Bootstrapping, Default Formatting, and Skeleton Editing in the Spoofax Language Workbench

Benchmark Solutions for LWC@SLE 2016

Gabriël Konat Luís Eduardo de Souza Amorim Sebastian Erdweg Eelco Visser

Delft University of Technology

g.d.p.konat@tudelft.nl, l.e.desouzaamorim-1@tudelft.nl, s.t.erdweg@tudelft.nl, e.visser@tudelft.nl

1. Introduction

Language workbenches are tools that help language designers to design and implement (domain-specific) programming languages, aiming to produce a full featured programming environment from a high-level language description. A recent paper, resulting from a series of language workbench challenge workshops, describes a collection of benchmark problems for language workbench research [5]. In this paper, we describe solutions to two of these benchmark problems in the Spoofax Language Workbench [6], i.e. default formatting in Section 3 and skeleton editing in Section 4. In addition, we introduce a new benchmark problem — bootstrapping of meta-languages in a workbench — and describe the support for bootstrapping we developed for Spoofax in Section 2.

2. Bootstrapping

We propose a solution to the problem of bootstrapping the meta-languages of language workbenches [1].

Problem A bootstrapped compiler is a compiler that can compile its own source code, because the compiler is written in the compiled language itself. It is common practice to bootstrap the compiler of a general-purpose language, such as the GCC compiler for the C language.

Language workbenches provide high-level meta-languages for defining the compilers and environments of domain-specific languages (DSLs). Thus, users of a language workbench implement the compiler of their DSL $\texttt{L}_\texttt{c}$ not in `L`, but in a high-level (domain-specific) meta-language `M` instead. Therefore, bootstrapping of $\texttt{L}_\texttt{c}$ is no longer necessary, which is good since many DSLs have limited expressiveness and are often ill-suited for compiler development.

What we desire instead is bootstrapping of a language workbench's meta-language compiler $\texttt{M}_\texttt{c}$. We want to use our meta-languages for implementing our meta-language compilers.

[1] A full paper about the problem and our solution to bootstrapping meta-languages of language workbenches is under submission

Bootstrapping in language workbenches poses three challenges:

1. Most language workbenches provide separate meta-languages for describing the different language aspects such as syntax, analysis, code generation, and editor support. Thus, to build the definition of one meta-language compiler, multiple meta-language compilers are necessary. This entails intricate dependencies that bootstrapping needs to handle.
2. Most language workbenches provide an integrated development environment (IDE) in which programs and languages can be interactively developed. Therefore, to be able to bootstrap meta-language compilers in the IDE environment, interactive bootstrapping is required [8]. Importantly, since meta-language changes can be defective, it also needs to be possible to roll back to a working version of the meta-language if bootstrapping fails.
3. Since meta-languages in language workbenches depend on one another, it can become difficult to implement breaking changes that require the simultaneous modification of a meta-language and existing client code. To be able to evolve meta-languages, bootstrapping breaking changes should be supported.

Note that this is a problem at the meta-level. That is, it is a problem of the meta-language engineer, or language workbench engineer, *not* the language engineer that creates DSLs with the language workbench.

Assumptions We assume that the language workbench runtime and environment are deterministic, and that all meta-language compilers are deterministic. If this is not the case, bootstrapping may never terminate.

We also assume that an existing baseline for the meta-language compilers exist, to kickstart the bootstrapping process. Without an existing (manually constructed) baseline, meta-language compilers cannot be implemented in itself.

Finally, we assume that the meta-languages are defined in Spoofax.

Implementation An implementation requires a method for *sound bootstrapping*, *interactive bootstrapping*, and *bootstrapping of breaking changes*.

In general, there is no way to know how many iterations are necessary until a defect materializes or after how many iterations it is safe to stop. Therefore, for sound bootstrapping it is required to iterate until reaching a fixpoint, that is, until the build stabilizes. Fixpoint bootstrapping applies meta-language compilers to a meta-language definition in iterations. To determine if a fixpoint has been reached, we must be able to compare the binaries that meta-languages generate. We have reached a fixpoint if the generated binaries in iteration $k+1$ are identical to the binaries generated in iteration k.

Besides having a bootstrapping system that satisfies the requirements above, we also need to support bootstrapping in the interactive environments of language workbenches. In particular, an interactive environment needs to provide operations that (1) start a bootstrapping attempt, (2) load a new baseline into the environment after bootstrapping succeeded, (3) roll back to an existing baseline after bootstrapping failed, and (4) cancel non-terminating bootstrapping attempts.

