Combining Processing with Racket

Hugo Correia and António Menezes Leitão(✉)

INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa, Rua Alves Redol 9, Lisboa, Portugal
{hugo.f.correia,antonio.menezes.leitao}@tecnico.ulisboa.pt

Abstract. Processing is a programming language created to teach programming in a visual context. Despite its success, Processing remains a niche language with limited applicability in the architectural field, as no Computer-Aided Design (CAD) application supports Processing. This work presents an implementation of Processing for the Racket platform, that transforms Processing code into semantically equivalent Racket code. Our Processing implementation is developed as a Racket module language for interoperability with Racket and other module languages of Racket's language ecosystem. Our implementation allows us to take advantage of Rosetta, a Racket library that provides access to several CAD back-ends (e.g. AutoCAD, Rhinoceros, SketchUp). As a result, architects and designers can take advantage of our implementation to use Processing with their favourite CAD application.

Keywords: Processing · Racket · Compilers · Interoperability

1 Introduction

Processing [1] is a programming language and development environment created to teach programming in a visual context. The language has grown over the years, creating a community where users can share their artistic works. Many examples and educational materials are available to newcomers, reducing their effort to learn the language. Moreover, Processing offers a wide range of 2D and 3D drawing primitives, as well as an Integrated Development Environment (IDE) that provides tools to programmatically create innovative designs.

Nonetheless, Processing is a niche programming language with limited applicability in the architectural field, as architects depend on traditional heavy-weight CAD applications (e.g. AutoCAD, Rhinoceros 3D, etc.), that provide APIs tailored for that specific CAD tool. Unfortunately, no CAD application allows users to write scripts in Processing. As a result, architects that have learnt Processing cannot use the language or any of the publicly available examples to program in the context of their favourite CAD tool. This problem is addressed in this paper, showing how our solution combines Processing with the Racket programming language.

Racket [2] is a descendent of Scheme, which encourages developers to tailor their environment to project-specific needs, offering an ecosystem that allows

J.-L. Sierra-Rodríguez et al. (Eds.): SLATE 2015, CCIS 563, pp. 101–112, 2015.
DOI: 10.1007/978-3-319-27653-3_10

the creation of new languages which have direct interoperability with other Racket libraries. For instance, Rosetta [3] is a Generative Design tool built on top of Racket, that encompasses Racket's philosophy of using different languages. Rosetta allows programmers to generate 2D and 3D geometry in a variety of CAD applications, namely AutoCAD, Rhinoceros3D, SketchUp, and Revit, using several programming languages, such as JavaScript, AutoLISP, Racket, and Python. Furthermore, Racket offers a pedagogic IDE, DrRacket, which can be adapted to support new module languages of the Racket ecosystem.

Our solution is to implement a source-to-source compiler that translates Processing code to semantically equivalent Racket code, enabling architects to prototype designs using Processing in a CAD tool. Moreover, as Racket encourages developers to use and create different languages within the Racket ecosystem [4], we have developed an interoperability mechanism to access Racket libraries and to combine Processing with scripts written in other languages of the Racket ecosystem, such as Python [5] or Typed Racket [6].

2 Processing

Processing was developed at MIT media labs and was heavily inspired by the *Design by Numbers* [7] project. The language was created to teach computer science to artists and designers with no previous programming experience. Processing has grown over the years with the support of a large community, which has written several educational materials, demonstrating how programming can be used in the visual arts.

Processing can be considered a dialect of the Java programming language, that significantly simplifies the original language. For instance, in Java, developers have to implement a large set of steps to develop simple examples, namely a public `class` that implements public methods and a static `main` method. These constructs bring an initial overhead and verbosity for novice programmers, which are cumbersome for beginners that want to quickly try out new ideas. To solve this problem, Processing allows users to write simple scripts (i.e. simple sequences of statements) that do not have the verbosity of Java, thus enabling them to quickly create new designs.

