Injecting language workbench technology into mainstream languages

Michael Ballantyne
PLT at Northeastern University, Boston, MA, United States

Matthias Felleisen
PLT at Northeastern University, Boston, MA, United States

Abstract

Eelco Visser envisioned a future where DSLs become a commonplace abstraction in software development. He took strides towards implementing this vision with the Spoofax language workbench. However, his vision is far from the mainstream of programming today. How will the many mainstream programmers encounter and adopt language workbench technology? We propose that the macro systems found in emerging industrial languages open a path towards delivering language workbenches as easy-to-adopt libraries. To develop the idea, we sketch an implementation of a language workbench as a macro-library atop Racket and identify the key features of the macro system needed to enable this evolution path.

2012 ACM Subject Classification Software and its engineering → Macro languages; Software and its engineering → Domain specific languages; Software and its engineering → Development frameworks and environments

Keywords and phrases Language workbenches, macro systems, language adoption

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.10

Funding This work was partially supported by NSF grants SHF 2007686 and 1823244.

Acknowledgements The authors are grateful for technical feedback from Michael Delmonaco, Siddhartha Kasivajhula, and Alexis King. Also thanks to William Byrd and Siddhartha Kasivajhula for helpful comments on early drafts, and to Shriram Krishnamurthi and Ryan Culpepper for rubbing Matthias's nose in Spoofax and language workbench ideas.

1 Bringing Eelco Visser's legacy to the mainstream

Eelco Visser envisioned a future where creating a new programming language becomes a common means of abstraction. One of his well-known examples is that of a web programming DSL. In WebDSL [24], a programmer specifies only the data model, page flow, and page layouts. The DSL compiler generates all operational elements such as request handlers.

In order to implement such DSLs, Visser sought to provide programmers with a "language designer's workbench". A programmer specifies a DSL in a declarative manner, and the workbench derives compilers, IDEs, verifiers, and documentation [25]. The Spoofax Language Workbench [18] represents Visser's last step in a long chain of research accomplishments towards this vision.

Sadly, Visser did not see his vision spread to the mainstream of software development. We hypothesize one contributing cause. Spoofax requires teams to adopt language-building wholesale or not at all. A programmer must cease to be a Java programmer, and instead become a Spoofax programmer. Worse, the programmer's teammates must use a new IDE, a new build system, and possibly other tools.

We share Visser's broad goal and call it language-oriented programming [12,27]. In contrast to the vision behind Spoofax, our working hypothesis calls for an incremental evolution path from the current state of affairs to one where developers embrace the construction of DSLs.

Programming technologies created by researchers most often reach the mainstream when they are adopted by existing industrial languages. Hence to achieve mainstream adoption, language-building technology must be integrated into existing language ecosystems.

This paper sketches one feasible evolution path by which mainstream programmers and languages can integrate language workbench ideas. This evolution requires removing hurdles to adoption at two levels. The first concerns friction at the moment an individual programmer chooses to use or create a DSL. If adopting a DSL or a language workbench requires installing and using new tools, the programmer may find the cost too high. Instead, a programmer should be able to begin using a DSL or creating a DSL as easily as using or creating a library. In a typical programming ecosystem, only the language itself is universally available to all programmers. Different programmers may use different IDEs and build tools. Thus to ensure any programmer can easily begin using and making DSLs, the language workbench must be made available as part of the language itself.

The second hurdle thus concerns the integration of DSL-building tools into a language. The creators of existing industrial languages are unlikely to integrate and standardize complex language workbench features before they are widely used. However, a number of widely-used languages including Rust, Scala, and Julia are extensible via macro systems. If language workbench features can be built as macro-libraries, they can be used as part of the language without being built-in to the core.

For the past year, we have investigated this path by building a language workbench as a macro library in Racket, a general-purpose language with a sophisticated macro system. In the process we have identified the essential macro system features needed to support a language workbench macro-library and those features that are superfluous. We anticipate that this experience can guide the design of extensions to macro systems in industrial languages and thus the development of language workbenches as libraries.

Our initial prototype realizes a part of Visser's ideal language designer's workbench. It accounts for only some aspects of language specification, and it does not yet generate rich IDE services or verification infrastructure. We expect that further development can close these gaps and make a great many of the technologies pioneered by Visser accessible in existing languages. However, some tradeoffs are fundamental to the approach. Our language workbench libraries give up generality in order to specialize to the conventions of their host. We also focus specifically on the case of DSLs intended for professional software engineers, whereas language workbenches are also used to create DSLs for domain experts and end-users.