All operations should work within the same language workbench environment, without requiring a restart of the environment, or a new environment to be started. This requires *dynamic (re)loading* of the meta-language compilers into the running language workbench environment. Rolling back defective changes requires *versioning* of meta-languages and their dependencies, such that a rollback to a previous working version can occur.

Bootstrapping helps to detect changes that break a language implementation. However, sometimes breaking changes are desirable, for example, to change the syntax of a meta-language. If we change the syntax definition of some language `M` and the code written in `M` simultaneously, bootstrapping fails to parse the source code in a later iteration because the baseline only supports the old syntax of `M`. The bootstrapping environment should provide operations for bootstrapping breaking changes, which also require dynamic loading and versioning.

Spoofax supports bootstrapping of meta-languages with dynamic loading, dependencies between meta-languages, versioning, and a fixpoint bootstrapping implementation. Meta-languages can be bootstrapped interactively in the Spoofax Eclipse plugin. More details about the implementation can be found in the artifact.

Usability The user interface for bootstrapping is shown in Figure 1.

A meta-language engineer can import meta-language definitions into the Eclipse environment, make changes to the definitions, and run bootstrapping operations on the definitions to produce new baselines. The Eclipse console displays information about the bootstrapping process, e.g. when a new iteration starts, which artifacts were different during language product comparison, and any errors that occur during bootstrapping.

When bootstrapping fails, changes are reverted, and the console shows observed errors. Bootstrapping can also be cancelled by cancelling the bootstrapping job in the Eclipse progress view. When bootstrapping succeeds, the new baseline and meta-language definitions are dynamically loaded, such that the meta-language engineer can start making changes to the definitions and run new bootstrapping operations.

Impact If there is no existing baseline for the meta-languages yet, this has to be created. If a meta-language compiler is not deterministic, it has to be changed such that it is. However, no other artifacts have to be changed in order to perform bootstrapping operations. Therefore, the solution is very modular; any Spoofax meta-language with a deterministic compiler and an existing baseline can be bootstrapped.

Composability The bootstrapping solution composes well with the rest of the language workbench, because fixpoint bootstrapping reuses the existing compilation and dynamic loading infrastructure of Spoofax, and plugs into Spoofax via a context menu. Other parts of the language workbench are not affected by bootstrapping. It composes well with the other benchmark problems, because none of the other benchmarking problems are about compilation or dynamic loading. Furthermore, any meta-language defined in Spoofax can be bootstrapped, because the support for bootstrapping is general.

Limitations One limitation of fixpoint bootstrapping is that it can be slow. Many meta-language definitions are compiled with many meta-language compilers, multiple times until a fixpoint is reached. For example, bootstrapping Spoofax's meta-languages, depending on the kind of change, can take 10 to 20 minutes.

Uses and Examples Bootstrapping yields four main advantages:

1. A bootstrapped compiler can be written in the compiled high-level language,
2. it provides a large-scale test case for detecting defects in the compiler and the compiled language,
3. it shows that the language's coverage is sufficient to implement itself, and
4. compiler improvements such as better static analysis or the generation of faster code applies to all compiled programs, including the compiler itself.

Most general-purpose languages, such as C/C++ [11, 1], Java [2], and C# [10], are bootstrapped because of these advantages.

Meta-languages can benefit from the same advantages when bootstrapped. For example, the Stratego meta-language is bootstrapped [15], providing a significant test case for the

