DSLs in Racket: You Want It How, Now?

Yunjeong Lee
National University of Singapore
Singapore
yunjeong.lee@u.nus.edu

Kiran Gopinathan
National University of Singapore
Singapore
mail@kirancodes.me

Ziyi Yang
National University of Singapore
Singapore
yangziyi@u.nus.edu

Matthew Flatt
University of Utah
USA
mflatt@cs.utah.edu

Ilya Sergey
National University of Singapore
Singapore
ilya@nus.edu.sg

Abstract

Domain-Specific Languages (DSLs) are a popular way to simplify and streamline programmatic solutions of commonly occurring yet specialized tasks. While the design of frameworks for implementing DSLs has been a popular topic of study in the research community, significantly less attention has been given to studying how those frameworks end up being used by practitioners and assessing utility of their features for building DSLs "in the wild".

In this paper, we conduct such a study focusing on a particular framework for DSL construction: the Racket programming language. We provide (a) a novel taxonomy of language design *intents* enabled by Racket-embedded DSLs, and (b) a classification of ways to utilize Racket's mechanisms that make the implementation of those intents possible. We substantiate our taxonomy with an analysis of 30 popular Racket-based DSLs, discussing how they make use of the available mechanisms and accordingly achieve their design intents. The taxonomy serves as a reusable measure that can help language designers to systematically develop, compare, and analyze DSLs in Racket as well as other frameworks.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**.

Keywords: domain-specific languages, meta-programming

ACM Reference Format:
Yunjeong Lee, Kiran Gopinathan, Ziyi Yang, Matthew Flatt, and Ilya Sergey. 2024. DSLs in Racket: You Want It How, Now?. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3687997.3695645

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

SLE '24, October 20–21, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695645

1 Introduction

When developing software, programmers often want domain-specific languages (DSLs) that are tailored for the problem at hand: such as using "regex" languages to express regular expressions, `make`-like DSLs for expressing build dependencies, or `SQL` languages for relational queries. Unfortunately, constructing such DSLs from scratch can be a challenge, because it requires the developer to write a front-end with a parser, lexer, *etc.*, as well as suitable libraries and definitions for use within the language itself. These challenges can dissuade a developer from creating and using a DSL. The Racket programming language [12] addresses this challenge and aims to popularize and streamline the development of DSLs. Racket encourages its users to *embed* their DSLs within the language itself, and it supports a litany of features to simplify this process. Most prominently, it supports customizable parsers and macro-based metaprogramming support to define a DSL's syntax, and it provides a module system for reusing libraries between different embedded languages. Together, these facilities provide a framework for language-oriented programming.

The past decade has seen both the Racket language and ecosystem mature and flourish, with contributions both from researchers extending the language and from developers using these features to build new DSLs. The research side includes work that extends Racket with new techniques and facilities to support DSL construction: support for debugging macros [9], tooling for writing macros with robust error handling [10] or ensuring hygienic treatment of scope and variables [15], and recently a framework for constructing macro-based DSLs that are also extensible by end users [3]. At the same time, the Racket ecosystem has seen a steady flow of new users and developers adopting the language-oriented paradigm and building their own DSLs in Racket. For example, at the time of writing, the *Racket Package Index* lists a substantial number of 2106 packages [32], providing libraries and DSLs for others to use.

Given the state of the ecosystem, it is possible to now pose interesting questions about its use. Most importantly, are the various macro-programming facilities that Racket provides for language design actually used "in the wild"? And if so,

what specific intents of the language designers themselves are effectively supported by Racket facilities? The answers to these questions have previously been unknown; to the best of our knowledge, no prior work has conducted any study of the real-world use of Racket mechanisms across its ecosystem or how they are employed in practice by language designers to implement their DSLs.

In this paper, we answer these questions by performing a study of the real-world uses of Racket for designing DSLs, presenting our findings from the perspective of a language designer and investigating how the various features of Racket are employed in practice to facilitate the goals of language design. The main contributions of this work are as follows:

- We identify a family of language design *intents*—implicit goals that a language designer has in mind with regard to how their language will look and feel—and present a systematic taxonomy based on a study of the Racket ecosystem (Sec. 3).
- For each of these design intents, we investigate and classify how the various features of Racket can be and are employed to achieve them (Sec. 4). We provide example implementations of DSLs using those mechanisms to achieve each design intent.
- Finally, using this classification system, we provide an extensive analysis of the 30 popular DSLs from the Racket ecosystem in terms of design intents and mechanisms. We also present a number of suggestions for future work for the Racket developers, its users, and the programming language research community (Sec. 5).

2 A Racket DSL: the User's Perspective

Self-styled as a *programming-language programming language* [12], the Racket ecosystem seeks to provide a metaprogramming framework for creating DSLs. In the context of designing DSLs, what differentiates Racket from other languages is in its ability to empower its users—*i.e.*, DSL authors—to easily change and customize the look and feel of their DSLs. In particular, Racket allows DSL authors to both adopt an arbitrary syntax for their DSL and to piggy-back on top of Racket, allowing them to use Racket's metaprogramming facilities to define custom semantics for their programs. This approach of embedding DSLs into Racket, rather than writing them from scratch, enables language authors to take advantage of Racket's own rich ecosystem while also being able to fully customize the way the DSL code is processed and run, all without implementing a full-fledged language from scratch. In this section, to illustrate the main components of Racket's approach to DSL design, we will provide a step-by-step walk-through of Racket's meta-programming capabilities from the perspective of an end user, programming a simple game as a running example.

2.1 A Racket Primer

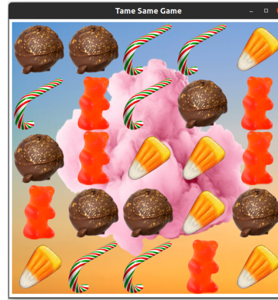

Fig. 1. Clickomania screen

Consider the task of developing a "Clickomania" game in Racket. Clickomania is a one-player, logical game where a board is populated with blocks of different colors or images, as shown in Fig. 1. At each turn, the player is allowed to remove all adjacent blocks of the same color or image around a given block. In this game, the player's goal is to clear as many blocks as possible from the board. To implement the game, the developer will need to develop functions that encode its various logical and graphical aspects. These include operations such as choosing random images, resizing images based on the frame or the block size, setting background images, removing blocks when they are adjacent and are of the same image or color, calculating scores upon removal, populating images in the empty blocks, running the game itself, and many more. Due to the number and complexity of these adjustments, it would be inconvenient and tedious to package these routines into a small number of functions with multiple parameters. Instead, a customizable Clickomania would be far more suited as a candidate for a dedicated DSL.

To begin, consider two particular functions from the Clickomania implementation. The first one encodes the internal behavior of the game itself, while the second one customizes images for rendering the user interface. Fig. 2 shows these functions implemented in vanilla Racket. We first specify a record type `wstate` to represent the game state, containing fields for an (x, y) coordinate of a mouse click, a counter to add up the number of mouse clicks, an image that represents a group of live blocks, a remaining number of live blocks, and an accumulated score (line 3). We also create a global variable `curr-idxs`, which is a mutable matrix—a list of lists—of indexes to keep track of states of blocks. The matrix is populated with randomly chosen indexes based on the number of colors and the total number of blocks (lines 5-7). To encode the behavior of removing same images which are represented with their respective index numbers, the `negate-same-indexes-around` function first collects indexes of images that are adjacent in `lst` (lines 10-13), recursively traverses the adjacent blocks and collect all the column and row indexes of the blocks to negate (lines 15-16), and negates all the indexes of the collected blocks (lines 19-20). Subsequently, this function is used to update the current game state of type `wstate` upon each mouse click input, thereby populating the new game screen.

To customize images for background as well as the blocks, the function `adjust-image-with-path` resizes images by

```
1  #lang racket
2  (require 2htdp/image)
3  (define-struct wstate [position counter image remaining score])
4
5  (define curr-idxs ; (Listof (Listof Number))
6    (let ([idxs (choose-rand-idxs num-colors num-blocks)])
7      (lst->lstlst idxs edge-size)))
8
9  (define (negate-same-indexes-around col-row curr-idx)
10   (let* ([lst (if (= -2 curr-idx)
11                   (list) ; If empty (-2), return empty list
12                   (list (col-row->u col-row) (col-row->d col-row)
13                         (col-row->l col-row) (col-row->r col-row)))]
14          ; Traverse adjacent blocks and collect col-row's to negate
15          [col-rows-to-negate (collect-col-rows-to-negate
16                               lst curr-idx (list col-row))]
17          [num-negate-blocks (length col-rows-to-negate)])
18     ; Negate indexes in curr-idxs based on col-rows-to-negate
19     (for ([col-row col-rows-to-negate])
20       (set-at col-row -1))
21     (* num-negate-blocks block-score)))
22
23 (define (adjust-image-with-path target-size img-path)
24   (let* ([img (bitmap/file img-path)]
25          [img-width (image-width img)]
26          [img-height (image-height img)])
27     ; Return an image whose width and height are equal to targe-size
28     (scale/xy (/ target-size img-width)
29               (/ target-size img-height) img)))
```

Fig. 2. Selected parts of the Clickomania game in vanilla Racket

adjusting the width and height of an image, provided with its path (`img-path`), based on a target length (`target-size`). It returns a resized image, as shown at lines 23-29 in Fig. 2. Notice that the functions `negate-same-indexes-around` and `adjust-image-with-path` are needed in any Clickomania game, in addition to other functions for encoding the above-mentioned logics (*e.g.*, choosing random images, populating images in the empty blocks, *etc*). This does not mean, however, that anyone who wants to create the game needs to implement all of them from scratch. Instead, one can build games by accessing `negate-same-indexes-around` directly or via a function `run` that incorporates all of the logic needed for running a game's instance.

