P2R

Implementation of Processing in Racket

Hugo Correia
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
hugo.f.correia@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
antonio.menezes.leitao@ulisboa.tecnico.pt

ABSTRACT

Processing is a programming language and development environment created to teach programming in a visual context. In spite of its enormous success, Processing remains a niche language with limited applicability outside the visual realm. Moreover, architects that have learnt Processing are unable to use the language with traditional Computer-Aided Design (CAD) and Building Information Modelling (BIM) applications, as none support Processing.

In the last few years, the Rosetta project has implemented languages and APIs which enable programmers to work with multiple CAD applications. Rosetta is implemented on top of Racket and allows programs written in JavaScript, AutoLISP, Racket, and Python, to generate designs in different CAD applications, such as AutoCAD, Rhinoceros 3D, or Sketchup. Unfortunately, Rosetta does not support Processing and, thus, is not available to the large Processing community.

In this paper, we present an implementation of Processing for the Racket platform. Our implementation allows Processing to use Rosetta's APIs and, as a result, architects and designers can use Processing with their favourite CAD application. Our implementation involves compiling Processing code into semantically equivalent Racket source code, using a compiler pipeline composed of parsing, code analysis, and code generation phases. Processing's runtime is implemented purely in Racket, allowing for greater interoperability with Racket code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Languages

Keywords

Processing; Racket; Compilers; Language implementation

1. INTRODUCTION

Many programming languages have been created to solve specific needs across a wide range of areas of expertise. 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 are encouraged to share their artistic works. As a result, a wide range of Processing examples are freely available, making it easier for anyone with little, or even no programming knowledge, to experiment with Processing. Among the many benefits that Processing offers, are a wide range of 2D and 3D drawing primitives, and a simple Integrated development environment (IDE), that provides a basic development environment to create new designs.

Despite its enormous success, Processing is a niche programming language with limited applicability outside the visual realm. Architects, for instance, depend on traditional heavyweight CAD and BIM applications (i.g. AutoCAD, Rhinoceros 3D, Revit, etc), which provide APIs that are tailored for that specific CAD. Unfortunately, no CAD application allows users to write scripts in Processing. Therefore, architects that have learnt Processing cannot use their programming knowledge or any of the publicly available examples to program for their favourite CAD tool.

On the other hand, Racket is a descendent of Scheme, that has a wide range of applications, such as teaching newcomers how to program, developing web applications, or creating new languages. Racket encourages developers to tailor their environment to project-specific needs, by offering an ecosystem that allows for the creation of new languages, having direct interoperability with DrRacket and existing Racket's libraries. For instance, Rosetta [2], is Generative Design tool built on top of Racket, that encompasses Racket's philosophy of using different languages to solve specific issues. 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.

Our implementation enables Processing to use Rosetta, therefore allowing architects to prototype designs in their favourite CAD application, using Processing. Our implementation involves compiling Processing code to semantically equivalent Racket source code, using Rosetta's modelling primitives and abstractions. Furthermore, Racket allows us to take advantage of language creation mechanisms [3], that simplify the language development process and its integration with DrRacket. Also, as Racket is our target language, Processing developers gain access to Racket libraries and vice versa. Lastly, as Racket encourages developers to use different languages within the Racket ecosystem, Processing developers could potentially combine their scripts with other languages, such as Python [4].

The following sections describe in greater detail the Processing language and other language implementations that are relevant to our work. Additionally, we describe the main design decisions that were taken for our implementation and a sample of the results obtained so far.

2. PROCESSING

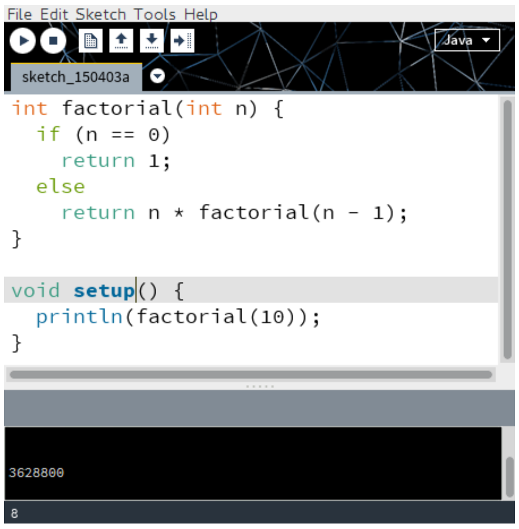
Processing was developed at MIT media labs and was heavily inspired by the *Design by Numbers* [5] project, with the goal to teach computer science to artists and designers with no previous programming experience. The language has grown over the years with the support of an academic community, which has written several educational materials, demonstrating how programming can be used in the visual arts. Also, an online community[1] has been created around the language, allowing users to share and discuss their works. The existence of an online community, good documentation, and a wide range of publicly available examples, has been a positive factor for the language's growth over the years.