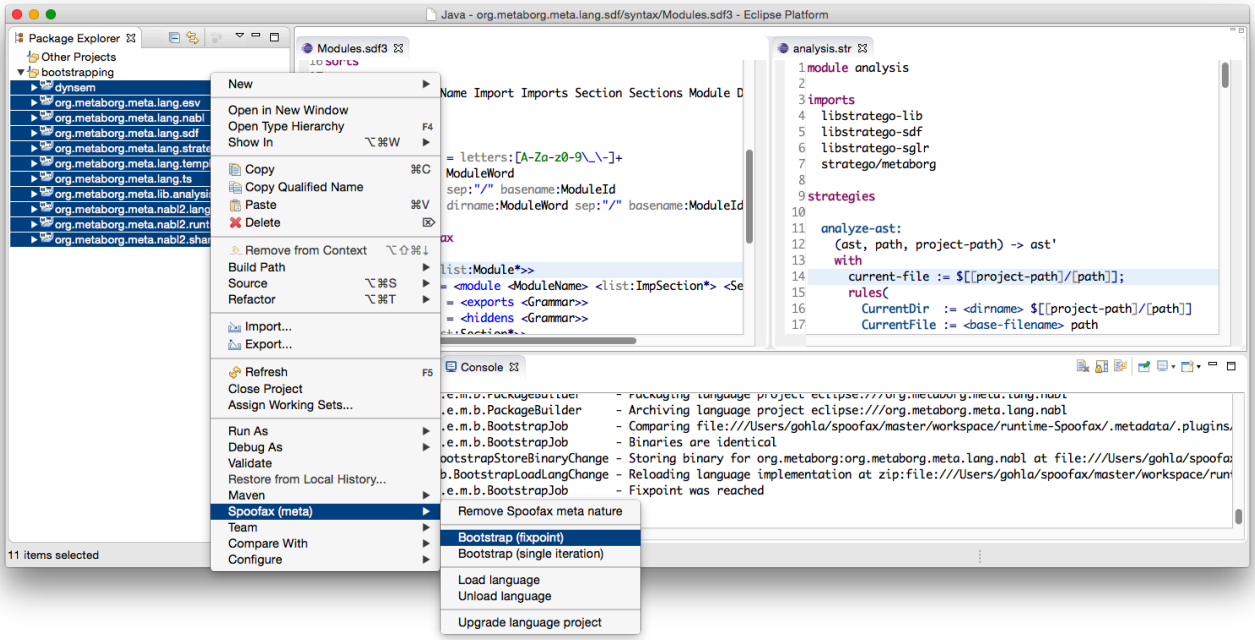

Figure 1. The Spoofax Eclipse plugin with support for bootstrapping. Meta-language definitions are imported as projects on the left, which are developed interactively in the (SDF and Stratego) editors on the right. Once the meta-language engineer is done changing the meta-languages, all meta-language projects are selected and 'Bootstrap (fixpoint)' is selected to start a fixpoint bootstrapping operation. Bootstrapping with a single iteration for breaking changes is also supported. The console on the bottom displays the progress of the bootstrapping operation, in this case that (after several iterations) a fixpoint was reached.

Stratego language and compiler. Recently, the Rascal compiler has been bootstrapped [3], simplifying the development of the compiler because of higher-level specification.

However, support for sound bootstrapping in language workbenches is limited, typically lacking a solution for fixpoint bootstrapping, interactive bootstrapping, and rollbacks. For example, MPS [17] does not support versioning or undoing changes, making rollbacks impossible if defects are introduced. Ensō [9, 14] and Rascal [7] do not provide a general solution for fixpoint bootstrapping. Older versions of Spoofax bootstrapped a meta-language by a single compilation step, which is not sound because it does not uncover defects that occur after several fixpoint iterations.

We think that meta-languages should be bootstrapped to benefit from the advantages listed above, and that meta-languages should be bootstrapped in a sound way to ensure that all defects of a meta-language are found.

Effort The effort of creating an existing baseline ranges from minutes to months. If a meta-language is already bootstrapped, it only requires minutes to manually construct a baseline for that meta-language. However, if a meta-language is not already bootstrapped, the meta-language has to be completely reimplemented in itself, which can take months. In the case of Spoofax, the meta-languages were already bootstrapped, it only took a day to construct a baseline for all the meta-languages.

The effort for making a meta-language compiler deterministic can also vary a lot. In the case of Spoofax, the meta-languages were not deterministic because name generation was not deterministic. It took us two days of debugging and refactoring to make the compilers completely deterministic.

Other than that, no effort is required to start bootstrapping a meta-language.

Artifact We have evaluated our bootstrapping solution by applying it to eight of Spoofax's meta-languages against seven changes. The Spoofax bootstrapping repository `https://github.com/spoofax-bootstrapping/bootstrapping` contains the evaluation of the bootstrapping solution, and links to the source code implementing fixpoint bootstrapping. To try out bootstrapping in Spoofax, the nightly version of Spoofax is required, which can be downloaded from our website at `http://metaborg.org/en/latest/source/release/note/nightly.html`.