2.2 Removing Boilerplate with a DSL

Let us see how creating the same game would differ using a dedicated `clickomania` DSL. The entire code is shown in Fig. 3. A function called `set-background-image` sets the background image, and `run` performs all the internal behaviors mentioned above, starting the game. Given these two functions and `adjust-image-with-path` from Sec. 2.1, they can be packaged together as the DSL *bindings* (*i.e.*, user-accessible identifiers) that one can use to create a game with a selected image resized and set as a background. Adjusting an image based on the frame size and setting it as the background can be simply done using `adjust-image-with-path` and `set-background-image`, as shown at lines 3-4 of Fig. 3. While the design of the `Clickomania` DSL is arguably concerned with a relatively simple problem domain, the benefits of abstraction are more visible for more complex domains and their operations. For example, Verilog [36] is a DSL designed for digial systems, enabling its users to easily describe complex electronic circuits for verification and testing.

2.3 Adding Custom Syntax

Not every developer is comfortable with Racket's syntax based on S-expressions. For instance, those familiar with C/Java-like languages might prefer grouping of statements using curly braces, which are understood as normal parentheses in vanilla Racket (though typically recommended for a specific use [17]). In Racket, this particular syntax can be accommodated easily, as shown in Fig. 4 (lines 4-9), which creates a new image by rotating an image and overlaying it on top of another, following with setting it up and running a game instance.

Customizing the DSL syntax in Racket is made possible by its meta-programming capabilities, including language front-end tools that are part of the Racket's standard libraries.

```
1 #lang clickomania
2
3 (define sunset (adjust-image-with-path 500 "./sunset.png"))
4 (set-background-image sunset)
5 (run 500 5 #:theme "candy" #:num-colors 4 #:num-clicks 10)
```

Fig. 3. Clickomania game instance implemented in the `clickomania` DSL

```
1 #lang clickomania
2
3 (define (create-background-image bsize bg-img-path img-path)
4   (if (<= 400 bsize) {
5         (define back-img (adjust-image-with-path bsize bg-img-path))
6         (define img (adjust-image-with-path (* bsize 0.8) img-path))
7         (define rotated-img (rotate -5 bimage))
8         (underlay back-img rotated-img)
9       }
10      (adjust-image-with-path bsize bg-img-path)))
11
12 (define sunset-cloud (create-background-image 500 "./sunset.png" "./cloud.png"))
13 (set-background-image sunset-cloud)
14 (run 500 5 #:theme "candy" #:num-colors 4 #:num-clicks 10)
```

Fig. 4. A `clickomania` program using curly braces for a block of statements.

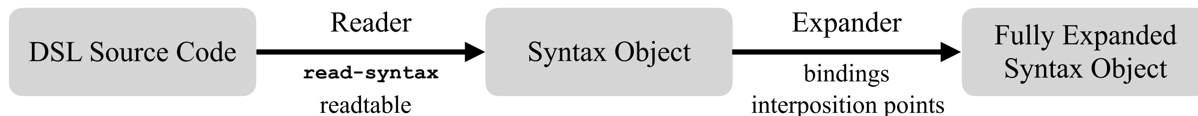

Fig. 5. From DSL source code to Racket; components beneath the arrows are explained in Sec. 4.

We discuss the details of how to enable the syntax of Fig. 4 (and more) in Sec. 4. For now, let us take a brief look at the underlying pipeline of converting a program in DSL into forms that can be processed by Racket's compiler to produce executable code, as shown in Fig. 5.

Racket builds on the traditional Lisp split of syntax handling into two layers: the *reader* and the *expander* [21]. DSL authors can extend or replace the reader, and they customize the expander's behavior by binding names in the expander's environment. A reader is a DSL-specific parser that converts its code into the Racket notation based on S-expressions, known as a *syntax objects*—data components containing the source code fragment itself, along with its metadata such as source location. In Racket, the reader is imported with the rest of the DSL implementation from the respective module using the `#lang` line. The rest of the file following `#lang` is parsed following its rules. Once the reader generates a syntax object, it gets passed to the expander, responsible for rewriting its syntax object into the Scheme-like core forms that are understood by the compiler. The expansion process relies on bindings in the lexical environment of a term as defined and implemented by a DSL designer.

Going back to the earlier `clickomania` code in Fig. 4, support for curly braces is achieved using the reader customization. At the same time, customizing the expander environment is a matter of defining *macros*, *i.e.*, compile-time functions that transform one syntax object to another. In the light of this explanation, we now look at what other language-specific behaviors that we can achieve via DSL embedding.

2.4 Tailoring Language-Specific Behavior

Using the meta-programming features of Racket, we can customize the behavior of DSL programs through imported and exported bindings, allowing an interaction between vanilla Racket code and DSL constructs. As a first example of this feature, we can make any `clickomania` program export a definition of `game-info`, shown in Fig. 6, for using in client applications. A failure to provide a binding for `game-info` when defining a game instance will cause the compilation of the `clickomania` program to break, triggering an error message that `game-info` is missing. This exporting constraint is achieved via `clickomania`'s expander environment (*cf.* Sec. 4.3) by defining a macro that controls how an overall module body is used. In particular, the macro accommodates

```
#lang clickomania                                    config.rkt
(provide run
         create-background-image)

(define (game-info)
  (game-title "Tame Same Game")
  (game-author "Racketeer")
  (values (game-title) (game-author)))
```

Fig. 6. `clickomania` program with a binding for `game-info`

```
#lang clickomania                                    mac.rkt
(provide adjust-image-with-path-macro)
; 'game-info' defined somewhere here
(define-syntax (adjust-image-with-path-macro
     stx)
  (syntax-parse stx
    [(_ size:number path:string)
     #'(adjust-image-with-path size path)]))
```

Fig. 7. `clickomania` code featuring a Racket macro

```
#lang racket                                         client.rkt
(require racket/gui/base
         "config.rkt")
(provide author)
(define-values (title author) (game-info))
(define bg-frame
  (new frame%
       [label (format "~a's Window" author)]))

; Generate a game upon clicking a button
(define (on-button-click b e)
  (set-background-image
    (create-background-image 1000
               "./dirt.png" "./tree.png"))
  (run 1000 10 #:theme "gem"
               #:num-colors 5 #:num-clicks 20))

(new button% [parent bg-frame]
  [label (format "Play ~a!" title)]
  [callback
    (lambda (b e) (on-button-click b e))])
(send bg-frame show #t)
```

Fig. 8. Racket program creating a window where a button click produces a Clickomania game

the module body's expansion to `provide` (*i.e.*, export) the `game-info` definition. Accordingly, when the module from Fig. 6 is imported by another program (line 2, Fig. 8), the `game-info` binding becomes automatically available in the importing program, as illustrated by the usage of `game-info` binding at line 4 in Fig. 8.

Second, we can write macros in the DSL to extend the set of language constructs, perform static checking, or both. As an example of simple static checking, Fig. 7 includes a macro named `adjust-image-with-path-macro`, which matches the input syntax `stx` against an AST pattern involving two arguments `size` and `path` that should respectively belong to the `number` and `string` syntax classes. Those syntax classes constrain the syntax to immediate number and string literals. Here, we use macro-generating terms such as `syntax-parse` [10] without having to import additional libraries that contain the terms, as shown at line 6 in Fig. 7. When the pattern matching succeeds, the macro rewrites, or *expands* to, the pattern with a template that calls the function `adjust-image-with-path` such as the one defined in Fig. 2 at lines 23-29. The constraint in this example is simple, but macros can be provided as DSL bindings to allow for the compile-time checking of various statically enforced policies, ranging from syntactic well-formedness of inputs to full-blown type systems [6].

Another behavior we can enforce via DSL is the ability to write Racket code in the DSL as in vanilla Racket. In Clickomania, any code that does not involve curly brackets `{}` is treated the same as vanilla Racket code. The interplay is more complex in some languages, as we will see in DSLs like `rash` in Sec. 3. We could similarly design `clickomania` in a way that disallows its users from being able to read or write Racket code whatsoever.

Separate from whether DSL syntax and Racket syntax can be mixed statically, DSL and Racket can cooperate to different degrees dynamically. In case of `clickomania`, we want the DSL to be *interoperable* with Racket. That means we can create a Racket program that uses `clickomania` code at runtime, *e.g.*, by creating an interactive GUI for some application in Fig. 8, where a button widget is used to trigger a Clickomania game from Fig. 6. Specifically, this interaction demonstrates using the results of calling `run` and `game-info` to define the runtime behavior of the code from Fig. 8.

2.5 Putting it All Together

We wrap up this tour by showing how all the user-facing aspects of Racket-embedded DSLs from Sec. 2.2–2.4 work together in a `clickomania` program in Fig. 9. First, we use `clickomania` functions such as `set-game-end-image` or `run`. Second, we use a modified syntax involving curly braces to arrange a group of code at lines 12-22. Third, exportation of `game-info` is enforced by the DSL. Although this enforcement is not apparent through the code snippet alone, we explain how this is enabled in Sec. 4.3. Fourth, we use a macro from Fig. 7 at line 11, as the DSL lets us write and work with macros. Fifth, when using the macro, Racket's macro facility is used to statically check the validity of input arguments to the `adjust-image-with-path` function at lines 5-8 in Fig. 7.

```
1  #lang clickomania
2  (require "client.rkt" "mac.rkt" )
3
4  (define (game-info)
5    (game-title "Same Game Name")
6    (game-author author) ; 'author' imported from client.rkt
7    (values (game-title) (game-author)))
8
9  (define (my-end-image size img-path txt)
10   (define bg ; Use a macro from mac.rkt
11     (adjust-image-with-path-macro size img-path))
12   (if (<= 200 size) { ; Return an image where a lambda is drawn
13        (define over-text (text/font txt (* 0.5 size) "white" #f
14                                  'modern 'italic 'normal #f))
15        (define-values (end mid) (values (- size 20) (/ size 2)))
16        (define line-drawn-bg (add-curve bg 20 20 0 1/3 end end 0 1/3 "white"))
17        (underlay (add-curve line-drawn-bg mid mid 0 0 20 end 0 0 "white")
18                  (text/font txt (* 0.1 size) "white" #f
19                             'modern 'italic 'normal #f))
20        } bg))
21 (set-game-end-image (my-end-image 200 "./neon-grid.png" "Try again"))
22 (run 400 4 #:num-colors 3 #:num-refills 3)
```

Fig. 9. `clickomania` program generating a Clickomania game with a personalized ending image

Sixth, as we implicity use Racket language constructs (*e.g.*, `define`, `require`, *etc.*) on top of Racket's standard image library functions (*e.g.*, `text/font`, `add-curve`, `underlay`), we are able to read and write plain Racket code throughout the program in Fig. 9. Lastly, we can let the DSL program interact with Racket program by importing and using a binding `author` from Fig. 8 at line 7.