The Processing language is built on Java. It is statically typed sharing Java's C-style syntax and implements a wide range of Java's features. The decision of developing Processing as a Java "preprocessor", was due to Java being a mainstream language, used by a large community of programmers. Moreover, Java has a more forgiving development environment for beginners when comparing with other languages used in computer graphics such as C++.

As Processing is meant for beginners, several features were introduced to simplify Java, and consequently, allow users to quickly test their design ideas. In Java, developers have to implement a set of language constructs to develop a simple example, namely a public `class` that implements public `methods` and a static `main` method. These constructs only bring verbosity and complexity to the program. As a result, Processing simplifies this by removing the these requirements, allowing users to write scripts (i.e. simple sequences of statements) that produce designs.

Processing also introduces the notion of a *sketch*, a common metaphor in the visual arts, acting as a sort of project that artists can use to organize their source code. Within a *sketch*, artists can develop their designs in several Processing source files, but that are viewed as a single compilation unit. A *sketch* can operate in one of two distinct modes: *Static* or *Active mode*. *Static mode* supports simple Processing scripts, such as simple statements and expressions. However, the majority of Processing programs are in *Active mode*, which allow users to implement their *sketches* using more advanced features of the language. Essentially, 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: the `setup()` and `draw()` functions. On one hand, the `setup()` function is called once when the program starts. Here the user can define the initial environment properties and execute initialization routines that are required to create the design. On the other hand, the `draw()` function runs after the `setup()` and executes the code that draws the design. The control flow is simple, first `setup()` is executed, setting-up the environment; followed by `draw()` called in loop, continually rendering the sketch until stopped by the user.

[1]`http://openprocessing.org/`



```
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}

void setup() {
  println(factorial(10));
}
```

```
3628800
```

Figure 1: Processing Development Environment

Furthermore, Processing offers users a set of design and drawing tools that are specially tailored for visual artists, providing 2D and 3D drawing primitives along with different 2D and 3D rendering environments. Also, a set of built-in classes are provided, specifically tailored to help artists create their designs. For instance, the `PShape` class serves as a data type that enables users to easily create, manipulate, and reuse custom created design shapes throughout his *sketches*.

On top of being a programming language, Processing offers its users a development environment (presented in Fig 1) called PDE (Processing Development Environment). Users can develop their programs using this simple and straightforward environment, which is equipped with a tabbed editor and IDE services such as syntax highlighting and code formatting. Moreover, Processing users can create custom libraries and tools that extend the PDE with additional functionality, such as, Networking, PDF rendering support, color pickers, sketch archivers, etc.

3. RELATED WORK

Several different language implementations were analysed to guide our development. Our focus was on different implementations for the Processing environment, namely Processing.js, Ruby-processing, and Processing.py. Additionally, we analysed ProfessorJ due to the similarities that Java shares with Processing, and that Scheme shares with Racket.

3.1 Processing.js

Processing.js [6] 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. Therefore, greater interoperability is allowed between both languages, as Processing code is seamlessly integrated with JavaScript and Processing's data types are directly implemented in 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[2] is an alternative online IDE for Processing.js, that allows users to create and test their design ideas online and share them with the community.

3.2 Ruby-processing & Processing.Py

Ruby-processing[3] and Processing.py[4] produce Processing 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 run when new changes are saved) and live coding, which are functionalities that are not present in any other implementation.

3.3 ProfessorJ

ProfessorJ [7, 8] was developed to be a language extension for DrScheme [9], providing a smoother learning curve for students that are learning Java and offering a set of language levels that progressively cover more complex notions of the language.

ProfessorJ implements a traditional compiler pipeline, that starts with a `lex` and `yacc` parsing phase, that produces 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, therefore name mangling techniques were implemented to correctly support Java's multiple namespaces in Scheme.