3. Specification of Default Formatting

Syntax definition formalisms allow language designers to specify a language's syntax. From this specification they generate parsers that construct an abstract representation from a given program in that language. However, in most

cases the inverse is also necessary, as part of language development involves pretty-printing, i.e. presenting abstract representations back to the user in some visual form. Thus, parsers and pretty-printers are important elements to consider when defining a language.

In our solution to implement a default formatter/pretty-printer we use SDF3 [16]. SDF3 is a syntax definition formalism that combines the specification of a parser and a pretty-printer in the same formalism. Production rules in SDF3 can be written as templates, specifying not only the structure of elements in a language but also retaining layout information necessary to define the concrete form of such elements.

Assumptions We assume a language specification written in SDF3.

Implementation We derive a formatter based on the SDF3 definition by generating rewrite rules responsible for traversing the abstract syntax tree and produce the concrete representation for its elements. In our implementation, this traversal produce elements in the Box language [13], as this language provides more flexibility on how to format terms, e.g. considering horizontal and/or vertical alignment and indentation.

Templates in SDF3 consist of a list of lines that might contain literal strings, placeholders or layout as its elements. To generate a pretty-printer that produces formatted code in the box language, each line in a template produces an horizontal box (*H*). We construct this box by iterating over the elements of a template line. Literal or layout elements produce string boxes (*S*), whereas a placeholder element recursively constructs the boxes necessary to pretty-print the top-level term.

Figure 2 shows an example of a pretty-printer generated from a small syntax definition. Note that we generate vertical boxes (*V*) for elements in lists that are separated by newlines. Note also that elements of the list are indented according to the layout defined in the template. The final result is wrapped in a vertical box corresponding containing all the lines of the complete program. To produce the formatted program, we convert this top-level box to a string.

Usability To define a default formatter from the language specification, language engineers can write template productions and edit its layout. For regular context-free productions, the generated formatter separates symbols by a single whitespace. The user can apply the formatter via a menu in the editor.

Figure 3 illustrates an example of formatting code using the formatter derived from SDF3. In the figure, on the left we see the SDF3 definition for the QL language using template productions, in the middle, a program containing unformatted elements and the same program formatted from applying the pretty-printer generated from SDF3. Finally, on the right, the abstract syntax tree of the same program.

Impact Template productions in SDF3 only retain layout for pretty-printing, thus, they do not affect other aspects of the syntax definition such as parser generation. The language engineer must define its productions in the grammars as templates, and include the desired layout in each production. However, the impact is still reasonably low, as it is not necessary to change anything other than the syntax definition.

Uses and Examples All mainstream IDEs provide support for formatting a program, thus, language workbenches aim to somehow generate such tools from the language definition. In our solution, we integrate the definition of parsing and pretty-printing into a single formalism. Other syntax definition formalisms such as SYM [4] and Extended SDF [12] also unify the language specification to produce a parser and a pretty-printer. However, our approach avoids redundancy of operators, as the specification of layout is explicit in SDF3 templates, and the language engineer does not need to use special operators to specify layout and formatting like in other formalisms.

Composability The generated pretty-printer is a separate artifact generated from the language specification and can be reused for other purposes, e.g., when pretty-printing an abstract representation of a code template inside a completion proposal. Furthermore, language projects can depend on external pretty-printers, using them to produce formatted code when generating code in another language, for example.

Limitations Currently, the default formatter generated from SDF3 is not comment-preserving. Furthermore, when writing template productions users might need to specify how to tokenize lexical elements inside a template. SDF3 allows the specification of tokenize rules separately, and they affect all template productions inside a file.

Effort The default formatter is automatically derived from SDF3 syntax definitions, thus, no extra effort is required from the user. In spite of that, it is necessary that the language designer integrates the desired layout into the syntax definition writing grammar rules as template productions, otherwise, the default formatter separates elements by a single whitespace.

Artifact SDF3 is part of Spoofax Language Workbench and can be downloaded at our website.

4. Skeleton Editing

Skeleton editing helps new users to discover language features and increases coding efficiency as the user can insert larger pieces of code at once. In our solution, we implement skeleton editing through syntactic code completion. We automatically derive proposals as templates from the the grammar and expand such templates to construct larger skeletons.