The Processing language introduces the notion of a *sketch*, which is used to organize source code. A *sketch* can operate in one of two distinct modes: *Static* or *Active*. *Static mode* supports simple Processing scripts, such as simple statements and expressions. *Active mode* allows users to implement their *sketches* using more advanced features of the language. If a function or method definition is present, the *sketch* is considered to be in *Active mode*. Within each *sketch*, Processing users can define two functions to aid their design process: `setup` and `draw`. On one hand, the `setup` function is called once when the program starts. In `setup` the user can define initial environment properties and execute initialization routines needed to create the design. On the other hand, the `draw` function runs after `setup` and executes the code to draw the design. The control flow is simple: first `setup` is executed, setting-up the environment; followed by `draw` called in a loop, rendering the sketch until it is stopped by the user.

Moreover, Processing offers users a set of tools that are specially tailored for visual artists. For instance, 2D and 3D drawing primitives are made available, rendering designs in different 2D and 3D rendering environments. Processing also offers a simple but effective development environment called the PDE (Processing Development Environment), where users can develop their programs using a tabbed editor with IDE services such as syntax highlighting and automatic code formatting.

3 Related Work

The following section presents different approaches that influenced our work, and an analysis of their main features.

3.1 Processing.js

Processing.js [8] is a JavaScript implementation of Processing for the Web that enables developers to create scripts in Processing or JavaScript. Using Processing.js, developers can use Processing's approach to design 2D and 3D geometry in a HTML5 compatible browser. Processing.js uses a custom-purpose JavaScript parser, that parses both Processing and JavaScript code, translating Processing code to JavaScript while leaving JavaScript code unmodified. Moreover, Processing.js implements Processing drawing primitives and built-in classes directly in JavaScript, allowing for greater interoperability between both languages, as Processing code is seamlessly integrated with JavaScript. To render Processing scripts in a browser, Processing.js uses the HTML canvas element to provide 2D geometry, and WebGL to implement 3D geometry. Processing.js encourages users to develop their scripts in Processing's development environment, and then render them in a web browser. Additionally, Sketchpad [9] is an alternative online IDE for Processing.js, where users can create and test their design ideas online and share them with the community.

3.2 Processing.py and Ruby-Processing

Ruby-Processing [10] and Processing.py [11] produce Processing/Java as target code. Both Ruby and Python have language implementations for the JVM, allowing them to directly use Processing's drawing primitives. Processing.py takes advantage of Jython to translate Python code to Java, while Ruby-Processing uses JRuby to provide a Ruby wrapper for Processing. Processing.py is fully integrated within Processing's development environment as a language mode, and therefore provides an identical development experience to users. On the other hand, Ruby-Processing is lacking in this aspect, by not having a custom IDE. However, Ruby-Processing offers *sketch* watching (code is automatically executed when new changes are saved) and live coding, which are functionalities that are not present in any other implementation.

3.3 ProfessorJ

ProfessorJ [12,13] was developed to be a language extension for DrScheme [14]. ProfessorJ implements a traditional compiler pipeline, that starts with lexing and parsing phases, producing an intermediate representation in Scheme. Subsequently, the translated code is analysed, generating target Scheme code by using custom defined functions and macro transformations. ProfessorJ implements several strategies to map Java code to Scheme. For instance, Java classes are translated into Scheme classes with certain caveats, such as implementing static methods as Scheme procedures or by changing Scheme's object creation to appropriately handle Java constructors. Also, Java has multiple namespaces while Scheme has a single namespace, hence name mangling techniques were implemented to correctly support multiple namespaces. Moreover, Java's built-in primitive types and some classes are directly implemented in Scheme, while remaining classes are implemented in Java. Strings, Arrays, and Exceptions are mapped directly into Scheme forms. Implementing them in Scheme is possible (with some constraints) due to similarities in both languages which, in turn, allow for a high level of interoperability. Finally, ProfessorJ is fully integrated with DrScheme, providing a development environment that offers syntax highlighting, syntax checking, and error highlighting for Java code.