The remainder of the paper proceeds as follows: Section 2 illustrates our objective by showing the experience of using and defining DSLs as libraries. Section 3 describes the implementation of the meta-DSL and the set of macro system features that are and are not needed to support it. Section 4 contrasts our work to related approaches, and suggests further ideas that we may adapt from language workbenches in the future. Section 5 anticipates the issues that need to be addressed to create language workbench macro-libraries for other host languages. The last section provides a summary and an outlook.

2 An integrated language workbench

The best way to understand how developers should be able to use workbenches inside their ecosystem is to work through an example. Consider a programmer who is implementing a user interface that loads and displays a CSV file from a remote server. The controller for the UI component is easily understood as a state machine. The states include the initial state, before a URL has been entered; the loading state; and the final state where the table

```
#lang racket

(require state-machine)

;; GUI elements and data processing code in Racket (simplified)
(define table (new list-box%))
(define url-field (new text-field%))
(define load-button
  (new button% [callback (lambda _ (send csv-controller load-click))]))
(define (load-data url)
  ;; elided code to load CSV from the URL
  (send csv-controller loaded data))

;; Controller
(define csv-controller
  (machine
   #:initial-state no-data
   (state no-data
     (on-enter (set-display url-message)))
   (state loading
     (on-enter (set-display loading-message)
               (load-data (send url-field get-value)))
     (on (loaded data)
       (set-data data)
       (-> display)))
   (state display
     (on-enter (set-display table)))
   (on (load-click)
     (-> loading))))
```

Figure 1 CSV browser UI with a controller implemented using a state machine DSL

is displayed. Asynchronous user actions and network events drive the transitions between states.

In a conventional programming language the programmer must map these concepts to language elements such as classes and methods. The programmer might create a class representing the machine, an interface for states including methods for each transition, and classes implementing the interface for each state. The original machine structure easily becomes swamped by implementation details, and it may be difficult to decipher when returning to the code sometime later.

2.1 Using a DSL as a library

In contrast, writing the controller using a state machine DSL directly expresses the structure the programmer has in mind. Figure 1 shows a program using a state machine DSL implemented with our language-workbench library. It uses the DSL alongside general purpose GUI and data processing code written in Racket. The library import `(require state-machine)` makes the DSL available in the module. Subexpressions written in the host language (here

Racket, displayed in lighter font) integrate the state machine with the GUI code. The state machine DSL's compiler generates code that uses Racket's object-oriented programming facilities to implement the state machine with classes for states and methods for transitions. Racket code that triggers machine transitions interacts with the controller as an object using Racket's method call form `send`. Thus the generated implementation is much the same as the programmer would write without access to a DSL, but the mapping from domain concepts to host-language constructs is encapsulated in the DSL implementation.

The ease with which a programmer can integrate this DSL code is essential to its value. A programmer is likely to use this approach if it merely means importing a library. However, a programmer is unlikely to find the benefit sufficient if it requires a major change in workflow such as a new IDE or build system component. Thus a platform for programming with DSLs should minimize the cost of integrating new languages into a program. Racket's "languages as libraries" [22] and SugarJ's "sugar libraries" [11] realize this goal.

2.2 Implementing a DSL as a library

Of course, someone must first have done the work to implement the state machine language. The meta-DSLs offered by language workbenches promise to simplify this task. Our meta-DSL, integrated with Racket, aims to provide the advantages of a language workbench without requiring a programmer to leave the familiar host language. Each aspect of our meta-DSL design aims at making it easy for programmers familiar with Racket to understand and adopt.

Figure 2 shows how the state machine language is defined in our meta-DSL.[1] Like other DSLs in Racket, the meta-DSL is a "language as a library", so it is used in a normal Racket module via the `(require syntax-spec)` declaration. Similarly, the `machine` syntax is exported with `provide` in the same way as any normal function or value definition. Thus implementing a DSL involves no more friction than implementing a library.

The state machine language definition specifies syntax via grammar nonterminals and name binding via binding rules associated with nonterminal productions. The definition of `machine` establishes the interface between the state machine DSL and the host language. It includes a call to a back-end compiler for the DSL, `compile-machine`, which is implemented in compile-time Racket code.