In this section, we have seen what it's like to use a DSL embedded in Racket. Let us now dissect its design into a series of intents (Sec. 3) and show how each of the intents can be implemented in Racket (Sec. 4).

3 Language Design Intents

In this section, we present the first main contribution of this paper: a general taxonomy of the main *design intents* observed in use across real-world code found in the Racket ecosystem. As seen through the Clickomania example in the previous section, a language designer may have any number of different implicit goals and objectives in mind when designing their language. From a pragmatic perspective, it will be useful to precisely categorize these. To this end, we have identified seven design intents that capture the main goals and objectives that we have seen in use by developers in the Racket ecosystem (*cf.* Tab. 1). We gathered the list of intents through manual inspection of popular DSLs in Racket.

In the rest of this section, we will describe each of these intents in detail, contextualizing each by reference to how they apply in one of the three selected real-world DSLs: (1) the `frog/config` language for writing configuration files for a static blog generator [26], (2) `typed/racket`, an extension to Racket language with types [39], and (3) `rash`, a shell-like DSL for Racket [25]. While we cannot expect to capture all possible intentions that motivate language design, we expect that this taxonomy will encompass the most common objectives DSL authors have in mind when designing and developing their languages—a conjecture we quantitatively evaluate on a large corpus of case studies in Sec. 5.

```
1 #lang frog/config
2
3 (define/contract (init) (-> any)
4   (current-title "My Blog"))
5 (define/contract (enhance-body xs)
6   (-> (listof xexpr/c) (listof xexpr/c))
7   (~> xs (syntax-highlight #:line-numbers? #t))
8 (define/contract (clean) (-> any) (void))
```

Fig. 10. `frog/config` code for a blog generator

3.1 Custom Syntax

The most obvious motivation for constructing a DSL is, of course, to have a custom syntax. Racket's default syntax is inherited from its Lisp roots and based on heavily-parenthesized S-expressions, providing a suite of expression and definition forms for functions, datatypes, *etc.*, which can be quite verbose and might drive some users to seek alternatives. As we shall see, the three selected languages all embody the main different choices in this design space.

To begin with, the `frog/config` language (*cf.* Fig. 10) is an example of a DSL that opts not to change the default

Table 1. Description of language design intents

Design Intent	Description
Custom Syntax (§3.1)	Custom-built syntax that differs from Racket's default syntax
Custom Semantics (§3.2)	DSL-specific meanings assigned to the language constructs
Import/Export Control (§3.3)	Management of import- and/or export-related behaviors
Macro Support (§3.4)	Language facility to write and work with macros
Syntactic Embeddability (§3.5)	Ability to syntactically embed Racket into the DSL
Interoperability (§3.6)	Ability to work with Racket code at runtime
Static Checking (§3.7)	Analysis of certain properties at compile time

syntax at all by fully retaining the host language's characteristic Lisp-based S-expression syntax. In particular, the syntax for definitions, expressions and other forms are entirely unchanged in this language and can be used verbatim as in normal Racket code, fitting the purpose of `frog/config` designed for writing configuration files.

The `typed/racket` (*cf.* Fig. 11) language, on the other hand, presents a DSL with a slight deviation from the default Racket syntax, because it allows programs to include a type-annotation form using the "`:`" symbol. In particular, line 5 in the example snippet contains a type annotation of the form `(: <function-id> <function-type>)` placed before declaring the function. This annotation form is handled and interpreted as a type for the corresponding function `my-print-lst` at the time of parsing, while the rest of the program is parsed the same as in untyped `racket`. This way, `typed/racket` retains similarity with its host language, while, at the same time, *extending* the host language with new forms needed for typing.

Finally, DSL authors may choose to diverge from Racket's default syntax altogether and incorporate non-S-expression-based forms in their DSL, as illustrated by `rash` (*cf.* Fig. 12). In fact, `rash` happens to be a nuanced point in this design space, as its code is actually a mix of S-expressions, *e.g.* the argument to `ls` at line 5, and non-S-expressions combining both default Racket syntax as well as a custom one. This mirrors `rash`'s purpose as a shell language that can integrate with Racket code, using the non-S-expression syntax for shell commands and Racket syntax for Racket code. Of course, the inclusion of Racket syntax in `rash` is a deliberate design decision, not a requirement. Moreover, in practice, DSL authors have developed languages with even more divergent and esoteric syntax [19, 33, 41].

3.2 Custom Semantics

Beyond providing nicer front-ends for the user, the next obvious design goal is in having a bespoke semantics for programs written in the DSL. While most DSLs have customized semantics, there is still some nuance to be found, as shall be shown in the running examples.

Consider the `frog/config` code presented in Fig. 10. What would happen if we were to write the same program in vanilla Racket, *i.e.*, change the line 1 to `#lang racket`? Although programs written in `frog/config` mostly follow the semantics of Racket, the DSL introduces some small and subtle changes to Racket, in particular providing access to additional identifiers that are not available in `racket`, such as the functions `current-title` and `syntax-highlight`. In this way, programs in the `frog/config` DSL can leverage domain-specific functions where it makes sense, and otherwise fall back to Racket's semantics.

The `typed/racket` program in Fig. 11 again presents a larger deviation on this front, introducing an entirely new static semantics that is used to type-check the program. In particular, the listing includes terms that are *not bound* to any value or function in `racket` itself but only available for the purposes of type-checking, such as `define-type`, `Pairof`, `String`, `Number`, *etc.* Once type-checking is completed, then similarly to the previous example, `typed/racket` leverages the host language's semantics to actually run the program.

Finally, the `rash` DSL exhibits a different choice in the space of customized program semantics, and diverges further and attempts to replicate the behaviors expected of a shell scripting language. The language exposes commands such as the pipe operator `|`, or the `ls` command as shown in Fig. 12 to the user, which are defined to behave like their shell counterparts rather than as Racket functions, such as propagating error codes in a pipeline, consuming from standard input, or producing to standard output. Finally, as with the previous examples, the `rash` language again reuses the vanilla Racket semantics to give meaning to the vanilla Racket forms that it allows users to integrate into the DSL.

These three examples demonstrate how DSLs in Racket are able to provide a range of custom semantics by exposing bindings tailored to be specific to their domains, but also can

typed-fun.rkt

```
#lang typed/racket

(define-type StrNum (Pairof String Number))

(: my-print-lst (-> (Listof StrNum) Void))
(define (my-print-lst lst)
  (for ([elem lst])
    (printf (car elem) (cdr elem))))
```

Fig. 11. Defining a typed function in `typed/racket`.

leverage the bindings provided by Racket itself to reuse parts of its semantics as needed.[1]

3.3 Import/Export Control

Shifting away from syntax and semantics, we now move to some more nuanced aspects of language design, starting with the control of imports and exports of a DSL. In particular, in order to effectively shape the look and feel of their languages, designers often need mechanisms to constrain and refine the ways their users are allowed to import or export the DSL code. This is especially the case as we have seen that the set of exposed bindings, as imported and exported by libraries, can heavily impact the semantics of the DSL itself, and so users may be required to import or export certain bindings or other DSL code in a specific manner. This kind of behavior is demonstrated by DSLs with an import/export control as part of their design objectives, as shown in DSLs such as `frog/config` or `typed-racket`.

Returning back to our `frog/config` code in Fig. 10, in order to ensure that the user has defined *all* the necessary configuration variables to build the blog, the language itself enforces this constraint as an export requirement. Omitting any one of the three definitions, `init`, `enhance-body`, and `clean`, will actually cause the program to fail to compile with an error message that the user *must* provide a function for the missing identifier. Moreover, the `frog/config` language also prevents users from exporting any other bindings—these would be irrelevant for website configuration—by preventing users from accessing Racket's `provide` operator. Consequently, the only accessible identifiers available from any program written in `frog/config` will be the three that are automatically exported by the configuration language.

The `typed/racket` DSL also introduces constraints on its imports and exports, although in a different way. At first glance, it may seem that users can simply import or export other `typed/racket` code just as in `racket` by using `require` or `provide`. However, the underlying behaviors of `require` and `provide` in `typed/racket` have been altered to also incorporate type information when importing/exporting other code. For example, when importing from a vanilla Racket library, the user must provide type ascriptions for any bindings they introduce, while importing from another `typed/racket` library places no constraints.

Finally, `rash` is an example of a DSL *without* any import/export control. As a shell scripting language, `rash` is designed so that its users can write Racket code alongside other shell scripts, and as such allows users to import or export other code without constraints.

```
#lang rash

echo "Hello rash"
echo Show hidden files or a long listing:
ls (if (even? (random 2)) '-a '-l) | tac
```

Fig. 12. Combining Racket and bash scripting in `rash`.