4 Compilation Process

Observing all previous implementations, it is clear that an IDE is an important feature to have in any implementation of the Processing language (as only Ruby-Processing is lacking one). Regarding the runtime system, Processing.js and ProfessorJ implement it in the target language to achieve greater interoperability; while, Ruby-Processing and Processing.py take advantage of JVM language implementations to provide Processing's runtime.

Finally, we observe that none of the presented approaches offers a solution that allow us to explore Processing in the context of a CAD environment. Neither Processing.js, Processing.py, or Ruby-Processing allow designs to be visualized in a CAD tool. Alternatively, other external Processing libraries could be explored to connect Processing with CAD applications. For instance, OBJExport [15] is a Processing library to export coloured meshes from Processing as OBJ or X3D files. These files can then be imported into some CAD applications. However, using this approach, we lose the interactivity of programming directly in a CAD application, as users have to generate and import the OBJ file each time the Processing script is changed, creating a cumbersome workflow. Moreover, as shapes are transformed to meshes of triangles and points, there is a considerable loss of information, as the semantic notion of the shapes is lost.

Our proposed solution was to develop Processing as a new Racket language module, using Rosetta for Processing's visual needs, and integrating Processing with DrRacket's IDE services. We chose Racket, firstly, because it simplifies the development of new languages, providing libraries to implement the lexical and syntactic definitions of the Processing language, as well as offering mechanisms

to generate semantically equivalent Processing code. Secondly, Racket's capabilities enable us to easily adapt our Processing implementation to work with DrRacket (Racket's educational IDE), providing an IDE to its users. Moreover, after analysing ProfessorJ, we concluded that many parts of the lexical and syntactical definitions, and type-checking procedures could be adapted, due to the similarities between Java's and Processing's language definitions. Finally, our implementation allows us to take advantage of Rosetta to augment Processing with capabilities that make the language suitable for architectural work.

Our Processing implementation follows the traditional compiler pipeline approach (illustrated in Fig. 1), composed by three separated phases, namely parsing, code analysis, and code generation.

4.1 Parsing Phase

The compilation process starts with the parsing phase, which is divided in two main steps. First, Processing source code is read and transformed into tokens. Secondly, tokens are given to an LALR parser, building an abstract syntax tree (AST) of Racket objects which will be analysed in subsequent phases. To implement the lexer and parser specifications, we used Racket's `parser-tools` [16] library, adapting parts of ProfessorJ's lexer and grammar specification to fit Processing's needs.

4.2 Code Analysis

Following the parsing phase, an analysis of the AST must be made, due to differences between Processing's and Racket's language definitions. For instance, Processing has static type-checking and different namespaces for methods, fields, and classes, while Racket is dynamically typed and has a single namespace. As a result, custom tailored mechanisms were needed to correctly type-check the AST and support Processing's scoping rules.

Firstly, the AST is traversed passing scope information to child nodes. When a new definition is created, be it a function, variable, or class, the newly defined binding is added to the current node's scope along with its type information. Each time a new scope is created in Processing, a new custom scope is created to represent it, referring to the current scope as its parent. These mechanisms are needed to implement Processing scoping and type-checking rules. For example, the information of the return type, arity, and argument types are needed to type-check a function call.

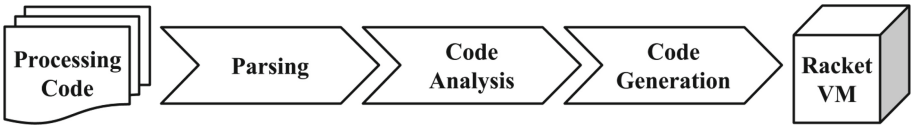


Fig. 1. Overall compilation pipeline

Secondly, the type-checking procedure runs over the AST starting topmost AST node. As before, it repeatedly calls the type-checker on child nodes until the full AST is traversed, using previously saved bindings in the current scope to find out the types of each binding. During the type-checking procedures, each node is tested for type correctness and, in some cases, promoting types if necessary. In the event that types do not match, a type error is produced, signalling where the type error occurred.