We choose to account for syntax, binding rules, and the interface with the host language in our meta-DSL. These portions of a language implementation are the most uniform across DSLs, because their structure relates as closely to the host language as to the domain of the language being defined. This connection also means that these elements require the most intricate interaction with the host language implementation. The underlying implementation must interact with the host's data representations of syntax, scope, and binding environments. In an extensible language such as Racket, the processes of parsing and name resolution interleave in complex ways. Thus a DSL implementation must perform operations in the correct order to ensure that data is available when needed. Hiding these elements via the meta-DSL means that programmers need not be aware of these effectful operational details. We leave programmers to implement the back-end compilation to Racket using conventional procedural Racket code augmented by the existing `syntax-parse` pattern matching and templating DSL [8].

[1] The meta-DSL is available at `https://github.com/michaelballantyne/syntax-spec`, and the state machine example at `https://github.com/michaelballantyne/syntax-spec/tree/main/demos/visser-symposium`.

```
#lang racket

(provide machine)
(require syntax-spec)

(syntax-spec
  (binding-class state-name)

  (host-interface/expression
    (machine #:initial-state s:state-name d:machine-decl ...)
    #:binding {(recursive d) s}
    (compile-machine #'s #'(d ...)))

  (nonterminal/two-pass machine-decl
    (state n:state-name
      e:event-decl ...)
    #:binding (export n)
    e:event-decl)

  (nonterminal event-decl
    (on-enter e:racket-expr ...)
    (on (evt:id arg:racket-var ...)
      e:racket-expr ...
      ((~datum ->) s:state-name))
    #:binding {(bind arg) e}))
```

Figure 2 Syntax and binding rules declaration for the state machine DSL

Our meta-DSL is optimized for specifying syntaxes and binding structures similar to those of Racket itself. This restriction helps programmers understand programs in the meta-DSL because of the programmers' knowledge of the host language. Furthermore, restricting the expressivity of the meta-DSL means that the DSLs created with it share a common structure, so users of a collection of such DSLs may develop transferrable intuitions. Following this design idea, DSLs created in our framework re-use the S-expression syntax of Racket. They feature tree-structured scope and binding, with a two-pass operational model of name analysis and macro expansion. The DSL front-end defined in the meta-DSL checks the same degree of static semantics as in Racket: syntax and name binding. Any other static semantics must be checked in subsequent passes of the DSL compiler.

3 A language workbench as a macro-library

Our meta-DSL is implemented via procedural macros that transform the specification of a DSL into macros that process DSL programs. Concretely, the `syntax-spec` macro generates compile-time datatypes and analysis functions corresponding to binding categories and nonterminals. The analysis functions traverse the DSL syntax and create a representation of scopes and name bindings. The `host-interface/expression` syntax generates a macro that serves as the entry point into a DSL implementation. The state machine language's `machine` is an example of such a macro. The generated macro calls the DSL's analysis functions. It

then invokes the programmer-defined DSL compiler on the result of this analysis to obtain host language code, which the macro returns as its expansion.

3.1 Essential macro system features

Our approach relies on a host language with a procedural macro system sporting a basic set of features:

- Macros consume a data representation that is flexible enough to support DSL syntaxes.
- Macros are capable of generating further procedural macros and other compile-time code.
- Macros can generate fresh names and names that reliably reference exports of a given library.

A variety of languages have macro systems that meet this most basic set of requirements: Common Lisp, R6RS Scheme, Clojure, Rust, Scala, Template Haskell, Julia, and Elixir.

Less commonly available macro system features are required to support other aspects of language workbenches. To allow fine-grained intermixing of host-language code within DSL syntax, the host macro system needs to come with a reflective API. The analysis function for a DSL nonterminal must be able to (1) determine whether a name has a meaning established in the surrounding context; (2) create an extended binding environment with entries for new names; and (3) analyze a host language subexpression in such an extended environment. In Racket, definition contexts, `local-expand`, and manipulation of scope sets provide these capabilities [15, 16]. Our implementation builds on top of an API that provides higher-level abstractions over these concepts [2].

Finally, to allow parts of DSL programs to be written in separately compiled modules, the host macro system must provide a means to persist a compile-time environment across separate compilations. Racket's notion of "visits" allows macros to leave behind expressions that are re-evaluated every time one module is loaded to support the analysis of another [14]. Such expressions may construct compile-time tables associating name bindings with arbitrary values.

3.2 The full power of Racket's macros is not needed

Interestingly, our experience suggests that some of the most-heralded features of Racket's macro system are *not* essential to support a language-workbench library.