3.4 Macro Support

DSLs can allow their users to write and use their own macros within DSL programs. There are two different flavors in which this intent is manifested in DSLs: (1) letting the DSL users to write macros just as in Racket and (2) making a DSL-specific macro system available so that the users write macros in the corresponding DSL. We can easily identify the first case by noting if a DSL designed to be a language on top of or used in combination with Racket. For instance, consider the DSLs discussed in earlier sections. Since `rash` can be understood as a shell language where users can also read and write normal Racket code, the DSL clearly intends its users to be able to enjoy Racket's macro system. As a typed language on top of Racket, `typed/racket`, is a DSL that also belongs to this category, as it allows its users to write Racket-like macros in `typed/racket`. On the other hand, `frog/config` is intended for writing configuration files for a blog generator, and is not meant to provide macro support to its users. The second case of DSL-specific macro system is identifiable by a language that provides macro-generating terms that are custom-built in the DSL. This turns out to be challenging to support, and is mostly found in DSLs constructed for research [19, 27, 28].

3.5 Syntactic Embeddability

In addition to providing macro support, a DSL can be designed to let its users read and write its host language code *as is* in the host language within the DSL program. When a DSL is flexible enough to allow its host language, Racket, to be *embedded* syntactically, the DSL is claimed to have *syntactic embeddability*. Intuitively, a DSL has this design intent if `racket` code can be copy-pasted in a DSL program and successfully compiles.

When it comes to `frog/config`, `racket` code can be read and written as is in `racket` and successfully compiles most of the time, except when the code has to export bindings other than `init`, `enhance-body`, and `clean` (see Fig. 10). As long

[1] The host language refers to `racket` or `racket/base` in `#lang` pragma.

```
1 #lang typed/racket
2 (require "typed-fun.rkt")
3 (require/typed racket
4                [string-append (-> String String String)]
5                [for-each (-> (-> StrNum Void) (Listof StrNum) Void)])
6
7 (: my-println-lst (-> (Listof StrNum) Void))
8 (define (my-println-lst lst)
9   (for-each (lambda ([elem : StrNum])
10              (let ([new-str (string-append (car elem) " ~a\n")]
11                    [num (cdr elem)])
12                (printf new-str num)))
13            lst))
```

Fig. 13. typed/racket code importing racket code

as provide is not used in a program, however, racket code which exists in the frog/config program is effectively embedded, making frog/config an example of DSLs with syntactic embeddability. Another example DSL with this intent is rash, as the language allows its users to switch between Racket and bash language (see Fig. 12). In this case, most racket code can be embedded successfully as is in racket except when rash has to read or write code as shell commands. On the other hand, when you look at typed/racket code in Fig. 11, normal Racket code cannot be written (or read) as if it is in racket since the DSL requires its users to annotate bindings with type information. When valid types are specified for the corresponding Racket code, as shown in lines 3-5 in Fig. 13, the typed code can compile with Racket bindings (*e.g.*, for-each on line 9, string-append on line 10). Hence, typed/racket is an example of DSLs syntactically embeddable under certain condition. Additionally, some of the #lang DSLs, such as rash or typed/racket, can be provided as a library in the form of (require <dsl-name>). And when a DSL is used as a library in Racket programs, we automatically know that the DSL is capable of syntactically embed Racket language.

3.6 Interoperability

Language designers can develop a DSL to make it work with with its host language. That is, if running the DSL code results in values that can work with Racket code, then we say that the DSL is *interoperable* with Racket. All of frog/config, rash, and typed/racket are interoperable with Racket, because executing programs written in these DSLs results in values that can work with Racket programs at runtime. Interoperability is almost always a feature and goal for Racket-embedded DSLs, as illustrated by the examples. DSLs, though rarely, may *not* be interoperable with Racket, as we will see in the survey results in Sec. 5.

3.7 Static Checking

Another useful language design goal is to equip a DSL with an ability to *statically check* certain properties. A common form of static checking is a type system to catch type errors at compile time. For example, typed/racket is a DSL where a type system is added to normal Racket, as illustrated in Fig. 11 and Fig. 13. On the other hand, since rash is meant to work as normal Racket as well as a shell language, it does not statically type check or perform any other static checking. Similarly, frog/config, a configuration language that works like Racket except for import/export control, is not intended to provide static checking. Static checking does not necessarily have to be in the form of type system, and Racket itself has static checks such as disallowing free variables. An *infix-notation*-based language built on Racket, rhombus [19], for example, lets its users add static information on expressions or bindings to control the way that they are used within other expression, including rejecting an expression that's used in a way incompatible with its static information (but with no soundness guarantee overall).

4 Enabling Mechanisms of Design Intents

This section describes the different ways of employing the various language design mechanisms that Racket provides in order to implement each one of the language design intents discussed in Sec. 3. Throughout this section, we refer back to the examples from Sec. 2 and clarify *how* the previously discussed design objectives concerning syntax and semantics of the game-developing DSL are enabled via Racket's metaprogramming facilities.

4.1 Custom Syntax

The main mechanism for syntax customization is by adjusting the DSL's reader (Fig. 5). In particular, it is only DSLs that are made available as libraries, *i.e.*, accessible via (require <dsl-name>), that always use the same reader

clickomania/lang/reader.rkt

```
1  #lang racket
2  (require syntax/strip-context)
3  (provide (rename-out [my-read-syntax read-syntax]))
4
5  (define (curly-parsing-handler char in src line col pos)
6    (let ([lst (read-syntax/recursive src in char #f)])
7      (cons 'block lst)))
8
9  (define updated-readtable
10   (make-readtable (current-readtable)
11                   #\{ 'terminating-macro curly-parsing-handler))
12
13 (define (my-read-syntax src in)
14   (define (my-read in) (read-syntax src in))
15   (parameterize ([current-readtable updated-readtable])
16     (let ([body (port->list my-read in)])
17       (strip-context
18        #`(module demo clickomania/main
19            #,@body)))))
```

Fig. 14. Extended reader for clickomania DSL

as vanilla Racket. In constrast, when a DSL is made available via through a #lang form, the compiler instead looks for a <dsl-name>/language/reader module, which in turn must export a read-syntax function that determines how to "read" the module, and parse its text contents into a syntax object.[2] In this work, we will assume we are dealing with the #lang DSLs, unless specified otherwise.

Even within DSLs that customize their readers, there are a variety of forms this can take. In particular, if a DSL has S-expression syntax but parses only certain notations differently from vanilla Racket, as was the case in clickomania, the DSL's reader can use read-syntax from Racket but adjust it by parameterizing a Lisp-style *readtable* that controls the reader. This readtable represents a mapping of "special" characters to their corresponding customized parsing handlers and allows extending the vanilla Racket parser with new behaviors. In cases when the syntax diverges even further, then users must write their own readers from scratch. Based on the extent to which read-syntax is replaced or adjusted, we classify a DSL's reader as one of the four categories: (1) *default*, (2) *extended*, (3) *custom*, or (4) *hybrid*.

First, when a DSL's reader is the same as Racket's built-in reader, *i.e.*, using the default read-syntax without modification of a readtable, the reader is called a *default reader*. In other words, when DSL authors want their language to have the Racket's default syntax, as in frog/config DSL, they can reuse and provide Racket's reader as the DSL's reader.

Second, a DSL that is primarily S-expression-based but has some non-S-expression elements—as in typed/racket—can be implemented by an *extended reader* which uses the Racket's default read-syntax with an updated readtable.

[2]Alternatively, a reader can be a submodule in the <dsl-name> module.

That is, by accessing and modifying a built-in readtable from Racket, a reader can parse certain characters differently from vanilla Racket. In the case of typed/racket, the extended readtable enables a #{} syntax for local type annotations. As shown in clickomania programs from Fig. 4 and Fig. 7, the clickomania DSL's reader is also an example of an extended reader, since its S-expression-based syntax differs from Racket's syntax by parsing curly brackets as a block of code. Fig. 14 shows a simplified implementation for the extended reader.[3] In this extended reader, my-read-syntax is provided as read-syntax (line 3), where read-syntax used in the definition of my-read-syntax at line 14 is provided by racket and called when the current-readtable, which refers to the racket's readtable, is parameterized by updated-readtable (lines 14-16). The updated-readtable is created by extending the Racket's default readtable with a parsing handler called curly-parsing-handler for a curly bracket (lines 9-11). Lines 5-7 then show how this parsing handler converts the code between curly brackets by joining the code by 'block. The binding for 'block is expected to be provided by the clickomania's expander environment.

Another way to customize syntax is by implementing a DSL-specific read-syntax function, in which case the reader is classified as a *custom reader*. This type of reader is needed to enable entirely non-S-expression-based syntax, as illustrated by shell language part of the rash code in Fig. 12. To demonstrate this case, a variant of clickomania called clickomania-infix reads with infix notation, similar to Python or Rhombus, Fig. 15 shows simplified code for the

[3]This example of an extended reader is motivated by a StackOverflow post: https://stackoverflow.com/questions/38369817/curly-brackets-to-replace-begin-in-racket.

```
1 #lang racket          clickomania-infix/lang/reader.rkt
2
3 (require syntax/strip-context)
4 (provide (rename-out [my-infix-read-syntax
      read-syntax]))
5
6 (define (my-infix-read-syntax src in)
7   (let ([body (custom-parse in)])
8     (strip-context
9      #`(module any clickomania-infix/main
10          #,body))))
```

Fig. 15. Custom reader for `clickomania-infix` DSL

DSL's custom reader. In this reader, the input program—passed to the second argument of `my-infix-read-syntax`, *i.e.*, `in` at line 5—is handled via a DSL-specific `custom-parse`[4] which parses and converts code written with infix notation to S-expressions. For instance, given an expression `n1 bop n2` consisting of a mathematical binary operation `bop` and two numbers `n1` and `n2`, `custom-parse` converts the input to its corresponding S-expression `(bop n1 n2)`.[5] Fig. 16 shows `clickomania-infix` code that is semantically equivalent to `clickomania` code in Fig. 4, where the custom reader is used to parse the code.