[2]`http://sketchpad.cc/`
[3]`https://github.com/jashkenas/ruby-processing`
[4]`http://py.processing.org/`

Moreover, Java's built-in primitive types and some classes are directly implemented in Scheme, while remaining classes are implemented in Java. Classes such as Strings, Arrays, and Exceptions are mapped directly to Scheme forms. Implementing these classes in Scheme is possible (with some constraints) due to similarities in both languages which, in turn, allow for a high level of interoperability between both languages.

Finally, ProfessorJ is fully integrated with DrScheme, providing a development environment that offers syntax highlighting, syntax checking, and error-highlighting for Java code. This is possible due to preserved source location information throughout the compilation pipeline.

4. SOLUTION

Although previously presented implementations are relevant for our solution, they do not fit entirely in our work's scope. Analysing Processing.js and Processing.py, we observe that having an IDE is a fundamental feature for Processing users. Also, both Processing.js and ProfessorJ implement their APIs directly in their target language, permitting greater interoperability between the source and target language. However, Processing.js, Processing.py, and Ruby-processing, do not allow designs to be visualized in a CAD, and, in spite of Java and Processing sharing many features, the differences require a custom tailored solution. Finally, Ruby-processing presents some relevant features that are useful for designs, namely live coding. Yet, as both Processing.py and Ruby-processing translate to the JVM, they are not relevant to our work.

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. The following sections explain how our compiler was developed and structured, presenting the main design decisions taken.

4.1 Module Decomposition

To better understand the main modules of our compiler, **Fig 2** illustrates the main Racket modules that are used, as well as the dependencies between them. We divided the modules in two major groups: the *Compiler* and *Runtime* modules. In the following paragraphs, we provide a detailed description of the most important modules.

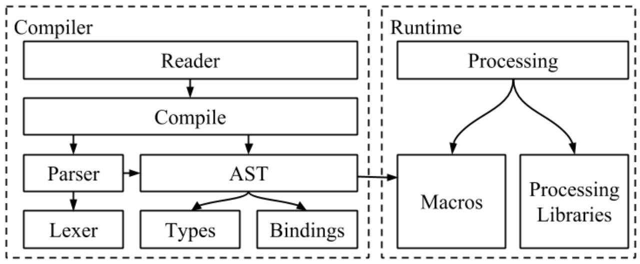


Figure 2: Main module decomposition and dependencies. The arrows indicate a uses relationship between modules - module A uses (⟶) module B

4.1.1 Compiler Modules

Reader Module. To add Processing as a new language module [10], a new specifically tailored `reader` is needed for Processing. This enables Racket to parse Processing source code and transform it to target Racket code. The `reader` must provide two important functions: `read` and `read-syntax`, and receive an `input-port` as input, differing in their return value. The former produces a list of S-expressions, while the latter generates `syntax-objects` (S-expressions with lexical-context and source-location information). The `reader` uses functions provided from the *Compile module*, to create and analyse an intermediate representation of the source Processing code, and to generate target Racket code.

Compile Module. The *Compile module* defines an interfacing layer of functions that connects the *Reader module* with the *Parse* and *AST* modules. The main advantage is to have a set of abstractions that manipulate certain phases of the compilation process. For instance, the *Compile module* provides functions that parse the source code, create an AST, check types, and generate Racket code.

Parser & Lexer Modules. The *Parse* and *Lexer* modules contain all the functions that analyse the syntactic and semantic structure of Processing code. To implement the lexer and parser specifications, we used Racket's `parser-tools` [11], adapting parts of ProfessorJ's lexer and grammar according to Processing's needs. The *Lexer* uses `parser-tools/lex` to split Processing code into tokens. To abstract generated tokens by the *Lexer module*, Racket's `position-tokens` are used, as they provide a simple way to save the code's original source locations. Processing's parser definition is implemented using Racket's `parser-tools/yacc`, which produces a LALR parser.

AST Module. Parsing the code produces a tree of `ast-node%`, that abstracts each language construct such as, statements, expressions, etc. These nodes are implemented as a Racket class, containing the original source locations and a common interface which allows the analysis and generation of equivalent Racket code. Each `ast-node%` provides the following methods:

- `->check-bindings`: traverses the AST populating the current scope with defined bindings and their type information;
- `->type-check`: checks each AST node for type errors, promoting types, if necessary;
- `->racket`: generates Racket code using custom defined functions and macros, wrapping them within a `syntax-object` along with the original source information.