4.3 Code Generation

After the AST is fully analysed and type-checked, semantically equivalent Racket code can be generated. To achieve this, every AST node generates Racket code by using custom defined macros and functions. Afterwards, Racket will expand the defined macros and load the generated code into Racket's VM. By using macros we can create boilerplate Racket code that can be constantly modified and tested by the developer

Racket and Processing follow the same evaluation order on their programs, thus most of Processing's statements and expressions are directly mapped into Racket forms. However, other statements such as `return`, `break`, or `continue` need a different handling, as they use control flow jumps. To implement this behaviour, we used Racket's *escape continuations* [17] in the form of `let/ec`.

Furthermore, Processing has multiple namespaces, which required an additional effort to translate bindings to Racket's single namespace. To support multiple namespaces in Racket, binding names were mangled with custom tags. For instance, a `fn` tag is appended to functions, so function `foo` internally would be `foo-fn`. The use of `'-'` as a separator allows us to solve the problem of name clashing with user defined bindings, as Processing does not allow `'-'` in names. Also, as we have function overloading in Processing, we append specific tags that represent the argument's types to the function's name. For instance, the following function definition: `int foo(float x, float y){ ... }` would be translated to `(define (foo-FF-fn x y) ...)`.

To correctly support Processing's distinctions between *Active* and *Static* mode, we used the following strategy. We added a custom check in the parser that signals if the code is in *Active mode*, i.e. if a function or method is defined. In this mode, global statements are restricted, thus when generating code for global statements we check if the code is in *Active mode*, if so we signal an error indicating the invalid statement.

5 Runtime

Our runtime is implemented directly in Racket, allowing for greater interoperability with Racket libraries, namely Rosetta. However, this presents some important issues. First, as Racket is a dynamically typed language, the type-checker, at compile time, cannot know the types of Racket bindings. To solve

this problem, we introduced a new type in the type hierarchy, which the type-checker ignores when type checking these bindings. Furthermore, as Processing primitives and built-in classes are implemented in Racket, we also have the problem of associating type information for these bindings. Therefore, we created a simple macro that allows us to associate type information to Racket definitions, by adding them to the global environment, thus the type-checker can correctly verify if types are compatible.

Moreover, Processing's drawing paradigm closely resembles OpenGL's traditional push & pop matrix style. To provide rendering capabilities in our system, we use Rosetta, as it provides design abstractions that not only let us generate designs in an OpenGL render, but also give us access to several CAD back-ends. Custom interface adjustments are needed to implement Processing's drawing primitives in Racket, as not every Processing primitive maps directly into Rosetta's. Furthermore, Rosetta also enables us to provide with additional drawing primitives that are unavailable in the original Processing environment.

6 Interoperability

One of the advantages of developing a source-to-source compiler is the possibility of combining libraries that are written in different languages. The Racket platform encourages the use and development of different languages to fulfil programmers' needs, offering a set of extension mechanisms that can be applied to many of the language's features. The combination of Racket's language modules [6] and powerful hygienic macro system [18] enables users to extend the base Racket environment with new syntax and semantics that can be easily composed with modules written in different dialects.

To achieve interoperability with Racket, we developed Processing's compilation units as a Racket language module, adding Processing to Racket's language set. Nonetheless, compatibility issues between languages arise when accessing exported bindings from a Racket module. First, a new `require` keyword was introduced to specifically import bindings from other modules. This `require` maps directly to Racket's `require` form, receiving the location of the importing module. By using Racket's `require` we have access to all of Racket's require semantics, enabling the programmer to select, exclude, or rename imported binding from the required module.