The binding specifications provided as part of nonterminal definitions would allow the analysis functions to implement name hygiene on top of macro systems with relatively naive macro hygiene. The form of hygiene available in Clojure, for example, would suffice. The more sophisticated macro hygiene found in Scheme and Racket that infers the intended scoping structure from the expansion of a macro is unnecessary.

Similarly, the careful phase-separation and separate compilation guarantees provided by Racket's macro system are essentially unnecessary. The meta-DSL implementation handles the symbol table operations performed during analysis. Because all the needed side-effects occur within the meta-DSL implementation, there is less opportunity for programmer error in DSL implementations to create effect dependencies that break separate compilation.

Even though the reflective APIs discussed above are critical to support our meta-DSL, they are only needed in restricted form. For example, DSL analysis functions need to invoke the host analyzer on subexpressions but do not need to examine the results; opaque values would suffice. Likewise, the analysis functions need a way of associating compile-time information with names, but this could take a simpler form than Racket's support for visit-time evaluation with managed side-effects.

4 Related approaches

4.1 Internal and embedded DSLs

DSLs created with our language-workbench library might at first appear similar to internal or embedded DSLs. However, both of those terms tend to refer to DSLs where the DSL code not only appears within host-language syntax but is encoded in the host language syntax and semantics. For example, the syntax of an embedded state machine DSL might consist of host-language function calls to methods with names such as "state" and "on-event". Embedded DSL semantics are defined in one of two ways [17]. In *shallow* embeddings, the evaluation of the embedding host-language code immediately realizes the DSL evaluation. In *deep* embeddings, the evaluation of host-language code that embeds DSL code first produces a data representation of DSL abstract syntax, which is then interpreted or compiled. Thus, in deep embedding a DSL compiler can perform any kind of static checking or optimization, but it all takes place at run time of the host language.

In contrast, our style of DSLs use only the lexical or reader syntax of the host language. Their syntax and semantics are defined by custom compiler components (parser, static checker, code generator) that run at host-language compile time. Running as part of host-language compilation means that any static errors are raised at compile time in the IDE.

4.2 Language workbenches

Our meta-DSL adapts ideas from the language workbench tradition into a new context where we anticipate an opportunity for broad adoption.

Spoofax aims to be general, with meta-DSLs expressive enough to implement the full gamut of designs compatible with well-established programming language theory. This principled, research-focused approach pushes the expressivity of language workbench meta-DSLs. That expressivity comes with a cost, however: programmers need to learn an expansive language and have some familiarity with the underlying theory. To reduce this cost and flatten the learning curve we focus instead on the restricted scenario of building DSLs that fit together on top of a single host language.

SugarJ represents an alternative approach to integrating language workbench meta-DSLs with a host language [11]. It provides a front-end that integrates the syntax of Java together with (predecessor versions of) Spoofax meta-DSLs for syntax and transformation. While SugarJ achieves linguistic integration, it requires a non-standard compiler. It is not as easy to adopt as a library, which is the key insight for our approach.

4.2.1 IDE services

Language workbenches such as Spoofax and SugarJ generate more from a DSL definition than just a compiler. They also automatically generate IDE services. These services range from basic syntax coloring and bracket matching to rich semantic services. For example, Spoofax generates code completion that takes into account the syntax, static semantics, and name binding of the language [20]. With additional effort DSL creators can also implement custom transformations such as refactorings.

DSLs created with our prototype meta-DSL integrate with the DrRacket IDE in the same manner as other macro-based DSLs in Racket [13]. However, the provided services are limited to error highlighting, jump-to-definition, and rename refactorings. Macro-extensible languages have long faced challenges providing advanced IDE services. Syntax defined by a macro is specified only via its procedural expansion into host-language syntax. This lack of

structure makes it difficult to implement parsing, name analysis, or typechecking processes that recover from errors. Autocompletion faces a similar problem taking into account the grammar and static semantics of a macro-defined DSL.

Looking ahead, the syntax and binding rule declarations in our meta-DSL provide the structure that is missing from conventional macro definitions. Thus it should be possible to derive rich IDE services for DSLs specified in our language-workbench library. The open challenge lies in connecting the editor-service code that would be generated by the meta-DSL to the IDE. One possible approach is to modify the interface for macros to separate out analysis and compilation procedures. The analysis procedure would return the information needed to implement IDE services. The host language could then expose these services via the Language Server Protocol [4, 7]. While the task of implementing such an analysis interface for each language extension would be a burden for a traditional macro author, the meta-DSL can generate it automatically.