Types and Bindings Module. The *bindings module* provides auxiliary data structures needed to store and manage different Processing bindings. We created a `binding%` class to abstract binding information (e.g. modifiers, argument and return types, etc), and a custom `scope%` class to handle Processing's scoping rules. Each `scope%` has a reference to its parent `scope%` and has a hash table that associates identifiers to `binding%` representation. The *types module* has all the necessary functions to check if two types are compatible or if they need to be promoted. As many of Processing's typing rules are similar to Java's, we adapted ProfessorJ's type-checking functions to work with our compiler.

4.1.2 Runtime Modules

The *runtime module* provides all the necessary macros, functions, and data types, that are required by generated Racket code. These functions are provided by the *Processing module* using the *Macros* and *Processing libraries* modules. The former contains necessary source transformations required to generate equivalent Racket code. The latter provides an interface that implements Processing's built-in classes and drawing primitives, using Rosetta to generate designs in several CAD backends.

4.2 Compilation Process

Our Processing implementation follows the traditional compiler pipeline approach (illustrated in Fig 3), composed by three separated phases, namely parsing, code analysis, and code generation.

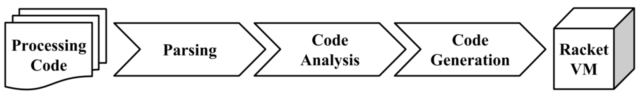


Figure 3: Overall compilation process

Parsing. The initial compilation process starts by the parsing phase, which is divided in two main steps. First, Processing source code is read from the `input-port` and transformed into tokens. Secondly, tokens are given to LALR parser, building an AST of Racket objects, that will be analysed in subsequent phases.

Code Analysis. Following the parsing phase, a series of checks must be made to the generated AST. This is due to some differences between Processing's and Racket's language definitions. For instance, Processing has static type-checking and has 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 solve compatibility issues, in order to generate semantically equivalent Racket code.

Initially, the AST is traversed by repeatedly calling `->check-bindings` on child nodes, passing the current `scope%`. When a new definition is created, be it a function, variable, or class, the newly defined binding is added to the current scope along with its type information. Each time a new scope is created in Processing code, a new `scope%` object is created to represent it, referring to the current `scope%` as its parent. These mechanisms are needed to implement

Processing scoping rules and type-checking rules. For example, to type-check a function call, the information of the return type, arity, and argument types is needed to correctly type-check the expression.

Secondly, the type-checking procedure runs over the AST by calling `->type-check` on the topmost AST node. As before, it repeatedly calls `->type-check` 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 error occurred.

Code Generation. After the AST is fully analysed and type-checked, semantically equivalent Racket code can be generated. To achieve this, every AST node implements `->racket`, which uses custom defined macros and functions to produce Racket code. This code is then wrapped in a `syntax-object` along with source information saved by the AST. Subsequently, these `syntax-objects` will be consumed by `read-syntax` at the reader level. Afterwards, Racket will expand the define macros and load the generated code into Racket's VM. By using macros, we can create human-readable boilerplate Racket code that can be constantly modified and tested.

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*, 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, `func` 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: `float foo(gloat x, float y){ ... }` would be translated to `(define (foo-FF-fn x y)...)`.

An initial implementation of classes has been developed, by mapping Processing classes into Racket classes, reusing some of ProfessorJ's ideas. Instance methods are translated directly into Racket methods, therefore instance method `public void foo()` is translated to Racket's public methods, `(define/public (foo-fn)...)`. On the other hand, static methods are implemented as a Racket function, by appending the class name to the function's name. For example, a method `static void foo()` of class `Foo` will be translated to `(define (Foo.foo-fn)...)`. Also, there is an issue with constructors, as Processing can have multiple constructors. This problem is solved by adapting `new` to find the appropriate constructor to initialize the object.

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. While in active mode, global statements are restricted, thus when generating code for global statements we check if the code is in *Active mode*, signalling an error if true.

4.3 Runtime

Our runtime is implemented directly in Racket, due to the necessity of integrating our implementation with Rosetta. Processing offers a set of built-in classes that provide common design abstractions that aid users during their development process. For instance, `PVector`, abstracts 2 or 3 dimensional vectors, useful to describe positions or velocities. These will be implemented directly in Racket, allowing for greater performance and interoperability.

