Case Studies in Bidirectionalisation
PART 1 OF 3

Bidirectional Transformations
From FP to BX: unidirectional
From FP to BX: unidirectional
From FP to BX: unidir.pair
FROM FP TO BX: lens
From FP to BX: synchroniser
FROM FP to BX: sustainer
From FP to BX: sustainer
Bidirectionalisation

Make or infer BX from FP
PART 2 OF 3

GRAMMAR TRANSFORMATIONS
In GrammarLab:

importG

expr ::= atom "+" expr;
atom ::= ID | INT | "(" expr ")";

abstractize atom ::= ID | INT | <>:"(" expr <>:"")";  
vertical in atom.
unite atom with expr.
abridge expr ::= expr;  

**Grammar Transformations**

\[
\begin{align*}
\text{expr} & : \ldots; \\
\text{atom} & : \text{ID} \mid \text{INT} \mid (\text{expr}); \\
\end{align*}
\]

MINI-LENS IN ACTION
SAME PROBLEMS WITH...

- eliminate
  - introduce what?
- inline
  - extract what?
- unlabel
  - designate what?
- etc
PART 3 OF 3

PARSING & UNPARSING
(Un)Parsing

- Parsing: recognising structure
  - text $\rightarrow$ tree
  - parse tree $\rightarrow$ AST
  - disambiguation of trees/forests
  - tokeniser vs. scannerless

- Unparsing: representing structure
  - model $\rightarrow$ picture
  - tree $\rightarrow$ text
  - (re)formatting
  - serialisation

3 (Un)parsing case study

3.1 Problem description

Different approaches and phases of software language processing feature different kinds of artefacts, which can be considered to fit into one of twelve categories, depicted on Figure 1. Consider three of them:

- **Str** — a purely textual flat string-like representation of a program, easy to edit, transfer and maintain and familiar to what mainstream programmers are used to for the last six decades. There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different entity (such as a parse tree).

- **Ast** — an abstract syntax tree, a conceptual representation of a program which is the most suitable for automated program analysis and assigning semantics. It lacks certain details specific to **Str** such as line numbers and indentation.
Fig. 1. Megamodel of various kinds of parsing and unparsing. Dotted lines denote mappings that rely on either lexical or syntactic definitions; solid lines denote universally defined mappings \[12\]; loops are examples of transformations we consider in this paper.

3. (Un)parsing case study

3.1 Problem description

Different approaches and phases of software language processing feature different kinds of artefacts, which can be considered to fit into one of twelve categories, depicted on Figure 1 \[12\]. Consider three of them:

- **Str**— a purely textual flat string-like representation of a program, easy to edit, transfer and maintain and familiar to what mainstream programmers are used to for the last six decades. There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different entity (such as a parse tree).

- **Ast**— an abstract syntax tree, a conceptual representation of a program which is the most suitable for automated program analysis and assigning semantics. It lacks certain details specific to **Str** such as line numbers and indentation.

- **Cst**— a concrete syntax tree, a detailed representation of a program which is the most suitable for automated program analysis and assigning semantics. It has the same details as **Str**.

Textual  
Structured  
Graphical
Fig. 1. Megamodel of various kinds of parsing and unparsing. Dotted lines denote mappings that rely on either lexical or syntactic definitions; solid lines denote universally defined mappings [12]; loops are examples of transformations we consider in this paper.

3 (Un)parsing case study

3.1 Problem description

Different approaches and phases of software language processing feature different kinds of artefacts, which can be considered to fit into one of twelve categories, depicted on Figure 1 [12]. Consider three of them:

Str — a purely textual flat string-like representation of a program, easy to edit, transfer and maintain and familiar to what mainstream programmers are used to for the last six decades. There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different entity (such as a parse tree).

Ast — an abstract syntax tree, a conceptual representation of a program which is the most suitable for automated program analysis and assigning semantics. It lacks certain details specific to Str such as line numbers and indentation.
Fig. 1. Megamodel of various kinds of parsing and unparsing. Dotted lines denote mappings that rely on either lexical or syntactic definitions; solid lines denote universally defined mappings [12]; loops are examples of transformations we consider in this paper.

3 (Un)parsing case study

3.1 Problem description

different approaches and phases of software language processing feature different kinds of artefacts, which can be considered to fit into one of twelve categories, depicted on Figure 1[12]. Consider three of them:

Str—a purely textual flat string-like representation of a program, easy to edit, transfer and maintain and familiar to what mainstream programmers are used to for the last six decades. There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different entity (such as a parse tree).