Last but not least, a *hybrid reader* refers to the type of reader that enables multiple syntaxes for a single DSL. We can create a hybrid reader by conditionally combining different types of readers discussed so far: that is, default, extended, and custom readers. Fig. 17 shows a simple hybrid reader that is a mix of a default reader and a custom reader. When the reader *peeks* a character and sees an open parenthesis `(`, it uses the default reader by calling `read-syntax` from `racket` (first clause of the `cond` conditional in lines 9-13). When the peeked character is not equal to `(`, it treats the code almost the same way as in custom reader from Fig. 15 (`else` clause in lines 14-18). Assuming that `custom-parse` at line 15 is constructed for reading code with infix notation, as discussed in the custom reader example, this hybrid reader[6] can be used for reading `clickomania-infix` code from Fig. 16 as well as normal Racket code.

In summary, a reader can be customized by defining the `read-syntax` function and/or configuring a readtable to enable custom syntax, which naturally leads to a taxonomy for readers: (1) default reader, (2) extended reader, (3) custom reader, and (4) hybrid reader. In addition, in case more than one type of reader is needed for the same DSL, as described in the hybrid case, language designers are expected to specify an additional condition per additional reader needed. Racket provides only limited support for composition of arbitrary readers, and while defining a new `read-syntax` offers an escape hatch to other parsing technologies, DSLs in the Racket ecosystem typically limit customization to the forms that Racket makes convenient.

[4]The `custom-parse` used in Fig. 15 is comparable to `parse-all` from `shrubbery/parse` library used in `rhombus` DSL.

[5]For simplicity, we do not cover all possible patterns with infix notation.

[6]This illustrative example takes the entire program and determines which reader to use, instead of choosing reader line by line or based on encountering certain characters. To achieve `rash`-like reading behavior, a more fine-grained switching of reader is is necessary.

4.2 Custom Semantics

Though a DSL may replace Racket's `read-syntax` wholesale to implement a bespoke syntax at the reader level, it will practically almost always use the Racket default macro expander as-is, refining the DSL's semantics through bindings provided in the expander's environment. That is to say, a DSL's *expander environment* is the main mechanism that is used to customize its semantics, and consists of an initial set of bindings, which the Racket expander uses to recursively *expand* forms in the syntax object to produce a fully-expanded program (Fig. 5).

Bindings will take the form of either functions and constants as seen before, or macros, which can interact with the expander itself. In particular, a macro is a function that consumes and produces syntax objects during the process of a program's expansion. Macros typically rearrange the syntax objects they are given as part of their input, and may introduce literal syntax-object fragments— constructed in code using either the `syntax` or `#'` forms. Of particular note, Racket also provides a restricted set of special *interposition points* bindings which facilitate more subtle customization of semantics. These interposition point forms are automatically inserted by the expander around every form in a program and have names that are prefixed with `#%`, including `#%module-begin` for declaration of a module, `#%app` for function application, `#%top-interaction` for REPL, and more [18]. Putting it all together, expanding any program in a DSL embedded in Racket employs the bindings in the expander context to produce an final Abstract Syntax Tree (AST) in terms of core Racket forms.

Of the various dimensions of expander customization, we identify five main strategies that are used in practice: (1) using default bindings only, (2) using custom bindings or (3) using custom bindings but exposing default bindings for end-user macros, and finally customization of either (4) only `#%module-begin` or (5) other interposition points such as `#%app`. Let us now explore the ways in which these can be used by DSL authors to provide custom semantics alongside the other language design intents.

Language designers enable custom semantics by making DSL-specific, *custom bindings* available through the DSL's expander environment. While Racket language provides *default bindings*—a predefined set of bindings supplied by Racket—to its users, custom bindings could be newly defined or override the default bindings with different meanings. Looking at the `clickomania` code in Fig. 4, for example, we

```
1  #lang clickomania-infix
2
3  fun create_background_image (bsize, bg_img_path, img_path):
4    if 400 <= bsize
5    | def back_img = adjust_image_with_path (bsize, bg_img_path)
6      def img = adjust_image_with_path (bsize * 0.8, img_path)
7      underlay (back_img, img)
8    | adjust_image_with_path (bsize, bg_img_path)
9
10 set_background_image (
11   create_background_image (500, "./sunset.png", "./cloud.png"))
12 run (500, 5, ~theme: "candy", ~num_colors: 4, ~num_clicks: 10)
```

Fig. 16. clickomania-infix code example

```
clickomania-hybrid/lang/reader.rkt
1  #lang racket
2
3  (require syntax/strip-context)
4  (provide (rename-out [my-hybrid-read-syntax
5                        read-syntax]))
6
7  (define (my-hybrid-read-syntax src in)
8    (define (my-read x) (read-syntax src x))
9    (let ([peeked (peek-char in 1)])
10     (cond
11       [(equal? #\( peeked)
12        (let ([body (port->list my-read in)])
13          (strip-context
14           #`(module sexp racket
15               #,@body)))]
16       [else
17        (let ([body (custom-parse in)])
18          (strip-context
19           #`(module nonsexp
      clickomania-hybrid/main
20               #,body)))])))
```

Fig. 17. Hybrid reader for clickomania-hybrid DSL

called custom bindings such as adjust-image-with-path, set-background-image, and run. Assuming that these function definitions (*e.g.*, adjust-image-with-path at lines 23-29 in Fig. 2) are written in "image.rkt" and "game.rkt", these functions become available in the clickomania DSL by providing them through its expander environment, as specified by export of the bindings at line 7 in Fig. 18. With all the custom bindings available via the expander environment, the clickomania program in Fig. 4 would generate a game screen that initially looks like the one in Fig. 1. Additionally, DSL authors can provide different semantics for *other* interposition points—option (5) from the above list—such as #%app or #%top-interaction to enable custom semantics. Although we do not discuss this alternative option in detail, it is not difficult to envision its implementation based on earlier discussions of macros.

4.3 Import/Export Control

We identify three different approaches that developers use to enforce import/export control. The first approach is through reinterpretation of #%module-begin to check constraints. For example, clickomania's check for exporting game-info (*e.g.*, lines 4-7 in Fig. 6) is enabled this way. Specifically, the expander environment in Fig. 18 defines my-module-begin and provides it as #%module-begin; where my-module-begin introduces (provide game-info) in addition to the module content form . . . as shown in the macro definition (lines 10-15).[7] Similarly, the frog/config DSL ensures its users provide definitions for init, enhance-body, and clean by supplying a customized #%module-begin. In this manner, other DSLs could require libraries to be imported by adding require forms in their #%module-begin expansion.

Another way to control imports and exports in a DSL is to provide customized semantics for the associated require and provide forms, or any DSL-specific language constructs used in importing or exporting DSL code. For example, the typed/racket DSL is equipped with reinterpreted require and provide so that these forms can accommodate type information when they are used for importing or exporting typed/racket or vanilla Racket code.

Lastly, import/export control can also be achieved by simply omitting any or all forms that provide imports or exports from a DSL's expander environment and thereby prevent users of the DSL from importing or exporting any DSL code at all. This scenario is also demonstrated by the frog/config DSL, which prevented its users from exporting any bindings—other than automatically exporting the three above-mentioned identifiers—by *not* supplying the form provide through its expander environment.

[7] In the my-module-begin macro, with-syntax works similarly to a let statement, arranging a variable to be used in the template. The format-id function, whose result is bound to the required-sym variable, is applied to format an identifier game-info with the corresponding lexical context.

clickomania/main.rkt

```
1 #lang racket
2 (require "image.rkt" "game.rkt" "sound.rkt"
3          (for-syntax racket syntax/parse))
4
5 (provide (except-out (all-from-out racket) #%module-begin)
6          (rename-out [my-module-begin #%module-begin])
7          (all-from-out "image.rkt" "game.rkt" "sound.rkt")
8          (for-syntax (all-from-out syntax/parse)))
9
10 (define-syntax (my-module-begin stx)
11   (syntax-parse stx
12     [(_ form ...)
13      (with-syntax ([required-sym (format-id stx "game-info")])
14        #'(#%module-begin form ...
15                          (provide required-sym)))]))
```

Fig. 18. Expander environment for `clickomania` DSL

interop.rkt

```
1 #lang racket
2 (provide title author)
3 (require "config.rkt")
4 (define-values (title author)
5   (game-info))
```

(a) `racket` code using `clickomania` code

REPL

```
1 > (require "interop.rkt")
2 > title
3 "Tame Same Game"
4 > author
5 "Racketeer"
```

(b) REPL demonstrating `interop.rkt`

Fig. 19. Interaction between `racket` code `clickomania` code

4.4 Macro Support

The `clickomania` DSL allows its users to write and work with Racket-like macros by making Racket's default bindings available at expansion time. For example, a macro named `adjust-image-with-path-macro` is defined in Fig. 7 and used in Fig. 9. Racket manages compile-time and run-time computations through a *phase system* that separates run-time bindings from compile-time bindings to help keep the computations and constraints of each time separate. Compile time is considered a higher *phase* than run time. With this system, a clear way to enable macro support is by making macro-generating libraries such as `syntax/parse` accessible at phase 1 but not phase 0. The `clickomania`'s expander environment provides `syntax/parse` bindings at phase 1, as indicated at line 8 in Fig. 18.[8] If the expander environment did not provide the macro library at compile time, it would not have been clear whether the `clickomania` DSL is designed with an intention to support its users to write macros as the DSL is originally intended for creating a specific game.

While reusing Racket's macro system for a DSL is relatively straightfrward, implementing a DSL-specific macro system is considered more difficult in the Racket ecosystem, as demonstrated by DSLs such as `hackett` [28], `rhombus` [19], and `qi` [27]. This is because developing a DSL-specific macro system requires non-trivial efforts to result in a macro system comparable to Racket's macro system in terms of composability [14], robustness [10], hygiene [16, 20], and cooperation [20]. Recent work [3] directly addresses this problem, but more time will be needed for packages in the Racket ecosystem to take advantage of the solution.