Templates may contain placeholders to define points to further expand the program. We construct a placeholder for each non-terminal in the syntax definition and make them part of the language. We derive placeholder expansions from

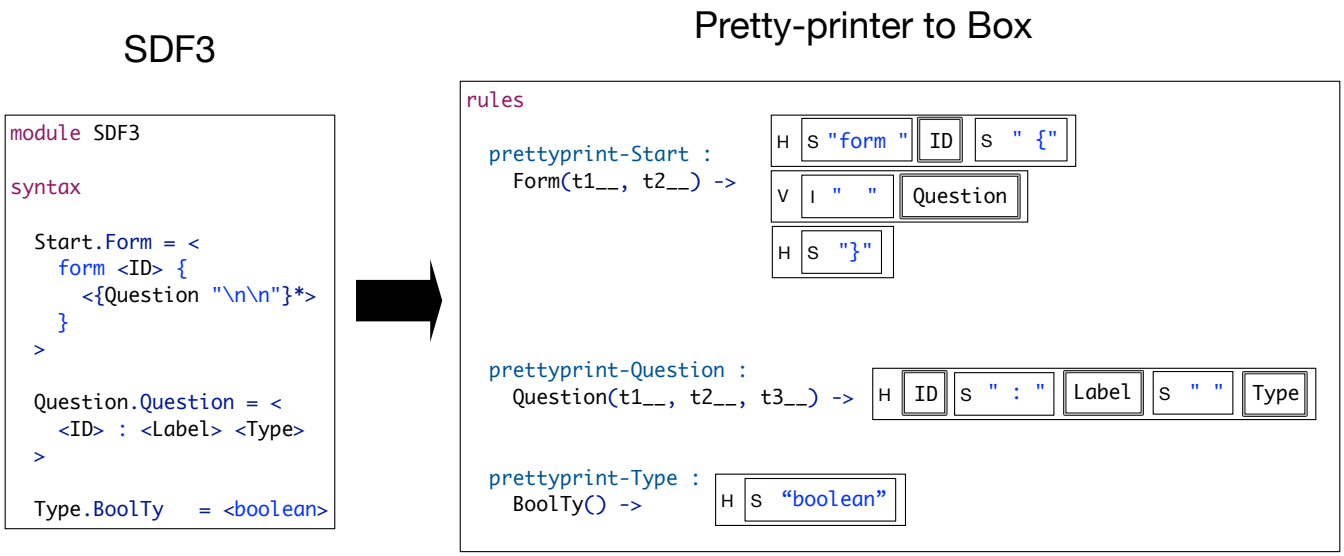

Figure 2. Deriving pretty-print rules from SDF3 templates.