Furthermore, Racket and Processing have different naming rules. For instance, function `foo-bar!` is a valid identifier in Racket but not in Processing, thus we cannot reference the `foo-bar!` function in our Processing code. To solve this issue, we use a translation procedure that takes a Racket identifier and transforms it into a valid Processing identifier. For example, `foo-bar!` would be translated to `fooBarBang`. Therefore, for each provided binding of a required module, we apply the translation procedure on each binding, making it available to the requiring module. By providing an automatic translation, the developer's effort is reduced, as he can quickly use any Racket module with his Processing code. Notwithstanding, as developers may not be satisfied with our

```
#lang racket

(provide foo-bar)
(define (foo-bar foo)
  ...)
```

Fig. 2. The `foo-bar` module in Racket

```
#lang processing

require "foo-bar.rkt";
void checkFoo(String s) {
  println(fooBar(s));
}
```

Fig. 3. `checkFoo` in Processing

automatic translation procedure, they can develop their custom mappings in a Racket module adhering to Processing identifier's rules.

Another issue that arises by importing foreign bindings, is making them accessible to our custom environment and type-checker, as they are needed during the *code analysis* phase. To solve this issue, we dynamically load the required module, saving exported bindings along with their arity. As Racket is dynamically typed, we use a special type for arguments and return types that the type-checker skips. As a result, when using bindings with this type, typing errors will only be observed when these bindings are executed at runtime. To illustrate the interoperability mechanism consider the `foo-bar` module Fig. 2, which provides the `foo-bar` function, and the Processing code illustrated in Fig. 3.

As illustrated in Fig. 3, the function `checkFoo` uses the `foo-bar` procedure from `foo-bar.rkt`. Note that our automatic translation procedure has been applied to provided bindings from the `foo-bar.rkt` modules. So in `checkFoo`, we use the automatically translated `fooBar` identifier to refer to `foo-bar`.

To understand how this is accomplished, our `require` uses a custom macro that receives the module's path (i.e. the location of the required module), as well as a list of pairs that map the original bindings of the module into their mangled form. To compute this list, we used Racket's `module->exports` primitive to provide the list of exported bindings. However, this information does not suffice, as we need to know the arity of each exported binding. This is information is needed to produce a compatible biding (i.e. a mangled binding) with our generated code. Therefore, we analysed each exported binding by `module->exports`, and retrieved its arity using the `procedure-arity` primitive. This way we can correctly perform the translation of external bindings to valid Processing identifiers and generate bindings that work with our code generation process. Lastly, when generating Racket code, our custom macro expands to Racket's `require` form, making each mangled binding available in the requiring module.

7 Example

Developing a source-to-source compiler has the advantage of allowing us to explore libraries written in another language. We provide an example of our implementation, showing how Processing code can take advantage of libraries

```
require "fib.rkt"; require "draw.rkt";
void echo(int n, Object pos, float ang, float r) {
  if ( n == 1) {
    fullArc( pos, r, ang, HALF_PI, 20);
  } else {
    fullArc( pos, r / fib(n), ang, HALF_PI, 20);
    echo(n-1, pos, ang, r);
  }
}
void mosaics(float l, int size) {
  for(int i = 0; i < size;  i++) {
    for (int j = 0; j < size; j++) {
      echo(10, xyz(i*l, j*l, 0), 0, l);
      echo(10, xyz(i*l+l, j*l, 0), HALF_PI, l);
      echo(10, xyz(i*l+l, j*l+l, 0), PI, l);
      echo(10, xyz(i*l, j*l+l, 0), 3/2 * PI, l);
    }
  }
  frame(xyz(0,0,0), size * l, 20);
}
```

Fig. 4. Processing code to generate mosaics

that were previously built for 3D modelling. This is still a work in progress, thus the compilation results are likely to change.

Consider the Processing code presented in Fig. 4. The `mosaics` procedure generates a grid of mosaics given the length of each mosaic and the total size of the grid. This function uses `echo` to generate the interior pattern of each mosaic, progressively generating smaller arcs from each corner of the mosaic. After generating the interior pattern, the `frame` generates the full outer boundary of the grid.

