

1 Injecting language workbench technology into 2 mainstream languages

3 Michael Ballantyne ✉

4 PLT at Northeastern University, Boston, MA, United States

5 Matthias Felleisen

6 PLT at Northeastern University, Boston, MA, United States

7 — Abstract —

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

16 **2012 ACM Subject Classification** Software and its engineering → Macro languages; Software and its
17 engineering → Domain specific languages; Software and its engineering → Development frameworks
18 and environments

19 **Keywords and phrases** Language workbenches, macro systems, language adoption

20 **Digital Object Identifier** 10.4230/OASICS.EVCS.2023.10

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

22 **Acknowledgements** The authors are grateful for technical feedback from Michael Delmonaco, Sid-
23 dhartha Kasivajhula, and Alexis King. Also thanks to William Byrd and Siddhartha Kasivajhula for
24 helpful comments on early drafts, and to Shriram Krishnamurthi and Ryan Culpepper for rubbing
25 Matthias’s nose in Spoofox and language workbench ideas.

26 **1** Bringing Eelco Visser’s legacy to the mainstream

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

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

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

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



© Michael Ballantyne and Matthias Felleisen;
licensed under Creative Commons License CC-BY 4.0

Eelco Visser Commemorative Symposium (EVCS 2023).

Editors: Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann; Article No. 10; pp. 10:1–10:11

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Injecting language workbench technology into mainstream languages

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

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

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

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

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

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

84 **2 An integrated language workbench**

85 The best way to understand how developers should be able to use workbenches inside their
86 ecosystem is to work through an example. Consider a programmer who is implementing a
87 user interface that loads and displays a CSV file from a remote server. The controller for
88 the UI component is easily understood as a state machine. The states include the initial
89 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

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

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

98 2.1 Using a DSL as a library

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

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

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

117 2.2 Implementing a DSL as a library

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

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

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

134 We choose to account for syntax, binding rules, and the interface with the host language
 135 in our meta-DSL. These portions of a language implementation are the most uniform across
 136 DSLs, because their structure relates as closely to the host language as to the domain of the
 137 language being defined. This connection also means that these elements require the most
 138 intricate interaction with the host language implementation. The underlying implementation
 139 must interact with the host’s data representations of syntax, scope, and binding environments.
 140 In an extensible language such as Racket, the processes of parsing and name resolution
 141 interleave in complex ways. Thus a DSL implementation must perform operations in the
 142 correct order to ensure that data is available when needed. Hiding these elements via the
 143 meta-DSL means that programmers need not be aware of these effectful operational details.
 144 We leave programmers to implement the back-end compilation to Racket using conventional
 145 procedural Racket code augmented by the existing `syntax-parse` pattern matching and
 146 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

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

157 3 A language workbench as a macro-library

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

10:6 Injecting language workbench technology into mainstream languages

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

167 3.1 Essential macro system features

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

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

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

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

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

191 3.2 The full power of Racket’s macros is not needed

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

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

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

204 Even though the reflective APIs discussed above are critical to support our meta-DSL,
205 they are only needed in restricted form. For example, DSL analysis functions need to
206 invoke the host analyzer on subexpressions but do not need to examine the results; opaque
207 values would suffice. Likewise, the analysis functions need a way of associating compile-time
208 information with names, but this could take a simpler form than Racket’s support for
209 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