Ast—an abstract syntax tree, a conceptual representation of a program which is the most suitable for automated program analysis and assigning semantics. It lacks certain details specific to Str such as line numbers and indentation.

Gra—visual representation of a program, suitable for visual editing and inspection. It is the most concrete representation of a program and can be directly manipulated by a user.
Fig. 1. Megamodel of various kinds of parsing and unparsing. Dotted lines denote mappings that rely on either lexical or syntactic definitions; solid lines denote universally defined mappings [12]; loops are examples of transformations we consider in this paper.

3 (Un)parsing case study

3.1 Problem description

Different approaches and phases of software language processing feature different kinds of artefacts, which can be considered to fit into one of twelve categories, depicted on Figure 1 [12]. Consider three of them:

Str — a purely textual flat string-like representation of a program, easy to edit, transfer and maintain and familiar to what mainstream programmers are used to for the last six decades. There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different entity (such as a parse tree).

Ast — an abstract syntax tree, a conceptual representation of a program which is the most suitable for automated program analysis and assigning semantics. It lacks certain details specific to Str such as line numbers and indentation.
Astärtefacts are depicted on various kinds of artefacts, which can be considered to fit into one of twelve categories.

3.1 Problem description

### (Un)parsing case study

Definition of mappings that rely on either lexical or syntactic definitions; solid lines denote universally used approaches and phases of software language processing feature different entity (such as a parse tree).

It lacks certain details specific to different programs. There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different program instance. There is the most suitable for automated program analysis and assigning semantics. The code has around 3000 line of documented code in Rascal and is publicly available from a dedicated repository: http://github.com/grammarware/bx-parsing

Two practical cases of bidirectionalisation have been described in this abstract. The source code of all discussed prototypes is released as open source through the code. One of the open questions left unanswered is dealing with sustainers. The sustainers are much more interesting because they can model annotations for added ones and disregard the parts of the old instance.

The filter in Fig. 1, a functional convergence graph with nodes-grammars and edges-transformations was presented. It visualises a part of the megamodel of various kinds of parsing and unparsing. Dotted lines denote mappings that effectively correspond to any fragments of the updated code, and recalculates all inferred information. The visualisation makes it easier to understand the principle of bidirectionalisation in the future.

For the last six decades, there is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different program instance. In essence, it is the most suitable for automated program analysis and assigning semantics.

Visually editing bidirectionalised by iteratively adding more information to the transformations until the mapping became bijective. In essence, it is the most suitable for automated program analysis and assigning semantics. Bidirectionalisation of the existing unidirectional mappings and thus achieving preservation of locally significant information while updating the changed fragments. The results are rendered by default.

Two practical cases of bidirectionalisation have been described in this abstract. For the last six decades, there is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different program instance. In essence, it is the most suitable for automated program analysis and assigning semantics.

The filter in Fig. 1, a functional convergence graph with nodes-grammars and edges-transformations was presented. It visualises a part of the megamodel of various kinds of parsing and unparsing. Dotted lines denote mappings that effectively correspond to any fragments of the updated code, and recalculates all inferred information. The visualisation makes it easier to understand the principle of bidirectionalisation in the future.
depicted on kinds of artefacts, which can be considered to fit into one of twelve categories.

3.1 Problem description

3 (Un)parsing case study

defined mappings that rely on either lexical or syntactic definitions; solid lines denote universally

Fig. 1. 

It lacks certain details specific to is the most suitable for automated program analysis and assigning semantics.

but it is not apparent until the language instance is processed and turned

into a different approaches and phases of software language processing feature di

There is some structure in such a program, transferred and maintained familiar to what mainstream programmers are

— a purely textual flat string-like representation of a program, easy to edit,

— an abstract syntax tree, a conceptual representation of a program which

transfer and maintain and familiar to what mainstream programmers are

used to for the last six decades. There is some structure in such a program,

used to for the last six decades. There is some structure in such a program,

into a different approaches and phases of software language processing feature di

is required for the general case of model synchronisation of the update strategy in the case of more than two instances (like in

error-correcting strategies. However, it is yet unclear how to prove termination

instead of maintainers:

Two practical cases of bidirectionalisation have been described in this abstract.

4 Conclusion

http://github.com/grammarware/bx-parsing

language for program analysis and transformation. It is released as open source

and is publicly available from a dedicated repository:

and the source code of all discussed prototypes is released as open source through

Two repositories referenced above. It mostly consists of documented Prolog

for systematic development of new methods of automated and semi-automated

are of practical and engineering nature, but their generalisation may be useful

locally significant information while updating the changed fragments. The results

tion of the existing unidirectional mappings and thus achieving preservation of

mented as traversals of ADT instances operating on the result of the superposi-

until the mapping became bijective. In

bidirectionalised by iteratively adding more information to the transformations

in the future.