[8] The phase 1 is specified by `for-syntax`, each usage of which indicates one phase higher.

4.5 Syntactic Embeddability

While the intents discusssed so far are enabled by customizing either reader or expander, DSL's support for embedding of Racket forms requires both of the mechanisms to be tailored in particular ways. That is, a DSL is able to syntactically embed Racket by using the default reader—or in the terminology of Sec. 4.1, a default, extended, or hybrid reader— and providing all (or perhaps most) default bindings with compatible meanings of the bindings. For example, the `clickomania` DSL in Sec. 2 allows its users to syntactically embed Racket code by having its extended reader Fig. 14 parse DSL code almost identically to Racket, except when it encounters curly brackets, and also by ensuring that its expander environment provides all the default bindings that are semantically equivalent to those from Racket.

Redefining default bindings, including interposition points, in a DSL's expander environment does *not* automatically mean that the default bindings are not provided in the DSL. What matters is how much the semantics of the bindings

```
1  #lang typed/racket
2  (require typed/2htdp/image)
3  (provide (all-defined-out))
4
5  (: adjust-image-with-path (-> Real String Image))
6  (define (adjust-image-with-path target-size img-path)
7    (let* ([img : Image (bitmap/file img-path)]
8           [img-width : Nonnegative-Integer (image-width img)]
9           [img-height : Nonnegative-Integer (image-height img)])
10     (typed-scale/xy (/ target-size img-width)
11                     (/ target-size img-height) img)))
```

Fig. 20. `typed/clickomania` function annotated with types

changes through reinterpretation. In the `clickomania` case, although the `#%module-begin` is redefined to require exporting the `game-info`, this newly defined `#%module-begin` does *not* semantically differ from the one in Racket as it preserves the original contents of the module `form ...` (line 14, Fig. 18). And all other bindings from Racket are provided without any modification (line 5, Fig. 18).

4.6 Interoperability

The main mechanism that enables interoperability of a DSL in Racket is that the DSL code compiles (through expansion) to Racket, which means that the DSL uses the same runtime system and value representations as Racket programs. In this regard, interoperability of a DSL is limited only when those value representations are opaque or when Racket's runtime system is not used after all (as turns out for `urlang` DSL in Sec. 5). Like other Racket-embedded DSLs that are designed with interoperability in mind, as assessed in Sec. 5, `clickomania` is designed to work with Racket programs at runtime. The interoperability of `clickomania` DSL is demonstrated by the interaction between `clickomania` code in Fig. 6 and `racket` code in Fig. 8.

Fig. 19 illustrates another simple example of runtime interaction between `clickomania` and `racket` programs. First, `racket` program `interop.rkt` imports the `clickomania` code from Fig. 4, which automatically provides a binding for `game-info`, as discussed in Sec. 4.3. Next, it stores results of calling `game-info` in identifiers `title` and `author` which are or exported from the program (lines 2 and 4-5, Fig. 19). Then, we can call these values imported from `interop.rkt` and check that they refer to the ones from the `clickomania` program, as shown in Fig. 19. In addition, it is also possible to import `racket` libraries from the `clickomania` program and show that they can work together at runtime, although we do not explicitly demonstrate this interaction.

4.7 Static Checking

Racket enables general forms of static checking, including the implementation of a type system, by providing DSL authors access to the full Racket language at compile time. A DSL can inspect the syntax objects that make up a program either before or after expansion to Racket's core forms, depending on the level that makes sense for the language. The `typed/racket` DSL, for example, fully expands programs to check types [37].

We can create another typed variant of `clickomania` DSL that performs type checking at compile time, similar to `typed/racket`. What is needed is (1) to determine types representing classes of values and (2) to specify a set of rules called *type inference rules* for inferring types of expressions based on types of their sub-expressions. In particular, since type checking is to be done statically at compile time, raising syntax errors during expansion when types don't match the inferred types, custom bindings specified for type checking are provided in the form of macros or functions that are invoked at compile time. Racket is well equipped with various resources [6, 11, 13] to facilitate implementation of type systems. Although we do not delve into details of constructing DSL-specific types as well as typing rules due to limited space, a Racket-embedded meta-DSL called `turnstile` [7], based on the idea of *type system as macros* [6], in particular is useful in facilitating the creation of type systems for DSLs.

The `typed/clickomania` DSL does not have to be implemented from scratch, since the DSL is expected to behave like `typed/racket`. We can make use of `typed/racket` bindings (that incorporate the type system) in addition to specifying types for `clickomania`'s custom bindings, as illustrated by a custom function annotated with type information in Fig. 20. When the type-annotated custom bindings and overridden default bindings that perform type checking are available through the DSL's expander environment, calling the example function with invalid inputs will cause the type checker to throw a type error at compile time. For example, calling `(adjust-image-with-path "100" "./cloud.png")` results in `type mismatch, expected: Real given: String in: "100"`.

To conclude, Tab. 2 summarizes design intent-enabling mechanisms discussed in this section.

Table 2. The summary of racket mechanisms for different design intents

Design Intents	Enabling Mechanisms
Custom Syntax	Customization of reader (extended, custom or hybrid)
Custom Semantics	Custom bindings, Customization of other interposition points (*e.g.*, `#%app`, `#%top`, etc.)
Import/Export Control	Allow or restrict import/export-related bindings, Management of require/provide in `#%module-begin`
Macro Support	Racket bindings at phase 1
Syntactic Embeddability	Default, default-like extended, hybrid reader (a mix of default and others) *and* default bindings
Interoperability	Default bindings, custom bindings, `#%module-begin`, and other interposition points compiling to Racket forms via the expander
Static Checking	Custom bindings and customization of `#%module-begin` and other interposition points

5 Analysis of Design Trends

We now look at how popular Racket-embedded DSLs achieve the language design intents from Sec. 3 by customizing their readers and expander environments based on the reader taxonomy and expander customization options discussed in Sec. 4. In this process, we make observations and verify hypotheses that we had or are commonly believed in the Racket community about the ways in which DSL authors make use of reader and expander environment to design DSLs in Racket. We believe the hypotheses are unbiased as the hypotheses *H1.1*, *H2.2* and *H2.3* are respectively derived from the official guide [22], a book [5] and a paper [11]. The rest of hypotheses are reasonable because the language designers, also Racket programmers, are expected to prefer their DSLs to be similar to and be able to work with Racket.

Selection of case studies. DSLs analyzed in this paper were chosen based on their popularity, which was measured by the GitHub stars. Tab. 3 lists these DSLs, their language specifications and the numbers of GitHub stars. They were largely chosen from the *Racket Package Index* [32]. In some cases, though not many, DSLs are a small part of the corresponding projects (e.g., `frog/config` used for configuration in Frog project [26]). Moreover, while earlier sections primarily consider those DSLs specified as `#lang <dsl-name>`, DSLs considered here could be packaged and imported as a library with `(require <dsl-name>)`. These language specifications, enumerated in Tab. 3, help clarify which of the existing DSL specifications to look for when examining the corresponding DSL. It also shows that 7 out of the 30 studied DSLs are provided as libraries. In addition, there were several projects where (1) multiple DSLs exist in a single project (*e.g.*, `#lang minipascal` and `#lang minipascal simple`) or (2) the same DSL is provided both as a `#lang` language or as a library (*e.g.*, `#lang anarki` and `(require anarki)` from Anarki [2]). In both of these cases, we chose a representative as the one that we believe the DSL designer would intend its users to choose. That is because looking at multiple ones per DSL can make the statistics in Sec. 5.1-Sec. 5.3 biased or confusing. Racket-embedded DSLs often come with documentation that clarifies how the DSL authors intend their languages to be used. When there are multiple versions of languages and when such documentation exists, we relied on it to determine which version to use as a representative.

5.1 Reader Customization

Prior to studying the DSLs, we made the following hypotheses about how DSL authors would choose to customize the readers for their languages, which type of reader is expected to be more (or less) prevalent, and why this may be the case.

- *H1.1.* A DSL is more likely to extend Racket syntax than to have its own custom syntax. If a DSL extends Racket syntax, the syntax extension is expected to be implemented via `readtable` customization. This expectation, also noted in the *Racket Guide* [22], means that DSLs with extended readers are likely to be more common than the ones with custom readers.
- *H1.2.* DSL designers prefer S-expression syntax over non-S-expression syntax, resulting in DSLs with custom syntax being less common than the DSLs with S-expression syntax. In other words, we expect more use of default and extended readers than custom readers.
- *H1.3.* DSLs with multiple syntaxes (*e.g.*, `rash`) are expected to be rare. That is to say, we hypothesized that DSLs are least likely to have a hybrid reader.

Tab. 4 shows the breakdown of different types of reader used in the DSLs. The results largely confirm the above hypotheses about the reader customization. First, with regard to *H1.1*, there are more extended readers (20.0%) than custom readers (13.3%). While we expected extended readers to be much more prevalent than the custom ones, however, the

Table 3. A selection of Racket DSLs for our analysis

DSL	Specification	Stars	DSL	Specification	Stars
anarki	(require anarki)	1149	malt	(require malt)	137
hackett	#lang hackett	1137	turnstile	#lang turnstile	129
frog/config	#lang frog/config	906	video	#lang video	123
pie	#lang pie	661	racket-clojure	#lang clojure	118
rosette	#lang rosette	618	sketching	#lang sketching	106
rash	#lang rash	529	racket-r7rs	#lang r7rs	92
typed-racket	#lang typed/racket	494	redex	(require redex)	89
urlang	(require urlang)	296	minipascal	#lang minipascal	88
rhombus	#lang rhombus	286	algebraic	#lang algebraic	73
beautiful-racket	#lang br	282	brag	#lang brag	62
rackjure	#lang rackjure	234	sham	(require sham)	66
cur	#lang cur	215	lens	(require lens)	73
scribble	#lang scribble	189	heresy	#lang heresy	69
nanopass	#lang nanopass	174	qi	(require qi)	51
cKanren	#lang cKanren	152	racket-lua	#lang lua	50

difference turns out to be not large. Second, the number of default and extended readers combined is indeed greater than the number of custom readers, confirming *H1.2.* The difference (70%) is significant and shows the popularity of relying on the default reader. Lastly, hybrid readers turned out to be the least common type, as there was only one DSL, `rash`, that falls into this category, which is likely caused by them being the most difficult to construct.