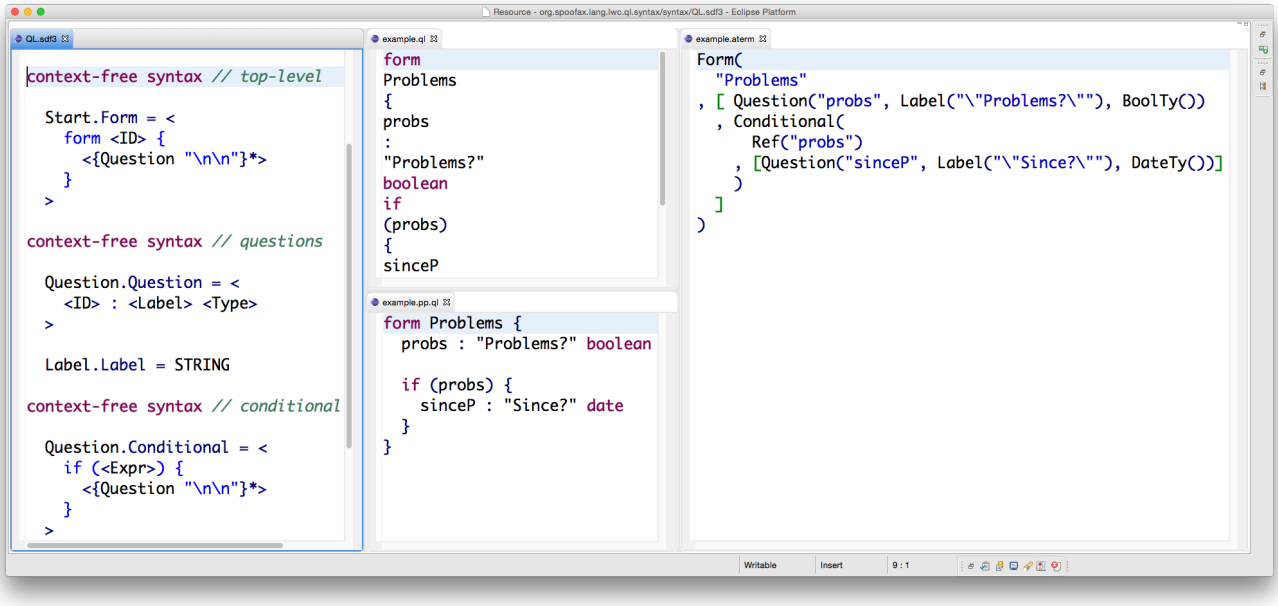

Figure 3. Spoofax Environment containing syntax definition, unformatted program, formatted program and abstract syntax tree.

the productions of the placeholder's non-terminal which guarantees that selecting an expansion results in a correct program. Our solution[2] is based on definitions written in SDF3 and built inside Spoofax. We reuse the SDF3 generated pretty-printer to properly format proposals.

Assumptions We assume a language definition written in SDF3.

Implementation Our solution derives syntactic code completion from a syntax definition in SDF3. First of all, we extend the syntax definition with rules for placeholders that can explicitly appear in the program or in a completion proposal. The language engineer can define the characters of a placeholder in a configuration file so placeholders do not clash with elements of the language. For example, the language engineer can write:

```
placeholder:
  prefix : "#"
  suffix : "#"
```

and all placeholders would be of the form #NAME#, with #NAME# being the name of the placeholder. Hence, for each non-terminal in the language, we generate a placeholder of that non-terminal, allowing an explicit placeholder to appear wherever the non-terminal can appear in the program.

Next, we produce all terms that a placeholder can expand to. Each grammar rule with a constructor defines a placeholder expansion. We encode placeholder expansions as rewrite rules, and apply these rules whenever code completion is triggered when the cursor is in a placeholder node.

In Figure 4 we show the template rules for allowing explicit placeholders in a program and the rewrite rules, both automatically generated from a definition written in SDF3. Note that we only specified a single character (#) that prefixes the placeholder name, thus, our placeholders are of the form #NAME.

We use origin tracking to identify whether a placeholder node contains the cursor. If it does, we show all placeholder expansions formatted according to the grammar as completion proposals. Completing the program, i.e. selecting a proposal, replaces the placeholder by its formatted expansion that may or may not contain more placeholders. Users construct larger skeletons by navigating through placeholders and selecting one of its expansions or finally replacing a placeholder textually.

Variants Our approach also supports expanding a program in cases where the cursor is not in a placeholder node. We can also expand programs by expanding a recursive structure or adding an optional element that is not part of the program. By using the respective non-terminal that corresponds to such terms, we inject a selected proposal that e.g., adds an element to a list, or expands an expression term to another one,

[2] A full paper named *Principled Syntactic Code Completion using Placeholders* is under submission at SLE'16.

```
module QL

context-free syntax //regular production rules

  Start.Form = <
    form <ID> {
      <{Question "\n\n"}*>
    }
  >

  Question.Question = <
    <ID> : <Label> <Type>
  >

  Type.BoolTy   = <boolean>
```

```
context-free syntax //derived rules for placeholders

  Start.Start-Plhdr       = <#Start>
  Question.Question-Plhdr = <#Question>
  ID.ID-Plhdr             = <#ID>
  Label.Label-Plhdr       = <#Label>
  Type.Type-Plhdr         = <#Type>
```

```
rules  //rewrite rules for placeholder expansions

  rewrite-placeholder:
    Start-Plhdr()    -> Form(ID-Plhdr(), [])

  rewrite-placeholder:
    Question-Plhdr() -> Question(ID-Plhdr()
                                 , Label-Plhdr()
                                 , Type-Plhdr())

  rewrite-placeholder:
    Type-Plhdr()     -> BoolTy()
```

Figure 4. Generating syntactic code completion from an language definition written in SDF3.

keeping the original expression and using placeholders for the inserted terms. To expand the program in such way, whenever the cursor is in the surrounding layout of a recursive structure or an optional term, we calculate the expansions according to the grammar, and inject the selected proposal in the program. Skeleton editing continues from there, by either expanding explicit placeholders or injecting placeholder expansions into recursive or optional nodes.