4.2.2 DSLs for domain experts

Some DSLs are designed with non-programmer domain experts in mind, rather than software engineers. Implementing a DSL for non-programmers demands different design considerations than those addressed by our meta-DSL. Fluid integration with general-purpose code and standard programming tools is less critical. On the other hand, providing structured editing to reduce errors, guidance via autocompletion or form-like interfaces, and live reaction to edits to show their effects become more important. Visual representations of domain concepts can also make their manipulation more tangible. Thus language workbenches such as JetBrains MPS [26] that produce external DSLs equipped with projectional editors provide advantages for these situations.

Nonetheless, our language-workbench library may sometimes be a good way to create DSLs for domain experts. Especially for initial prototyping, the ease with which a library-based DSL integrates into an existing software project may present an attractive tradeoff against the advantages of a more sophisticated implementation in a standalone workbench. As we improve the IDE services provided by our language-workbench library, this tradeoff becomes more advantageous. Furthermore, several directions of prior work suggest ways to integrate the custom editor services and visual and interactive elements that are important for domain experts. “Editor libraries” allow DSLs to provide custom editor services such as refactorings [10]. “Visual syntax” provides a mechanism for using interactive visualizations in place of textual syntax for macro-defined language extensions [1]. Along similar lines, “livelits” provide interactive syntactic elements that take on a more limited linguistic role but integrate with live evaluation [19].

5 Scaling up to mainstream host languages

Our meta-DSL design is specialized to Racket’s conventions for syntax and static semantics. When applying the idea in another host language these design elements would vary.

Furthermore, Racket is designed with extensibility in mind. S-expression syntax makes language extension within that syntax easy, and Racket features a module system designed specially to support macros and macro libraries [14]. Racket thus has all the macro system features we need to host a language workbench as a library. This makes Racket a good platform for prototyping our idea, but also the easiest case.

We anticipate that replicating our meta-DSL in another dynamically typed, Lisp-family language such as Clojure would be relatively easy. Clojure features S-expressions, procedural

macros, a form of macro hygiene, and reflection on the compile-time environment. Small extensions to the macro system would be needed to allow macros to record data about DSL variables in the compile-time environment and to invoke Clojure's macro expander in an extended environment.

Applying the idea in a language such as Rust presents a greater, yet surmountable challenge. Like Clojure, Rust supports procedural macros, includes macro hygiene, and uses a representation of syntax that is rich enough to support interesting DSL syntax [21]. Extensions similar to those discussed above for Clojure would suffice for implementing a basic language workbench library. In the context of Rust, these APIs need to be made type-safe and the added compile-time information needs to be plumbed between separate compilations.

In a typed language like Rust programmers would benefit substantially from integration between the type systems of each DSL and the host language. Ideally a language-workbench library for Rust would feature a meta-DSL like Statix [23] for specifying DSL type systems, suitably modified to integrate with Rust's type system. Unfortunately, Rust typechecking occurs only after macro expansion, meaning that type information is not available to macros, and any type errors are currently reported in terms of expanded code. A macro API that provides the right kind of interaction with the host language type system is an open research problem. Previous research on integrating macro expansion and typechecking in the Turnstile meta-DSL for Racket [5, 6] and the Klister language [3] suggest potential directions.

Unfortunately the metaprogramming systems present in many of today's languages are inadequate to properly host a language-workbench library. For example, Java's annotation processors [9] cannot introduce new syntax. Annotation arguments can only be simple values or arrays. Annotation processors also cannot change the way the processed class compiles; they can only generate additional classes. If our approach succeeds in those languages that currently have capable macro systems, we hope that this success will inspire the creators of other mainstream languages to add support for macros.

6 Conclusion

This paper has outlined what we consider a feasible evolution path for introducing language workbench technologies into an existing, mainstream programing language. Integrating a language workbench meta-DSL into an existing language ecosystem reduces friction at the moment a programmer chooses to adopt or create a DSL. Tailoring the design of the meta-DSL to use concepts familiar from the host language narrows the gap between programmers' existing knowledge and DSL creation. Implementing the meta-DSL as a macro library minimizes the additions needed to the core of a host language to support the approach.

Our experience from prototyping such a meta-DSL in Racket revealed that certain macro system features missing from mainstream languages are necessary to support a language workbench. However, we also found that the most complex features of Racket's macro system are not essential for this application. Given the additional information afforded by a declarative specification, the meta-DSL implementation can implement behaviors that would otherwise need to be provided in the host. Thus a relatively simple set of host-language extensions can add a great deal of power.