A substantial number of readers are default type, amounting to 19 out of 30, which is over three times more frequent than the second most common type. Even considering that a DSL provided as a library automatically uses the default reader, the default type is twice as much as the extended type upon excluding the 7 DSLs provided as a library (Tab. 3). Again, this implies that many DSLs (*e.g.*, `frog/config` [26], `rosette` [40], `cur` [4], `pie` [8], etc.) are built using the Racket's default reader *as is* rather than customizing the reader.

5.2 Expander Customization

We made the following three hypotheses about popularity of the five expander customization options (Sec. 4) to better understand the ways in which language designers choose to determine a set of bindings for their DSLs.

- *H2.1.* Custom bindings are the most commonly employed expander customization option. DSLs are created for specific domains, and they operate based on DSL-specific, custom bindings. This expectation suggests that DSLs that provide (a subset of) default bindings *alone* are likely to be rare, if any.
- *H2.2.* Changing the meaning of interposition points is uncommon among DSLs, as noted in *Beautiful Racket* [5], due to the complexity involved. That is, we expect tha tDSLs that do *not* customize the meanings of `#%module-begin`, and other interposition points are expected to be more common than the ones that do.
- *H2.3.* Out of the interposition points, customization of `#%module-begin` is more common than that of other interposition points. We expect `#%module-begin` to be a popular customization choice because its reinterpretation can help DSL authors to eliminate boilerplate code and communicate context-sensitive information during expansion, as noted by the Racket developers [11].

Table 4. Types of readers used in DSLs

	Default	Extended	Custom	Hybrid
Number of DSLs	19	6	4	1
Percentage of DSLs	63.3%	20.0%	13.3%	3.3%

Usage of the expander customization options, as shown in Tab. 5, confirm two out of the three above-mentioned hypotheses. First, as speculated in *H2.1*, custom bindings are indeed the most popular option to customize expanders of the studied DSLs. In fact, all of the 30 DSLs provide custom bindings via their expanders, indicating that none of the DSLs operate with default bindings alone. Second, there are more DSLs that do not customize interposition points than the ones that do, supporting *H2.2.* However, the percentage

Table 5. Expander customization options used in DSLs

	Default bindings	Subset of default bindings	Custom bindings	Default bindings at phase 1	Custom `#%module-begin`	Other custom implicit forms
Number of DSLs	18	6	30	5	11	13
Percentage of DSLs	60.0%	20.0%	100.0%	16.7%	36.7%	43.3%

of DSLs that reinterpret `#%module-begin` and other interposition points are respectively 36.7% and 43.3%, indicating that they are a small minority. Based on this, we recognize that there is significant need for DSL authors to customize these special macros to tailor default behaviors of DSLs. Third, DSLs that customize `#%module-begin` turn out to be fewer than the ones that customize other implicit forms, contrary to our expectation in *H2.3*. Looking at the DSLs that do not customize `#%module-begin` but do customize other interposition points, `#%app` and `#%top` (used for module-level or top-level bindings of variables) are frequently reinterpreted. This result suggests that these other interposition points play a more significant role in designing DSLs than previously expected. Additionally, we observe that 80% of the DSLs provide all or subset of default bindings, suggesting Racket-embedded DSLs have a strong tendency to utilize bindings from their host language.

5.3 Language Design Intents

Most importantly, we made hypotheses about and subsequently investigated popularity of the language design intents among the surveyed DSLs. In particular, we took the relationship between reader and expander customizations and the correspondingly enabled design intents into consideration, as illustrated in Tab. 2.

- *H3.1*. DSLs are likely to provide macro support. That is, we expect that DSL authors have designed their DSLs in a way that allows its users to extend the DSLs, as needed.
- *H3.2*. DSLs are expected to syntactically embed their host language, Racket. We expect that DSL authors would like to allow the DSL users to be able to read and write (or escape to) Racket code.
- *H3.3*. DSLs are likely to be able to work with Racket. In other words, we expect more DSLs that support interoperability than ones that do not. Similarly to *H3.2*, this hypothesis is based on our expectations about DSL authors' preference.

In the process of designing languages, a DSL author is expected to have written macros and accordingly appreciate extensibility of Racket based on its macro system. This naturally led us to hypothesize that DSLs are likely to be designed to support its users to write and work with macros. Popularity of the design intents (Tab. 6) confirms the hypothesis *H3.1*, with about two-thirds of DSLs equipped with macro support. On the other hand, it is interesting to note that one third of the DSLs do *not* provide macro support, suggesting that DSLs may intend to prevent its users from extending the DSLs. Moreover, 70% of the DSLs are syntactically embeddable, validating *H3.2*. This outcome is related to the fact that a majority of DSLs choose to use default reader and provide all (or subset of) default bindings. Note that when a DSL's reader and expander environment do not fully satisfy the requirements for enabling syntactic embeddability (Sec. 4.5) by a minor difference (*e.g.*, a character is parsed differently as `clickomania`, all the default bindings except for `provide` are provided as `frog/config`) we determine whether the DSL achieves the intent based on qualitative understanding of the purpose and use cases of the DSL. The last hypothesis *H3.3* is supported by a large majority of DSLs, 83.3%, being interoperable with Racket. Most of the bindings in the surveyed DSLs generate values that Racket programs can work with. However, 16.6% of DSLs turned out to *not* support embedding of Racket forms. For example, some DSLs are meant to be standalone languages without S-expression syntax—as in `brag` [41], `minipascal` [33] or `pie` [8], and these languages are presumably not meant to work with Racket in any ways. Surprisingly, there is a DSL called `urlang` [34] with Racket's default syntax but running the DSL program would produce strings representing JavaScript code, results that cannot work with Racket code at runtime.

Looking at the rest of intents, custom semantics is the most popular design intent among DSLs, as DSLs are assumed to operate under their domain-specific operations. This result is aligned with the hypothesis *H2.1* from Sec. 5.2. The least popular intent turned out to be static checking, whereas most of the DSLs with static checking achieved it in the form of type systems, with `rhombus` [19] being a notable exception (*cf.* Sec. 3.7). In addition, while a majority of the intents are enabled by exactly one (combination of) customization of the mechanism, as summarized in Tab. 2, import/export control can be enabled in multiple ways. Out of 12 DSLs with import/export control, 2 of them achieve the intent through reinterpretation of `#%module-begin`, 7 of them by redefining the Racket import/export bindings (*i.e.*, `require`, `provide`), 4 of them by preventing all or subset of import- or export-related bindings.[9]

[9] One of the 12 DSLs, `frog/config`, both reinterprets `#%module-begin` and prevents import/export bindings.

Table 6. Language design intents in DSLs

	Custom Syntax	Custom Semantics	Import/Export Control	Macro Support	Embed-dability	Interoper-ability	Static Checking
Number of DSLs	11	30	12	20	21	25	6
Percentage of DSLs	36.7%	100.0%	40.0%	66.7%	70.0%	83.3%	20.0%

5.4 Threats to Validity

Several factors threaten the validity of this survey. First, the studied DSLs were selected among the listed Racket packages [32] based on GitHub stars. However, DSLs with the most GitHub stars are not guaranteed to be the most widely used or representative Racket-embedded languages. Furthermore, one of our examples was selected from a GitHub repository where the DSL was a small part of the project (*e.g.*, `frog/config` used in configuring a blog generator), implying that the popularity is about the project, not about the language. Second, some of the surveyed DSLs are created by the same author, or same (sub)set of authors. This could mean that DSLs they implement are likely to have a similar set of design intents, potentially making this study biased. Third, DSLs may not have been *fully* implemented at the time of this study, despite the fact that we took these DSLs from published packages. For example, `hackett` and `cur` specifically mention that development of the languages is incomplete at the time of writing. Lastly, 30 may not be a big enough number as there exist many more Racket-embedded DSLs that can be taken into consideration. While this may impact the generality of our results, we believe the studied DSLs are a representative sample of practical DSL usage. That is because (1) for practicality, DSLs have been selected prioritizing those with higher number of GitHub stars and (2) for representation, filtering packages in *Racket Package Index* [32] by the tag "language" or "lang" produces fewer than 100 results at the time of writing this paper. Although there are likely even more DSLs in the Racket ecosystem, it seems reasonable that our sample captures a representative portion of actively used DSLs in Racket.

6 Related Work

This paper connects several lines of work: (a) on classification of design intents requiring one to build a new domain-specific language, (b) techniques and patterns for implementing DSLs, and (c) applications of Racket meta-programming mechanisms allowing one to do so.

Classification of language intents. Past work has classified DSL intents and developed taxonomies of DSLs, although not in the context of a specific framework to implement them (*e.g.*, Racket). In particular, Kövesdán *et al.* [29] developed a catalog of DSL intents based on the functionalities and usage patterns. Their classification relies on a set of *intent properties*, largely inspired by design pattern formalization that was originally developed for objected-oriented software [24]. Unlike our taxonomy, Kövesdán *et al.*'s classification is not based on design objectives and does not discuss the specific implementation mechanisms.