One of the open questions left unanswered is dealing with sustainers. However, it is yet unclear how to prove termination

instead of maintainers:

Two practical cases of bidirectionalisation have been described in this abstract.

4 Conclusion

http://github.com/grammarware/bx-parsing

language for program analysis and transformation. It is released as open source

and is publicly available from a dedicated repository:

and the source code of all discussed prototypes is released as open source through

Two repositories referenced above. It mostly consists of documented Prolog

for systematic development of new methods of automated and semi-automated

are of practical and engineering nature, but their generalisation may be useful

locally significant information while updating the changed fragments. The results

tion of the existing unidirectional mappings and thus achieving preservation of

mented as traversals of ADT instances operating on the result of the superposi-

until the mapping became bijective. In

bidirectionalised by iteratively adding more information to the transformations

in the future.

One of the open questions left unanswered is dealing with sustainers. However, it is yet unclear how to prove termination

instead of maintainers:

Two practical cases of bidirectionalisation have been described in this abstract.
### 3.1 Problem description

Consider three of them:

- A purely textual flat string-like representation of a program, easy to edit, but it is not apparent until the language instance is processed and turned into a different approach and phases of software language processing feature different entity (such as a parse tree).

- A layoutless abstract representation of a program in such a way that all different approaches and phases of software language processing feature different entity (such as a parse tree).

- An abstract syntax tree, a conceptual representation of a program which is rendered by default.

It lacks certain details specific to different approaches and phases of software language processing feature different entity (such as a parse tree).

There is some structure in such a program, but it is not apparent until the language instance is processed and turned into a different approach and phases of software language processing feature different entity (such as a parse tree).

There are of practical and engineering nature, but their generalisation may be useful for systematic development of new methods of automated and semi-automated program restructuring.

The code has around 3000 line of documented code in Rascal, and is publicly available from a dedicated repository: http://github.com/grammarware/bx-parsing.
depicted on kinds of artefacts, which can be considered to fit into one of twelve categories.

3.1 Problem description

3 (Un)parsing case study

defined mappings \([\text{unparse}]\) that rely on either lexical or syntactic definitions; solid lines denote universally

Fig. 1. Layoutless

erent approaches and phases of software language processing feature di

It lacks certain details specific to

is the most suitable for automated program analysis and assigning semantics.

into a di

but it is not apparent until the language instance is processed and turned

used to for the last six decades. There is some structure in such a program,

transfer and maintain and familiar to what mainstream programmers are

raw (text editing

— an abstract syntax tree, a conceptual representation of a program which

Megamodel of various kinds of parsing and unparsing. Dotted lines denote map-

loops are examples of transformations we consider in this paper.

Textual

structural

and is publicly available from a dedicated repository:

Tokenisation

and

is released as open source through

language for program analysis and transformation. It is released as open source

Local contextual information

presentation of the existing unidirectional mappings and thus achieving preservation of

mented as traversals of ADT instances operating on the result of the superposi-

In

bidirectionalised by iteratively adding more information to the transformations

Two practical cases of bidirectionalisation have been described in this abstract.

4 Conclusion

and is publicly available from a dedicated repository:

Two repositories referenced above. It mostly consists of documented Prolog

for systematic development of new methods of automated and semi-automated

are of practical and engineering nature, but their generalisation may be useful

locally significant information while updating the changed fragments. The results

tion of the existing unidirectional mappings and thus achieving preservation of

composition

in the case of more than two instances (like in

error-correcting strategies. However, it is yet unclear how to prove termination

instead of

instead of maintainers:

One of the open questions left unanswered is dealing with sustainers

The code has around 3000 line of documented code in Rascal

The source code of all discussed prototypes is released as open source through

and is publicly available from a dedicated repository:

4.1 Software ecosystems

Two repositories referenced above. It mostly consists of documented Prolog

for systematic development of new methods of automated and semi-automated

are of practical and engineering nature, but their generalisation may be useful

locally significant information while updating the changed fragments. The results

tion of the existing unidirectional mappings and thus achieving preservation of
Thanks for your attention!

- Bidirectionalisation: pushing FP to BX

- This was Dr. Vadim Zaytsev a.k.a. grammarware
  - grammarware.net, twitter.com/grammarware, grammarware.github.com, …

- Slides are CC-BY-SA: [http://grammarware.net/talks/#TFP2014](http://grammarware.net/talks/#TFP2014)