Usability The user construct skeletons of code by iteratively navigating through placeholders and selecting a proposal to expand the program. After selecting a proposal the developer can expand or textually replace placeholders inserted by the proposal. Users can also navigate to a part of the program that does not have a placeholder but a recursive or an optional node, and expand the program by variant described before.

Figure 5 shows some of the iterations to construct the skeleton and eventually the final program containing a question and a conditional, as the user textually replace placeholders for lexical symbols in the skeleton.

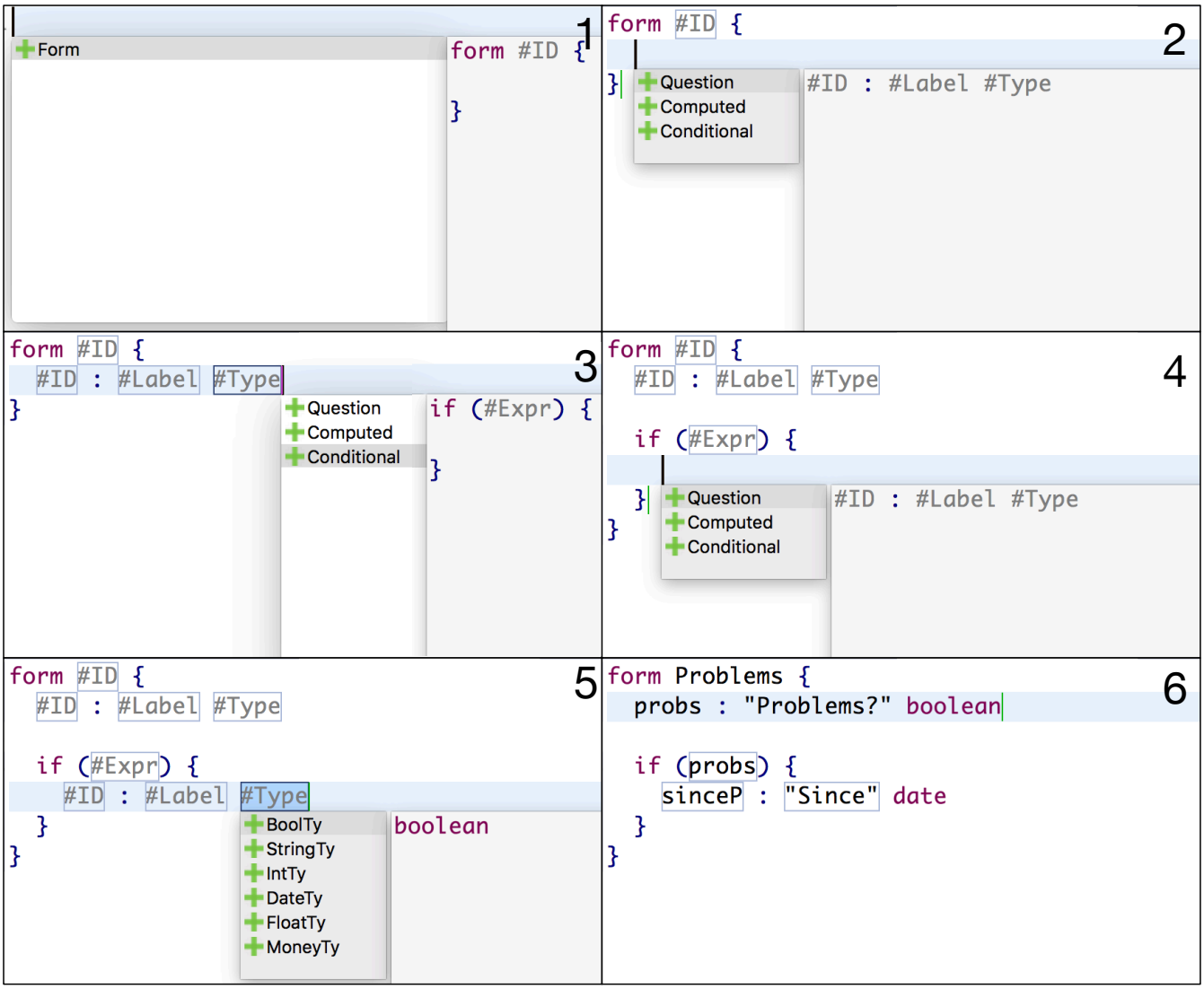

Figure 5. Constructing a program step by step by syntactic code completion.

