Second, we will optimize the prototype implementation of `BIT` to improve its runtime efficiency. Finally, we will try to refine our approach by defining a bidirectional semantics for the newly proposed string calculus [53].

Acknowledgement

This work is funded by National Key Research and Development Program of China (No. 2023YFB3002903), Natural Science Foundation of Fujian Province for Youths (No. 2021J05230), Beijing Natural Science Foundation (No. 4192036).

Declaration of Generative AI and AI-assisted technologies in the writing process

Statement: During the preparation of this work the authors used ERNIE Bot 3.5 in order to grammar checks. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

References

[1] O. M. Group, Model driven architecture, `http://www.omg.org/mda`.

[2] A. W. Brown, Model driven architecture: Principles and practice, Software and Systems Modeling 3 (4) (2004) 314–327.

[3] A. Rodrigues da Silva, Model-driven engineering: A survey supported by the unified conceptual model, Computer Languages, Systems & Structures 43 (2015) 139–155.

[4] E. Umuhoza, M. Brambilla, Model driven development approaches for mobile applications: A survey, in: M. Younas, I. Awan, N. Kryvinska, C. Strauss, D. v. Thanh (Eds.), Mobile Web and Intelligent Information Systems, Springer International Publishing, Cham, 2016, pp. 93–107.

[5] I. Boussaïd, P. Siarry, M. Ahmed-Nacer, A survey on search-based model-driven engineering, Automated Software Engineering 24 (2) (2017) 233–294.

[6] D. Akdur, V. Garousi, O. Demirörs, A survey on modeling and model-driven engineering practices in the embedded software industry, Journal of Systems Architecture 91 (2018) 62–82.

[7] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, D. Varró, Survey and classification of model transformation tools, Software & Systems Modeling 18 (4) (2019) 2361–2397.

[8] O. M. Group, Mof model to text transformation language, version 1.0, https://www.omg.org/spec/MOFM2T/1.0/About-MOFM2T/ (jan 2008).

[9] Xtend templates, https://eclipse.dev/Xtext/xtend/documentation/203_xtend_expressions.html#templates (dec 2023).

[10] X. He, P. Avgeriou, P. Liang, Z. Li, Technical debt in mde: A case study on gmf/emf-based projects, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 162–172. doi:10.1145/2976767.2976806.
URL https://doi.org/10.1145/2976767.2976806

[11] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, S. Gottmann, T. Engel, Model synchronization based on triple graph grammars: correctness, completeness and invertibility, Software & Systems Modeling 14 (1) (2015) 241–269.

[12] H. Giese, R. Wagner, From model transformation to incremental bidirectional model synchronization, Software & Systems Modeling 8 (1) (2009) 21–43.

[13] F. Hermann, H. Ehrig, C. Ermel, F. Orejas, Concurrent model synchronization with conflict resolution based on triple graph grammars, in: J. de Lara, A. Zisman (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 178–193.

[14] F. Orejas, E. Pino, M. Navarro, Incremental concurrent model synchronization using triple graph grammars, in: H. Wehrheim, J. Cabot (Eds.), Fundamental Approaches to Software Engineering, Springer International Publishing, Cham, 2020, pp. 273–293.

[15] Y. Xiong, H. Song, Z. Hu, M. Takeichi, Synchronizing concurrent model updates based on bidirectional transformation, Software & Systems Modeling 12 (1) (2013) 89–104.

[16] N. Macedo, A. Cunha, Least-change bidirectional model transformation with qvt-r and atl, Software & Systems Modeling 15 (3) (2016) 783–810.

[17] L. Samimi-Dehkordi, B. Zamani, S. Kolahdouz-Rahimi, Evl+strace: a novel bidirectional model transformation approach, Information and Software Technology 100 (2018) 47–72.

[18] T. Buchmann, M. Bank, B. Westfechtel, Bxtenddsl: A layered framework for bidirectional model transformations combining a declarative and an imperative language, Journal of Systems and Software 189 (2022) 111288.

[19] A. Boronat, Emf-syncer: scalable maintenance of view models over heterogeneous data-centric software systems at run time, Software and Systems Modeling (Jun 2023).

[20] X. He, Z. Hu, Putback-based bidirectional model transformations, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 434–444.

[21] X. He, Z. Hu, N. Meng, A theoretic framework of bidirectional transformation between systems and models, Science China Information Sciences 65 (10) (2022) 202103.

[22] H.-S. Ko, Z. Hu, An axiomatic basis for bidirectional programming, Proc. ACM Program. Lang. 2 (POPL) (dec 2017).