While our prototype is a first step on the evolution path to bring Visser's vision to fruition in mainstream languages, he imagined a much wider set of capabilities. Much work remains to be done to adapt these ideas and we must adjust our approach for varying host languages. In particular, providing rich IDE services and type system integration both seem to require widening the interface for macros beyond simple syntax-to-syntax transformation.

References

1. Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding interactive visual syntax to textual code. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428290.
2. Michael Ballantyne, Alexis King, and Matthias Felleisen. Macros for domain-specific languages. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428297.
3. Langston Barrett, David Thrane Christiansen, and Samuel Gélineau. Predictable macros for Hindley-Milner. In *The Workshop on Type-Driven Development*, TyDe '20, 2020. URL: https://davidchristiansen.dk/pubs/tyde2020-predictable-macros-abstract.pdf.
4. Hendrik Bünder and Herbert Kuchen. Towards multi-editor support for domain-specific languages utilizing the language server protocol. In *Model-Driven Engineering and Software Development*, MODELSWARD 2019, pages 225–245, 2020. doi:10.1007/978-3-030-37873-8_10.
5. Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. Dependent type systems as macros. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371071.
6. Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Proc. Principles of Programming Languages*, POPL 2017, pages 694—705, 2017. doi:10.1145/3009837.3009886.
7. Microsoft Corporation. Language server protocol. URL: https://microsoft.github.io/language-server-protocol/.
8. Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(4-5):439—476, 2012. doi:10.1017/S0956796812000275.
9. Joe Darcy and Oracle. Java specification request 269: pluggable annotation processing. URL: https://jcp.org/en/jsr/detail?id=269.
10. Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. In *Proc. Generative Programming and Component Engineering*, GPCE '11, pages 167–176, 2011. doi:10.1145/2047862.2047891.
11. Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-based syntactic language extensibility. In *Proc. Object-Oriented Programming Systems, Languages & Applications*, OOPSLA '11, pages 391–406, 2011. doi:10.1145/2048066.2048099.
12. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, February 2018. doi:10.1145/3127323.
13. Daniel Feltey, Spencer P. Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Languages the Racket way. *Language Workbench Challenge*, 2016. URL: https://users.cs.northwestern.edu/~robby/pubs/papers/lwc2016-ffkscfff.pdf.
14. Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. International Conference on Functional Programming*, ICFP '02, pages 72—83, 2002. doi:10.1145/581478.581486.
15. Matthew Flatt. Binding as sets of scopes. In *Proc. Principles of Programming Languages*, POPL '16, pages 705—-717, 2016. doi:10.1145/2837614.2837620.
16. Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together: compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming*, 22(2):181–216, March 2012. doi:10.1017/S0956796812000093.
17. Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings. In *Proc. International Conference on Functional Programming*, ICFP '14, pages 339–347, 2014. doi:10.1145/2628136.2628138.
18. Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proc. Object-Oriented Programming Systems, Languages & Applications*, OOPSLA '10, pages 444–463, 2010. doi:10.1145/1869459.1869497.

19 Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live GUIs. In *Proc. Programming Language Design and Implementation*, PLDI '21, pages 511—525, 2021. doi:10.1145/3453483.3454059.

20 Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. doi:10.1145/3527329.

21 The Rust Project. The Rust reference: Procedural macros. URL: https://doc.rust-lang.org/reference/procedural-macros.html.

22 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. Programming Language Design and Implementation*, PLDI '11, pages 132–141, 2011. doi:10.1145/1993498.1993514.

23 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276484.

24 Eelco Visser. WebDSL: a case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School*, GTTSE '07, pages 291–373, 2007. doi:10.1007/978-3-540-88643-3_7.

25 Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat. A language designer's workbench: a one-stop-shop for implementation and verification of language designs. In *Proc. International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 95—111, 2014. doi:10.1145/2661136.2661149.

26 Markus Voelter and Sascha Lisson. Supporting diverse notations in MPS' projectional editor. In *Proc. International Workshop on The Globalization of Modeling Languages*, GEMOC 2014, pages 7–16, 2014. URL: http://ceur-ws.org/Vol-1236/paper-03.pdf.

27 Martin P Ward. Language-oriented programming. *Software Concepts and Tools*, 15(4):147–161, 1994. doi:10.1007/978-1-4302-2390-0-12.