However, this presents some important issues. First, as Racket is a dynamically typed language, the type-checker, at compile time, cannot know what are the types of Racket bindings. To solve this issue we introduced a new type in the type hierarchy, acting as an opaque super type that the type-checker ignores when type checking these bindings. On the other hand, as Processing primitives and built-in classes are implemented in Racket, we also have the problem of associating type information for these bindings. To solve this issue, we created a simple macro (**Fig 4**), 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. Alternatively, instead of using a custom macro, we could of written Processing's APIs in `typed/racket`, as types can be associated to Racket definitions. Yet, this alternative was not used due to possible type incompatibilities with Processing's and Racket's type hierarchy. Secondly, because the gain of using `typed/racket` [12] would not be significant due to Rosetta and the compiler being written in untyped racket.

```
(define-syntax-rule
  (define-types (id [type arg] ... -> rtype)
     body ...)
  (begin
    (add-binding! rtype 'id (type ...))
    (define (id arg ...) body ...)))
```

Figure 4: Macro that associates Processing types to a definition

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 lets us generate designs in an OpenGL render, but also gives 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 supply Processing developers with different drawing primitives unavailable in Processing's core environment. Therefore we are able to augment Processing's core capabilities with additional drawing primitives and design approaches, that empower users to explore different designs.

4.4 Interoperability

Mapping Processing constructs directly into Racket's allows for greater interoperability between both languages. At the moment, each Processing module is translated to a Racket module. As a result, to use Racket code within a Processing module, a custom import mechanism was created. A `require` statement was introduced that maps into Racket's `require`, allowing Racket modules (or any other language of the Racket ecosystem) to be referenced within a Processing module. Nonetheless, this decision has a major issue in regard to Processing's identifiers, as they are not compatible with Racket's. Racket allows for identifiers to be composed of characters such as '?', '-', or '+', yet Processing does not. As result, we are unable to use these bindings in our Processing scripts. This issue can be solved by using two different approaches.

The first approach is to automatically rename all of provided bindings of the required module, by using a custom set of name renaming rules. For instance, Racket bindings separated with '-' characters are translated to camel case, i.e. `foo-bar` is converted to `fooBar`. The second approach is to force the developer to create his custom name mappings by creating a Racket module that does the name conversion. Clearly, the first approach provides a clearer and quicker way of using a Racket module. However, as renaming rules applied are debatable, the developer is free to create his custom mapping from our initial transformation.

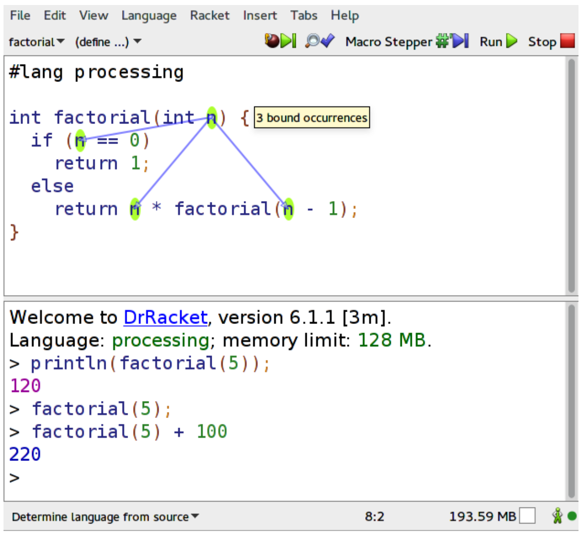
On the other hand, we want to be able to use a Processing module in other languages of the Racket ecosystem. To correctly implement these mechanisms, Processing's modifiers (i.e. `public`, `private`, etc.) are used to provide bindings to other modules, mapping them into Racket's `provide`.

4.5 Integration with DrRacket

Processing developers are familiar with an IDE (the PDE) that offers them a set of common IDE tools, such as syntax highlighting or code formatting. DrRacket as a pedagogical IDE, shares some of the PDE's features, providing a similar development environment to Processing developers and allowing them to easily make the transition to our system. DrRacket's IDE services use source locations to operate, therefore by saving this information and passing it along through the compilation process, we can easily integrate our Processing implementation with DrRacket's features (illustrated in Fig 5).

A relevant feature that Racket offers is a REPL, which is common in many Lisp descendants. However, currently no Processing implementation provides a REPL to its users. Therefore, having a REPL would be a major advantage to the PDE environment, as it would provide users a mechanism to test specific parts of their code, being a good mechanism for beginners to learn and immediately experiment new ideas.