Uses and Examples Existing IDEs and language workbenches often propose erroneous code templates that yield syntax errors when inserted. For example, Eclipse largely ignores the syntactic context at the cursor position and proposes the insertion of an *else*-block without a corresponding *if*-statement, yielding a syntax error after insertion as we can see in Figure 6.

Furthermore, the set of code templates available to the user in IDEs is limited, and it is not possible to construct a more complex skeleton by only using code completion. In our implementation, code completion is sound and complete, i.e., completing a program does not introduce syntax errors and it is possible to construct any program by only triggering syntactic code completion, only manually rewriting placeholders for lexical elements.

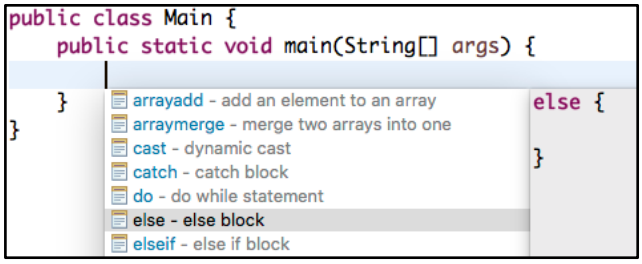

Figure 6. Eclipse: Unsound completion yields syntax error.

Impact and Composability Our solution depends on the language definition as we extend the syntax with extra productions to parse explicit placeholders. We also need a mechanism to format code templates, so it is necessary to change

the language pretty-printer. However, since we implement our solution in SDF3, parser and pretty-printer are specified together and a formatter is automatically derived. Another important point to note is that our solution allows editing incomplete programs (containing explicit placeholders) in a textual editor without having syntax errors on incomplete structures.

Limitations Currently placeholder expansions only contain syntactic elements. Moreover, since we do not consider the language's semantics, even though expansions are syntactically correct, they might produce programs with type errors. Another limitation is that code templates might contain only small structures, therefore it might take a large number of iterations to generate more complex skeletons.

Effort No additional effort on the user nor language engineer is necessary as syntactic code completion is automatically derived from SDF3 syntax definitions. However, one should consider the effort of specifying the concrete syntax of the language in SDF3 templates to produce placeholder expansions that are properly formatted.

Artifact Our solution is available in the latest nightly version of the Spoofax. The project containing an implementation for the questionnaire language and the examples shown in this paper is available at `https://github.com/MetaBorgCube/metaborg-ql/tree/lwb-2016`.

References

[1] GCC, the GNU compiler collection. `https://gcc.gnu.org/`. [Online; accessed 28-July-2016].

[2] JDT core. `http://www.eclipse.org/jdt/core/`. [Online; accessed 28-July-2016].

[3] Rascal compiler is bootstrapped! `https://twitter.com/jurgenvinju/status/651786592521224196`. [Online; accessed 28-July-2016].

[4] R. Boulton. *Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing*. Number 390. University of Cambridge, Computer Laboratory, 1996.

[5] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly 0001, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[6] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[7] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society, 2009.

[8] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Towards live language development. In *Proceedings of Workshop on Live Programming Systems (LIVE)*, 2016.

[9] Alex Loh, Tijs van der Storm, and William R. Cook. Managed data: modular strategies for data abstraction. In Gary T. Leavens and Jonathan Edwards, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, pages 179–194. ACM, 2012.

[10] Microsoft. .NET compiler platform ("Roslyn"). `https://roslyn.codeplex.com/`, 2014. [Online; accessed 28-July-2016].

[11] University of Illinois. clang: a C language family frontend for LLVM. `http://clang.llvm.org/`. [Online; accessed 28-July-2016].

[12] N. Pouillard. Extending SDF. Technical Report 0407, EPITA, jul 2004.

[13] Mark G. J. van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering Methodology*, 5(1):1–41, 1996.

[14] Tijs van der Storm, William R. Cook, and Alex Loh. Object grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2012.

[15] Eelco Visser. A bootstrapped compiler for strategies (extended abstract). In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 73–83, Trento, Italy, July 5 1999.

[16] Tobi Vollebregt. Declarative specification of template-based textual editors. Master's thesis, Delft University of Technology, Delft, The Netherlands, April 2012.

[17] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 41–61. Springer, 2014.