This example illustrates the use of two external Racket libraries. First, we `require` the `fib.rkt` module to use `fib` to compute the reducing factor of arches size. This illustrates how we can use simple Racket code with Processing. Secondly, we `require draw.rkt`, which allows us to access `fullArc` and `frame`. These functions enable us to generate the arcs and produce the enclosing boundary, showing how we can use previously created Racket drawing libraries with our Processing implementation. Moreover, observe the use of `xyz` primitive. Rosetta provides custom mechanisms to abstract coordinate systems, namely cartesian (`xyz`), polar (`pol`), and cylindrical (`cyl`) which can be used and combined interchangeably. As a result, we made these abstractions (`xyz`, `pol`, and `cyl`) available in our system, so that users can take advantage of them in their designs. Figure 5 illustrates an execution of the `mosaics` function in AutoCAD.

Observe the generated Racket code for `echo` displayed in Fig. 6. The first point that is immediately visible is that function identifiers are renamed to support multiple namespaces. We can see that the `echo` identifier is translated to

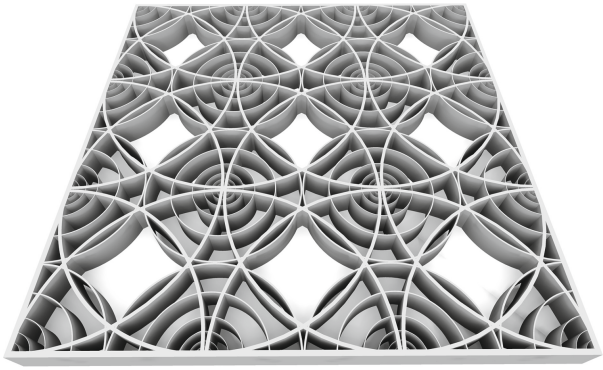


Fig. 5. Mosaics generated in AutoCAD

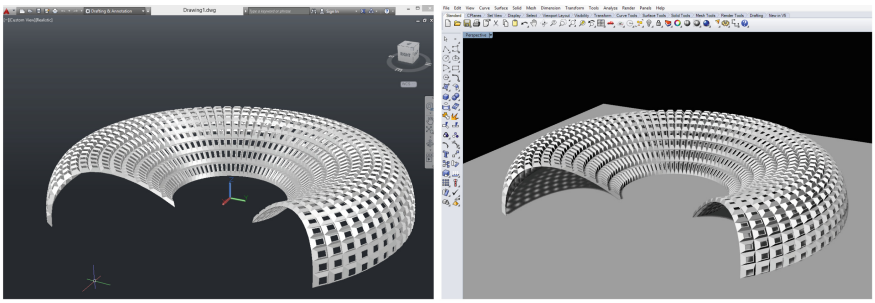
`echo-IOFF-fn`. Theses tags indicate the argument types of the function, where **F**, **O**, and **I**, represent the types `float`, `Object`, and `int`. Also note that imported bindings `full-arc` use the type **O** for their arguments, enabling the type-checker to correctly deal with these imported bindings. Functions and macros such as `p-div`, `p-sub`, or `p-call` are used to implement Processing's language primitives. Function are defined within a `let/ec` form, to support return semantics in functions. However, `let/ec` is not always needed and can be removed, for instance, in the case of unnecessary tail returns or when functions have return type `void`.

```
(p-function (echo-IOFF-fn n p a r)
  (let/ec return
    (p-block
    (p-if (p-eq n 1)
      (p-block (p-call fullArc-OOOOO-fn p r ang HALF_PI h))
      (p-block (p-call fullArc-OOOOO-fn p (p-div r (p-call
   fib-O-fn n)) a HALF_PI h)
        (p-call echo-IOFF-fn (p-sub n 1) p a r))))))
```

Fig. 6. Generated Racket code for `echo`

We demonstrate another example (shown in Fig. 7) of our Processing implementation using libraries that are written in another language and renders designs in AutoCAD and Rhinoceros 3D. To produce this example, our Processing code `requires "elliptic-torus.rkt"`, a library written in the Racket language that is capable of generating highly parametric elliptic torus. Using this library, we can specify in Processing, the domain range, the thickness of the surface, the size of the surfaces' holes, etc.