DSL design patterns. Prior to our work, several surveys offered constructive guidance on how to decided whether developing a DSL would be beneficial for solving a task at hand, and, if so, what functionality its design should incorporate. Mernik *et al.* [31] offered a family of DSL design patterns, arranging a set of domains based on the identified usage scenarios. The work analyzes the patterns based on application domains and requirements, focusing on specific functionalities rather than language designers' needs and goals with regard to tailoring the look and feel of the resulting language. Fowler [23] lists several reasons why one should consider creating a domain-specific language, using a series of examples, following the DSL categorization based on types of design framework. However, the work neither offers a clearly defined set of DSL design intents nor does it explore DSL design patterns that can be allowed by concrete implementation strategies. Spinellis [35] proposes eight DSL design patterns, with the classification is based on *how* a DSL can be created by harnessing the existing programming languages available in 2001. Our work not only offers a set of DSL design intents but look at how these play out among DSLs embedded specifically into the Racket ecosystem.

Racket-embedded DSLs. Despite Racket's rich history of serving as a foundation DSL embedding and a variety of tools it provides to help programmers to develop domain-specific languages [3, 11], no prior work has been done to systematically survey Racket-embedded DSLs and their implementation trends. Regardless, several efforts have been made to showcase individual DSLs and the advantages they provide for solving the respective programming/scripting tasks [4, 6, 25, 38, 40]. Additionally, several small-scale case studies were conducted to demonstrate the effectiveness of developing languages for particular domains [1] or using specific elements of Racket, *e.g.*, macros[3]. While there exist tutorials on step-by-step construction of toy DSLs for educational purposes [5, 13], they do not immediately provide the answers to the questions we have posed in Sec. 1, and, thus, cannot serve as studies of trends in real-life Racket DSLs.

7 Conclusion

The design space of domain-specific languages is vast and diverse. Existing efforts to classify DSLs have largely been concerned with DSL application domains and usage patterns. By focusing on DSL intents irrespective of a concrete design framework, they often lacked coherent explanations as to *what* methodologies enable those intents. Moreover, while it is feasible to identify more language design intents if we consider multiple existing frameworks for DSL construction, many of these frameworks have relatively small user base. Therefore, we focus on Racket as a flatform due to its popularity and the diversity of available DSL implementations.

Our work proposed a taxonomy of DSL design intents as well as their enabling mechanisms specific to Racket's ecosystem. In particular, we analyzed existing Racket-embedded DSLs and made observations about how experienced Racket programmers use Racket's meta-programming facilities to achieve their language design objectives. We expect that our analysis will be helpful to shed light on the common design objectives and practices in the Racket community. For the language designers, our findings will be informative regarding the capabilities and limitations of Racket as an implementation platform. For Racket users, our work can serve as a guide for better understanding the underlying behavior of the Racket-based programming tools.

While our work is Racket-specific, we believe that findings from this paper pose interest to the programming language community and could be applied to other languages with similar metaprogramming facilities such as Rust, Scala, OCaml, Haskell, etc. More specifically, designers of DSLs embedded in non-Racket languages can use our framework and taxonomy describing the main intents in DSL design to better plan and systematically develop their DSLs. Moreover, given that other languages like Rust or Scala enable their users to achieve subsets of language design intents identified and presented in our work, developers of non-Racket languages can get ideas about what metaprogramming facilities they can add or modify within their languages to allow the users, *i.e.*, DSL designers, to achieve more design objectives that can be popular among users.

Acknowledgments

We thank the SLE'24 anonymous reviewers for their constructive and insightful comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001 and by Sergey's MoE Tier 1 grant T1 251RES2108 "Automated Proof Evolution for Verified Software Systems".

Data Availability

The artifact accompanying this paper is available online [30]. It contains source code implementing the `clickomania` DSL as well as the implementation of reader customization and expander customization from Sec. 2 and Sec. 4.

References

[1] Leif Andersen, Stephen Chang, and Matthias Felleisen. 2017. Super 8 languages for making movies (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 30:1–30:29. https://doi.org/10.1145/3110274

[2] Ross Angle, Kartik Agaram, and Zachary Kanfer. 2016. Anarki: Community-Managed Arc Variant. https://docs.racket-lang.org/anarki/. Last updated: 13 Mar 2016.

[3] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for domain-specific languages. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 229:1–229:29. https://doi.org/10.1145/3428297

[4] William J Bowman. 2016. Growing a Proof Assistant. In *The Workshop on Higher-order Programming with Effects (HOPE)*.

[5] Matthew Butterick. 2013. Beautiful Racket. https://beautifulracket.com/.

[6] Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type systems as macros. In *POPL*. ACM, 694–705. https://doi.org/10.1145/3009837.3009886

[7] Stephen Chang, Alex Knauth, Ben Greenman, Milo Turner, and Michael Ballantyne. 2018. The turnstile language. https://docs.racket-lang.org/turnstile/index.html. Last updated: 16 Apr 2021.

[8] David Thrane Christiansen and Daniel P. Friedman. 2018. The Pie Reference. https://docs.racket-lang.org/pie/. Last updated: 7 Jul 2021.

[9] Ryan Culpepper and Matthias Felleisen. 2007. Debugging macros. In *GPCE*. ACM, 135–144. https://doi.org/10.1145/1289971.1289994

[10] Ryan Culpepper and Matthias Felleisen. 2010. Fortifying macros. In *ICFP*. ACM, 235–246. https://doi.org/10.1145/1863543.1863577

[11] Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi. 2019. From Macros to DSLs: The Evolution of Racket. In *SNAPL (LIPIcs, Vol. 136)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:19. https://doi.org/10.4230/LIPICS.SNAPL.2019.5

[12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *SNAPL (LIPIcs, Vol. 32)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 113–128. https://doi.org/10.4230/LIPICS.SNAPL.2015.113

[13] Matthias Felleisen, Matthew Flatt, Robby Findler, Jay McCarthy, and Jesse Tov. 2019. Racket School 2019: The How to Design Languages track. https://school.racket-lang.org/2019/plan/.

[14] Matthew Flatt. 2002. Composable and compilable macros: you want it when?. In *ICFP*. ACM, 72–83. https://doi.org/10.1145/581478.581486

[15] Matthew Flatt. 2013. Submodules in Racket: you want it when, again?. In *GPCE*. ACM, 13–22. https://doi.org/10.1145/2517208.2517211

[16] Matthew Flatt. 2016. Binding as sets of scopes. In *POPL*. ACM, 705–717. https://doi.org/10.1145/2837614.2837620

[17] Matthew Flatt. 2024. Reading Pairs and Lists. In *The Racket Reference*. PLT. https://docs.racket-lang.org/reference/reader.html#%28part._parse-pair%29.

[18] Matthew Flatt. 2024. Syntactic Forms. In *The Racket Reference*. PLT. https://docs.racket-lang.org/reference/syntax.html.

[19] Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay A. McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. 2023. Rhombus: A New Spin on Macros without All the Parentheses. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 574–603. https://doi.org/10.1145/3622818

[20] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that Work Together - Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.* 22, 2 (2012), 181–216. https://doi.org/10.1017/S0956796812000093

[21] Matthew Flatt and Robert Bruce Findler. [n. d.]. Lists and Racket Syntax. In *The Racket Guide*. PLT. https://docs.racket-lang.org/guide/Pairs_Lists_and_Racket_Syntax.html#(part._lists-and-syntax).

[22] Matthew Flatt and Robert Bruce Findler. 2024. Using `#lang s-exp syntax/module-reader`. In *The Racket Guide*. PLT. https://docs.racket-lang.org/guide/syntax_module-reader.html.

[23] Martin Fowler. 2011. Domain-Specific Languages. Addison-Wesley.

[24] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. 707 (1993), 406–431. https://doi.org/10.1007/3-540-47910-4_21

[25] William Gallard Hatch and Matthew Flatt. 2018. Rash: from reckless interactions to reliable programs. In *GPCE*. ACM, 28–39. https://doi.org/10.1145/3278122.3278129

[26] Greg Hendershott. 2022. frog: Static blog generator application. https://docs.racket-lang.org/frog/index.html. Last updated: 26 Dec 2023.

[27] Siddhartha Kasivajhula. 2024. Qi. https://github.com/drym-org/qi.

[28] Alexis King. 2020. Hackett. https://github.com/lexi-lambda/hackett.

[29] Gábor Kövesdán, Márk Asztalos, and László Lengyel. 2014. A classification of domain-specific language intents. *International Journal of Modeling and Optimization* 1, 4 (2014), 67–73.

[30] Yunjeong Lee, Kiran Gopinathan, Ziyi Yang, Matthew Flatt, and Ilya Sergey. 2024. *DSLs in Racket: You Want It How, Now? Software Artifact.* https://doi.org/10.5281/zenodo.13709851

[31] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (2005), 316–344. https://doi.org/10.1145/1118890.1118892

[32] PLT. 2023. Racket Package Index. https://pkgs.racket-lang.org/. Accessed: 2023-12-30.

[33] Jens Axel Søgaard. 2021. Minipascal. https://github.com/soegaard/minipascal.

[34] Jens Axel Søgaard. 2023. Urlang. https://github.com/soegaard/urlang.

[35] Diomidis Spinellis. 2001. Notable design patterns for domain-specific languages. *J. Syst. Softw.* 56, 1 (2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3

[36] Donald E. Thomas and Philip Moorby. 1995. *The Verilog hardware description language (2. ed.).* Kluwer.

[37] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL*. ACM, 395–406. https://doi.org/10.1145/1328438.1328486

[38] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *PLDI*. ACM, 132–141. https://doi.org/10.1145/1993498.1993514

[39] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. 2014. The Typed Racket Guide. https://docs.racket-lang.org/ts-guide/.

[40] Emina Torlak and Rastislav Bodík. 2013. Growing Solver-Aided Languages with Rosette. In *Onward!: ACM Symposium on New Ideas in Programming and Reflections on Software*. ACM, 135–152. https://doi.org/10.1145/2509578.2509586

[41] Danny Yoo and Matthew Butterick. 2022. brag: a better Racket AST generator. https://docs.racket-lang.org/brag/.