Due to Racket's language development capabilities, this feature was easily implemented by creating a custom function to compile REPL interactions for Processing. However, as Processing is a statement based language, REPL interactions will not produce expressions. So we created a new parser rule to implement REPL interactions, adding it to the


File Edit View Language Racket Insert Tabs Help
factorial (define ...)
Macro Stepper Run Stop
3 bound occurrences
Determine language from source 8:2 193.59 MB

```
#lang processing

int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: processing; memory limit: 128 MB.
> println(factorial(5));
120
> factorial(5);
> factorial(5) + 100
220
>
```

Figure 5: Processing in DrRacket

parser generator's start symbols. This way Racket's `parser-tools` produces different parsing procedures for each start symbol, which we can use according to the type of interaction we are manipulating. The interactions shown in **Fig 5** shows how we can use the REPL for Processing. For example, note that `println(factorial(5));` returns the result of factorial of 5 by producing a print side-effect, while in `factorial(5) + 100` the returned result is the actual expression that is produced by the add operator.

5. EXAMPLE

In this section, we illustrate an example of code that generates a double helix (**Fig 6**) using our system. Our current implementation is still a work in progress, hence the compilation results are subject to change. The code illustrated in **Fig 7** shows a Processing example that generates the helix illustrated in **Fig 6**.

The double helix is drawn by using a recursive function that repeatedly renders a pair of spheres connected by a cylinder, along a rotating axis. This example is a case of an *Active mode* sketch, as function definitions are present. Also, Processing's design flow is demonstrated by the use of `setup()` and `draw()`. In `setup()`, we use the `backend` function (provided by Rosetta) to define the rendering backend to use, which in this case is AutoCAD. On the other hand, `draw()` executes `helix()` to produce the design in AutoCAD. **Fig 8** presents the Racket code that is produced by our compiler.

The first point worth mentioning is that function identifiers are renamed to support multiple namespaces. We can see that `helix` identifier is translated to `helix-FF-fn`. The **F** is to indicate that the function has 2 arguments that are of type `float`. Also, we can see that `setup()` and `draw()` are mangled as well, by appending `fn` to their name. Functions and macros such as `p-mul`, `p-sub`, or `p-call`, are defined in the runtime modules, implementing Processing's semantics. Variable definitions are translated by using `p-declaration`, which is a macro that generates a Racket `define-values` form, using a sequence of `identifier` and `value` pairs. To

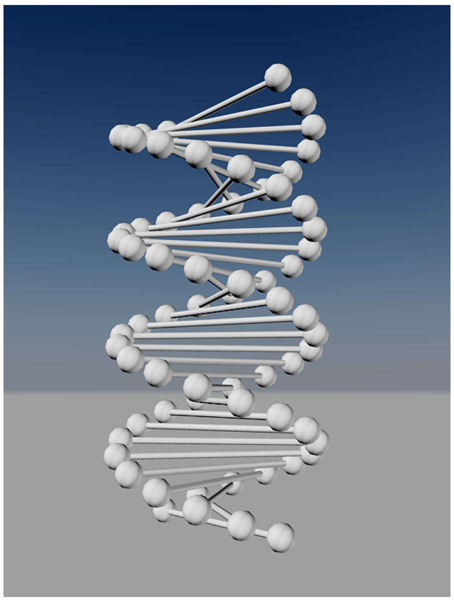


Figure 6: Double helix generated from Processing code using AutoCAD

declare variables (i.e. `float x,y;`), the `value` is stored using Racket's `undefined`. Mathematical operators (`p-mul`, `p-add`) are implemented as Racket functions, yet, do not overflow as Processing. On the other hand, the `sphere` and `cylinder` drawing primitives, are specially tailored to map into Rosetta's operators. For instance, `cylinder` is a good example of an operator that Rosetta provides, but that is not available in the current Processing environment.