The possibility of accessing libraries written in different languages of the Racket ecosystem enables Processing users to take advantage of the capabilities



```
require "elliptic-torus.rkt";
float aMin = QUARTER_PI, aMax = 7 * aMin, h = .005;

ellipticTorus(xyz(0,0,0), h, .03, .5, aMin, aMax, 0, TWO_PI);
```

Fig. 7. Elliptic torus generated in AutoCAD and Rhinoceros 3D

of these libraries in their artistic endeavours. Moreover, these examples demonstrate that users can effortlessly migrate to our system and directly use libraries that were previously developed in Racket.

8 Conclusion

Translating a high-level language to another enables the possibility of accessing libraries that are written in different languages. Combining Processing with Racket, allows users to access libraries written in any language of the Racket ecosystem. One particularly important library is Rosetta, a portable Generative Design library that allows architects to use Processing to generate designs in a CAD application, thus providing a motivating reason for the architecture community to use our system.

Our implementation follows the common compiler pipeline architecture, generating semantically equivalent Racket code and loading it into Racket's VM. Our approach was to develop the parts of the language that Processing users most need, that empower them to write simple scripts. In future, our goal is to further develop our existing work, progressively introducing more advanced mechanisms, such as implementing Processing's class system and exception handling.

Acknowledgements. This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/ 2013, and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

References

1. Reas, C., Fry, B.: Processing: programming for the media arts. AI Soc. **20**(4), 526–538 (2006)
2. Flatt, M., Findler, R.B.: The racket guide (2011). http://docs.racket-lang.org/guide/. Accessed 02 May 2014
3. Lopes, J., Leitão, A.: Portable generative design for CAD applications. In: Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture, pp. 196–203 (2011)
4. Flatt, M.: Creating languages in racket. Commun. ACM **55**(1), 48–56 (2012)
5. Ramos, P.P., Leitão, A.M.: An implementation of python for racket. In: 7th European Lisp Symposium, p. 72 (2014)
6. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 132–141. ACM (2011)
7. Maeda, J.: Design by Numbers. MIT Press, Cambridge (1999)
8. Resig, J., Fry, B., Reas, C.: Processing. js (2008)
9. Bader-Natal, A.: Sketchpad (2011). http://sketchpad.cc/. Accessed 28 April 2015
10. Ashkenas, J.: Ruby-processing (2015). https://github.com/jashkenas/ruby-processing. Accessed 28 April 2015
11. Feinberg, J., Gilles, J., Alkov, B.: Python for processing (2014). http://py.processing.org/. Accessed 28 April 2015
12. Gray, K.E., Flatt, M.: Compiling java to PLT scheme. In: Proceedings of 5th Workshop on Scheme and Functional Programming, pp. 53–61 (2004)
13. Gray, K.E., Flatt, M.: ProfessorJ: a gradual introduction to java through language levels. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 170–177. ACM (2003)
14. Findler, R.B., Flanagan, C., Flatt, M., Krishnamurthi, S., Felleisen, M.: DrScheme: a pedagogic programming environment for scheme. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 369–388. Springer, Heidelberg (1997)
15. Louis-Rosenberg, J.: Objexport (2013). http://n-e-r-v-o-u-s.com/tools/obj/. Accessed 29 April 2015
16. Owens, S.: Parser tools: lex and yacc-style parsing (2011). http://docs.racket-lang.org/parser-tools/. Accessed 22 September 2014
17. Flatt, M, Findler, R.B.:. The racket guide, chapter 10.3 continuations (2011). http://docs.racket-lang.org/guide/conts.html?q=continuations. Accessed 05 May 2014
18. Flatt, M.: Composable and compilable macros: you want it when? SIGPLAN Not. **37**(9), 72–83 (2002)