[23] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, B. C. Pierce, Matching lenses: Alignment and view update, in: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 193–204.

[24] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for bi-directional tree transformations: A linguistic approach to the view update problem, in: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 233–246.

[25] Z. Hu, S.-C. Mu, M. Takeichi, A programmable editor for developing structured documents based on bidirectional transformations, Higher-Order and Symbolic Computation 21 (1) (2008) 89–118.

[26] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano, Bidirectionalizing graph transformations, in: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 205–216.

[27] S. Hidaka, K. Asada, Z. Hu, H. Kato, K. Nakano, Structural recursion for querying ordered graphs, in: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 305–318.

[28] V.-D. Tran, H. Kato, Z. Hu, Programmable view update strategies on relations, Proc. VLDB Endow. 13 (5) (2020) 726–739.

[29] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, L. Montrieux, Maintaining invariant traceability through bidirectional transformations, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, 2012, pp. 540–550.

[30] R. van Tonder, C. Le Goues, Lightweight multi-language syntax transformation with parser parser combinators, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 363–378.

[31] X. Zhang, Z. Hu, Towards bidirectional live programming for incomplete programs, in: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022, pp. 2154–2164. doi:10.1145/3510003.3510195.

[32] X. Zhang, G. Guo, X. He, Z. Hu, Bidirectional object-oriented programming: Towards programmatic and direct manipulation of objects, Proc. ACM Program. Lang. 7 (OOPSLA1) (apr 2023).

[33] X. He, Incremental and bidirectional template language for model-text synchronization. doi:10.5281/zenodo.10433994.

[34] `ApacheVelocityProject`, `https://velocity.apache.org` (dec 2023).

[35] Apache freemaker project, `https://freemarker.apache.org` (dec 2023).

[36] Django templates, `https://docs.djangoproject.com/en/4.2/topics/templates/` (dec 2023).

[37] Nunjucks project, `https://mozilla.github.io/nunjucks/templating.html` (dec 2023).

[38] Mustache project, `http://mustache.github.io` (dec 2023).

[39] Acceleo project, `https://eclipse.dev/acceleo/` (dec 2023).

[40] Java emitter template project, `https://projects.eclipse.org/projects/modeling.m2t.jet`.

[41] L. M. Rose, N. Matragkas, D. S. Kolovos, R. F. Paige, A feature model for model-to-text transformation languages, in: Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE '12, IEEE Press, 2012, pp. 57–63.

[42] S. Hidaka, M. Tisi, J. Cabot, Z. Hu, Feature-based classification of bidirectional transformation approaches, Software & Systems Modeling 15 (3) (2016) 907–928.

[43] Z. Diskin, H. Gholizadeh, A. Wider, K. Czarnecki, A three-dimensional taxonomy for bidirectional model synchronization, Journal of Systems and Software 111 (2016) 298–322.

[44] H. Chivers, R. F. Paige, Xround: A reversible template language and its application in model-based security analysis, Information and Software Technology 51 (5) (2009) 876–893.

[45] M. Lemerre, Reverse template processing using abstract interpretation, in: M. V. Hermenegildo, J. F. Morales (Eds.), Static Analysis, Springer Nature Switzerland, Cham, 2023, pp. 403–433.

[46] S. Fischer, Z. Hu, H. Pacheco, The essence of bidirectional programming, Science China Information Sciences 58 (5) (2015) 1–21.

[47] M. Hofmann, B. Pierce, D. Wagner, Symmetric lenses, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 371–384.

[48] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, A. Schmitt, Boomerang: Resourceful lenses for string data, in: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 407–419.

[49] Z. Zhu, H.-S. Ko, Y. Zhang, P. Martins, J. Saraiva, Z. Hu, Unifying parsing and reflective printing for fully disambiguated grammars, New Generation Computing 38 (3) (2020) 423–476.

[50] K. Matsuda, M. Wang, Flippr: A system for deriving parsers from pretty-printers, New Generation Computing 36 (3) (2018) 173–202.

[51] L.-y. Xia, D. Orchard, M. Wang, Composing bidirectional programs monadically, in: L. Caires (Ed.), Programming Languages and Systems, Springer International Publishing, Cham, 2019, pp. 147–175.

[52] K. Matsuda, M. Wang, Embedding invertible languages with binders: A case of the flippr language, in: Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 158–171.

[53] W. Crichton, S. Krishnamurthi, A core calculus for documents: Or, lambda: the ultimate document, in: Proceedings of the 51st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '24, Association for Computing Machinery, New York, NY, USA, 2024, accepted.