At the moment, we observe that all function definitions have their body wrapped in a `let/ec` form. This is injected to support `return` statements within functions. Although performance will be limited by the chosen connection interface and rendering backend, the usage of `let/ec` is a clear example that brings additional performance overhead with no possible gains, as the return type is `void`. Thus an optimization is required to remove `let/ec`, for cases that jump statements are not present or when the last statement is a `return`.

```
float r =  15, height = 2;

void helix(float z, float ang) {
  float x1 = r*cos(ang), y1 = r*sin(ang);
  float x2 = r*cos(PI+ang), y2 = r*sin(PI+ang);

  sphere(x1, y1,  z, 2);
  cylinder(x1, y1,  z, 0.5, x2, y2, z);
  sphere(x2, y2,  z, 2);

  if(ang > 0) helix(z + height, ang - PI/8);
}
void setup() { backend(autocad); }
void draw()  { helix(0, 4 * PI); }
```

Figure 7: Processing code example of the Double Helix

```
(p-declaration (r 15.0) (height 2.0))

(define (helix-FF-fn z ang)
 (let/ec return
  (p-block
   (p-declaration
     (x1 (p-mul r (p-call cos-F-fn ang)))
     (y1 (p-mul r (p-call sin-F-fn ang))))
   (p-declaration
     (x2 (p-mul r (p-call cos-F-fn
                     (p-add PI ang))))
     (y2 (p-mul r (p-call sin-F-fn
                     (p-add PI ang)))))
   (p-call sphere-FFFI-fn x1 y1 z 2)
   (p-call cylinder-FFFFFFF-fn
         x1 y1 z 0.5 x2 y2 z)
   (p-call sphere-FFFI-fn x2 y2 z 2)
   (when (p-gt ang 0)
    (p-call helix-FF-fn
           (p-add z height)
           (p-sub ang (p-div PI 8.0)))))))

(define (setup-fn)
 (let/ec return
  (p-block (p-call backend-0-fn autocad))))

(define (draw-fn)
 (let/ec return
  (p-block (p-call helix-FF-fn 0 (p-mul 4.0 PI)
      ))))

(p-initialize))
```

Figure 8: Generated Racket code

Finally, a `p-initialize` macro is added to implement Processing's workflow semantics. This macro is responsible of ensuring that the `setup()` and `draw()` functions are called, if defined by the user. `p-initialize` is implemented by using Racket's `identifier-binding` to check if the `setup-fn` and `draw-fn` are bound in the current environment.

6. CONCLUSION

Implementing Processing for Racket benefits architects and designers, by allowing them to develop with Processing in a CAD environment. Also, the ability to provide new design paradigms offered by Rosetta is a strong reason for the architecture community to use our solution. The implementation follows the common compiler pipeline architecture, generating semantically equivalent Racket code and loading it into Racket's VM. Our strategy was to implement Processing's primitives directly in Racket to easily access Rosetta's features and allow a greater interoperability with Racket. Also, we have developed mechanisms to access Processing code from Racket and vice versa.

Currently, our development approach was to first fulfil the most basic needs of Processing users (the ability to write simple scripts) and present visual results in a CAD application. Afterwards, our goal is to build-upon our existing work, and progressively introduce more advanced mechanisms, such as implementing inheritance and interfaces in classes, support live coding, or adapt Processing's exception system to Racket. To provide a better environment for Processing developers, we plan to further adapt DrRacket by creating an editor mode with better syntax highlighting for Processing. Also, adding visual support to REPL interactions would be a huge advantage to our implementation, as it would allow users to immediately visualize geometric shapes in the IDE without loading up the rendering backend. Finally, optimizations can be made to generated Racket code to improve the quality and performance of the generated Racket code.

7. 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.

8. REFERENCES

[1] Casey Reas and Ben Fry. Processing: programming for the media arts. *AI & SOCIETY*, 20(4):526–538, 2006.

[2] José Lopes and António Leitão. Portable generative design for cad applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pages 196–203, 2011.

[3] Matthew Flatt. Creating languages in racket. *Communications of the ACM*, 55(1):48–56, 2012.

[4] Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. *7 th European Lisp Symposium*, page 72, 2014.

[5] John Maeda. *Design by Numbers*. MIT Press, Cambridge, MA, USA, 1999.

[6] John Resig, Ben Fry, and Casey Reas. Processing. js, 2008.

[7] Kathryn E Gray and Matthew Flatt. Compiling java to plt scheme. In *Proc. 5th Workshop on Scheme and Functional Programming*, pages 53–61, 2004.

[8] Kathryn E Gray and Matthew Flatt. 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*, pages 170–177. ACM, 2003.

[9] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *Journal of functional programming*, 12(02):159–182, 2002.

[10] M Flatt and RB Findler. Creating languages: The racket guide.

[11] Scott Owens. Parser tools: lex and yacc-style parsing.

[12] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.