10:8 Injecting language workbench technology into mainstream languages

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

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

267 4.2.2 DSLs for domain experts

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

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

289 5 Scaling up to mainstream host languages

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

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

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

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

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

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

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

325 **6 Conclusion**

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

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

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

345 — **References** —

- 346 **1** Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding interactive visual syntax
347 to textual code. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/
348 3428290.
- 349 **2** Michael Ballantyne, Alexis King, and Matthias Felleisen. Macros for domain-specific languages.
350 *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428297.
- 351 **3** Langston Barrett, David Thrane Christiansen, and Samuel Gélineau. Predictable macros
352 for Hindley-Milner. In *The Workshop on Type-Driven Development*, TyDe '20, 2020. URL:
353 <https://davidchristiansen.dk/pubs/tyde2020-predictable-macros-abstract.pdf>.
- 354 **4** Hendrik Bünder and Herbert Kuchen. Towards multi-editor support for domain-specific lan-
355 guages utilizing the language server protocol. In *Model-Driven Engineering and Software Devel-*
356 *opment*, MODELSWARD 2019, pages 225–245, 2020. doi:10.1007/978-3-030-37873-8_10.
- 357 **5** Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. Dependent
358 type systems as macros. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:
359 10.1145/3371071.
- 360 **6** Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Proc.*
361 *Principles of Programming Languages*, POPL 2017, pages 694–705, 2017. doi:10.1145/
362 3009837.3009886.
- 363 **7** Microsoft Corporation. Language server protocol. URL: [https://microsoft.github.io/
364 language-server-protocol/](https://microsoft.github.io/language-server-protocol/).
- 365 **8** Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(4-5):439–476,
366 2012. doi:10.1017/S0956796812000275.
- 367 **9** Joe Darcy and Oracle. Java specification request 269: pluggable annotation processing. URL:
368 <https://jcp.org/en/jsr/detail?id=269>.
- 369 **10** Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann,
370 and Eelco Visser. Growing a language environment with editor libraries. In *Proc. Generative*
371 *Programming and Component Engineering*, GPCE '11, pages 167–176, 2011. doi:10.1145/
372 2047862.2047891.
- 373 **11** Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-
374 based syntactic language extensibility. In *Proc. Object-Oriented Programming Systems, Lan-*
375 *guages & Applications*, OOPSLA '11, pages 391–406, 2011. doi:10.1145/2048066.2048099.
- 376 **12** Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzila-
377 y, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language.
378 *Communications of the ACM*, 61(3):62–71, February 2018. doi:10.1145/3127323.
- 379 **13** Daniel Feltey, Spencer P. Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper,
380 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Languages the Racket way.
381 *Language Workbench Challenge*, 2016. URL: [https://users.cs.northwestern.edu/~robby/
382 pubs/papers/lwc2016-ffkscfff.pdf](https://users.cs.northwestern.edu/~robby/pubs/papers/lwc2016-ffkscfff.pdf).
- 383 **14** Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. International*
384 *Conference on Functional Programming*, ICFP '02, pages 72–83, 2002. doi:10.1145/581478.
385 581486.
- 386 **15** Matthew Flatt. Binding as sets of scopes. In *Proc. Principles of Programming Languages*,
387 POPL '16, pages 705–717, 2016. doi:10.1145/2837614.2837620.
- 388 **16** Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that
389 work together: compile-time bindings, partial expansion, and definition contexts. *Journal of*
390 *Functional Programming*, 22(2):181–216, March 2012. doi:10.1017/S0956796812000093.
- 391 **17** Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow
392 embeddings. In *Proc. International Conference on Functional Programming*, ICFP '14, pages
393 339–347, 2014. doi:10.1145/2628136.2628138.
- 394 **18** Lennart C. L. Kats and Eelco Visser. The Spofax language workbench: rules for declarative
395 specification of languages and IDEs. In *Proc. Object-Oriented Programming Systems, Languages*
396 *& Applications*, OOPSLA '10, pages 444–463, 2010. doi:10.1145/1869459.1869497.

- 397 19 Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling
398 typed holes with live GUIs. In *Proc. Programming Language Design and Implementation*,
399 PLDI '21, pages 511–525, 2021. doi:10.1145/3453483.3454059.
- 400 20 Daniel A. A. Pelsmaecker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser.
401 Language-parametric static semantic code completion. *Proc. ACM Program. Lang.*, 6(OOP-
402 SLA1), April 2022. doi:10.1145/3527329.
- 403 21 The Rust Project. The Rust reference: Procedural macros. URL: [https://doc.rust-lang.
404 org/reference/procedural-macros.html](https://doc.rust-lang.org/reference/procedural-macros.html).
- 405 22 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias
406 Felleisen. Languages as libraries. In *Proc. Programming Language Design and Implementation*,
407 PLDI '11, pages 132–141, 2011. doi:10.1145/1993498.1993514.
- 408 23 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as
409 types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276484.
- 410 24 Eelco Visser. WebDSL: a case study in domain-specific language engineering. In *Generative
411 and Transformational Techniques in Software Engineering II, International Summer School*,
412 GTTSE '07, pages 291–373, 2007. doi:10.1007/978-3-540-88643-3_7.
- 413 25 Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto
414 Passalaqua, and Gabriël Konat. A language designer's workbench: a one-stop-shop for
415 implementation and verification of language designs. In *Proc. International Symposium on
416 New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014,
417 pages 95–111, 2014. doi:10.1145/2661136.2661149.
- 418 26 Markus Voelter and Sascha Lisson. Supporting diverse notations in MPS' projectional editor.
419 In *Proc. International Workshop on The Globalization of Modeling Languages*, GEMOC 2014,
420 pages 7–16, 2014. URL: <http://ceur-ws.org/Vol-1236/paper-03.pdf>.
- 421 27 Martin P Ward. Language-oriented programming. *Software Concepts and Tools*, 15(4):147–161,
422 1994. doi:10.1007/978-1-4302-2390-0-12.